

Whitemarsh  
Information Systems Corporation

# Database Management Systems: Understanding and Applying Database Technology

Michael M. Gorman  
January 2006

Whitemarsh Information Systems Corporation  
2008 Althea Lane  
Bowie, Maryland 20716  
Tele: 301-249-1142  
Email: [whitemarsh@wiscorp.com](mailto:whitemarsh@wiscorp.com)  
Web: [www.wiscorp.com](http://www.wiscorp.com)

## 2<sup>nd</sup> Edition

This second edition is largely the same as the first with the following changes. Throughout the text, the terms: data record type, data column, and data record instance have been changed to table, column, and row. This is because these terms are the most commonly used today and because most DBMSs support the SQL language. Since the SQL 1999 standard, tables and columns can take on significant complexity. Tables can have subtyped tables. Columns can have sets, multi-sets, and methods. This makes SQL based DBMSs to be as simple as pure relational DBMSs and as complex as the most sophisticated network DBMS.

In Chapter 1 material was added that extended the progression of ANSI database language standards beyond 1992. It is now up to date with 2005.

In Chapter 3, because the SQL facilities have developed so significantly beyond relational, a new section has been added to explain the SQL Data Model. Simply stated, there are no more relational DBMSs. Chris Date might even state that there never were. Regardless, the majority of DBMSs today support the SQL language. SQL 1999 dramatically broke with the relational model to the extent that most modern SQL DBMSs have adopted some or all of the network, hierarchical and independent logical file data model facilities.

Because the SQL 1999 and SQL 2003 data model is largely a melding of data model facilities of the network, hierarchical, independent logical file, and relational data models, a good quantity of the material from Chapter 3 did not have to be removed to make this book “up to date.” What goes around, comes around, I guess. In some sense, the original architectures of DBMSs in the early 1960s has been validated. Updated and changed of course has been the size of databases and the quantity of transactions per second. So, while the scale has changed, the fundamental DBMS technology building blocks of Logical, Physical, Interrogation, and System Control has remained.

The goal of this book is to provide an overall technology-based framework and components within the DBMS technology framework for understanding DBMS technology. Every DBMS represents a subset collection of these components. Thus, the book is an overarching architecture for DBMSs. This was true in 1986 when this book was first drafted, and it is true today, almost 20 years later.

Michael M. Gorman, Secretary  
ANSI INCITS H2 Technical Committee on Database Languages

January, 2006



## **To: ANSI NCITS H2 Technical Committee on Database**

There is no measure to the debt the DBMS community owes to the ANSI INCITS H2 Technical Committee on Database Languages. ANSI means the American National Standards Institute. INCITS is the International Committee on Information Technology Standards. H2 is just the letter-number reference to the committee's official name, Technical Committee on Database Languages. This organization has met continuously since the Spring of 1978 to create the database language standards that exist today and that are accepted world wide. Within the H2 committee, Don Deutsch of Oracle Corporation, Phil Shaw of Sybase Corporation, Jim Melton of Oracle Corporation, and Len Gallagher of NIST (formerly National Bureau of Standards) deserve special mention. They are clearly the unsung heroes of DBMS.

Don Deutsch has been the chairman of H2 for twenty-five of its twenty-seven years. While on the committee, Don also managed the DBMS standards work of the U.S. National Institute of Standards and Technology (NIST, formerly the U.S. National Bureau of Standards), and for the last several years, he has worked on the vendor side of the DBMS community at General Electric, Sybase and Oracle. Don has managed the committee superbly. The committee has had over 100 different corporate and individual members, and about 2500 technical papers have been presented. Don not only crafts the agendas and manages the meetings, he has also managed to bring to the market the first two DBMS standards: NDL and SQL. Under Don's chairmanship, H2 has finished two additional SQL standards: one for referential integrity, and the other for the interface specifications between SQL and six different programming languages (Ada, COBOL, C, Fortran, Pascal, and PL/I). These standards are described in Chapter 1, 3, and on the Whitmarsh website, [www.wiscorp.com](http://www.wiscorp.com) on the SQL standards page. The quality of H2's work is a direct result of Don's leadership, patience, and persistence.

Phil Shaw was the IBM representative to H2 since the first meeting in 1978. Phil provided the vast majority of the technical papers for the network data language standard, NDL, and because of Phil, NDL is a very high quality specification. Phil retired from IBM but couldn't just "walk on the beach." So he went back to work for Sybase. While much is said in the press and on the lecture and seminar circuit, of the work of the originators of the SQL language, they are its fathers in name only, because it was Phil Shaw, over the years, without notice or attention, who has toiled days, weeks, and years drafting, redrafting, and word smithing the ANSI SQL 1986 standard. And, once the American NDL and SQL 1986 and SQL 1989 standards were finished, it was Phil who was the key American representative to the International Standards Organization (ISO), enabling both NDL and SQL 1986 and SQL 1989 to become international standards.

Jim Melton of Oracle Corporation has been the editor of the SQL base documents since 1987. Jim serves as the editor to both ANSI/H2 and the ISO Database Languages rapporteur groups. The SQL document set has grown from a little over 100 pages (SQL 1986) to well over 4500 pages (SQL/1992, SQL/1999, SQL/2003, and SQL/200n). Jim spends 40 to 60 hours every week integrating the results of each ANSI/H2 meeting (six per year) and of the two ISO meetings. The progress of H2 and the ISO/DBL groups is a direct consequence of Jim's work.

Len Gallagher of the NIST was both the international representative from H2 to the International Standards Organization (ISO), and the convener of the ISO database languages rapporteur group through the middle 1990s. Len's ISO group consisted of representatives from



about a dozen countries, who must review proposals and then arrive at a consensus. Under Len's stewardship, the ISO/DBL rapporteur group managed to avoid confrontation and the development of nationalistic fractions. Len's ISO representation was taken over by an Krishna Kulkarni from IBM. It was and is because of Jim, Len, Phil, and Krishna consensus, not dominance, moves the agenda.



# Table of Contents

<b>Preface</b> .....	xxi
----------------------	-----

## 1

<b>ANSI DATABASE STANDARDS</b> .....	1
1.1 In the Beginning .....	1
1.2 Reference Models .....	2
1.2.1 The 1975 ANSI Reference Model .....	2
1.2.2 The ISO Reference Model .....	5
1.2.2.1 Levels of Abstraction .....	6
1.2.2.2 Levels of Data .....	7
1.2.2.3 Generalized Reference Model .....	10
1.2.2.4 Detailed Reference Models .....	11
1.2.2.5 ISO Reference Model Summary .....	11
1.3 ANSI Database Committees: H2 and H4 .....	11
1.4 The Battle over Data Models .....	12
1.5 DBMS Standards .....	12
1.6 ANSI/NDL .....	15
1.7 ANSI/SQL .....	20
1.8 ANSI/SQL (1999) .....	26
1.8.1 Foundation Components .....	27
1.8.2 Call Level Interface .....	28
1.8.3 SQL/Multi Media (MM) Components .....	28
1.8.4 SQL Persistent Stored Module Language Components .....	29
1.8.5 SQL Transaction and Connection Management .....	29
1.9 The ANSI/IRDS .....	30
1.10 Database Standards Summary .....	32

## 2

<b>DBMS APPLICATIONS AND COMPONENTS</b> .....	37
2.1 Application Classifications .....	37
2.2 Static and Dynamic Relationships .....	38
2.3 The Nature of a Static Relationship Application .....	39
2.4 The Nature of a Dynamic Relationship Application .....	40
2.5 Problems and Benefits of Static Relationship DBMS .....	40
2.6 Problems and Benefits of Dynamic Relationship DBMS .....	42
2.7 Implementing the Static Relationship Application .....	43
2.7.1 The Logical Database .....	43
2.7.2 The Physical Database .....	44
2.7.3 Interrogation .....	44
2.7.4 System Control .....	45
2.7.5 Static Relationship Application Summary .....	45
2.8 Implementing the Dynamic Relationship Application .....	45
2.8.1 The Logical Database .....	45
2.8.2 The Physical Database .....	46
2.8.3 Interrogation .....	46
2.8.4 System Control .....	47
2.8.5 Dynamic Relationship Application Summary .....	47
2.9 DBMS Components and Subcomponents .....	47

2.10	The DBMS .....	49
2.10.1	The Logical Database .....	49
2.10.2	The Physical Database .....	51
2.10.3	Interrogation .....	52
2.10.4	System Control .....	52
2.11	DBMS Issues .....	53
2.12	Application Components and DBMS Components .....	55
2.13	DBMS Requirements Summary .....	56

3

<b>THE LOGICAL DATABASE</b> .....	59
3.1 Definition .....	59
3.2 Logical Database Components .....	59
3.2.1 Domains .....	60
3.2.2 Columns .....	61
3.2.2.1 Column Names .....	61
3.2.2.2 Data Types .....	63
3.2.2.3 Column Structures .....	64
3.2.2.4 Column Integrity Rules .....	67
3.2.3 Tables .....	67
3.2.3.1 Table Name Clause .....	68
3.2.3.2 Table Integrity Clauses .....	68
3.2.3.3 Columns .....	69
3.2.3.4 Column Roles .....	71
3.2.3.5 Physical Implications of Table structures .....	73
3.2.3.6 Table Summary .....	74
3.2.4 Relationships .....	74
3.2.4.1 Relationship Types .....	75
3.2.4.1.1 One-to-Many Relationships .....	76
3.2.4.1.2 Owner-Multiple-Member Relationships .....	78
3.2.4.1.3 Singular-Single-Member Relationships .....	81
3.2.4.1.4 Singular-Multiple-Member Relationships .....	82
3.2.4.1.5 Recursive Relationships .....	82
3.2.4.1.6 Many-to-Many Relationships .....	88
3.2.4.1.7 One-to-One Relationships .....	91
3.2.4.1.8 Inferential Relationships .....	93
3.2.4.1.9 Relationship Type Summary .....	94
3.2.4.2 Relationship Implementation Mechanisms .....	94
3.2.4.2.1 Dynamic Relationships .....	98
<b>3.2.4.2.2 Static Relationships</b> .....	102
3.2.4.3 Relationship Integrity .....	105
3.2.4.3.1 Static Insertion Referential Integrity .....	105
3.2.4.3.2 Dynamic Insertion Referential Integrity .....	106
3.2.4.3.3 Static Retention Referential Integrity .....	106



	3.2.4.3.4	Dynamic Retention Referential Integrity . . . . .	107
	3.2.4.3.5	Referential Integrity Surprises . . . . .	107
3.2.5	Operations . . . . .		109
	3.2.5.1	Row Operations . . . . .	109
	3.2.5.2	Relationship Operations . . . . .	114
	3.2.5.3	Combination Operations . . . . .	125
3.2.6	Logical Database Components Summary . . . . .		125
3.3	Data Models . . . . .		126
	3.3.1	Data Models and DBMSs . . . . .	136
	3.3.2	Static Data Models . . . . .	137
	3.3.2.1	Network Data Models . . . . .	138
	3.3.2.2	Hierarchy . . . . .	141
	3.3.3	Dynamic Data Models . . . . .	144
	3.3.3.1	Independent Logical File Data Model . . . . .	144
	3.3.3.2	Relational Data Model . . . . .	148
	3.3.3.3	Dynamic Data Model Summary . . . . .	149
	3.3.4	Data Model Summary . . . . .	150
	3.3.5	ANSI Standard SQL Data Model . . . . .	150
	3.3.5.1	Data Model: Data Structures . . . . .	152
	3.3.5.2	Data Model: Relationships Background . . . . .	153
	3.3.5.3	Data Model: Operations Background . . . . .	154
3.4	Data Definition Language . . . . .		155
3.5	Logical Database Summary . . . . .		160

## 4

	<b>THE PHYSICAL DATABASE . . . . .</b>	<b>163</b>
4.1	Physical Database Components . . . . .	163
4.2	Storage Structure . . . . .	163
	4.2.1 Dictionary Component . . . . .	165
	4.2.2 Index Component . . . . .	166
	4.2.2.1 Index Alternatives . . . . .	172
	4.2.2.2 Problem Specification . . . . .	174
	4.2.2.3 Initial Solution . . . . .	175
	4.2.2.4 The Company's DBMS . . . . .	177
	4.2.2.5 Looking Beyond a Popular Myth . . . . .	179
	4.2.2.6 Index Effects on Access Strategies . . . . .	179
	4.2.2.7 Index Structures . . . . .	180
	4.2.2.7.1 Unique Value Component of an Index . . . . .	182
	4.2.2.7.2 Multiple Occurrence Component of an Index . . . . .	184
	4.2.2.8 Multiple Condition WHERE Clauses . . . . .	189
	4.2.2.9 Case Study Summary . . . . .	192



	4.2.2.10	Index Summary .....	193
4.2.3		Relationship Component .....	193
	4.2.3.1	Relationship Basics .....	194
	4.2.3.1.1	Static Relationship Basics .....	194
	4.2.3.1.2	Dynamic Relationships .....	201
	4.2.3.2	Sophisticated Relationships .....	202
	4.2.3.3	Relationship Summary .....	210
4.2.4		Data Component .....	210
	4.2.4.1	File Storage Formats .....	214
	4.2.4.1.1	One Table per File .....	214
	4.2.4.1.2	One File for All Tables .....	217
	4.2.4.1.3	Multiple Tables per File .....	219
	4.2.4.1.4	Multiple Files per Table .....	219
	4.2.4.1.5	Rows and Key Formatted Files .....	222
	4.2.4.2	Table Storage Formats .....	224
	4.2.4.2.1	Fixed Format, Fixed Length Tables .....	224
	4.2.4.2.2	Fixed Format, Variable Length Tables .....	226
	4.2.4.2.3	Variable Format Tables .....	228
	4.2.4.3	Column Storage Formats .....	230
	4.2.4.3.1	Simple Column Formats .....	231
	4.2.4.3.2	Complex Column Formats .....	231
	<b>4.2.4.4</b>	<b>Data Summary .....</b>	<b>234</b>
4.2.5		Data Storage Definition Language .....	236
	4.2.5.1	File Clauses .....	236
	4.2.5.2	Area Clauses .....	237
	4.2.5.3	Rows .....	237
	4.2.5.4	Columns Clauses .....	237
	4.2.5.5	NDL Set Clauses .....	238
	4.2.5.6	Index Clauses .....	238
	4.2.5.7	DSDL Summary .....	239
4.2.6		Storage Structure Summary .....	239
4.3		Access Strategy .....	240
	4.3.1	Static Access Strategy .....	240





4.3.2	Dynamic Access Strategy	245
4.3.3	Comparing Access Strategies	246
4.3.4	Buffer Management	247
4.3.5	Access Strategy Summary	248
4.4	Data Loading	248
4.4.1	<b>Static Data Loading</b>	248
4.4.2	Dynamic Data Loading	251
4.4.3	Load Engineering	251
4.5	Data Update	252
4.5.1	Table Changes	252
4.5.2	Column Changes	253
4.5.3	Relationship Changes	254
4.6	Database Maintenance	256
4.7	Physical Database Summary	257
5		
	<b>INTERROGATION</b>	259
5.1	Interrogation Evolution	259
5.2	DBMSs and Language Orientation	261
5.2.1	<b>Static Relationship DBMSs and HLIs</b>	262
5.2.2	Dynamic Relationship DBMSs and Natural Languages	262
5.2.3	Types of Interrogation Languages	262
5.3	View Facility	264
5.3.1	Language-Independent Specifications	265
5.3.2	Language-Dependent Specifications	265
5.3.2.1	Data Types	265
5.3.2.2	Conversion Rules	265
5.3.2.3	Names	265
5.3.2.4	Natural Language Editing and Default Formats	266
5.3.3	Data Interface Specifications	266
5.3.4	WHERE Clause Facilities	266
5.3.5	Rationale for Views	270
5.4	Screen Development	279
5.5	Relationship Between Screens and Views	282
5.6	<b>System Control Capabilities</b>	283
5.6.1	Audit Trails	283
5.6.2	Message Processing	284
5.6.3	Backup and Recovery	284
5.6.4	Concurrent Operations	285
5.6.5	Multiple Database Processing	287
5.6.6	Security and Privacy	287
5.6.7	Reorganization	288
5.7	Host Language Interface	288
5.7.1	HLI-DBMS Interaction Protocol	288



5.7.2	Access Language Form .....	292
5.7.3	View Access .....	294
5.7.3.1	View Operations .....	294
5.7.3.2	View WHERE Clauses .....	295
5.7.3.3	Run-Unit Sort Clauses .....	297
5.7.4	View Access .....	297
5.7.5	Screen Development and Utilization .....	299
5.7.6	HLI Cursors .....	300
5.7.7	Updating .....	305
5.7.8	DBMS and Database Invocation .....	307
<b>5.7.9</b>	<b>System Control Capabilities</b> .....	<b>308</b>
5.7.10	Static and Dynamic Differences .....	308
5.8	Natural Languages .....	309
5.8.1	Natural Language Types .....	309
5.8.2	Natural Language Execution Alternatives .....	310
5.8.3	Run-Unit Development .....	311
5.9	Procedure-oriented Language .....	311
5.9.1	Basic Capabilities .....	312
5.9.2	View Access .....	315
5.9.2.1	Run-Unit WHERE Clauses .....	315
5.9.2.2	Run-Unit Sort Clauses .....	315
5.9.3	Table Access .....	315
5.9.4	Screen Development and Utilization .....	315
5.9.5	Updating .....	316
5.9.6	Reporting .....	316
5.9.6.1	User Prompting .....	317
5.9.6.2	Statistical Operators .....	318
5.9.6.3	Titles and Subtitles .....	319
5.9.6.4	Control Break Formatting .....	320
5.9.6.5	Page Formatting .....	320
5.9.6.6	Sorting .....	321
5.9.6.7	Graphical Output .....	321
5.9.6.8	Derived Data .....	325
5.9.7	Batch Job Execution .....	325
5.9.8	System Control .....	327
5.9.8.1	Message Processing .....	327
5.9.8.2	Backup and Recovery .....	328
5.9.9	Reorganization .....	328
5.9.10	POL Summary .....	328
5.10	Report Writer .....	328
5.10.1	Basic Capabilities .....	331
5.10.2	View Access .....	331
5.10.3	Reporting .....	331
5.10.4	System Control .....	332



	5.10.4.1	Message Processing .....	332
	5.10.4.2	Multiple Database Processing .....	332
	5.10.4.3	Security and Privacy .....	333
	5.10.5	Report Writer Summary .....	333
5.11		Query-update Languages .....	333
	5.11.1	Basic Capabilities .....	335
	5.11.2	View Access .....	335
	5.11.3	Updating .....	335
	5.11.4	Reporting .....	336
	5.11.5	System Control .....	336
5.12		Choosing the Right Interrogation Language .....	336
5.13		Interrogation Summary .....	337

## 6

		<b>SYSTEM CONTROL .....</b>	<b>339</b>
6.1		System Control Components .....	339
6.2		Audit Trails .....	342
6.3		Message Processing .....	345
6.4		Backup and Recovery .....	352
	6.4.1	Database Backup .....	352
	6.4.2	Database Recovery .....	354
		6.4.2.1 Re-run the Job .....	356
		6.4.2.2 Roll-forward .....	356
		6.4.2.3 Rollback .....	357
		6.4.2.4 Rollback with Roll-forward .....	360
		6.4.2.5 Transaction Rollback .....	360
		6.4.2.6 Backup and Recovery Responsibility .....	364
	6.4.3	Checkpoint and Restart .....	365
	6.4.4	Database Lockout .....	366
	6.4.5	Differences between Static and Dynamic Relationship DBMSs .....	367
	6.4.6	Disabling the Journal File .....	367
	6.4.7	Backup and Recovery Summary .....	368
6.5		Reorganization .....	369
	6.5.1	Logical Database Reorganization .....	370
	6.5.2	Physical Database Reorganization .....	372
		<b>6.5.3 Impact of Reorganization .....</b>	<b>373</b>
	6.5.4	Reorganization Locking .....	373
	6.5.5	Static and Dynamic Differences .....	374
6.6		Concurrent Operations .....	375
	6.6.1	Single or Multiple User DBMS .....	376
	6.6.2	Single or Multiple Database .....	377
	6.6.3	Single and Multiple Threaded DBMS .....	377
	6.6.4	Commonly Existing DBMS Combinations .....	378
		6.6.4.1 Single User, Single Database DBMS .....	379



6.6.4.2	Multiple User, Single Database, Single Thread DBMS	379
6.6.4.3	Multiple User, Multiple Database, Single Thread DBMS	380
6.6.4.4	Multiple User, Single Database, Multiple Thread DBMS	381
6.6.4.5	Multiple User, Multiple Database, Multiple Thread DBMS	381
6.6.5	Complex Storage Structure Effects on Concurrent Operations	381
6.6.6	System Control Operation Conflicts	382
<b>6.6.7</b>	<b>Concurrent Operations Locks</b>	383
<b>6.6.8</b>	<b>Deadly Embrace</b>	383
<b>6.6.9</b>	<b>Static and Dynamic Differences</b>	384
6.6.10	Summary	384
6.7	Multiple Database Processing	384
6.7.1	Types of Multiple Databases	385
6.7.2	Rationale for Multiple Databases	386
<b>6.7.3</b>	<b>Critical Issues</b>	388
6.7.4	Logical Database Impact	389
6.7.5	Physical Database Impact	389
6.7.5.1	Storage Structure	390
6.7.5.2	Access Strategy	390
6.7.5.3	Data Loading	390
<b>6.7.5.4</b>	<b>Data Update</b>	391
6.7.6	Interrogation Impact	391
<b>6.7.7</b>	<b>System Control Impact</b>	392
6.7.7.1	Audit Trails	392
6.7.7.2	Backup and Recovery	392
6.7.7.3	Concurrent Operations	392
6.7.7.4	Security and Privacy	393
6.7.8	Static and Dynamic Differences	393
<b>6.7.9</b>	<b>Multiple Database Summary</b>	394
6.8	Security and Privacy	394
6.8.1	Rationale	394
6.8.2	Areas of Concern	395
6.8.2.1	Logical Database	395
6.8.2.2	Physical Database	395
6.8.2.2.1	Storage Structure	395
6.8.2.2.2	Data Loading	396
6.8.2.2.3	Data Update	397
6.8.2.2.4	Backup	397
6.8.2.3	Interrogation	397
<b>6.8.2.4</b>	<b>System Control</b>	398
6.8.3	Definition Alternatives	400
6.8.4	Trapping and Reporting Security Violations	401
6.8.5	Static and Dynamic Differences	401
6.8.6	Security and Privacy Summary	402
6.9	Installation and Maintenance	402



6.9.1	Initial Installation and Testing .....	403
6.9.2	Special Versions .....	403
6.9.3	Operating Environment Interfaces .....	404
6.9.4	DBMS Versions and Upgrades .....	405
6.9.5	TP Monitor Restrictions .....	406
6.9.6	Managing DBMS Bugs .....	406
<b>6.9.7</b>	<b>Installation and Maintenance Summary .....</b>	<b>406</b>
6.10	Application Optimization .....	407
6.10.1	Logical Database Analysis .....	407
6.10.1.1	Derived Data .....	407
6.10.1.2	Denormalization .....	414
6.10.1.3	Centralization Versus Decentralization .....	416
6.10.1.4	Column Specification .....	417
6.10.1.5	Relationships .....	417
6.10.1.6	Model Transformation .....	419
6.10.1.7	Views .....	420
6.10.2	Physical Database Analysis .....	420
6.10.2.1	Storage Structure .....	420
6.10.2.1.1	Dictionary .....	421
<b>6.10.2.1.2</b>	<b>Index Design .....</b>	<b>421</b>
<b>6.10.2.1.3</b>	<b>Relationships .....</b>	<b>422</b>
6.10.2.1.4	Data .....	422
6.10.2.2	Access Strategy .....	422
6.10.2.3	Data Loading .....	423
6.10.2.4	Data Updates .....	423
6.10.2.5	Database Maintenance .....	423
6.10.3	Interrogation .....	423
6.10.4	System Control .....	424
6.10.4.1	Audit Trails .....	424
6.10.4.2	Backup and Recovery .....	425
6.10.4.3	Reorganization .....	425
6.10.4.4	Security and Privacy .....	425
6.10.4.5	Multiple Database Processing .....	425
6.10.4.6	Concurrent Operations .....	426
6.10.4.7	DBMS Installation and Maintenance .....	427
6.10.5	Static and Dynamic Differences .....	427
6.10.6	Application Optimization Summary .....	427
6.11	System Control Static and Dynamic Differences .....	428
6.12	System Control Summary .....	428
<b>Index</b>	.....	<b>431</b>



## Figures

Figure 1.1 Data Levels and Level Pairs .....	8
Figure 1.2. Data levels and level pairs. ....	8
Figure 1.3 Simplified Diagram of the Abstract Reference Model .....	10
Figure 1.4 NDL Reference Model (1982) .....	16
Figure 1.5. Database management system components addressed by ANSI/NDL .....	18
Figure 1.6a ANSI/NDL versus IDMS/R, VAX-DBMS, & DMS1100 Facilities Schema Comparison .....	21
Figure 1.6b ANSI/NDL versus IDMS/R, VAX-DBMS, & DMS 1100 Facilities Set Definition Comparison .....	22
Figure 1.6c ANSI/NDL versus IDMS/R, VAX-DBMS, & DMS110 Facilities Subschema Comparison .....	23
Figure 1.7 SQL Reference Model (1988) .....	24
Figure 1.8 Database Management System Components Addressed by ANSI/SQL .....	25
Figure 1.9a ANSI/H2 Standards Activity Past, Present, and Predictions .....	33
Figure 1.9b ANSI/H2 Standards Activity Past, Present, and Predictions .....	33
Figure 2.1 Static Dynamic Relationship Application Characteristics .....	38
Figure 2.2 Static-Dynamic Relationship DBMS Characteristics .....	41
Figure 2.5 Database Management Systems Components .....	47
Figure 2.4 Positive and Negative Aspects of an Issue .....	55
Figure 2.4. Flexibility vs Performance impact on DBMS. ....	54
Figure 3.1. Domain Salary .....	60
Figure 3.2. Multi-valued columns .....	64
Figure 3.3. Multiple Dimension Columns .....	65
Figure 3.4. Group Column .....	66
Figure 3.5. Repeating Groups .....	66
Figure 3.6. Job-Type column definition. ....	67
Figure 3.7. Salesperson table definition. ....	69
Figure 3.8. Hierarchical data model with two databases, one with 6 and the other with 4 tables. .....	70
Figure 3.9. Network Data Model. One database with 10 tables. ....	71
Figure 3.10. Regular Owner-member set. Set type: regular, single member .....	76
Figure 3.11. One-to-many relationship .....	77
Figure 3.12. ANSI Network data model relationships. ....	79
Figure 3.13. Relationship Type: Multi-member set .....	80
Figure 3.14. Relationship type: Singular set. ....	81
Figure 3.15. Physical relationship illustration. ....	83
Figure 3.16. Recursive relationship .....	84
Figure 3.17. Hierarchical layout of company organization chart. ....	84
Figure 3.18. Recursive layout of company organization chart. ....	86
Figure 3.19b. Sets of rows loading an IDMS/R recursive relationship. ....	87
Figure 3.20. Many to many relationships .....	88
Figure 3.21. Many to many relationships illustration. ....	89



Figure 3.22a. Many to many relationship (direct) . . . . .	89
Figure 3.22b. Many to many relationship (direct) – instances. . . . .	90
Figure 3.23a. Many to many relationship (decomposed) . . . . .	91
Figure 3.23b. Many to many relationship (decomposed) –instances. . . . .	92
Figure 3.24. Inferential relationship illustration. . . . .	93
Figure 3.25. Inter-row relationships. Static and Dynamic DBMS Comparison. . . . .	95
Figure 3.26. Chair represents forward and backward alphabetical order. Logical relationship illustration. . . . .	96
Figure 3.27. Chain represents forward alphabetical order. Logical relationship illustration. . . . .	97
Figure 3.28. Relational data model: Ten simple tables . . . . .	99
Figure 3.29. Teaches Many to many relationship between student and teacher. . . . .	100
Figure 3.30. A many to many relationship: Taught By between student and teacher. Tree table approach. . . . .	100
Figure 3.31. A many to many relationship: Taught By between student and teacher. A two table approach. . . . .	101
Figure 3.32. Two many to many relationships: Taught By between student to teacher, and Tutors between teacher and student. . . . .	102
Figure 3.33. Examples of relative record addresses. . . . .	103
Figure 3.34a. Ambiguous referential integrity access path among tables. . . . .	108
Figure 3.35. Three relational tables: player, team and team-player. . . . .	110
Figure 3.36. Row Operations. . . . .	111
Figure 3.37. Relational Operation: Project (Columns B and D from Table Foo) . . . . .	111
Figure 3.38. Relational Operation: Project example. . . . .	112
Figure 3.39. Relational Operation: Select. . . . .	113
Figure 3.40. Relational Operation: Select example . . . . .	113
Figure 3.41. Relationship Operations . . . . .	114
Figure 3.42. Relational operation: Intersection. . . . .	115
Figure 3.43. Relational operation: Intersection example. . . . .	115
Figure 3.44. Relational operation: Difference . . . . .	116
Figure 3.45. Relational Operation: Difference example . . . . .	116
Figure 3.46. Relational Operation: Join T1 with T2, producing T3 . . . . .	117
Figure 3.47. Relational operation: Join example. . . . .	118
Figure 3.48. Relational operation: Divide. T1 divided by T2 produces T3 . . . . .	119
Figure 3.49. Relational operation: Divide example. . . . .	120
Figure 3.50. Relational operation Product T1 on T2 produces T3. . . . .	121
Figure 3.51. Relational operation. Product example. . . . .	122
Figure 3.52. Relational operation: Union. . . . .	123
Figure 3.53. Relational operation: Union example. . . . .	124
Figure 3.54. Combination operations. . . . .	125
Figure 3.55. Independent logical file data model. Five “files” with complex and simple record types. . . . .	128
Figure 3.56. Location of static and dynamic functionality. . . . .	128



Figure 3.57. Comparing static and dynamic data model type components .....	130
Figure 3.58. Data Model Components. ....	130
Figure 3.59a. The four data models .....	131
Figure 3.59b. The four data models: Legend. ....	132
Figure 3.60. DBMS data models. ....	133
Figure 3.61. Data model effects on application development. ....	134
Figure 3.62. Method of relationship implementation by data model. ....	135
Figure 3.63. Network data model. ....	139
Figure 3.64. Network row organization. ....	139
Figure 3.65. Indirect network relationships. ....	140
Figure 3.66. Network Operations. ....	141
Figure 3.67. Hierarchical column examples. ....	142
Figure 3.68. Hierarchical relationships. ....	143
Figure 3.69. Hierarchical operations. ....	144
Figure 3.70. Independent logical file column types. ....	145
Figure 3.71. Independent logical file. Many to Many relationship. ....	146
Figure 3.72. Independent logical file operations. ....	147
Figure 3.73. Relational data model terminology. ....	148
Figure 3.74. Relational table columns and rows. ....	149
Figure 3.75. Relational data model operations .....	149
Figure 3.76. ANSI/NDL data model data definition language .....	157
Figure 3.77. Hierarchical data model data definition language (IBM's IMS) .....	158
Figure 3.78. Hierarchical data model data definition language (System 2000) .....	159
Figure 3.79. Independent logical file data model DDL, IBI's Focus. ....	159
Figure 3.80. SQL Data definition language. ....	160
Figure 4.1. Physical database: static and dynamic relationship comparison. ....	164
Figure 4.2 Hierarchical Index Organization .....	167
Figure 4.3. Hash based primary key index organization. ....	168
Figure 4.4 Multiple Key Access (static or dynamic) .....	169
Figure 4.5. Project assignment .....	170
Figure 4.6. Indirect index organization for a student database. ....	171
Figure 4.7. Popular list of row processing DBMSs. ....	173
Figure 4.8. Logical database design. ....	176
Figure 4.9. Secondary index organized as a primary key index. ....	178
Figure 4.10. Primary and secondary index structures performance comparisons for single column select statements. ....	181
Figure 4.11. Hash-based secondary index organization. ....	182
Figure 4.12. Multiple occurrence component of an index structure for customer numbers. ...	185
Figure 4.13 Multiple Occurrence DBMS Record Reads .....	190
Figure 4.14. Summary of index strategy times. ....	193
Figure 4.15. Embedded static relationships. ....	195
Figure 4.16. Traditional next, prior, and owner chains. ....	196
Figure 4.17 Pointer Array: Next, Prior, and Owner Pointers .....	197





Figure 4.18 Separated Relationship Pointers .....	199
Figure 4.19 Employee Data Structure Data Record Instance Counts and Population Ratios ..	200
Figure 4.20. Value based relationships. ....	202
Figure 4.21. Data definition language for Personnel schema. ....	203
Figure 4.22. Network (static) program illustration. ....	205
Figure 4.23. ILF or relational (dynamic program illustration). ....	206
Figure 4.24. Summary comparison of the relative performance of relationship mechanisms. .....	211
Figure 4.25. Data file storage formats by relationship type, data model, and DBMS for various popular IBM mainframe DBMSs. ....	212
Figure 4.26 Data Record Instance Storage Formats by Relationship Type, Data Model, and DBMS for Various Popular IBM Mainframe DBMSs .....	213
Figure 4.27 Data Element Storage Formats by Relationship Type, Data Model, and DBMS for Various Popular IBM Mainframe DBMSs .....	213
Figure 4.28a. One table per file organization. ....	215
Figure 4.28b. One table per file organization. ....	216
Figure 4.29a One File for all Tables. ....	218
Figure 4.29b Example of One File For All Tables .....	218
Figure 4.30a. Multiple files per table. ....	220
Figure 4.30b. Example of multiple tables per file. ....	220
Figure 4.31a. Multiple files per table. ....	221
Figure 4.31b Example of Multiple Files Per Table .....	221
Figure 4.32a. Rows and key formatted files. ....	223
Figure 4.32b Example of Data Record Instances and Key Formatted Files .....	224
Figure 4.33a. Fixed format and fixed length rows (simple). ....	225
Figure 4.33b Example of Fixed Format and Fixed Length Records .....	225
Figure 4.34a Fixed Format and Variable Length Rows (Complex) .....	227
Figure 4.34b. Example of fixed format and fixed length rows. ....	227
Figure 4.35b. Example of variable format row. ....	229
Figure 4.35a Variable Format Row .....	229
Figure 4.36a. Simple column formats. ....	230
Figure 4.36b Example of Simple Column Formats .....	230
Figure 4.37a Complex Column Storage Formats: BNF .....	231
Figure 4.37b. Complex column storage formats: DDL .....	233
Figure 4.37c. Examples of complex column storage formats. ....	234
Figure 4.38 DBMS Based Pointers, Embedded Static .....	241
Figure 4.39 Relationships Case Study Record Type Diagram .....	243
Figure 4.40a. Tables for static relationship DBMS loading strategy. ....	249
Figure 4.40b. Manual strategy for static relationship DBMS loading. ....	250
Figure 4.40c Automatic Strategy for Static Relationship DBMS Loading .....	250
Figure 4.41. Static and dynamic effects on row adds and deletes. ....	252
Figure 4.42. Column change effects. ....	254
Figure 4.43 Static and Dynamic Effects of Relationship Change .....	256
Figure 5.1. Comparative development efforts by language type. ....	264



Figure 5.2. Report formatting. ....	266
Figure 5.3. Program view of orders. ....	271
Figure 5.5. Abbreviated ANSI NDL orders database schema. ....	272
Figure 5.6 Order List NDL View Specification ....	273
Figure 5.7 NDL Pseudo Code to Obtain Complete Order ....	274
Figure 5.8 Abbreviated ANSI/NDL Order Database Schema ....	275
Figure 5.9 SQL-like View Definition ....	276
Figure 5.10 SQL Pseudo Code to Obtain Complete Order ....	276
Figure 5.11 NDL View Specification ORDER ....	277
Figure 5.12 SQL View Specification ORDER ....	278
Figure 5.13 Pseudo Code for View Based Program ....	278
Figure 5.14. Business Transaction: Add Customer consisting of many run-unit transactions. .....	280
Figure 5.15. Data interface area. COBOL program, data division. View section. ....	290
Figure 5.16. Working storage section for DBMS interface messages. ....	291
Figure 5.17a Specialized CALLS interface. ....	293
Figure 5.17b Generalized CALLS Interface. ....	293
Figure 5.17c. Precompiler interface. ....	294
Figure 5.18 Basic language-like example of schema table access ....	300
Figure 5.19. Cursor complexity ....	301
Figure 5.20 IDMS/R Network Cursor Variables ....	302
Figure 5.21. System 2000 hierarchy cursor model. ....	303
Figure 5.22. ILF/Relational cursor model. ....	304
Figure 5.23 Cursor Types by Data Model ....	305
Figure 5.24 Static versus Dynamic HLI Differences ....	308
Figure 5.25 Static POL Example ....	313
Figure 5.26 Dynamic POL Example ....	314
Figure 5.27 Example of User-supplied Values from FOCUS ....	318
Figure 5.28 FOCUS Graphics Command Language for a CAR Pie Chart ....	321
Figure 5.29 CAR Database Master-File Description (DDL) ....	322
Figure 5.30. Car database logical data model diagram. ....	323
Figure 5.31. Pie chart graphic for subset of data from the cars database. ....	324
Figure 5.32 POL Run-unit Application Message ....	327
Figure 5.33 SYSTEM 2000 Example of Report Writer Program ....	330
Figure 5.34 SYSTEM 2000 Example of Query-Update Language Program ....	334
Figure 5.35 Interrogation Static and Dynamic Relationship DBMS Comparison ....	338
Figure 6.1 Static versus Dynamic System Control Comparison ....	341
Figure 6.2 Transaction Content ....	343
Figure 6.3 Static versus Dynamic Audit Trail Differences ....	344
Figure 6.4 Typical Message Types: Type = 1: DBMS Syntax Errors ....	346
Figure 6.5 Typical Message Types (cont.): Type = 2 Application Generated ....	346
Figure 6.6 Typical Message Types (cont.) : Type = 3 Application Generated ....	347
Figure 6.7 Typical Message Types (cont.): Type = 4: Database Integrity ....	347
Figure 6.8 Typical Message Types (cont.): Type = 5: DBMS Integrity ....	348



---

Figure 6.9 Severity Levels: Severity = 1: Information Only .....	348
Figure 6.10 Severity Levels (cont.): Severity = 2: Correctable .....	349
Figure 6.11 Severity Levels (cont.): Severity = 3: Run Unit Fatal .....	349
Figure 6.12 Severity Levels (cont.): Severity = 4: Database Fatal .....	350
Figure 6.13 Severity Levels (cont.): Severity = 5: DBMS Fatal .....	350
Figure 6.14 Cross Reference Between Message Types and Severity Levels .....	351
Figure 6.15 Messages Database Table Format .....	351
Figure 6.16 Audit Trail Columns For a Help Transaction .....	352
Figure 6.17. Recovery methods vs causes of failures. ....	355
Figure 6.18. Single User Recovery: Rerun the job. ....	356
Figure 6.19. Single-user recovery: Roll-forward (after image recovery) .....	358
Figure 6.20. Single user recovery: Rollback or before image form. ....	359
Figure 6.21. Multi-user recovery. ....	361
Figure 6.22 Transaction Rollback Alternatives .....	362
Figure 6.23 Transaction Rollback Alternatives (cont.) .....	362
Figure 6.24. Business Transaction: Add Customer consisting of many run-unit transactions. .....	363
Figure 6.25 Processor States Responsibilities in On-Line Transactions .....	365
Figure 6.26 Recovery: Static versus Dynamic DBMS. ....	368
Figure 6.27 Static versus Dynamic DBMS Logical Reorganization Comparisons .....	374
Figure 6.28 Sophistication Levels for Concurrent Operations Configurations .....	378
Figure 6.29 Typical Platforms for Concurrent Operations Configurations .....	379
Figure 6.30 Complex Storage Structure Effects on Concurrent Operation .....	382
Figure 6.31. System 2000 Security Illustration .....	401
Figure 6.32. Sales and marketing database design–initial .....	410
Figure 6.33. Sales and Marketing database design–second design .....	411
Figure 6.34. Sales and marketing database design–third design .....	412
Figure 6.35. Sales and marketing database design–Final Design .....	413





---

*Copyright 2006, Whitemarsh Information Systems Corporation  
Proprietary Data, All Rights Reserved*

## Preface

*The next DBMS generation is here. It contains DBMSs that look alike--on the outside. That is, they conform to ANSI/SQL, or ANSI/NDL, or to a combination of ANSI/NDL and ANSI/SQL. This does not mean all DBMSs are the same--on the inside. On the inside, DBMSs perform very differently: some slow, others fast; some rudimentary, others advanced. To buy a DBMS solely because it is ANSI standard is to also believe that all automobiles are the same. This book is about the critical DBMS differences.*

Database management systems (DBMSs) have evolved a great deal over the last 30 years. The first systems were created in the early 1960s. Through the years there have been several major technology mergers. Almost all of these mergers were the direct result of market pressures. The first evolution resulted from the merger of the capabilities of the host language DBMSs and the self-contained language DBMSs. The resulting DBMSs were almost all based on embedded, inter-table relationship mechanisms, known in this book as static relationships. Then there arose the dynamic relationship mechanism DBMSs (shared column values in different table instances). The first such systems were servicing hundreds of production databases in the late 1960s, years before the first papers *discovering* relational DBMSs were published. Throughout the late 1960s and 1970s, static and dynamic relationship DBMSs were installed and used with great success.

The second merger began around 1984. Some DBMS vendors discovered that the majority of the facilities contained in their respective products were independent of data model and independent of technology. Thus, they made their products support both network (static) and relational (dynamic) facilities and also be implemented on a variety of hardware types, that is, microcomputers, minicomputers, and mainframes. Consequently, this book addresses both network and relational DBMSs that operate on mainframes, minicomputers, and microcomputers.

The author first encountered a DBMS in 1969. It was an inverted access, hierarchical data model, natural language DBMS that became operational in the middle 1960s. This DBMS's maximum database size was about 1 million characters. It supported between 50 and 100 concurrent users and the computer required to support it filled a room. Today's room-sized computers support databases 10,000 times larger and service thousands of concurrent users. Today's desk top computers support databases 10 to 25 times larger than the 1969 database and service from 20 to 30 concurrent users.

If this book were written in 1969, its contents would be valid only for mainframe computers. Due to today's miniaturization and the increased sophistication of *personal* software, almost all of today's mainframe DBMS characteristics are found in today's micro- and minicomputer DBMSs. Thus this book is also valid for DBMSs operating on mini- and microcomputers.

In many ways, DBMSs are the same today as they were years ago. The same data models still exist, as do the types of natural and host language interfaces. Today, however, it is even more necessary to have storage efficient databases that can load, update, and retrieve data rapidly because database sizes have grown from millions to billions to trillions of characters. And it is even more necessary to provide security, audit trails, and multiple user access.



Whether you are selecting a micro-, mini-, or mainframe DBMS, to be sophisticated means that the DBMS supports screen generators, query languages, host language access, indexes, concurrent access, and multiple data model support, etc. These are all needed regardless of the hardware technology or the application.

Not everything has stayed the same, however. Since 1969, there has been an increase in the use of on-line facilities (screen generators, etc.) and of DBMSs for a wider range of applications. There is also a much better understanding of the fundamental aspects of a data model. Most important has been the realization that data model implementations are for the most part independent of their specification. This means that DBMSs of the same data model can have very different implementations, some substandard, most only acceptable, and a few of high quality.

Furthermore, the years since 1969 have seen an effort to specify standards for common DBMS facilities. The first of these efforts were by CODASYL. Starting in the early 1970s, ANSI/SPARC began an effort to standardize DBMS facilities by developing a framework from which sprang three standards: NDL, SQL, and the IRDS. NDL is for network databases, SQL for relational databases, and the IRDS for storing and manipulating metadata.

The ANSI standards process did not fix features, rather it standardized interfaces. An organization is now free to select various products that produce syntax conforming to an interface, or that can read conforming syntax. The real value of ANSI database standards is that the days of being locked in by a vendor are now over. For example, a company can purchase a sophisticated menu-driven report writer that produces interactive NDL or SQL data manipulation language. This NDL or SQL is read by the DBMS, which obtains the rows and passes them across the standard interface for final formatting by the report writer.

The next DBMS generation has been *born* and contains DBMSs that look alike, on the outside. That is, there are classes of DBMSs that conform to SQL, NDL, or to a combination of facilities from NDL and SQL. But these DBMSs are not all the same, any more than all paintings done with oils are the same. Clearly, it is possible to design a DBMS that fits the standards but is of poor quality nonetheless. Just as there is more to a quality painting than choosing paints, there is more to a quality DBMS design than following standards.

This book goes to considerable length to detail the components of a sophisticated DBMS and to provide the information DBMS users or evaluators need to look critically at DBMS products.

I would like to acknowledge the help of those who made this book possible. First there is my family, all of whom showed great understanding during the time it took to finish the book. Thanks go especially to my son, Michael, who spent hundreds of hours reading, editing, and pointing out the many areas of the book that needed to be reworked. I am grateful to Ruth Jennings for her many hours work on the over 200 figures contained in the book.

Thanks are owed to the members of the ANSI/H2 committee, whose corporate members, at this book's first publishing, numbered over 40. Now, in 2006 is it 12. The committee met between 1978 through 2002 for six meetings every year. Each meeting was an average of three days at which 75 to 100 papers were presented. Starting in 2003, the meeting schedule has been changed to four electronic meetings of a half-day each and then two "face to face" multi-day meetings. Ideas are presented and then subjected to intense review by implementors and users. In this way, standards progress one syntax and general rule at a time. Initially, standards' progress is



quick, but as months and years pass, the task becomes more and more complex, and the work slows. To acknowledge the H2 members who performed the work would take pages. Consequently, they shall remain named only in the actual NDL and SQL standards.







# 1

## ANSI DATABASE STANDARDS

A database management system (DBMS) is a very popular class of software (sometimes even hardware) rivaling word processors and spread sheets. DBMSs were first developed for large mainframes in the early 1960s. Shortly after the emergence of minicomputers and microcomputers, DBMSs began to appear on them too. Today there are probably over 200 different DBMS developers. DBMSs have been used on almost every imaginable type of database application--for example, medical, industrial, government, financial, personnel, educational, and engineering.

A database is not a technology. Rather, it is an expression of organization, clarity, and precision. It may or may not be computerized. If it is, it may exist on a microcomputer, a minicomputer, or a large mainframe. Finally, a database may or may not be centralized. However a database is implemented and operated, success is impossible without the codification of and adherence to data semantics, which are the rules for meaning, validity, and usage.

When a database is computerized, it represents the automation of the knowledge component of a business, which is manifest through the business's quality operation, planning, and management. With a successful database, the managers of a business can research the past, organize the present, and plan for the future.

When a class of software becomes as widely accepted as have DBMSs, standards must be established so there can be portability. While portability is important for moving database applications from one computer and DBMS to the next, the most important class of portability is staff. Today's staff must be mobile as there is such a diversity of hardware and DBMSs. When standards exist, database design, application implementation, and maintenance strategies can be taught at the University and be presumed knowledge when an employee is hired. Companies then have only to concentrate on application-specific training.

### 1.1 In the Beginning

In the middle 1960s, the Systems Committee of CODASYL (Committee On Data Systems and Languages) undertook two surveys of existing DBMSs. The CODASYL organization then created several committees to develop specifications for a network DBMS. The American National Standards Institute (ANSI) reviewed the CODASYL work and in the early 1970s began the database standards process by developing a reference model. The process to standardize syntax and semantics of DBMS began with the establishment of the committee H2 in 1978, which developed the NDL and SQL standards: NDL is for network databases, SQL for relational databases. Another ANSI committee, H4, developed the Information Resource Dictionary System (IRDS) standard for storing the metadata so critical for effective data processing (including database) environments.

This chapter presents an overview of reference models and of the three ANSI standards.



## 1.2 Reference Models

A reference model is a mechanism or framework for describing a database management system in terms of its interfaces, processing functions, and the flow of data through the DBMS. An early reference model was partially defined and presented through an ad hoc study committee of the ANSI Standards Planning and Requirements Committee (SPARC). This incomplete model was published in 1975 and became known as the ANSI/SPARC reference model. The complete reference model was never finished, and the ad hoc study committee was disbanded in 1975. While this 1975 partial model was a necessary first step for ANSI's creation of DBMS standards, it is now only of historical interest, as its architecture no longer relates to a modern, sophisticated DBMS, and it has been completely replaced by the IEC/ISO/TC1 reference model.

The most significant contribution of the ANSI/SPARC architecture was the recognition of the need for an organization to have a conceptual schema, which is an institutionalized information resource dictionary system (IRDS), to serve as a repository for information requirements of the business. This IRDS can be used to show where, how, and by whom information is used throughout an enterprise.

### 1.2.1 The 1975 ANSI Reference Model

ANSI is divided into a number of committees. One, X3, the Computers and Information Processing Committee, commissioned a study by its Standards Planning and Requirements (sub)Committee (SPARC) in the Autumn of 1972 to determine whether technical committees should be established to create database (really DBMS) standards. From 1972 to early 1975, the ad hoc committee set up by ANSI/X3/SPARC met to resolve this question. The membership of this committee included Honeywell, IBM, Eastman Kodak, Equitable Life, Boeing Computing Services, Sperry Univac, Deere and Company, NCR, University of Maryland, Exxon Corporation, Columbia University, and RAND. The following is the ANSI/SPARC committee's own summary description of its work:

There are three realms of interest in the philosophy of information. These realms are: real world, ideas about the real world existing in the minds of men, and symbols on paper or other (storage) medium representing these ideas.

In addition, there are several realms of interest in data processing, the manipulation of the symbols representing these ideas. Three of these realms have special significance ... These realms are: external, including a simplified model of the real world as seen by one or more applications; conceptual, including the limited model of the real world maintained for all applications; and internal, including the data in computer storage representing the limited model of the real world. (1)

While the committee identified three realms, they chose to use the more technical term *schema*. In addition to using the word *schema* to mean *realm*, the committee used the word *interface* to mean *schema*. That change of terminology is not so bad, in that an interface can also be called a



schema since it is a framework of reference, or an outline that is output by one processor in order to be input to the next processor.

As for the identified schemas themselves, the ANSI/SPARC architecture study actually identified 42. Some of the schema definitions were merely titles, while others were whole sections of text. Since the committee recognized that an explanation of 42 separate interfaces would certainly have turned the ANSI/SPARC data architecture report into volumes, three main classes of interfaces, cited above and analogous to the realms, were presented: external, conceptual, and internal. The committee's original names for these three interfaces were subschema, super-schema, and schema. As described in an edited version of the draft ANSI/SPARC report, the three schema are:

- An external schema that contains the specifications of the external objects, the associations and structures in which they are related, operations permitted upon these objects, and administrative matters.
- A conceptual schema that includes the specification of the conceptual objects, and their properties and relationships, operations permitted on these objects, consistency, integrity, security, recovery, and administrative matters.
- An internal schema that includes the specifications of the internal objects, indices, pointers, and other implementation mechanisms, other parameters affecting (optimizing) the economics of internal data storage, integrity, security, recovery, and administrative matters. (2)

The ANSI/SPARC committee saw the conceptual schema as a super-sized interface between all the different external schemas and the one or more internal schemas. Because of advances both in technology and in understanding, however, that role of the conceptual schema is seen as being too narrow.

The purpose of the ANSI/SPARC study was to identify areas for ANSI to standardize, rather than to define the standards. This is evident from the name SPARC--Standards Planning and Requirements Committee--which implies that the purpose of the committee's work was to plan for standards activities. The introductory section of the draft ANSI/SPARC report supports this assertion by stating that:

Among the responsibilities of the Standards Planning and Requirements Committee (SPARC) of the American National Standards Committee on Computers and Information Processing (ANSI/X3) is the generation of recommendations for action by the parent Committee on appropriate areas for the initiation of standard development efforts. (3)

X3 allowed ANSI/SPARC to convene another ad hoc database study group in 1982. One of the many goals accomplished by this ad hoc study committee was to review the 1975 draft reference model and issue a report providing corrections, revisions, and expansions. Among the many items addressed by the committee were problems left untouched by the 1975 draft report:



- Is metadata different from data?
- Are metadata and data stored separately?
- Are metadata and data described in terms of different data models?
- Is there a schema for metadata--a meta schema?
- Are there also external and internal meta schema?
- Are the interfaces used to retrieve and change metadata different from those used to retrieve and change data?
- Can a schema be changed on-line?
- How would on-line schema changes affect the data? (4)

During the process of answering these questions, many new ones arose. For example, it was determined that the 1975 draft reference model was too simplistic, as it did not address:

- Distributed schemas that may result from distributed database processing
- Open systems architecture
- Establishment of a unified concurrency model (teleprocessing and database interactions)
- Operating system (O/S) based security and privacy

The results of the 1982 ANSI/SPARC Reference Model subgroup have been incorporated in the work effort of the reference model subgroup of the IEC/ISO/JTC1/SC21/WG3 committee (Joint Technical Committee 1 (JTC 1) from both ISO (International Standards Organization) and IEC (International Electrical Committee), Standing Committee 21 (SC 21, Information Processing Systems), Working Group 3 (WG3, data management).

If the three ANSI database standards produced in the 1986-1988 time frame were forcibly mapped to the ANSI/SPARC three-schema architecture that was produced in 1975, then H4 (IRDS) would be standardizing some parts of the conceptual schema, while H2 (SQL and NDL) would be standardizing other aspects of the conceptual schema, as well as the DBMS side of the internal schema and the entire external schema. No ANSI committee is working on a computer side of the internal schema, as computer technology changes faster than it can be standardized.



### 1.2.2 The ISO Reference Model

The ANSI/SPARC reference model was created before there were any DBMS standards. Consequently, the developers of that 1975 reference model could only hypothesize what the interfaces should be and how they should be interconnected. Since that early work, three database standards, NDL, SQL, and IRDS, have been developed. The ISO reference model thus reflects what was learned by those standards committees.

The purpose of the IEC/ISO/JTC1 reference model is to provide a common basis for the coordination of standards development.

The IEC/ISO/JTC1 reference model is a framework for existing (NDL, SQL, IRDS, and RDA) and future data management standards. The reference model does not specify services or protocols for data management, nor does it specify implementation strategies, nor does it serve as a basis for appraising the conformance of implementations. The reference model, however, does define the common terminology and concepts pertinent to all data held within database applications. These concepts are then used by the individual database management and dictionary management committees to define their respective standards.

The IEC/ISO/JTC1 reference model is only concerned with the abstracted specification of the types of interfaces, across which the various DBMS components operate. The reference model specifies neither the actual interface, that is, its syntax and semantics, nor its components that are on either side of the interface. Both the specification of the interface and the components on either side of the interface are the responsibility of the ISO (ANSI) committees specifying the DBMS service.

As stated in the IEC/ISO/JTC1 reference model document, the objective of the reference model is to provide a framework for the following:

- The identification of interfaces
- The positioning of all such interfaces relative to each other
- The identification of facilities provided at each interface
- The identification of the process and, where appropriate, specific data which supports each interface
- The positioning of the use of the interfaces in terms of an information systems life cycle
- The identification of the binding alternatives associated with each appropriate identified interface(5)

In IEC/ISO/JTC1, the reference model specifies the types and kinds of services that are to be available from a DBMS. The six services identified by the IEC/ISO/JTC1 reference model are



- **Dictionary Definition:** the specification of the types of data that an organization requires in its dictionary system. This provides the schema for the dictionary database, referred to as the dictionary schema.
- **Dictionary Use:** the storage and retrieval of dictionary data relating to all aspects of other information systems with special dictionary capabilities, such as keeping many versions of data. The dictionary database includes the data definitions for application databases.
- **Application Schema Definition:** the specification of the schema in the form appropriate to the DBMS. Depending on support for the DBMS provided by an IRDS, the dictionary system may provide the schema based on its data descriptions.
- **Application Database Use:** the storage and retrieval of data from an arbitrary database structure.
- **Database Creation:** the establishment of a database.
- **Database Maintenance:** the changes to a database required as a result of changes to its schema. (6)

Through the definition of these six services, the IEC/ISO/JTC1 reference model defines the framework for coordinating the development of existing and future standards. The existing standards included in this data management reference model are: NDL, SQL, IRDS, and RDAP (remote data access protocol).

### 1.2.2.1 Levels of Abstraction

The IEC/ISO/JTC1 reference model is presented first at an abstract level, identifying only the essential features of the system being modeled and ignoring everything else not directly relevant. Thus, the IEC/ISO/JTC1 reference model at the highest level of abstraction is the same for both the IRDS and the DBMS. At the DBMS level, the NDL and the SQL are the same. At the NDL level, all NDL compliant DBMSs are the same. The reference model thus contains only the specification of the highest level model as it is the responsibility of the IRDS and the DBMS (NDL and SQL) committees to define their respective reference models. (7)

The reference model document states that a DBMS and/or IRDS would be conforming in any of the alternatives for either of the two interface types: user and processing. A user interface is the interface between the end-user and the facility providing the service. A processing interface is the interface between two software and/or hardware components. The user interface alternatives are:

- Panels (abstract screen formats)



- Language syntax, concrete or abstract

The alternatives for processing language interfaces are:

- Procedure calls
- Syntax (execution time interpretation)
- Service conventions
- Protocols which define permitted interactions

While all seven forms satisfies the IEC/ISO/JTC1 reference model, each lower level standard (SQL, for example) has its own conformance section specifying a more detailed conformance requirement.

### **1.2.2.2 Levels of Data**

Data exists in two forms: values and representations of values. TELEPHONE NUMBER is the semantic representation of the value 212-555-1234. In this regard, the following pairs of terms stand in this relationship: table and row, column and value, metadata and data. This kind of representation does not take place on only one level. For example, a table includes the definitions of column types. The IEC/ISO/JTC1 reference model describes four common levels of representation:

- Fundamental
- Dictionary definition
- Dictionary
- Application

Figure 1.1 illustrates the four levels, the databases associated with each level, and the schemas required to represent the contents. In the complete database environment, these four levels must all exist.



Data Levels And Level Pairs			
	Application Level Pair	Dictionary Level Pair	Dictionary Definition Pair
<b>Fundamental Level</b>			Dictionary Definition Schema
<b>Dictionary Definition Level</b>		Dictionary Schema	Dictionary Definition Database
<b>Dictionary Level</b>	Application Schema	Dictionary Database	
<b>Application Level</b>	Application Database		

**Figure 1.1** Data Levels and Level Pairs

At the fundamental level, the IRDS itself is defined, resulting in the IRDS' own schema and the associated data, editing, and validation rules, etc. necessary to establish an IRDS.

The dictionary definition level is employed to define the data stored in the data dictionary and the constructions that represent data stored in the dictionary. Data instances at this level are values such as Row, Column, relationship, and the like. Second, after the IRDS is established, individual databases can come into existence, each with its own schemas, data, editing and validation rules that are to serve all data represented in the database. It is at this level that individual data models exist.

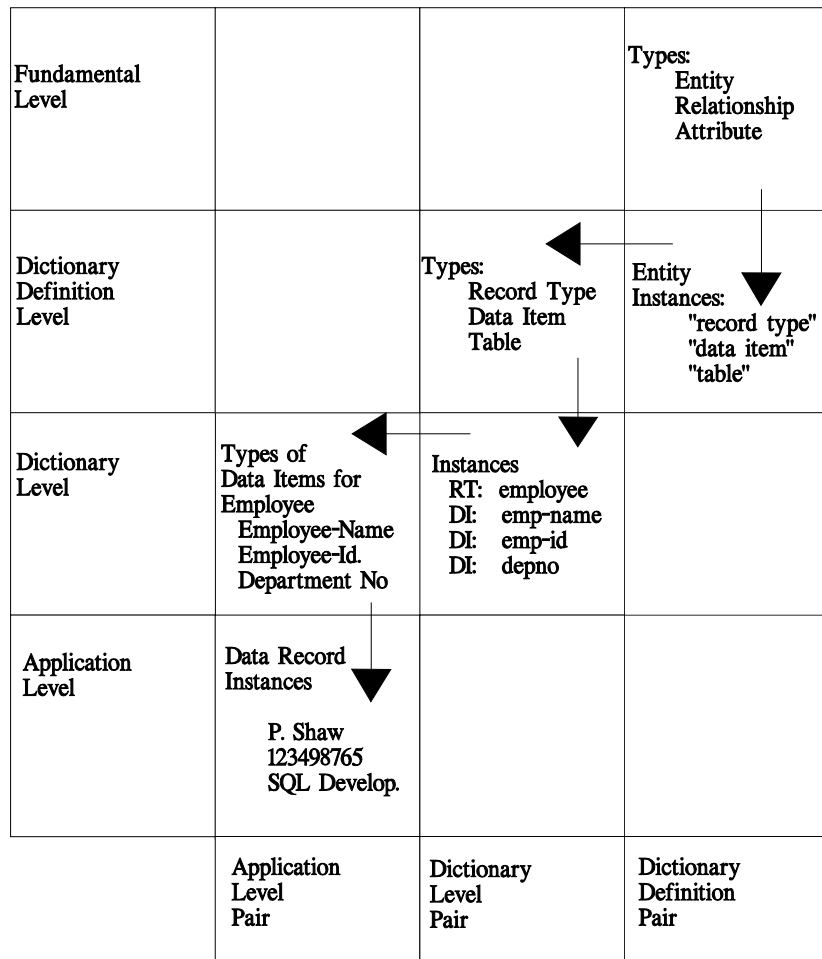
The dictionary level is employed to receive a request and to determine whether the column names are correctly referenced. To the dictionary, the data are the column names, SALARY or EMPLOYEE. Other data at the dictionary level can refer to the name of the program (for validation), schemas or views, referential integrity rules, table look-ups, proper subschemas, terminal identifiers, and the like.

Once the individual database is brought into existence, the fourth level, *application*, can exist. Of course, there may be many applications operating against a database that serve different purposes and are programmed in different languages. Each application program contains its own schematic of data (subschemas or views) and has its own set of data, editing, and validation rules that it controls whenever services are requested of the application. (8)

Figure 1.2 illustrates the interaction of these four levels.





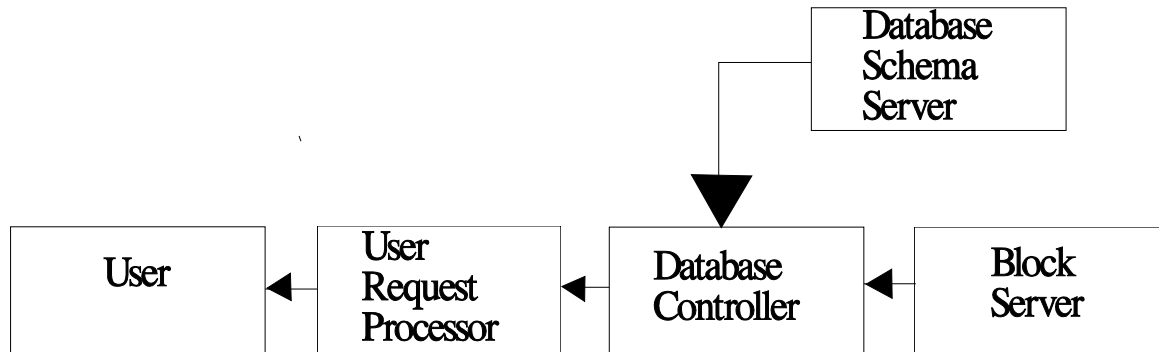


NOTE: RT means record type  
DI means data item

**Figure 1.2.** Data levels and level pairs.



### 1.2.2.3 Generalized Reference Model



**Figure 1.3** Simplified Diagram of the Abstract Reference Model

The IEC/ISO/JTC1 committee defined a generalized version of the reference model that is independent of any of the particulars necessary to view the individually bound components of the models. Figure 1.3 illustrates this model. It consists of the following five components:

- User
- User request processor
- Database controller
- Block server
- Schema server

User identifies the person requesting services from the user request processor. The request is accepted upon satisfactory identification.

The user request processor receives the user request and verifies that the service request is valid by checking the database schema server (through a subschema or view).

The database controller performs the largest component of work. It is tasked with establishing a session; noting the processing intent; performing database select, store, delete, and insert operations; bulk transferring data to and from the database; initiating and stopping transactions; performing recovery; performing database reorganization, etc.

The block servers stores and receives blocks of data (DBMS rows), maintain recovery and activity journals, and restore the database to a consistent state after a rollback or crash.

The schema server, whether it is database or blocks of data, stores and retrieves parts of a schema that are needed for user verification, data translation, etc.

These components of the generalized reference model interact to be a complete DBMS environment.



#### 1.2.2.4 Detailed Reference Models

The generalized reference model is intended to be a general representation of any detailed reference model. In this way, any of the following DBMS use scenarios can be modeled:

- Application access
- Proprietary DBMS vendor language access
- Application program use of an IRDS access
- IRDS definition access
- Multiple data model support access

In each of these cases, the user interacts with the named component (e.g., application program), which in turn accesses the IRDS schema and database if applicable and, in turn, accesses the block server to perform the data service request. (9)

#### 1.2.2.5 ISO Reference Model Summary

Included in the many products generated by the IEC/ISO/JTC1 reference model committee were answers to the questions left over from the ANSI/SPARC 1975 three schema effort. That is, the IEC/ISO/JTC1 reference model incorporates work from the IRDS, NDL, and SQL. Thus, it clearly resolves the metadata questions.

The IEC/ISO/JTC1 reference model not only supports the 1975 recommendation that there should be support for multiple data models, but also illustrates multiple data model usage.

Finally, the ISO model also indicates that a database environment should support multiple databases within a single session, as well as data from distributed processing.

### 1.3 ANSI Database Committees: H2 and H4

As previously stated, the 1975 ANSI/SPARC data architecture ad hoc study group did not intend their three schemas (conceptual, external, and internal) to be definitive, but illustrative, as these three schemas were merely the committee's shorthand way of discussing the three different subgroupings of the 42 different interfaces (schemas) that it had identified. As support for that statement, consider that each ANSI standard consists of five main components: conformance statements, field of applicability, (syntax) format, general rules, and syntax rules. The ANSI/SPARC study group produced, at best, the field of applicability section. If the remaining sections had been completed, then the report would have covered many thousands of pages.

As final evidence that the 1975 ANSI/SPARC committee was never created to produce standards, SPARC dissolved its ad hoc reference model subcommittee in 1975 and chose to create the ANSI committees, H2 and H4, with the expressed purpose of creating database standards.



## 1.4 The Battle over Data Models

The ANSI/SPARC committee in 1972 addressed the battle concerning data models with the following statement:

There is continuing argument on the appropriate data model: e.g., relational, network, hierarchical. If, indeed, this debate is as it seems, then it follows that the correct answer to this question of which data model to use is necessarily *all of the above*. (10)

The 1975 ANSI/SPARC committee saw, as confirmed by the IEC/ISO/JTC1 reference model committee in 1988, that a data model is merely a formalized method of defining tables, columns, the relationships among the tables, and the operations that are allowable on those columns, tables, and relationships. Today, there are four popular data models: network, hierarchical, independent logical file, and relational. H2, the ANSI database languages committee, has created NDL to manipulate network data model databases and SQL to manipulate relational data model databases. These four data models are presented in Chapter 3.

In the two years following the 1972-75 ANSI/SPARC committee's work, ANSI/X3, SPARC's parent committee, decided to establish a database standards committee, named H2. Its first meeting was in June 1978, and its initial charter was to develop an NDL standard. In 1981, H2's charter was expanded to include the development of an SQL standard. In addition to standardizing the data definition language (DDL) and the data manipulation language (DML) for NDL and SQL, H2 has standardized the use of these database facilities from the ANSI standard languages: COBOL, FORTRAN, PASCAL, PL/I, Ada, and C.

In addition to the DBMS standards work of H2, X3 established the H4 committee to create a standard for Information Resource Dictionary System (IRDS).

In sum, the ANSI database standards process began with SPARC, which identified the need for database standards through the definition of a draft reference model (1975). They recommended the establishment of a database standards committee (H2) to draft the standards NDL and SQL, and recommended the establishment of H4 to develop the IRDS. To ensure that these standards work together, H2 maintains liaisons with the IRDS committee, H4, and the various language development committees, such as PL/I (X3J1), BASIC (X3J2), Fortran (J3), COBOL (J4), Pascal (J9), and C (X3J11). Since there is currently no ANSI committee for Ada, liaison is conducted with the U.S. Department of Defense's Ada Joint Program Office in Washington, D.C. The liaison with these committees consists of shared organizational memberships, exchange of documents, occasional joint meetings, and joint votes on critical issues.

## 1.5 DBMS Standards

During 1986, ANSI ratified two database standards, one for the network data model, called NDL (network database language), and another for SQL (formerly meaning structured query language). ANSI/H2, the authoring committee of these two standards, is composed of individual experts from DBMS vendors, government agencies, corporations, and consulting organizations.



NDL was derived from the CODASYL data model, which had been implemented on computers from almost all major hardware vendors and by a few software vendors. The CODASYL 1978 specification was used as NDL's initial base document, and over a four-year period, the specification was streamlined until all physical aspects were removed. For example, the concept of AREA was removed because it is a technique for partitioning rows and has been implemented by DBMS vendors in very different ways. Over the four years the actual elapsed meeting time was less than five work months.

NDL has been substantively implemented by Digital (DEC). Their DBMS is quite logical, easy to implement, and cost effective to operate.

The other standard, SQL, founded on IBM's SQL, has been implemented on almost every brand of hardware and all three hardware tiers: micro-computers, mini-computers, and mainframes. H2 was initially chartered to develop a relational database standard by X3 in 1982. H2 determined that it is more expedient to develop a standard based on an existing product, rather than create a DBMS standard from scratch. The H2 member from IBM, Phil Shaw, created an SQL technical specification document at the request of the committee.

The two ANSI database standards are based on different data models: network and relational. The relationships in NDL are not required to be value-based. NDL relationships are defined in the data definition language, and rows which belong to a specific relationship instance can have their order (with respect to the relationship instance) specified. NDL can explicitly declare almost all the types of relationships. Referential integrity in NDL is defined through a set of clauses that govern row insertion and retention.

SQL relationships must be value-based, with the definition of the relationship occurring in the application program's view of data or in programming logic. Row orderings are requested by the application program and are performed by the DBMS prior to their presentation to the program. SQL can declare only one relationship type, but users can accomplish most of the others through application program logic. Finally, SQL has referential integrity to control insertion and retention.

These two standards are not conflicting but complementary. Any DBMS vendor could implement NDL, SQL, or both, either as separate DBMSs or as a combined set within one DBMS. During any database application implementation, there should be a DBMS selection step. If an application is an add-on to existing data already installed under a DBMS, then the choice is already made. If however, the application is for a completely different type of processing, for example, decision support rather than transaction processing, the DBMS selection and evaluation should focus first on the selection of data model, and then on the selection of accessories, and finally on performance.

A DBMS really consists of two classes of products: the database control system (DBCS), and accessories. The DBCS is the product that stores, selects, and manages the DBMS rows. An accessory tool may be a vendor's natural language that interfaces with the DBCS, a third generation language interfaced through a language embedding, or an application package that is implemented either through natural languages or language embeddings. ANSI committees have standardized the syntax and semantics of the third generation languages, that is, COBOL, FORTRAN, PL/I, Ada, C, and Pascal.

The ANSI process standardized the external specifications of two types of DBCSs: NDL and SQL. Each standard specifies the syntax and semantics of the language that communicates



the database's structure to the DBCS. Each standard also specifies the semantics of the operations that an accessory tool must use whenever it communicates with a standard DBCS to store, retrieve, and update data. Finally, each standard specifies the syntax appropriate for embedded language interface.

An organization is now free to select the DBCS for each application based on the DBCS' ability to select, store, and manage rows for the company's transaction processing (possibly NDL) or decision support needs (possibly SQL) without fear that all accessory tools will have to be relearned.

Independent of the decision on the DBCS, an organization can select, on the basis of individual requirements or styles of applications, accessory tools that interact with the DBCS through standard interfaces. Accessory products that have standard interfaces can operate against any DBCS that can *read* the interface language. If the interface is not generated, then the accessory product will not be as portable as an ANSI standard language run-unit (e.g., COBOL, FORTRAN, etc.). Once these products are selected and installed, data can be collected and stored by the accessory tools suited for that purpose, while other tools can be used for selecting and reporting data in tabular forms and graphics.

An application package is an accessory tool. It represents a *canned* database structure, various load and update programs, and an array of reports. These packages are affected by ANSI standards in that vendors can now concentrate on developing product features rather than nonstandard DBMS interfaces. This benefits application package users as well, since they have a broader set of standard products from which to select.

Both ANSI standards were created to be independent of the method of physical implementation. Traditionally, the physical implementation of NDL data has been through O/S files so that each file stores rows from multiple tables. Traditionally, the physical implementation of SQL data has been through O/S files so that each file stores rows (rows) from only one table (table).

Now that there are ANSI standard specifications for the interfaces, vendors are free to experiment with different physical implementation strategies to achieve different performances. While there have been traditional methods of physically implementing data models, a data model neither requires nor specifies a particular method of implementation. For example, the ANSI/NDL (network) data model has traditionally been implemented through embedded relative row addresses with the rows from one or more tables stored on one O/S file. Also traditionally, the SQL data model has been implemented by storing each relation in its own O/S file. Today, however, there are ANSI/NDL-like DBMSs with value based relationships and each table in its own physical file. And there are ANSI/SQL (relational) DBMSs that allow rows from different tables to be stored together to improve performance.

Today, sophisticated DBMS vendors recognize the value of both ANSI/NDL and ANSI/SQL, and allow database system designers to freely mix these types of relationships in a single database design. Furthermore, these DBMS vendors recognize that physical database design is, for the most part, independent of data model. Thus, physical database design can be tailored to meet the needs of flexibility or of high volume and performance. This tailoring can be directed to a database as a whole or to just one set of tables within a database.

In analyzing database applications, an organization may find that it needs different DBCSs adhering to the same data model, but with different physical implementations, as well as



DBCSs supporting different data models. Since all the DBCSs adhering to ANSI standards would be interacting through standard interfaces, different DBCSs can be selected, installed, and changed without having to reprogram existing ANSI standard language applications. If, however, an application's data model must be changed, say from NDL to SQL, then application programming logic has to be changed as well unless there is a sophisticated view facility in place (see Chapter 5).

Another very important benefit of these standard interfaces is that applications can be developed on a mainframe and ported to minicomputers or to microcomputers, providing that the applications on those computers use ANSI standard languages.

## 1.6 ANSI/NDL

NDL, the network database language, contains the syntax and semantics for defining tables, columns, and five different types of relationships. The relationships can define explicit network structures. NDL also defines the basic operations on those structures. Finally, NDL provides functional capabilities for designing, accessing, maintaining, controlling, and protecting the database. A high level DBMS reference model that illustrates the NDL interface languages is contained in Figure 1.4

Each NDL table can contain columns that are either single-valued or multi-dimensioned. A single-valued column are for data like SOCIAL SECURITY NUMBER. Multi-dimensioned columns is for data like MONTHLY SALES (one dimension), or MONTHLY SALES BY YEAR (two dimensions).

NDL also allows for the definition of five classes of relationships:

- Owner single member
- Owner multiple member
- Singular single member
- Singular multiple member
- Recursive

The owner single member relationship is the most common, and defines the relationship between a DEPARTMENT and its EMPLOYEES.

The owner multiple member relationship is less common, but quite useful. For example, for TERRITORY, the owner table is DISTRICT OFFICE, and the member tables are SALESPERSON and CUSTOMER.

A singular relationship is a relationship in which there is no defined owner table. If there is a need to maintain a list of top salespersons, then the singular set TOP-SALESPERSONS can be defined, involving only the table SALESPERSON.







One kind of NDL referential integrity rule declares that a row can be stored only if an owner row is already present. Another NDL referential integrity rule prohibits owner row deletions whenever there are member rows. A third NDL referential integrity rule automatically deletes all member rows whenever the owner row is deleted. Because these integrity clauses are defined in the data definition language, they are enforced on all languages that interact with NDL database.

NDL also specifies a module procedure language that can be employed by developers of NDL systems. This language specifies all the rules and regulations that must be followed if an NDL database is accessed through COBOL, FORTRAN, or any other type of language, such as report writers and query-update languages.

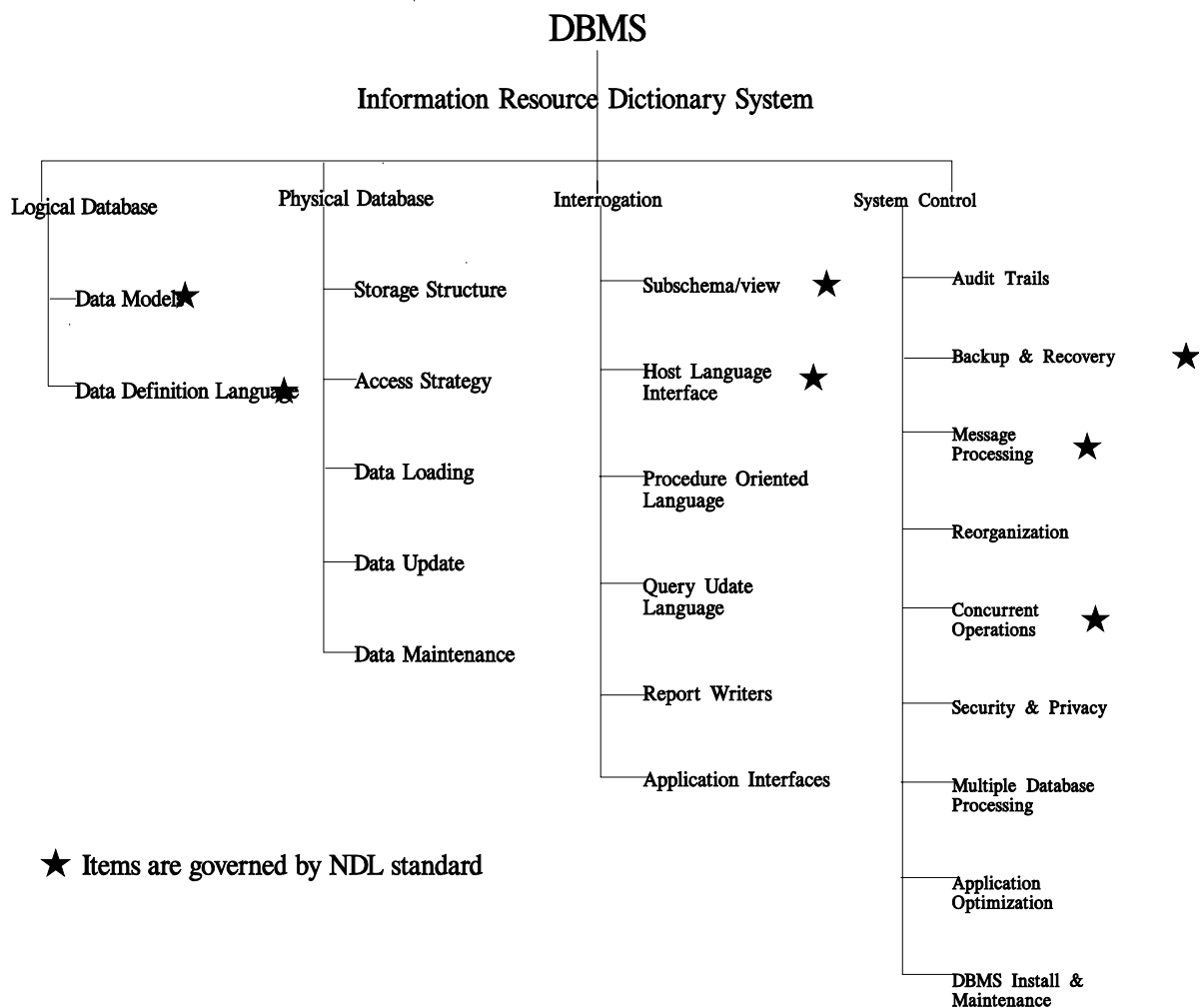
NDL is a complete specification of the DBCS component of the DBMS. It specifies neither the types of language that can interface, nor any mode of DBCS implementation. The components of DBMS that are standardized by NDL are identified in Figure 1.5.

To help understand the evolution of NDL, a review of its development history is appropriate. The Database Task Group (DBTG) of the CODASYL programming language committee delivered its 1970 report that served as the basis for several DBMS implementations (e.g., Honeywell's IDS and Cullinet's IDMS). The DBTG committee then split into two groups. The Data Description Language Committee (DDLC) took over the task of specifying the syntax and semantics of the data definition language portion of the CODASYL database specification. The Database Language Task Group (DBLTG) was a subordinate group within CODASYL COBOL. The DBLTG's function was to define the syntax and semantics for the subschema language and the data manipulation language (DML).

During the period 1971 to 1978, several DDLC documents, called journals of development (JOD) were issued. A JOD is more a working document than a strict implementable specification. For example, the CODASYL concept of AREA has been implemented somewhat differently in every CODASYL DBMS. In one CODASYL DBMS, an AREA is a collection of O/S files. In another, multiple AREAs are contained in a single O/S file, and in at least one other, an AREA means one O/S file.



Famous among the JODs are the 1973 and the 1978 issues. During this time, several more DBMS vendors implemented DBMSs based on these JODs, for example, NCR, Data General, Digital, and PRIME.



**Figure 1.5.** Database management system components addressed by ANSI/NDL

In the 1976-1978 time frame, ANSI/SPARC determined that the CODASYL approach should be standardized since there had been so many successful implementations. The ANSI committee H2 was formed to undertake the task. What was to be the relationship between the CODASYL DDL committee and the ANSI/H2 committee? In the development and standardization of the language COBOL, there exists a relationship between CODASYL and ANSI: the CODASYL COBOL committee develops and the ANSI J4 committee standardizes. That means that no COBOL language development is performed within the ANSI J4 committee. For example, if a facility of COBOL is determined deficient by the ANSI J4 committee, the only option for ANSI is to drop the facility until the CODASYL COBOL committee fixes the problem. That fix can take one or more years.



H2 found that approach unacceptable for database, and from its first official committee meeting (the meeting after the charter formation meeting), decided to do its own development. To implement this decision, H2 voted to *consider any proposal submitted by its own members*. The vote was 17 to 2 in favor! The relationship with CODASYL was also addressed in a motion that stated . . . *CODASYL Data Description Language Committee (DDL) is the development authority for the DDL and the CODASYL Journal of Development (JOD) is the only source . . .* The vote was 17 to 2 against!

H2 began its database standardization efforts in July 1978 by adopting the CODASYL DDL 1978 JOD as its base document. During the first six months of H2's work, almost half of this document was deleted, either because the facilities were poorly specified or because they were not portable from one machine environment to another. These deletions were for more than academic reasons. H2 felt that if the deleted facilities were implemented as formed, bad function would result. The respective CODASYL vendors saw this too, as each implemented these under-specified functions in different way. Since there was no standards committee to keep the various CODASYL implementors together, different versions of CODASYL systems were created from the same *standard*.

During the next several years, H2 added, deleted, and modified DDL facilities. Since H2 had reserved development authority to itself, the CODASYL DDL committee found itself without much to do and was disbanded in 1982.

The CODASYL COBOL committee's corresponding ANSI committee, J4, had a subcommittee, J4.1. This committee was charged with standardizing the syntax and semantics of the subschema and the data manipulation language. Their work ceased by 1980 for two reasons. First, J4 members were not greatly interested in database, especially since the committee's mission was COBOL. Second, the CODASYL DDL committee, which passed its JOD to CODASYL COBOL for their use in defining the syntax and semantics of the subschema and DML, ceased to exist. Thus, CODASYL COBOL no longer had a DDL source for standardizing development.

Beginning in 1980, H2 petitioned ANSI to broaden its charter to not only define the syntax and semantics for the data definition language, but also define the syntax and semantics for the subschema language and the semantics of the DML operations. That petition was approved. H2 then modified its base document to include the new areas and, during the summer of 1983, the document was approved.

After a base document is approved by the committee, it becomes a dpANS (draft proposed American National Standard) and is subject to a four month public review. If the public review is successful the document is voted by X3, and if the vote is successful, it is voted again by ANSI. If the ANSI vote is successful the dpANS becomes an American National Standard. Generally, the committee's work takes about three years, and the X3 and ANSI review and their votes take another two years.

The final steps of standardizing the NDL were delayed about six months so H2 could finish work on SQL. That enabled both NDL and SQL to be released together. In August 1986, both NDL and SQL were accepted as ANSI standards.

Once NDL was stabilized, J4 created an ad hoc committee to create a DML specification for COBOL that was issued as an J4 information bulletin.

To summarize NDL's development history:



- CODASYL produced JODs in 1971, 1973, 1978, and 1980. The 1980 JOD reflected ANSI's NDL deletions.
- All but one of the CODASYL DBMSs were implemented during the 1970s. Only DEC's NDL is based on the 1980 NDL dplans.
- Most CODASYL DBMSs differed from each other in areas critically important to portability. NDL does not contain those nonportable features
- The portable components of the *CODASYL standard* really are the ANSI/NDL standard. Thus, the NDL is the **ONLY** network DBCS specification that supports portability.

Figures 1.6a through Figure 1.6d presents a comparison of three popular DBMSs--CA's IDMS/DB, Unisys' DMS/1100, and the DEC's VAX-DBMS as a way of illustrating that these three DBMSs compare favorably to NDL. Figure 1.6a presents a comparison of ANSI/NDL schema language clauses for schema, table, and column. Figure 1.6b compares the set (relationship) clause. Figure 1.6c compares the subschema (view) clauses. Figure 1.6d compares the data manipulation language clauses. It must be noted, however, that the comparison was made only to the extent of general functional and command purpose. No attempt was made to determine exact syntax, or semantic compatibility

## 1.7 ANSI/SQL

In 1981, X3 assigned H2 another project. It was to create a standard language for relationally structured databases. Phil Shaw from IBM created a Backus Naur Form (BNF) specification for IBM's SQL product and submitted it to the committee as a proposed base document for the relational language. Appendix A illustrates BNF. The SQL specification was accepted as the base document, and the language was called RDL (relational data language). The DBMS reference model that applies to the resultant standard, ANSI/SQL, is illustrated in Figure 1.7. The components of DBMS that are standardized by ANSI/SQL are identified in Figure 1.8

The relational standards effort concentrated on IBM's SQL work. Since many vendors had already implemented their own version of IBM's effort, H2 developed an SQL baseline standard, so that all vendors could work together to develop further evolutions together.

The original RDL specification was modified throughout 1982 and 1983. It included, for example, a schema manipulation language and referential integrity. In 1984, the members of H2 determined that RDL had moved too far from existing SQL implementations, that is, too far from IBM, Data General, Digital, Oracle, UNISYS, and others. The committee decided to remove these incompatibilities and name the standard SQL rather than RDL. By 1985, SQL was accepted by H2.



ANSI/NDL Facility	ADDRESSED BY		
	IDMS/R	VAX-DBMS	DMS-1100
Schema Declaration	YES	YES	YES
Record Definition	YES	YES	YES
Record Clause	YES	YES	YES
Location Mode Clause	YES	NO	YES
Record Check Clause	NO	YES	YES
Element Definition	YES	YES	YES
Element Types	YES	YES	YES
Defaults	YES	YES	NO
Complex Elements	YES	YES	YES
Occurs	YES	YES	YES

**Figure 1.6a** ANSI/NDL versus IDMS/R, VAX-DBMS, & DMS1100 Facilities Schema Comparison



ANSI/NDL Facility	ADDRESSED BY		
	DMS/R	VAX-DBMS	DMS-1100
Set Definitions	YES	YES	YES
Set Name Clause	YES	YES	YES
Set Order Clause	YES	NO	YES
Owner Clause	YES	YES	YES
Owner Is System Clause	YES	YES	NO
Member Clause	YES	YES	YES
Retention Clause	YES	YES	YES
Insertion Clause	YES	YES	YES
Order Clause	YES	YES	YES
Member Check Clause	NO	YES	NO
Member Uniqueness Clause	NO	NO	NO

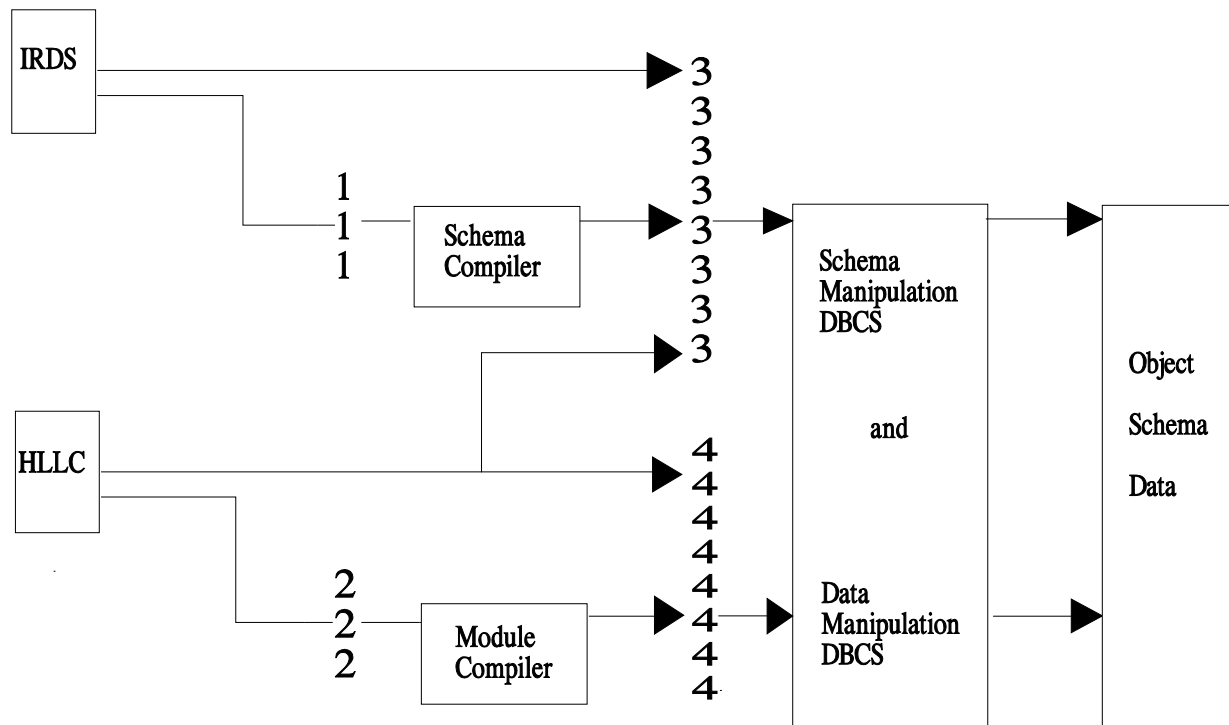
**Figure 1.6b** ANSI/NDL versus IDMS/R, VAX-DBMS, & DMS 1100 Facilities Set Definition Comparison



ANSI/NDL Facility	ADDRESSED BY		
	IDMS/R	VAX-DBMS	DMS-1100
<b>Subschema Definition</b>	YES	YES	YES
<b>Subschema Clause</b>	YES	YES	YES
<b>Schema Reference</b>	YES	YES	YES
<b>Record Definition</b>	YES	YES	YES
<b>Record Name Clause</b>	YES	YES	YES
<b>Element Definition</b>	YES	YES	YES
<b>Schema Set</b>	YES	YES	YES
<b>Set Names</b>	YES	YES	YES

**Figure 1.6c** ANSI/NDL versus IDMS/R, VAX-DBMS, & DMS110 Facilities Subschema Comparison





**Legend:**

- 1 = Schema Definition Language
- 2 = Procedural Language
- 3 = Schema Manipulation Language
- 4 = Data Manipulation Language

IRDS = Information Resource Dictionary System

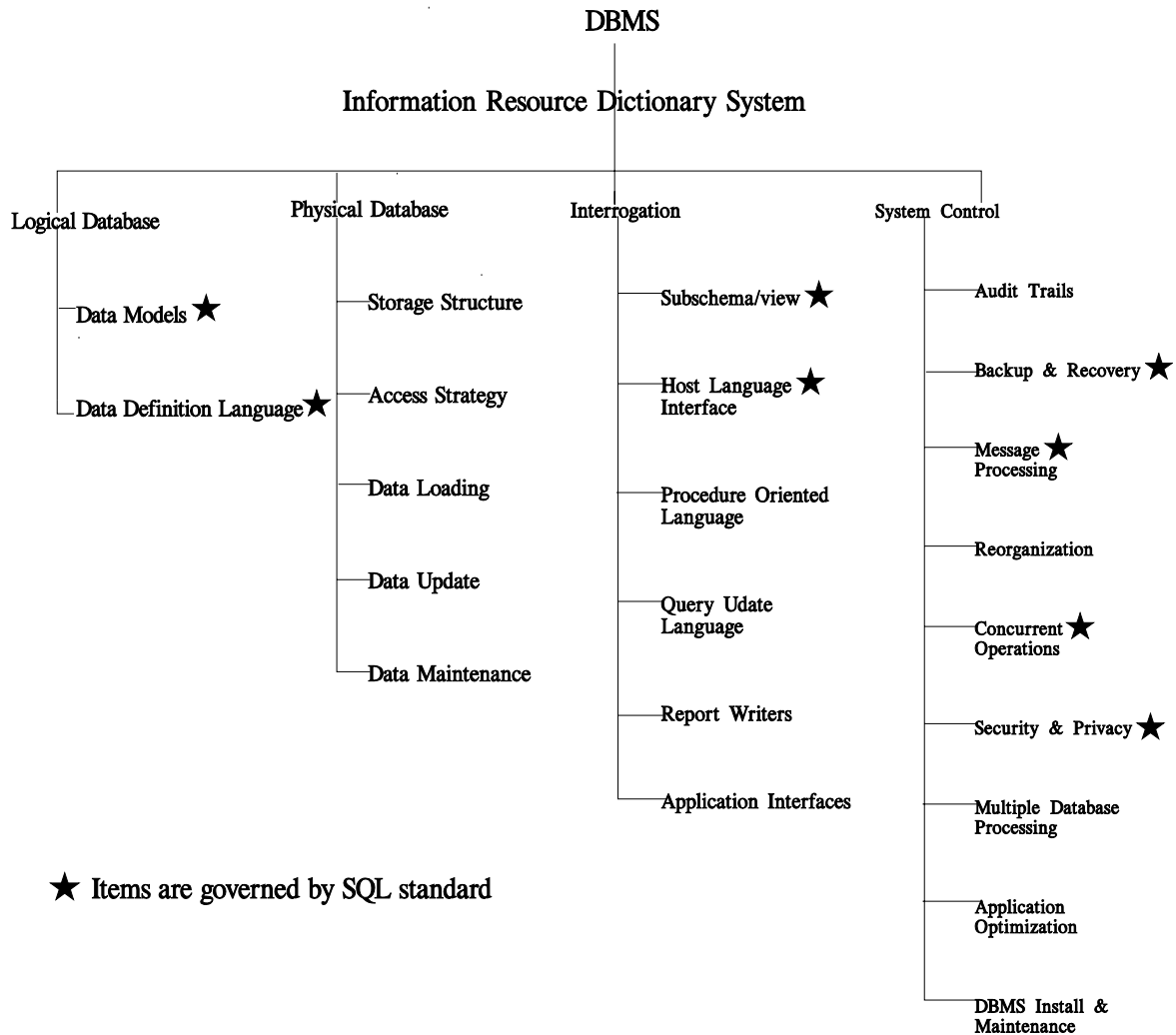
HLLC = High Level Language Compiler, e.g., Cobol, Query, POL

**Figure 1.7 SQL Reference Model (1988)**





Once the baseline SQL was completed, H2 immediately began to develop an extended SQL, calling it SQL2. Knowing that the development of SQL2 would take years, H2 also began



**Figure 1.8** Database Management System Components Addressed by ANSI/SQL

development of two interim SQL extensions: Addendum-1, and Embedded SQL. Addendum-1 contains a basic set of referential integrity clauses and actions. Embedded SQL contains the specifications for binding SQL to the programming languages COBOL, FORTRAN, PL/I, PASCAL, C, and Ada. Both Addendum-1 and Embedded SQL became ANSI standards in 1989. SQL 1986 with referential integrity along with minor corrections and clarifications is now known as SQL 1989.

The SQL2 project was technically complete during the first half of 1991, and became an ANSI standard by late 1992, taking on the name, SQL 1992. SQL2 contains both relaxations and *repairs* to the specifications contained in SQL 1986, Addendum-1, and Embedded SQL. In addition, there are extensions in the following areas:



- Datetimes and intervals
- Derived tables in FROM clause
- Dynamic SQL
- Intersection/Difference
- Multiple module support
- NULLIF and IFNULL/COALESCE
- Outer join
- PRIMARY KEY implies NOT NULL
- Query expressions in views
- Referential actions such as cascade delete
- Relaxed UNION compatibility rules
- Renaming columns in the select list
- Row expressions
- Schema manipulation language
- Schema information tables
- Self-referencing deletes
- Self-referencing updates
- SQLSTATE
- Subquery in value expressions
- Substrings and concatenation
- Variable length character strings

In general, SQL 1989 permits the definition of data as two-dimensional tables and defines the basic operations on these tables. The single valued columns are the columns, and the rows are the rows. Furthermore, SQL 1989 provides functional capabilities for designing, accessing, maintaining, controlling, and protecting the database.

SQL 1989 contains clauses for the definition of schemas and tables belonging to those schemas. Each table contains column (column) definitions. Each column may contain the definition of default values and table constraints for uniqueness, referential integrity, and various check clauses.

The data manipulation language for SQL 1989 includes the traditional relational operations such as select, project, and join.

## 1.8 ANSI/SQL (1999)

Starting in 1992, the H2 committee began the development of dramatic extensions to the SQL 1992 standard. The greatest change in the standard is that it no longer adheres to the 1970 relational data model. The second biggest change is that the SQL 1999 language now consists of individual parts that comprise a foundation and then a series of independently specified packages. The remainder of this section provides an overview, in outline form, of the contents of the SQL 1999 standard.



### 1.8.1 Foundation Components

- Tables that have been enhanced to support new built-in data types (boolean, enumerated, extensions to character sets, translations, and collations)
- BLOB and CLOB data types
- Abstract Data Types (user defined data type with behavior, an encapsulated internal structure, and access characteristics of public, protected, or private)
  - ◆ strong typing
  - ◆ subtypes and inheritance
  - ◆ encapsulation
  - ◆ virtual attributes
  - ◆ substitutability
  - ◆ polymorphic routines
  - ◆ dynamic binding
  - ◆ compile time type checking
  - ◆ value ADTs
- Array
- Row Types (table person (SSN, name(first, middle, last), address(street, city, state, zip(four, five))))
- User Defined Functions
- Predicate extensions (for all, for some, similar to, cursor extensions, null values, assertions, view updatability, joins)
- Triggers
  - ◆ Different triggering events, update, delete, and insert
  - ◆ Optional condition
  - ◆ Activation time: before and after
  - ◆ Multiple statement action
  - ◆ Several triggers per table
  - ◆ User-defined ordering
  - ◆ Condition and multiple statement action per each row or per statement
- Roles (enhancements to security), & Savepoints
- Recursion



### 1.8.2 Call Level Interface

The SQL Call Level Interface is the set of language specifications used by DBMS vendors to enable direct SQL engine access through completely specified call routines. Microsoft, for example has implemented SQL/CLI and calls it ODBC.

The CLI specification contains more than 50 different call specifications that address:

- Connection control to SQL servers
- Allocate and de-allocate resource
- Execute SQL statements
- Obtain diagnostic information
- Control transaction termination
- Obtain information about implementation

It also contains Resource Management Handle routines for:

- Environment
- Connection
- Statement
- Context

The CLI also contains a Descriptor Area that accommodates:

- Application parameter
- Application row
- Implementation parameter
- Implementation row

### 1.8.3 SQL/Multi Media (MM) Components

SQL/MM is itself a set of subparts that contain full specifications for a discrete set of data management functionality to address the data processing needs of:

- Full Text
- Spatial
- General Purpose
- Still Image



### 1.8.4 SQL Persistent Stored Module Language Components

SQL 1999 now has a complete embedded programming language to support the processing needs of its user defined data types, assertions, and triggers. The SQL/PSM language supports the following capabilities:

- Call
- Return
- Compound Statements (Begin ... End)
- If Statements
- Case Statements
- Loop
- Repeat
- While
- For
- Leave
- Assignment
- Signal and Resignal

### 1.8.5 SQL Transaction and Connection Management

Since the advent of distributed processing, client-server, and of course the Internet, the ability to manage transactions is critical. SQL 1999 now has facilities that address the following areas of transaction and connection management:

- Start transaction
- Set transaction
- Test completion
- Savepoint
- Release savepoint
- Commit
- Rollback
- Connect
- Set connection
- Disconnect statement



## 1.9 The ANSI/IRDS

The information resource dictionary system (IRDS) is designed to be an organization's repository of metadata. Uses of the IRDS include the following:

- A documentation tool
- A software life cycle and project management tool
- A data element standardization and management system
- An organizational planning tool
- A tool to support database administration, document administration, information resource management and data administration
- A tool for supporting a distributed processing and database environment
- A source and object library management system
- A configuration management facility
- The storage location for NDL & SQL schema and subschemas (views)

In IRDS, a data element is one level of abstraction higher than column. To handle such a broad set of requirements, the IRDS is not actually all those facilities. Rather, it is a core set of facilities with the ability to define and expand into the areas for which explicit definitions have not been provided.

The architecture of the IRDS contains four levels. The top level is implementation-defined. It is the actual software provided by an IRDS vendor to define, store, and operate on the data in the information resource dictionary system.

The second level defines the objects that comprise an IRDS schema. This level also holds definitions of the various control mechanisms, including naming rules, defaults, and validation information for the information resource dictionary contents.

The third level is for the dictionary. This level describes the environment being modeled. It describes the objects in the environment and the associations among the objects.

The fourth level, not described in the IRDS, is actual instances of the objects in the dictionary.

The minimum set of objects in the IRDS is specified in the IRDS part 2, Basic Functional Schema. It contains the following eight objects:

- data element
- record (a generic IRDS term for either table or row, as context determines)
- document



- file
- module
- program
- system
- user

Also included in the IRDS are four types of relationships to interconnect the IRD objects:

- One to many
- One to one
- Many to many
- Recursion

The following are illustrative combinations of objects and allowed relationships:

- Document contains document
- Document contains data element
- Document contains record
- Document derived from document
- Document derived from file
- Document derived from row

The IRDS also contains access control specifications for extending and modifying the objects contained in an IRD. The IRDS contains a complete language for manipulating the IRD contents. Typical verbs are DO, IF, CASE, and RETURN.

The ANSI IRDS does not specify any method of implementation. Thus, it may be an extension to a DBMS's data dictionary (NDL or SQL), or a completely stand alone product. In the former case it is active and in the later case it is passive.

By the end of the 1990s, the ANSI IRDS was effectively terminated. Between the standardization and IRDS-1 and that of IRDS-2, a philosophical battle raged between those in the United States who created IRDS-1 and those in the rest of ISO (most notably the British Computer Society) who created IRDS-2. The key difference was that in IRDS-2, the IRDS meta model was to be explicitly cast into SQL tables. Because this philosophical difference was too great the whole IRDS effort effectively dissolved. As of 2005 there is no IRDS standard. Thus, no standard meta model nor standard processes for metadata repositories and/or CASE tools. This turn of events has had as much negative effect on IT as SQL has had a positive effect.



### 1.10 Database Standards Summary

The ANSI database standards process started in the mid-1970s and continues today. Its milestones and achievements include:

- 1975 Creation of the Draft Reference Model (three schema)
- 1978 Establishment of H2 (NDL charter)
- 1979 Establishment of H4 (IRDS)
- 1981 Extension of H2's Charter extended for SQL
- 1983 Replacement of the ANSI/SPARC 1975 Reference Model
- 1986 Approval of the ANSI/NDL database standard
- 1986 Approval of the ANSI/SQL database standard (i.e., SQL 1986)
- 1986 Development of the ISO Reference model (replaces 1983 model)
- 1988 Approval of the ANSI/IRDS standard
- 1989 Approval of ANSI/SQL Addendum-1 (i.e., SQL 1989)
- 1988 Development of the IEC/ISO/TC1 Reference model (replaces 1986 model)
- 1989 Approval of ANSI/SQL (embedded SQL language support)
- 1992 Completion of ANSI/SQL 1992
- 1999 Completion of ANSI/SQL 1999
- 2003 Completion of ANSI/SQL 2003
- 2007 Completion of ANSI/SQL 2007 (estimated)

ANSI/H2 Event	Year									
	78	80	82	84	86	88	90	92	94	96
Start of H2	X									
NDL-DDL Development	X	X	X	X						
NDL-DML Development		X	X	X						
NDL Processing				X	X					
RDL Development			X	X						
Change RDL To SQL				X						
SQL Development				X	X					
SQL Processing				X	X					
SQL Referential Integrity Revision (SQL 1989)				X	X	X				





ANSI/H2 Event	Year									
	78	80	82	84	86	88	90	92	94	96
SQL Embedded Language Standard				X	X	X				
SQL2 Development						X	X			
SQL2 Processing (SQL 1992)								X		
	78	80	82	84	86	88	90	92	94	96

**Figure 1.9a** ANSI/H2 Standards Activity Past, Present, and Predictions

ANSI/H2 Event	Year									
	96	98	00	02	04	06	08	10	12	14
SQL1999 Development	X	X	X							
SQL1999 Processing				X						
SQL/MM Development	X	X	X	X	X					
SQL/XML Development				X	X					
SQL/XML Processing					X					
SQL/4 Development (SQL 2007?)				X	X	X				
SQL/4 Processing (SQL 2007?)						X	X			
	96	98	00	02	04	06	08	10	12	14

**Figure 1.9b** ANSI/H2 Standards Activity Past, Present, and Predictions

The specific milestones of H2 are portrayed in Figure 1.9. The dates for the start of the processing of SQL2 and the development and processing of SQL3 are estimated.

This book does not address IRDS other than to indicate its place in the set of database standards. Rather, this book concentrates on DBMS, and the effects ANSI/NDL and ANSI/SQL have on DBMS products.

The benefits derived from SQL and NDL standards are principally in two areas. First, database design efforts can proceed in the direction of either SQL or NDL confident that resulting products will not be locked into a particular vendor's products. Further, in the case of



SQL or NDL, database management systems that *read* SQL or NDL data definition language can compile database structures directly, rather than just conceptually.

Second, products that produce either or both of these two languages as their output can be procured safely. For example, if a company offers a screen-oriented, menu-driven product for producing database structures, it can be procured knowing that the syntax and semantics of the language it produces are governed by an ANSI standard rather than by a vendor who might change his side of the interface to force out competition.

The next DBMS generation is already here and contains DBMSs that look alike, superficially. That is, there are DBMSs that conform to SQL, or NDL, or to a combination of NDL and SQL. But this does not mean that they are the same. There are differences in taste, and dramatic differences in quality. Choosing one DBMS over another is becoming more and more difficult, as the real differences lie beyond simple conformity or non-conformity to ANSI standard syntax. Consequently, the remaining chapters of this book deal, not with syntax conformity, but with real DBMS discriminators.

## References

1. FDT, The Bulletin of ACM--SIGMOD, The Special Interest Group on Management of Data, Volume 7, Number 2, 1975, ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, 75-02-08, page II-1.
2. The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems, Tsichritzis and Klug, University of Toronto, Toronto, Canada, AFIPS Press 210 Summit Ave, Montvale, New Jersey 07645, page 9.
3. FDT, The Bulletin of ACM--SIGMOD, The Special Interest Group on Management of Data, Volume 7, Number 2, 1975, ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, 75-02-08, page I-1.
4. ACM's SIGMOD RECORD, Vol 15, Number 1, March, 1985, page 27.
5. IEC/ISO/JTC1/SC21/WG3/N2641, project 97.21.30 Reference Model of Data Management, Version 9, March, 1988, page 2.
6. IEC/ISO/JTC1/SC21/WG3/N2641, project 97.21.30 Reference Model of Data Management, Version 9, March, 1988, page 13.
7. IEC/ISO/JTC1/SC21/WG3/N2641, project 97.21.30 Reference Model of Data Management, Version 9, March, 1988, page 16.



8. IEC/ISO/JTC1/SC21/WG3/N2641, project 97.21.30 Reference Model of Data Management, Version 9, March, 1988, pages 18-21.
9. IEC/ISO/JTC1/SC21/WG3/N2641, project 97.21.30 Reference Model of Data Management, Version 9, March, 1988, pages 39-45.
10. FDT, The Bulletin of ACM--SIGMOD, The Special Interest Group on Management of Data, Volume 7, Number 2, 1975, ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, 75-02-08, page I-1.





## DBMS APPLICATIONS AND COMPONENTS

A DBMS is a large collection of computer processes that can be entirely software or firmware that operate in a generalized or specialized computer that provides languages or mechanisms to:

- Define the logical database component of a specific database either directly or through reference to the IRDS metadata, that is, the tables, columns, and relationships
- Define the physical component of a specific database, that is, the storage structure definition, and the techniques for access strategies, data loading and updates, and database backup.
- Specify interrogations that access data through one or more types of languages such as host languages like COBOL or FORTRAN, and/or natural languages like query-update, report writers, or procedure-oriented.
- Specify system control facilities, that is, audit trails, backup and recovery, concurrent operations, and the like.

This chapter provides an overview of the affects and characteristics of static and dynamic relationship applications. In addition, it contains an explanation of these DBMS components and how they interrelate in today's DBMSs. This chapter also explains how database management *issues* arise that lead to confusion on the part of evaluators. Finally, this chapter compares database applications with DBMS to show that they are inextricably linked.

### 2.1 Application Classifications

Applications can be classified in many ways. For example, applications can be classified by their major use, such as accounting, engineering, or personnel. Or, they can be classified by the types of audiences they serve: operations, support and control, or high-level MIS. For DBMS purposes, the real difference in applications is whether their design is constant or is variable.

DBMS facilities too can be classified in many ways, for example, host language or self-contained, network or hierarchical, embedded pointers or indexed based relationships, and so forth. While the third of these classifications hints at a major difference, the real difference in DBMSs is how they handle the differences in applications. One type of DBMS serves applications that are constant and unchanging, and another type of DBMS serves applications that are variable and changing.



## 2.2 Static and Dynamic Relationships

Either static or dynamic relationships dominate a database application. Seldom is there an application in which the quantity of relationship types is evenly divided. Figure 2.1 shows the critical differences between static and dynamic relationship applications.

If the application is based on static relationships, then the relationships among tables are normally well-defined, stable, and seldom-changing. Static relationship applications also tend to be production-oriented, with large transaction volumes that require high velocities. The principal access languages for static relationship applications are languages like COBOL.

A dynamic relationship application, in contrast, tends to have relationships that change almost as often as the data. The orientation of the application is MIS (management information system). These applications often typically have less data, lower transaction volumes, and consequently require lower velocities than the static relationship applications. The principal

APPLICATION CHARACTERISTIC	RELATIONSHIPS	
	STATIC	DYNAMIC
Fundamental Orientation	Corporate	Project
Required DBA Orientation	Centralized	Decentralized
Database Design Effort	Significant	Casual
Frequency Of Design Changes	Low	High
Design Change Control	Strict	Lax
Design Change Effects	Profound	Trivial
Typical Data Volume	Large	Small
Application And Database Design Interdependence	High	Low
Predominant Interrogation Language	Cobol	Query-Like

**Figure 2.1** Static Dynamic Relationship Application Characteristics

access languages for dynamic relationship applications are natural languages like query-update, procedure-oriented, and report writer.

A static relationship DBMS's DDL contains formally defined relationship clauses with all their attendant integrity, row storage, and retrieval subclauses. Rows are often stored in the order loaded, and are maintained in DDL defined ordering; or they are stored and maintained in the order of the value from a row's primary key. Rows are selected and presented according to the order implied by the primary key value or as specified in the DDL.



A dynamic relationship DBMS's DDL usually does not contain formally defined relationship clauses. Rows are often stored in the order presented and maintained in any order, or they are stored and maintained in the order of the row's primary key. The user of a dynamic database has sort clauses as a fundamental part of every access language. Rows are selected and sorted before presentation. Once presented, the rows can be re-sorted and re-presented. In most DBMSs, the rows are not actually sorted, their address identifiers are.

The effect of having a dynamically related set of rows is the inverse of having a statically related set of rows. With dynamic relationships, each interrogation has an equal chance of retrieval and update efficiency. There is no bias to the processing. Often there are no restrictive relationship integrity clauses. The burden of maintaining relationship integrity is placed on the user because the dynamic database has undefined row interrelationships. Thus it follows that the integrity clauses governing the interrelationships may also be undefined.

This lack of relationship integrity has prompted the builders of some dynamic relationship DBMSs to borrow a concept, inherent in static relationship DBMSs, called referential integrity. This is a two-part concept. First, it is the specification of the rules that relate two rows from two different types, and second, it is the automatic enforcement of the rules by the DBMS. This simple concept is, however, fraught with pitfalls, and must be carefully implemented. Referential integrity is covered in Chapter 3.

Knowledge of a DBMS's data model is important, but not as important as knowing whether the DBMS creates static or dynamic relationships. If the DBMS's relationships are static and the application is dynamic, application failure is almost always the outcome. If the DBMS's relationships are dynamic and the application is static, then the application will run, but the production performance will normally fall far below achievable levels.

## 2.3 The Nature of a Static Relationship Application

How many times since the creation of data processing has a bill of materials (BOM) system's design been substantively changed? Few times if ever. The reason is simple: the process's design is static. For this kind of production application, the software is written once and runs for years. And because it runs for years, and with voluminous transactions, the software written to process BOMs is *very close to the machine*. This software must be well designed and expertly implemented. A very large effort is justified because the design changes are few, and because the through-put required is very high.

Static relationship database applications are similar to BOM, in that they are less prone to change than others, and require very high through-put over a static set of relationships. These applications thrive on a certain kind of DBMS. Database benchmarks principally based on static relationship processing have, over the years, shown this assertion to be true. These DBMSs are designed to handle static relationships.

Applications implemented with static relationships should exhibit distinct characteristics, as enumerated in the static relationship column of Figure 2.1.

The critical characteristics of such DBMSs are cited in the static relationship column of Figure 2.2. Examples of DBMSs that contain static relationships are: DMS-1100, IDMS/R, IMS, SUPRA, and SYSTEM 2000.



## 2.4 The Nature of a Dynamic Relationship Application

In contrast to static relationship applications, there is a class of applications for which design change seems to be a way of life. Changes arise from two sources. First, there are changes that exist because the application is young and not yet completely evolved. Second, there are changes that are due to the very nature of the application. Dynamic relationship applications tend to require many changes to the tables and the relationship types among rows in order to keep the applications relevant to the user's changing needs. These applications tend to contain smaller amounts of data than do static relationship applications, and tend to be better suited to a natural language environment rather than to a production COBOL environment. This is because changes are needed faster than COBOL programs can be created or changed.

Applications implemented with dynamic relationship DBMSs should exhibit distinct characteristics, as enumerated in the dynamic relationship column of Figure 2.1.

To handle such applications, there is a class of DBMSs that are designed around dynamic relationships. The characteristics of these systems are cited in the dynamic relationship column of Figure 2.2.

Examples of DBMSs that contain dynamic relationships are: ADABAS, Datacom/DB, DB/2, DMS-170, FOCUS, IDMS/R, INQUIRE, MODEL-204, NOMAD, and SUPRA.

## 2.5 Problems and Benefits of Static Relationship DBMS

Applications implemented with static relationships can have problems in two areas. The first occurs when it is necessary to produce a report that does not mirror the database's structural definitions. While almost any output requirement can certainly be satisfied, some are ill-suited to the database's design. Complex programs must then be created that consume large amounts of computer resources to satisfy the report requests. Eventually, if that reporting requirement is frequent enough, the database's design needs to be modified to reduce the processing resources consumed.

The second problem pertains to the database itself. Redesigning a static database usually involves great amounts of computer and human work concerning the:

- Actual redesign of the database's individual tables
- Formal relationships among the tables
- Programs that process data from the databases





DBMS Component		Relationship Type & Effect	
		Static	Dynamic
<b>LOGICAL</b>	Tables per data base	Many	Few
	Relationship Mechanism	Pointers	Data Values
<b>Physical</b>	Relationship Building	Load/Update	Retrieval
	Row Keys	Single	Multiple
	Data Loading	Via Structure	Row by Row
	Relationship Change	Delete and Re-Add	Data Value Change
	Fundamental Bias	Toward Database design or Production	Neutral or ad hoc MIS-like
<b>Interrogation</b>	Host Language	Excellent	Good
	Natural Languages	Average	Excellent
<b>System Control</b>	Multiple Table Audit Trails	Easy	Hard
	Reorganization Logical Physical	Hard Expensive	Easy Reasonable
	Multiple DB Processing	Acceptable	Excellent
	Multiple Row Locks	Easy	Hard

**Figure 2.2** Static-Dynamic Relationship DBMS Characteristics

Programs are often involved because they are usually written with the database's structure in mind. As a result, once the database design is changed, the programs that process data from it often require changes. If the programs use the DBMS's view facilities that are described in Chapter 4, the extent of program changes can be minimized.

In contrast to the problems, the benefits derived from applications implemented with static relationships are significant. Because relationships in static relationship DBMSs are most often implemented through traditional pointers, all tables are under the control of a single schema. The DBMS then always knows all the tables and is able to perform relationship



maintenance (logical database reorganization) from this centralized point of view. Another benefit is the opportunity to build well-engineered and controlled databases. This results naturally from the data loading and maintenance process, which is performed principally through COBOL programs.

These two benefits greatly increase database integrity. As a final benefit, reports that mirror the database structure are highly efficient.

## 2.6 Problems and Benefits of Dynamic Relationship DBMS

Applications implemented with a dynamic relationship DBMS have almost the inverse set of problems and benefits described for the static relationship DBMS.

Database integrity can be a significant problem with a dynamic relationship DBMS. This is because the tables are related through column values that are under user control. If a DBMS does not have referential integrity and the value in an owner column is accidentally changed, then the owner's corresponding member rows are lost with respect to that relationship. Members can be placed in a wrong relationship if the data value in the member is updated incorrectly.

DBMS benchmarks have shown that whenever there are production-oriented data processing-intensive interrogations, dynamic relationship DBMSs perform more slowly than static relationship DBMSs. This is because dynamic relationship DBMSs often have to access the owner row to get the data value used to select--via secondary index searches--the member rows. The static relationship DBMS, on the other hand, simply follows a predefined *pointer-based* road map to another row that may be on the same physical page.

Other difficulties associated with applications of dynamic relationship DBMSs center around the implementation of multiple table applications. It is the very independence of the tables that causes the problems.

Whenever there is a need for a multiple table update, elaborate schemes must be implemented to ensure that all affected rows are updated in unison. Compounding the lock problem for the multiple table update, all well-designed dynamic database applications must have coordinated backup and recovery, and audit trails.

While these problems can be overcome by good user code and sophisticated techniques, they can cause concern to the application's users if not dealt with effectively.

The benefits associated with applications implemented with dynamic relationship DBMSs are just as significant as the problems. The benefits are flexibility, speed in some settings, and distributed processing. Flexibility heads this list. Tables associated with a dynamic relationship database are often separate and independent physical files, and thus their loading and maintenance can be carried out without fear of disturbance by processes accessing other table instances.

Additionally, these tables can often be processed sequentially--at great speeds--since all rows in the table's file are of the same type and format.

Finally, the greatest benefit of the dynamic relationship DBMS is that the tables that might traditionally operate together under a static relationship schema can enjoy some of the benefits of a centralized schema without being preloaded into one static database. For example, it



is possible to have multiple applications operating apart throughout the year, and then to bring them together at year's end under the control of a *virtual* schema for corporate-wide reporting.

## 2.7 Implementing the Static Relationship Application

The fundamental goals of a static relationship database can be only to:

- Create a data organization that mirrors the fundamental processes in the business organization
- Impose maximum DBA control over user updating and reporting of data in the database

There cannot be different goals, because the static relationship DBMSs will not permit them to be implemented.

### 2.7.1 The Logical Database

The database design process should take considerable time, as the designer should become involved in an in-depth analysis to determine the *natural* organizations of data.

Looking to existing data processing systems may only throw the designer off track as these systems are typically implemented to produce ad hoc reports. As a result, these systems had ad hoc data collections and ad hoc programs to transform the collected data into the report. When another report was needed, another system was created. Finally, all the month-end systems were combined into one to reduce the multitude of file extracts, sorts, and prints. These systems were set for data processing efficiency, not the organization's efficiency. *But organizational efficiency is what database is all about.*

Once a thorough information requirements analysis is complete, the real database design effort can begin, producing first a high-level design that serves the needs of the whole organization. Then a second level set of database designs should be built, each drawn from the high level design and addressing a specific database problem that has to be solved immediately. The combined scope of the second level designs should approximate that of the high level design. Finally, the lowest level of design should be constructed in the DBMS's data definition language syntax.

The products generated during the first two design levels are all part of the business model. At the highest level the design applies to the organization as a whole, and is a requirement specification for the middle level design. The middle level design applies either to a specific application or to a division of the organization. The middle level design also serves as a requirements specification for the lowest level design. The lowest level design applies both to a specific application and to the needs of a specific DBMS.



### 2.7.2 The Physical Database

For static relationship DBMSs the storage structure and the data access strategy are normally very complex. Included in their design are different kinds of pointers, indexes, relationships, storage organizations, overflow tables, access techniques, and the like. These things taken together give a static relationship database its speed. And, these things taken together tightly bind static relationship database designs to their applications.

There are two other significant parts of the physical database: the data loading subsystem and the data update subsystem.

The data loading subsystem is a one-time-only application. Database integrity begins with this subsystem. A static relationship database is best suited to be a non-redundant data store for a large collection of related applications. Prior to database, even related applications often had data in different formats, lengths, code tables, and so forth. All this data had to be brought together into one common format before loading into a database. The data loading subsystem becomes the single melting pot for diverse data. The data loading subsystems' design and implementation thus take a great deal of time; and the conversion of existing and running applications to the single database takes talent, skill, and cleverness.

The data update subsystem is typically required for both batch and on-line updating. For a static relationship database, most row access is through pointers that lead owner to owner, owner to member, member to member, and member to owner. This traversal process must be very carefully designed to make it efficient. Efficiency is required because the many different applications, brought together under one database, must have their formerly separate update and retrieval transactions coordinated and engineered to keep transaction response times acceptable.

### 2.7.3 Interrogation

Most static relationship database interrogations are through host languages like COBOL or FORTRAN. This is for two reasons:

- The database structure is often so complex that the sophistication of a COBOL-like language is typically required to make best use of it.
- A sophisticated natural language, which would allow any kind of interrogation to be formulated, traditionally operates too slowly.

A static relationship database derives its efficiency from its strict structure and pre-planned road maps of pointers. Because a sophisticated query-update language enables users to easily formulate interrogations that do not coincide with the database's design, most vendors of static relationship DBMSs have chosen not to produce very sophisticated query-update languages. They have, however, produced procedure-oriented languages, report writers, and elementary query-update languages.



### 2.7.4 System Control

System control covers the activities that control database, the DBMS, and the application. Figure 2.2 enumerates those system control functions that most affect static and dynamic relationship applications. Most significant to static relationship applications is reorganization. Since almost all programs are created with the database structure in mind, changing that structure is a complicated job. The process of reorganization must be very carefully planned so that the affected programs are identified and researched for the required reprogramming effort.

Whenever data is needed from several static relationship organized databases, the normal method of extraction is through host language interfaces (HLI). Most static relationship DBMSs do not permit multiple database processing capabilities for either their query-update or report writing languages.

### 2.7.5 Static Relationship Application Summary

The most important fact to remember about the static relationship application is that all aspects of its design should remain unchanged for as long as possible. This means that application designers should spend as much time as possible on analysis and design. If this effort is cut short, then there will be many expensive database reorganizations. But if the database is designed correctly from the start, then it will run efficiently for a long time.

## 2.8 Implementing the Dynamic Relationship Application

The fundamental goals of the dynamic database environment are:

- To create databases from existing collections of data processing files or single-purpose applications
- To allow the users maximum control over the updating and reporting from their databases

User control over updating and reporting cannot be restricted as the facilities enabling user control permeate dynamic relationship DBMSs.

### 2.8.1 The Logical Database

Database design takes place very informally. Simple flat file-like structures can be created from any existing data processing files by coding up the DDL and submitting it to the DBMS. For dynamic relationship DBMSs, the terms *file* and *table* have almost the same meaning. A complex structure, in which there are several interrelated tables, can also be created. These



interrelationships are encoded either into views or DML language expressions, which are executed by the DBMS when the interrogation is submitted. Some dynamic relationship DBMSs require the connecting columns to be of the same type, format, kind, and value. Other than that, the database design process is quite simple.

### 2.8.2 The Physical Database

Many dynamic relationship DBMSs implement each table as a distinct operating system (O/S) file. Some allow the combination of several of these tables into one O/S file. This is usually done to increase performance. When this is done, though, some of the flexibility of the independent O/S file approach is lost.

Most of the dynamic relationship DBMSs allow for the creation of indexes, both primary (unique value required) and secondary (non-unique values allowed). They use primary indexes to locate individual rows, and secondary indexes to locate sets of rows. Dynamic relationship DBMSs often use the primary index from one table as a secondary index of another table as a way to implement a parent-child relationship. In lieu of index connections, a dynamic relationship DBMS may merely require the user to state which two columns relate the tables. The DBMS takes care of all the processing.

The data loading subsystem for a dynamic relationship application is also informal. Dynamic relationship DBMSs often provide data loading utilities that transform a data processing file into a database. If the transformation is simple, the utility works. If the transformation cannot be handled, the user must accomplish data loading through a host language interface program.

Data update is performed by the users, one table at a time. Since each table is independent of any other, it is of little concern when other users update rows from other tables. Simple collections of tables can be updated by one of the DBMS's natural languages. Complex updates are normally done through host languages.

### 2.8.3 Interrogation

Most dynamic relationship DBMSs are supported by very well-developed natural languages. Some of these are actually very sophisticated programming languages that allow multiple table access, automatic report formatting, code table look-up, branching, looping, terminal prompting, definition, and invocation of stored procedures. Dynamic relationship DBMSs that do not employ the ANSI/SQL language often have under developed host language interfaces. The reason is simple: few users ever need to use COBOL or FORTRAN to process data from the database, given the sophistication of the natural languages.



## 2.8.4 System Control

Reorganization is a simple process with a dynamically organized database. A utility is invoked, the column types are added, deleted, or changed, and the DBMS automatically reorganizes the table. Whenever there is a new relationship between two tables, only one table is changed by incorporating the new column type. Following the addition of the column type, the user creates the update program to store values required by this new column for the relationship. Once valued, the basis of the relationship is in place, waiting to be used.

One drawback of a dynamically organized database application occurs when multiple table updates are attempted. The user or program attempting the update must acquire locks over all the tables involved. This is difficult, because a user is normally allowed to *open-for-update* only one table at a time, and a row must be opened before it can be locked.

Construction of a multiple table audit trail is also difficult. This is because the entries to the audit trail are usually made on a table basis. As a result, the integrated audit trail can only be created through the update program. When an audit trail is needed, the only safe vehicle for its creation is through a host language program. Consequently, the use of all other languages should be avoided.

## 2.8.5 Dynamic Relationship Application Summary

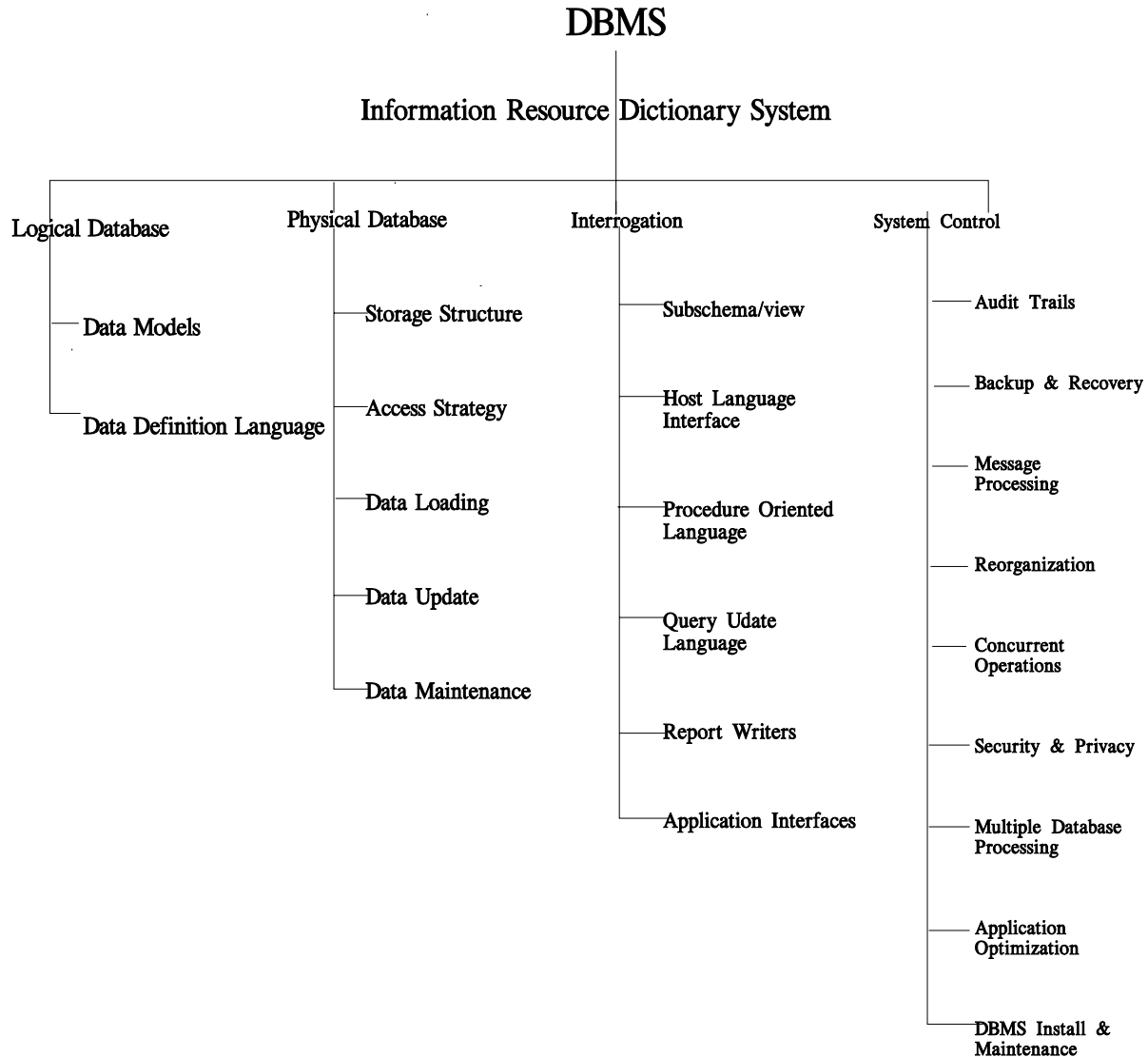
The most important fact to remember about a dynamic relationship application is that the user is in control of database definition, updating, reporting, and many of the system control activities. The dynamic relationship DBMS facility is a system where you *do your own thing*. This would be disastrous for payroll, accounts payable, and the like, but for some applications, like marketing research, it is ideal.

## 2.9 DBMS Components and Subcomponents

Figure 2.5 illustrates how the components of a DBMS are interrelated. Any data processing product contains interfaces, capabilities, and processes. Standard interfaces enable products to *connect*. Capabilities are the tools that a product brings to the user. Processing represents the transfer of the data (input and output) with which users ultimately deal.

ANSI/SPARC, in their 1975 draft reference model report, recognized the critical importance of interfaces by stating:





**Figure 2.5** Database Management Systems Components

In the course of the early discussions [about a reference model], it emerged that what any standardization should treat is interfaces. There is potential disaster and little merit in developing standards that specify how components are to work. What is proper for standards specification is how the components are meshed; in other words, the interfaces. (1)

To illustrate what is meant by interfaces, there are a number of vendor products that generate COBOL source programs, for example, MAGEC by Al Lee and Associates ([www.magec.com](http://www.magec.com)). These products enable users to define the various aspects of a COBOL program, which the products then generate. The resultant COBOL program is an interface between the product and the COBOL compiler. If the specification (content) of the interface is IBM COBOL, the programs can not be ported to DEC computers, because IBM COBOL does not work on DEC computers. If, however, the interface is ANSI standard COBOL, then a program can be specified





independent of the target computer environment, generated on either computer, and then run on both.

To illustrate what is meant by capabilities, this same program generator package might offer to interact with its users through menu-driven facilities, or through syntax input, and so on. Furthermore, the generator package might contain default capabilities, for example, that all files are VSAM, and thus might automatically generate COBOL code appropriate for VSAM. MAGEC, for example, automatically generates data access language statements to VSAM, Datacom/DB, and SQL. Interfacing to SQL means that the generated COBOL program can be used with any ANSI/SQL compliant DBMS. In comparing various types of program generator packages, various capabilities are evaluated to determine the best package. Since the purpose of a program generator is to save programmer time and produce standard bug-free code, the packages that enabled these things to be done efficiently and easily would rate highly.

To illustrate what is meant by processing, the program generator might have the same capabilities as another package, but generate COBOL code three times more slowly. All else being equal, that is, price, interface, and capabilities, the faster package would be the better buy. While DBMSs are a great deal more complicated than program generators, the ultimate selection goals are still the same. The package procured should be one that offers the most ANSI standard interfaces with the most capabilities and the most efficient processing.

## **2.10 The DBMS**

All fully functional DBMSs contain four distinct sets of facilities: the logical database, the physical database, interrogation, and system control. The first set supports the logical database and provides for the structuring of a database according to one or more of the four popular data models.

The DBMS's physical database facilities enable data loading, data update, various access strategies that affect storage structure design, and facilities to maintain the database, i.e., backup. Data update and database backup are included within physical database to understand their physical effect on the database.

The third set of facilities, interrogation, supports retrieval and update of the database's rows through a number of interrogation languages. These range from simple-to-use non-programmer facilities to sophisticated languages for professional programmers.

The fourth set of facilities, system control, provides protection to databases and database applications, and secures the DBMS and database from unwanted access. These facilities include: audit trails, backup and recovery, message processing, reorganization, concurrent operations, security and privacy, multiple database processing, DBMS installation and maintenance, and database application optimization.

### **2.10.1 The Logical Database**

The logical database component of a DBMS enables user definition of tables, columns, and explicitly defined relationships.



Column types include single-valued, multiple-valued, groups, and repeating groups. Column data types include character and numeric and fixed and variable lengths.

A defined table includes its name, primary key (if any), secondary keys (if any), and the various columns. If the table definition is for hierarchical or relational DBMSs then only single-valued columns are allowed. If the definition is for network (CODASYL or ANSI/NDL) or Independent Logical File (ILF) then, depending on the DBMS, some of the following columns can be defined: multi-valued, groups, repeating groups, and multi-dimensioned arrays.

Relationships are the expression of a semantic connection between rows from the same type, or between rows from two or more types. There are eight types of relationships that can be defined among tables:

- One-to-many
- Owner-multiple-member
- Singular-one-member
- Singular-multiple-member
- Recursive
- Many-to-many
- One-to-one
- Inferential

One-to-many relationships are the most common relationship type, and are also known as owner-member. An example of this relationship type is: COMPANY has many EMPLOYEES. The single relationship type name would be EMPLOYEES, and the two tables are COMPANY and EMPLOYEES.

An owner-multiple-member relationship is an owner-member relationship in which there are multiple-member tables. For example, a COMPANY has PART-TIME-EMPLOYEES, FULL-TIME-EMPLOYEES, and RETIRED-EMPLOYEES. The relationship type name would be EMPLOYEES, and the four tables participating in the one relationship are COMPANY, PART-TIME-EMPLOYEES, FULL-TIME-EMPLOYEES, and RETIRED-EMPLOYEES.

A singular-one-member relationship is so named because it is defined to contain only one table, a member. For example, the relationship HIGH-PROFIT-MARGIN-COMPANIES would only contain rows from the one table, COMPANY.

A singular-multiple-member relationship has no owner, but more than one member. There may be one or more member instances of each of the members for each relationship occurrence. For example, the relationship HIGH-ACHIEVERS could be defined to have rows from the PART-TIME-EMPLOYEES, FULL-TIME-EMPLOYEES, and RETIRED-EMPLOYEES tables.

Recursive relationships are those that can express owner-member relationships among rows that are from the same table. A typical example of a recursive relationship is COMPANY ORGANIZATION in which all the rows come from the table ORGANIZATIONAL UNITS. The relationship would keep track of the organization from the office of the president down to the mail room.

The many-to-many relationship type is common in database applications. This relationship, like the one-to-many relationship, involves two different tables. The relationship,



however, is one-to-many in BOTH directions. The relationship enables the first of the tables to be the owner of the second, and also the second table to be the owner of the first. A common example of the two parts of the many-to-many relationship ASSIGNMENTS is:

- 1) Each EMPLOYEE works on many PROJECTs and
- 2) Each PROJECT uses many EMPLOYEEs.

There is also the one-to-one relationship. The one-to-one relationship is not employed often. It relates a single row from one table to a single row from another table. The most common use of this relationship type is to segment the columns from one table into two or more tables. An example would be to segment respondent's profile information from the answers to a questionnaire. In this case, the two tables would be RESPONDENT's PROFILE and RESPONDENT's ANSWERS.

The inferential relationship relates rows from two types. This relationship, however, does not involve the primary key of either row. Rather, it involves a column from two different tables that is not uniquely defined in either table. For example, an EMPLOYEE table is defined to contain the column ASSIGNED PROJECT. Another table, BUILDING, has a repeating column HOUSES PROJECTS. The relationship LOCATED, between the two tables, is at best inferential as it is not known which building the employee is located in, since multiple buildings can house the same project.

In addition to having a data definition language to declare the tables, columns, and relationships to the DBMS, there must also be supplemental subclauses to define sophisticated editing and validation clauses and to execute procedures during (before and/or after) row, column value, and relationship instance updating.

Not all these logical database capabilities can be implemented in all DBMSs. When a well-defined set of facilities is implemented across a number of different DBMSs, that subset is commonly referred to as a data model. There are four recognized data models: network, hierarchical, independent logical file, and relational. Because of ANSI database standards, the relational model, which was most commonly represented via the SQL language is no longer represented by any commercial DBMS. That is because the SQL language has greatly expanded beyond relational. Chapter 3 presents the contents of the SQL 1999 (and 2003) data models.

### **2.10.2 The Physical Database**

The physical database component of any DBMS, regardless of whether it supports one or several data models, specifies the storage structure and access strategy for tables.

DBMSs allow an index to be defined for any column. These indexes are used in selection clauses to enhance processing. Sophisticated access strategies optimize the access of rows by using all the indexed columns referenced, or by ignoring the fact that a column is indexed when list processing is slower than row processing.

The physical database also supports variable physical designs such as including all of a table's data on one O/S file, storing the data from several tables on one O/S file, or spreading the



data from one table across multiple O/S files. Variable physical designs enable portions of a very large database to be moved off-line when there is no need for on-line access. Additionally, sophisticated DBMSs offer alternative O/S access methods, blocking factors, and DBMS row sizes to achieve even more highly tuned performances.

The single central-version, multiple-database DBMS enables databases to be brought on- and off-line at will, and so that database and/or DBMS failures will be isolated in their effects.

### 2.10.3 Interrogation

Interrogation is the process of selecting and retrieving data to satisfy simple queries or complex reports. Sophisticated DBMSs offer different languages, each suitable to a class of tasks. For example, a procedure-oriented language (POL) for quick application specification, a query-update language for simple reports and/or updates, and a report-writer for complicated on-line or batch reports. Additionally, sophisticated interfaces to compiler languages such as Ada, Pascal, C, COBOL, FORTRAN, and PL/I enable the development of complex database applications.

To help users access data, sophisticated DBMS vendors provide a mechanism for reducing the amount of knowledge that users are required to have of the database's data model or physical database organization. The mechanism, called a view, is a separately defined subset of columns from one or more tables. If multiple tables are involved in a view, then its definition must also implicitly or explicitly include logic to navigate between rows from different tables. Since views are simpler to comprehend, the available commands are also simplified: GET, STORE, MODIFY, and DELETE. To help users find the appropriate set of rows, the view should include a comprehensive set of WHERE clause facilities. Ideally, the view, its simplified commands, and the comprehensive set of select clauses is available to all the interrogation languages.

Of course, if the vendor does not provide a view capability, then access must take place through a more detailed set of commands that operate across the DBMS's data structures.

Some database operations are dependent upon the data model. For example, there are some special relational-only operations such as division, outer-join, and product. For NDL-related tables, there are special navigational and row-at-a-time operations such as GET OWNER, GET MEMBER, and GET NEXT.

### 2.10.4 System Control

System control capabilities provide support for controlling the database's audit trails, message processing, backup and recovery, reorganization, concurrent operations, security and privacy, multiple database processing, DBMS installation and maintenance, and database application optimization.

The DBMS's audit trails capture update transactions according to criteria such as user, table, date, time, and program. These transactions are selectively reportable, and available during backup and recovery operations.



Message processing facilities provide sophisticated on-line help for investigating the cause of a user, database, or DBMS error.

Backup and recovery facilities permit unaffected users to continue work while back-ups are taken of production databases and to undergo minimum interruptions while the DBMS recovers a database from a DBMS, user, or computing environment induced error.

Reorganization involves both logical changes to the database's column and relationship types, and physical reorganization of the database's storage structure components.

Concurrent operations enable a single run-time copy of the DBMS to support multiple concurrent update and retrieval transactions from multiple interrogation language run-units to the same or different databases without any compromise to database integrity, and with automatic detection and resolution of execution deadlock.

Quality DBMSs permit multiple databases to operate under a single central version. This allows logical database changes to be isolated and applied without affecting any of the other databases that may be running under that central version. Furthermore, if one database crashes, then only that database is affected, and while it is being recovered, the others remain operating.

Security and privacy support the definition of a comprehensive set of user, profile, and passwords to prevent and then report on illicit data access and database operations. The DBMS security provides protection at the database, table, and column level for at least update and retrieval access and ideally also for select clauses.

DBMS installation and maintenance facilities, if sophisticated, enable the generation of special run-time version that favors certain types of update or retrieval processing.

Finally, if the DBMS provides performance assessment aids, physical database designers can tune the database's logical and physical design to achieve high performance on critical applications. These DBMS application optimization facilities generate statistics on database access efficiency, overlay usage, DBMS routine invocation, and the like to guide physical and logical reorganization, or to guide the creation of a special DBMS version. A detailed presentation of the system control component is contained in Chapter 6.

## 2.11 DBMS Issues

Use of some DBMS facilities provokes argument. Each side of the argument usually has sound logic behind its position. Consider, for example, the issue of sorting. For the proponent of dynamic facilities, performing a sort of the data before it is presented to the user is a critical operation in any dynamic DBMS language. To have the DBMS maintain rows in a predetermined order is a waste of time, so the argument goes, since users will always want rows in a different order from a DBMS stored order.

On the other side of the sorting issue is the static relationship DBMS proponent who feels that if an organization has done a *proper* job of designing the database, it will rarely, if ever, be necessary to waste computer time sorting rows because the DBMS would have already sorted them into a *proper* order. Besides, if sorting does have to be done, the user should retrieve the rows and let the operating system utility sort them, relieving the DBMS of such a ridiculous and unimportant task.



The issue is actually more complicated than either of these sides allows. For example, both sides are presuming that static relationship DBMS facilities automatically maintain order among rows and do not allow sorting, and that dynamic DBMS facilities do not automatically maintain order among rows and do allow sorting. While this is generally true, SYSTEM 2000, a static relationship DBMS, does not automatically maintain rows in a sorted order, and does provide a sorting command to both host and natural languages. And FOCUS, a dynamic DBMS, maintains rows in a sorted order, and also provides sort verbs in its languages.

In other words, many issues are often founded on long-standing practices or on specific DBMS implementations, rather than on the fundamental technology of DBMS or on DBMS standards.

Nonetheless, all issues have both performance and flexibility impacts. Regrettably, performance and flexibility are almost always opposed. What you gain on one, you lose from the other.

For example, if a DBMS automatically maintains row orders, then resources must be expended to maintain these orders. As a new row is stored, its proper place must be found. Generally, the DBMS stores the row without regard to sorting, then traverses the rows according to their sort order, locating the proper *before* row. This row's *next* pointer, which points to the row's old *next* row in the sequence, is modified to point to the newly stored row. The newly-stored row then has its *next* pointer modified from NULL to the value of the old *next* row. This is a different process from merely storing the row and leaving the problem of sorting to the retrieval program.

The benefits of row sorting are the inverse of its drawbacks. If many different retrieval programs use rows, all according to the same order, then run-time sorting would not have to be performed if the rows were already stored in that order.

There are often two sides to issues like sorting, and flexibility and performance are typically on different sides. A two dimensional matrix occurs, like the one in Figure 2.4, can be employed to set down the differences. To select one alternative requires a detailed analysis of the database applications to identify the frequency of sorting during retrievals versus the cost of

Issue and Consequences	DBMS Maintained Sorting	User Maintained Sorting
Flexibility	Reduces Ability to Create Different Sort Orders on Reports	Users Can Develop Reports or Updates in Different Order
Performance	Slows Updates But Speeds Reports Using DBMS Provided Record Ordering	Speeds Updates, But Slows Reports as Each Run Unit Must Perform its Own Record Sorting

**Figure 2.4.** Flexibility vs Performance impact on DBMS.



maintaining the row orders during updates. Once the analysis is complete, the decision can be made.

## 2.12 Application Components and DBMS Components

There are significant differences between the components of a database application and the components of a DBMS. Despite the differences, the application and the DBMS are inextricably linked. The differences are related to perceptions and use sequencing.

In a database project, a need is specified on behalf of an application's uses. Then the need is addressed by implementing a facility through the DBMS. Finally, the new facility is used by the application's users to meet their business needs.

The difference between the logical database component of a database application and the logical database component of a DBMS is analogous to the difference between specifying the data to be handled in a database application and implementing that specification with the DBMS. While both represent the logical database, the representations are from different viewpoints.

For example, a database application requires the column TELEPHONE NUMBER. When implemented in a DBMS, the TELEPHONE NUMBER may be declared to be CUST-TEL-NBR TYPE IS INTEGER 9(10). The difference may arise when the DBMS can only handle names with only 12 characters, not allowing blanks in names, and requiring a data type specification.

The difference between logical database from the application point of view and logical database from the DBMS point of view extends to tables and relationships. A table, from the point of view of the application, may merely be a table name and the identifications of the contained columns. In contrast, the table from the DBMS point of view typically requires a name, column identifiers, access clauses, check clauses, and so on.

An application logical database relationship may only identify the tables involved and an enumeration of the rules that specify how to determine which tables belong to the relationship. In contrast, the DBMS relationship contains the names of the tables, which table is the owner and which are the members, whether the members are to be *sorted* within the context of the owner, and any referential integrity clauses governing row membership in these relationships.

A database application has requirements relating to the physical database, that is, an estimation of the size of each table, the volume and volatility of rows, and the nature and frequency of updates. The DBMS's requirements for the physical database relate to how these rows are stored, that is, fixed or variable lengths; how they are accessed, that is, through indexes and embedded pointers; and how update operations are carried out, that is, by updating in place, moving rows around, or splitting rows so that the different parts can be stored in different locations.

Interrogation, to a database application, is the identification and specification of necessary reports and queries, including their specification, life span, data volumes, and frequency of invocation. To the DBMS, interrogation relates to the various types of languages that are used to implement and produce the reports, including issues related to language sophistication, ease of use, and performance.

System control, to the database application, represents the identification of the requirements for audit trails, backup and recovery, and security and privacy. From the DBMS point of view, system control components represent the detailed specifications of how the DBMS



satisfies these requirements, including issues related to sophistication, ease of use, and performance.

### **2.13 DBMS Requirements Summary**

Sophisticated DBMSs support multiple data models. The two portable data models are ANSI/NDL (network) and ANSI/SQL (relational). Multiple data model supports enable databases to be built that are network only, hierarchical only, independent logical file only, or relational only. Today's popular combinations are network and relational (IDMS/R and SUPRA), hierarchical and relational (FOCUS), and independent logical file and relational (ADABAS and Model 204). The interface language of choice for the network and relational data models is NDL and SQL respectively. Regardless of the data model, all DBMSs must have sophisticated editing, validation, and referential integrity.

Sophisticated DBMSs also support multiple access methods for the files. Depending on the kind of performance needed, the relationships supported between tables and files should, at a minimum, be one-to-one, one-to-many, or many-to-many.

Modern DBMSs have a variety of languages, such as query-update languages, procedure-oriented languages, and report writers, so that whole applications can be implemented without resorting to the use of host languages.

Finally, the complete DBMS has a rich set of system control facilities for backup and recovery, and audit trails. Critical to an effective environment is multiple database processing, multiple threaded operations, and the ability to manipulate the DBMS software and buffers to achieve different performance characteristics.





## References

1. FDT, The Bulletin of ACM--SIGMOD, The Special Interest Group on Management of Data, Volume 7, Number 2, 1975, ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, 75-02-08, page I-3.





## 3

# THE LOGICAL DATABASE

### 3.1 Definition

The logical database is an expression of the data's organization represented in the database. The data organization is communicated to the DBMS through a data definition language (DDL). While most DBMSs, upon detailed examination, have a unique data organization, these organizations can generally be perceived as mapping to one of four main data models: network, hierarchical, independent logical file, and relational.

In addition to *plain vanilla* data model structures, quality DBMSs support the specification of various types of integrity rules and regulations. Integrity clauses govern the allowable operations on the database. These clauses can enable the execution of preprogrammed logic that can cause the acceptance or rejection column values, the addition/deletion/modifications of the row, and even cause triggering of other actions, which in turn might cause adds, deletions, or modifications of data within rows from other tables.

A DBMS must have a robust set of data structuring and integrity facilities to then allow use of natural languages for updates rather than requiring all update programs to be written in COBOL. COBOL, of course, supports the syntax to accomplish these editing and validation procedures through programmer created logic. If, however, these data structuring and integrity facilities are available through schema-based DBMS facilities, then a screen update program can be created in a natural language with only one or two hours work rather than one or two weeks. That's a 40:1 improvement in programmer productivity!

Even if COBOL were as easy to use as is a natural language, it would still be very important to have editing and validation facilities as a part of the DBMS, because that way all the update clauses can be defined and maintained in one place, without having to find, change, recompile, and retest many, many COBOL programs, and without risking a loophole by failing to change all affected programs.

This chapter presents detailed definitions of the components of the logical database (Section 3.2), describes how these components are combined to make up a data model (Section 3.3), and explains the language that communicates logical database constructs to the DBMS (Section 3.4). The chapter closes with a summary.

### 3.2 Logical Database Components

The logical organization of a database is defined through the specification of:

- Domains, which control ranges and types of column values
- Columns, which represent discrete business facts.



- Rows, which are collections of column value instances determined by the organization to have a collective business policy meaning.
- Relationships, which define connections established between rows of the same or different types for various reasons.
- Operations, which specify allowable actions on rows and on the relationships between and among rows.

### 3.2.1 Domains

A domain is a method of defining the business meaning of a class of values represented by columns. A column represents discrete classes of data values. These values have physical boundaries (minimum and maximum values), and represent certain business semantics, such as WEIGHT, LENGTH, and PRICE. Because a column represents business semantics, certain combinations of columns should be permitted, while others should not. For example, DATE should not be multiplied by PRICE as the result has no common business meaning.

A domain contains the specification of the type and value characteristics that might be used by various columns in different tables in the database. For example, sex would have the domain of male and female; salary would have the domain \$3,000 to \$60,000. An example of a domain appears in Figure 3.1.

DOMAIN IS SALARY TYPE IS NUMERIC PICTURE IS 99,999.99 RANGE IS 3000 TO 6000
--

**Figure 3.1.** Domain Salary

Because domains control values, they are in some sense *super-columns*. Consequently, domains have definitions similar to those of columns. The characteristics that describe domains reflect whether the data represented by the domain consists of fixed- and variable-length character strings, fixed- and floating-point numbers, integers, or special data types such as date and time.

More importantly, domains are employed to specify a series of statements that govern the values that are allowed in one or more columns. Included in these rule sets are:

- Valid value ranges
- Specific valid values
- Invalid value ranges
- Specific invalid values
- Duplicates allowed or disallowed
- Default values



- Conversion rules
- Null value allowed or disallowed

Finally, complete domains contain rules that define allowed interactions. For example, it might be perfectly legal to multiply a duration of time by an integer to arrive at a longer duration. But in an ordinary business situation, it would not make sense to take the square root of time. Some operations might be allowed on different domains, producing a third domain. For example, multiplying SQUARE FEET by POUNDS yields PRESSURE. These rules of engagement must be defined in the database so that when the widest possible audience is using the database through a natural language, database operations such as additions, modifications, and various combinations can be made in a manner deemed acceptable by the organization.

### 3.2.2 Columns

Columns represent discrete instances of business data. Examples are: SOCIAL SECURITY NUMBER, RECEIPT DATE, FIRST NAME, ADDRESS, and REMAINING AMOUNT. Columns exhibit many characteristics, all of which must be known before the column can be fully understood. Each column clause incorporates by reference the specifications contained in a domain specification, and possibly even augmentations to them. For example, the domain EMPLOYEE SALARY may have a value range \$1 to \$5,000, while the HOURLY EMPLOYEE SALARY may further restrict that range to \$4 to \$20. A column consists of its name, its data type, its structural composition, and the applicable integrity clauses.

#### 3.2.2.1 Column Names

Among the many issues surrounding columns, the issue of names is very important. To adopt a column naming standard that (for example) suffixes each column with a code indicating its use or characteristics trivializes the issue, however. Such a proposal would force the choice of a single suffix indicating that the column is a key, or a code, or a date, or a number, and so forth. Rather there must be a fully developed IRDS to store all the rules surrounding columns. In short, there is much more to a column than just its name, as the following examples show.

SOCIAL SECURITY NUMBER is typically represented as a 9-digit number, but is not arithmetic in nature. Consequently, it is senseless to compute the average SOCIAL SECURITY NUMBER for a sampling of EMPLOYEES. Additionally, there are legal and illegal values for SOCIAL SECURITY NUMBER. Thus, while, its form is certainly numeric, SOCIAL SECURITY NUMBER is certainly not numeric in the conventional sense. Thus, suffixing SOCIAL SECURITY NUMBER with the data type NUMERIC would hide its true nature, looks ridiculous, and take up an additional 8 character positions in the name.

RECEIPT DATE is another numeric business column. RECEIPT DATE has legal and illegal values and, if properly stored, can be used in some types of arithmetic logic. For example, a query might be constructed to print the DATE that is 15 days beyond a specific RECEIPT DATE. It would also make sense to compute the standard deviation, average, or median number



of days elapsed from the date an invoice was issued to the date it was paid. Such a calculation involves subtraction, summing, and square roots. Multiplying two dates, on the other hand, would make no sense. So, to say that RECEIPT DATE is a number is to mislead users as to allowable operations even though it can be involved in numeric calculations. To establish a data type called DATE is a good solution, but unless it is carefully thought out as to internal storage, the variety of external representations, and the problems caused by different time zones, this would cause more problems than it would solve.

FIRST NAME is most probably an alphabetic column. Its values are restricted to a combination of the instances of LETTERS, with the first letter being in upper case. In general, numbers would probably be an indication of an error, although not always. After all, the movie *Star Wars* gave rise to C3PO and R2D2! Determining the real column name here is difficult. After all, is NAME a part of the column's name, or is it an indication of the type of data (character)? Thus, the column's name could be NAME, or FIRST, or FIRST NAME, or *NAME FIRST*. Depending on how that question is answered, and presuming the domain to be CHARACTER, the following alternatives for first name are:

- CHARACTER NAME
- CHARACTER FIRST
- CHARACTER FIRST NAME
- CHARACTER NAME FIRST

Another type of column is ADDRESS. This column presents a different set of problems, the least of which is its data type. ADDRESS does not represent a discrete value, but a set of *lower level* columns, each of which has a discrete value. Each of the lower level columns may be of a different data type. For example, ADDRESS is most likely composed of EFFECTIVE DATE, STREET NUMBER, STREET NAME, CITY, STATE, ZIP, and COUNTRY. Again, each of these columns may have rules for the types of values, and rules for specific ranges of legal values. In the case of CITY and ZIP, these two have specific instances of legal values--in combination. In the case where the country was the United States, letters would be illegal in a ZIP code, while in CANADA, an all numeric ZIP code would be illegal. So, what is the domain suffix of ADDRESS, the first data type, the mode data type, or the average (whatever that means)? Since no meaningful domain prefix can be determined, it is inappropriate for any to be assigned.

A final example of a column is REMAINING AMOUNT. This column is obviously computed from some other source. Furthermore, it may have certain rules about its upper and lower limits (negative not allowed, for example). This column would probably also be subjected to rounding rules for MONEY, and be allowed to be operated upon in certain ways, but not in others. For example, the SQUARE ROOT of REMAINING AMOUNT might not be sensible, but the STANDARD DEVIATION (which involves square roots) may be. Again, identifying the data type as numeric would be misleading, as not all arithmetic operations ought to be permitted.

The point of all of these examples is that columns conform to certain rules of behavior, some of which are implied by their obvious data type, and some of which can be mistakenly implied by their data type.



Another attribute of a column is its use context. For example, the column REMAINING AMOUNT might be allowed to have negative values in some situations, but not in others. In a checking account, for example, a negative REMAINING AMOUNT might be allowed, resulting in INTEREST being charged by the bank. In the case of a mortgage payoff, the amount to be paid is required to be exactly that which is due, since the REMAINING AMOUNT on a mortgage cannot be negative. And, of course, if a REMAINING AMOUNT column were part of an INVENTORY-ON-HAND row, not only negative numbers but also fractions would not be allowed.

It should be quite clear that a column cannot be fully understood unless its scope, intent, rules, and other modifiers are known. These include:

- Data characteristics
- Use context
- Domain
- Valid, invalid, and range rules
- Allowable combinations of operations

As has been shown, these modifiers when applied to the same column--in different combinations--produce different results.

The only convention that leads to understanding, rather than to misunderstanding, is to give a column its most common business name and state all the modifiers that provide for complete definition. With this convention, column names can be used over and over where appropriate, changing only the name of the context in which it is used.

For example, when the column REMAINING AMOUNT is prefixed as follows, its meaning is almost obvious:

- MORTGAGE REMAINING AMOUNT
- INVENTORY REMAINING AMOUNT

But what about the issues of type, valid values, and so forth that were presented above? The answer is simple: provide the additional definition of a column's semantics (rules of meaning and usage) outside the confines of a column's name. With that approach, the semantics can be expressed in a form that is appropriate for the subject column.

### 3.2.2.2 Data Types

The types of data represented by a column include fixed- and variable-length character strings, integers, fixed- and floating-point numbers, and special data types such as date, time, and money.

Fixed-length character string columns are appropriate for formatted data. Data involving paragraphs of text, comments, and addresses can benefit from variable length character strings.

Numbers are represented either exactly or approximately. Exact numbers are represented with a precision and a scale. Precision denotes the number of digits in the number. The number



346.98 has a precision of 5 because the number has five digits. Scale is the number of digits to the right of the decimal point. The number 346.98 has a scale of +2.

Integers are exact numbers with a scale of zero (0). That means that integers represent only whole numbers. The magnitude of these numbers is usually hardware-dependent. A typical hardware restriction is some number that is a power of 2, for example, 2 to the 24th power, or about 16,777,220. Some integer numbers are DBMS software-dependent. For example, FOCUS has a maximum length of 9 digits for integers. That means that an integer number can range from 0 to 999,999,999.

Fixed decimal numbers are exact numbers with a scale greater than zero. 3289.445 is an exact fixed-decimal numeric with a precision of 7 and a scale of 3.

Approximate value numbers are commonly known as either single precision or double precision floating numbers. Approximate numbers consist of a mantissa and an exponent. For example, in the number  $5.87125 \times 10^3$ , 5.87125 is the mantissa, and  $10^3$  is the exponent. Single precision implies that the mantissa can only be a certain length. Double precision implies a precision greater than single precision, usually twice as great.

Data types such as date, time, and money represent special uses of numbers. In addition to specifying the legal values for dates and time, a date-time facility also provides rules for operations. A money data type for American dollars would have the \$ sign, commas, and appropriate rounding.

### 3.2.2.3 Column Structures

Structural composition clauses indicate whether a column is single-valued, multi-valued, multi-dimensioned, a group, or a repeating group.

A single-valued column, for example, PRODUCT-NAME, represents only one value per row. Multi-valued columns represent multiple occurrences of values, all of the same data type. Multiple-valued items have either one or more dimensions. A single-dimension, multiple-valued item is often called a vector or an array. Its values are distinguished positionally, that is, the first value, the second value, and so on. Figure 3.2 illustrates a multiple-valued item, TELEPHONE NUMBER, of a single dimension.

TELEPHONE NUMBERS			
3012495300	3012491142	7176485913	...

**Figure 3.2.** Multi-valued columns

If the multiple-valued item has more than one dimension, then a column like SALES-BY-DISTRICT-BY-MONTH results, as depicted in Figure 3.3. It represents 120 values, 12 per year for 10 DISTRICTS, for each product record. The values in this multi-valued item are distinguished through the intersection of the *by* values. That is, the sales for *district* = <value> and for *year* = <value>.





Some columns are groups. A group column has a name and a set of individually defined subordinate columns, each with their own column characteristics. For example, FULL-NAME might be subdefined into FIRST-NAME, MI, and LAST-NAME. A typical group column is ADDRESS, illustrated in Figure 3.4. Within the group are subordinately defined columns, each having its own data type specification.

The final column type is repeating group, a group with multiple occurrences. For example, there might be a discount structure for product sales. For each DISCOUNT, there might be a QUANTITY-MINIMUM column, a QUANTITY-MAXIMUM column, and a PRICE column. In some DBMSs, a repeating group is additionally allowed to contain multi-valued columns and other repeating group columns. Figure 3.5 illustrates the repeating group DEPENDENT in the EMPLOYEE row. Note that in this repeating group there is the additional repeating group column, HOBBIES.

SALES GROUP Column (12 X 10 X 1)			
DIMENSION	12	10	1
SUB-Column	MONTH	DISTRICT	VALUE
DATA VALUE	JAN	1	\$2500
	JAN	2	\$1800
	JAN	3	\$1700
	JAN	4	\$2900
	JAN	5	\$1100
	JAN	6	\$2300
	JAN	7	\$2400
	JAN	8	\$1600
	JAN	9	\$1800
	JAN	10	\$3300
	FEB	1	\$3560
	FEB	2	\$2200
	FEB	3	\$3400

**Figure 3.3.** Multiple Dimension Columns



Address
Number
Street
City
State
Zip

**Figure 3.4.** Group Column

Employee - ID
•
•
•
Dependents (RG)
Name
Sex
Birth-date
Hobbies (RG in Dependents)
Hobby name
Annual cost

**Figure 3.5.** Repeating Groups



### 3.2.2.4 Column Integrity Rules

Columns are governed by integrity rules. These rules typically involve:

- Valid value ranges
- Specific valid values
- Invalid value ranges
- Specific invalid values
- Numbers of occurrences
- Duplicates allowed or disallowed
- Default values
- Conversion rules
- Null
- Encodes and decodes

Figure 3.6 contains an example of the data definition language (DDL) for the column JOB-TITLE. The only valid column values for JOB-TITLE type are SENIOR, TRAINEE, or MANAGER. In this column, duplicate values are allowed, but no-value (NULL) and JUNIOR are illegal.

Column IS JOB - TITLE, TYPE IS CHAR, LENGTH IS 10,  NULL IS NOT ALLOWED, DUPLICATES ARE ALLOWED  VALID VALUES ARE SENIOR, TRAINEE, MANAGER
--

**Figure 3.6.** Job-Type column definition.

### 3.2.3 Tables

A table, in the context of a DBMS, is a collection of assigned columns. A DBMS allows rows, represented by the table's definition, to be inserted, deleted, and sorted, and to be related to rows of the same and different tables.

From the DBMS point of view, the row is just a formatted string of data representing the values from all the columns. Thus, if a row is a statement of management policy, then the definition of that policy must be completed before the database application's logical database is developed.

In general, the syntax for a table specification consists of the clause that identifies the name of the table, a clause identifying the primary key, clauses identifying other keys, one or more clauses enumerating the table-based integrity clauses, and a list of the columns identified as belonging to the table.



Whenever columns are assigned to a table, their role must also be defined. These roles are

- Primary key
- Secondary key
- Candidate key
- Foreign key (unique and nonunique)

### 3.2.3.1 Table Name Clause

The data name clause typically consists of only the name of the table and the name of the schema to which it belongs.

### 3.2.3.2 Table Integrity Clauses

A data integrity clause represents a collection of computable rules that govern the acceptance or rejection of the row as a whole. For example, row insertion might be denied if all the columns are not valued or if HIRE-DATE is less than the BIRTH-DATE or greater than the DEATH-DATE.

In the example of the table EMPLOYEE, shown in Figure 3.7, the table name is EMPLOYEE; and on the data manipulation (DML) verb STORE, the procedure STORE-REC-PROC is invoked to process the row before its actual inclusion in the database. The DBMS will not allow duplicates for this table in the database. The DBMS is instructed by the DDL to build indexes for SSN, JOB-TITLE, and REGION. Finally, the columns in the table include SSN, EMPLOYEE-NAME, etc. The actual data represented by a table's definition is called a row. A row for the table defined in Figure 3.7 might be

262625949|Ed Wallish|Manager|North East|. . .

In this example, the / symbol illustrates column value separation.



```

RECORD IS SALESPERSON;
ON STORE, CALL STORE - REC - PROC;
INDEXES ARE SSN, JOB - TITLE, REGION
DUPLICATES ARE NOT ALLOWED
FIELD IS SSN, TYPE IS . . . . ;
FIELD IS EMPLOYEE NAME, TYPE IS . . . . ;
FIELD IS JOB - TITLE, TYPE IS . . . . ;
FIELD IS REGION, TYPE IS . . . . ;
.
.
.
RECORD IS . . . . .

```

**Figure 3.7.** Salesperson table definition.

### 3.2.3.3 Columns

When data elements are assigned to tables, columns result. In this sense, a data element is a context independent business fact semantic template that is one level of abstraction higher than a column. When the DBMS allows only single-valued columns in each table, then its table is termed simple. When multi-valued columns, groups, and the like are allowed, the table structure is termed complex.

Some DBMSs allow tables of such complexity that it is difficult to distinguish tables from dependent structures within a table. For example, SYSTEM 2000's hierarchical data model permits any reasonable number of repeating groups on any level, up to 32 levels. Is the repeating group a table in its own right, or is it a dependent segment within a table? The following guideline establishes the difference:

A table can be stored, retrieved, and deleted on its own, independently from others, while dependent segments need the location and context of other table instances for their own storage, retrieval, and deletion.

Given this guideline, technically, in SYSTEM 2000 the database's root segment and all first level repeating groups within the root segment are tables. For the sake of simplicity, however, within this text, all System 2000 segments are referred to as tables.

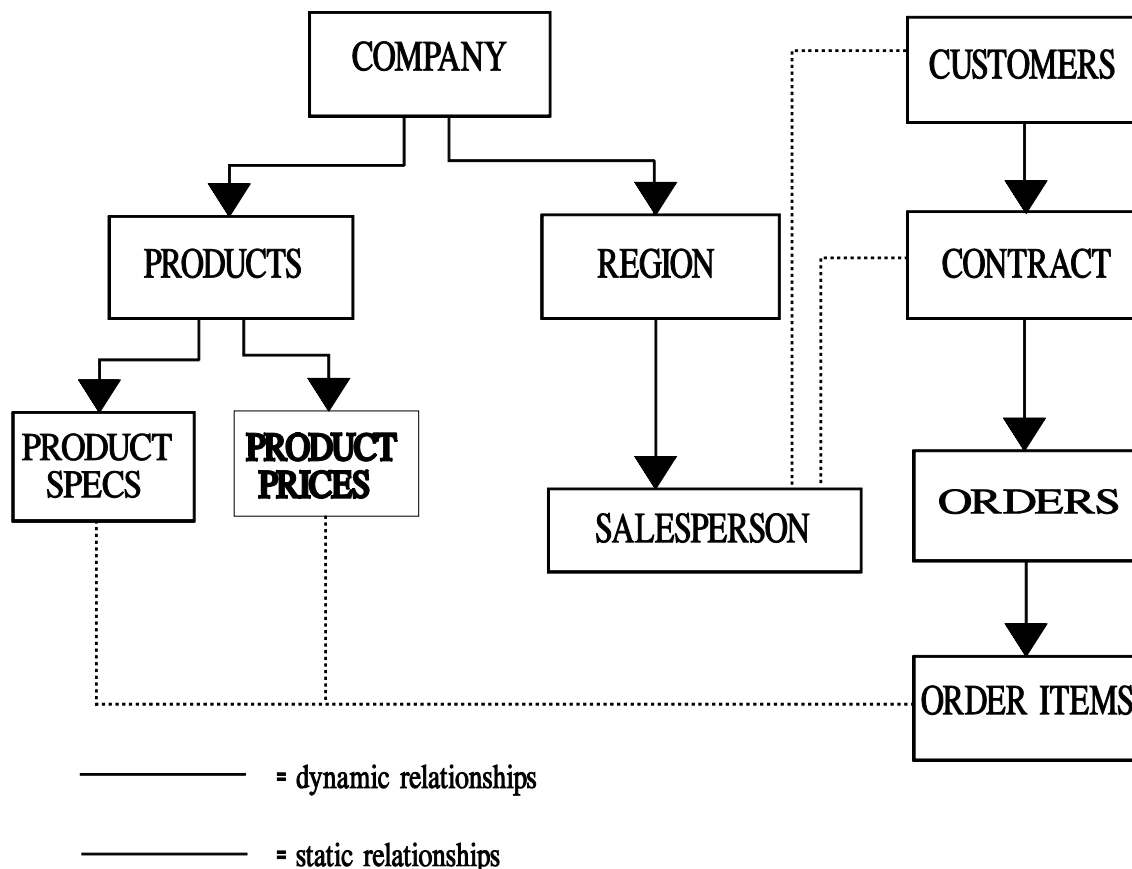
A DBMS table almost always has a defined primary key with all dependent segments defined in subordinate roles. When a row is deleted (after being selected by means of its primary key), all dependent segments are also deleted. In contrast, a dependent segment does not require a primary key of its own, especially in static relationships, because the dependent segment is accessed from within the context of a table that has a key. Further, the dependent segment can be deleted without affecting the higher level segment instance. However, its deletion often affects



lower level segments that may be defined in a subordinate relationship to the segment being deleted.

A SYSTEM 2000 implementation of the structure in Figure 3.8 results in two databases, each having one table. The first table has six segments, and the second has four. Each has only one table because a deletion of an instance of the COMPANY also removes all instances of the COMPANY's dependent segments. PRODUCT, PRODUCT SPECIFICATION, PRODUCT PRICE, REGION, and SALESPERSON are also deleted.

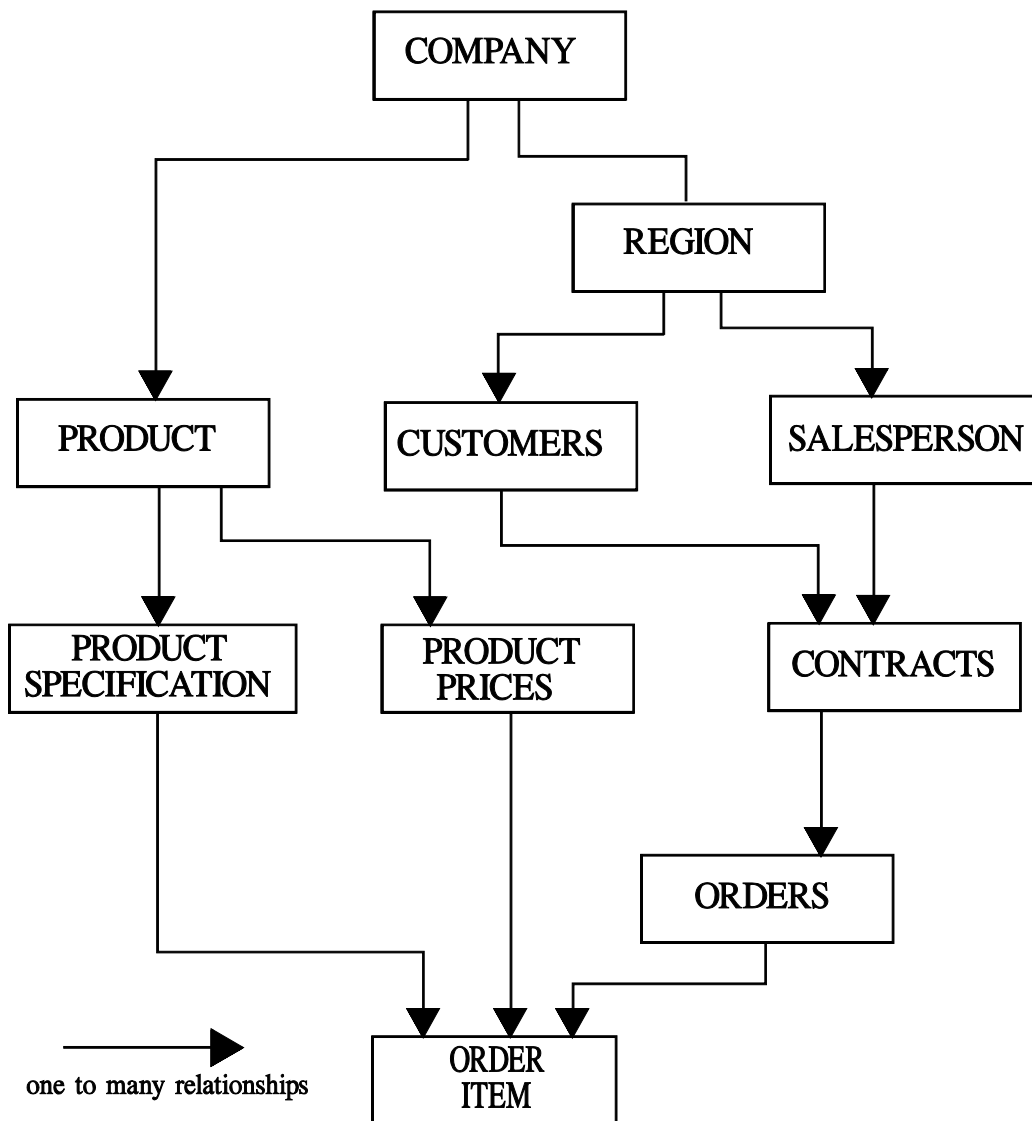
IDMS/R can implement the database illustrated in Figure 3.9 either as a single database with a set of simple independent tables, or as a single database that contains several complex tables along with several simple tables. The segments that are related hierarchically can be implemented as independent simple tables or as dependent segments with complex tables. The database designer must define formal relationships (ANSI/NDL or CODASYL sets) to relate



**Figure 3.8.** Hierarchical data model with two databases, one with 6 and the other with 4 tables.



independent tables. No formal relationships have to be defined to relate dependent segments. Formally defined relationships (sets) are defined only between and among tables, not among dependent segments.



**Figure 3.9.** Network Data Model. One database with 10 tables.

### 3.2.3.4 Column Roles

All columns are attributes of the table. That is, they represent a quality or characteristic of the table. It is important to note that *attribute* implies a semantic affinity. Thus, it would not make much sense to assign the column HORSEPOWER to the row EMPLOYEE.



Some of the columns assigned to a row can serve one or more additional roles. These roles are commonly known as:

- Primary keys
- Candidate keys
- Secondary keys
- Foreign keys

In this context, a key represents a certain integrity or access characteristic.

When a column takes on the role of primary key, the value represented by the column uniquely identifies a row. SOCIAL SECURITY NUMBER would likely be the primary key for an EMPLOYEE table. There may be more than one column assigned to the primary key role. In such a case, the primary key is represented through a compound column. For example, the primary key for a STUDENT GRADE table is likely to be the value represented by the combination of the values of five columns:

- STUDENT NUMBER and
- COURSE NUMBER and
- COURSE SECTION NUMBER and
- COURSE YEAR and
- SEMESTER.

The STUDENT NUMBER is the primary key of the student's table. The COURSE NUMBER, COURSE SECTION NUMBER, COURSE YEAR, and SEMESTER columns are together the primary key of the course-section table.

If a table EMPLOYEE has two columns representing unique values across all rows, for example, EMPLOYEE ID and SOCIAL SECURITY NUMBER, then only one of these columns can be the primary key. The other column is known as a candidate key. In the EMPLOYEE database, the primary key might be the combination of the two columns: EMPLOYEE NAME and BIRTHDATE. If this were the case, the SOCIAL SECURITY NUMBER would assume the role of a candidate key. The candidate key is a kind of alternate primary key, in that it can also be used to select a uniquely existing row.

A role different from either primary or candidate key is secondary key. The secondary key role implies that the values, more often than not, appear in multiple rows. Thus, a query on the basis of a secondary key's value usually retrieves more than one row. For example, a query to the EMPLOYEE row on the basis of DEGREE = BA will retrieve all those employees who have been awarded the BA degree. A secondary key can be defined singly or as a compound secondary key. For example, in the STUDENT GRADE RECORD, COURSE NUMBER and COURSE SECTION NUMBER and COURSE YEAR and SEMESTER would be the columns necessary to identify the complete set of students that had enrolled in the particular course section. Both columns, YEAR and SEMESTER, are needed to screen out the students who had taken the same course in earlier semesters.

When a primary key is a compound column, one or more of the columns comprising the primary key may also be a secondary key.





Foreign key is the last of the types of roles that a column can assume. A foreign key generally takes on the characteristics of a secondary key with the additional constraint that it is the primary key of another table. Thus, in the example of the STUDENT GRADE RECORD where the primary key is represented by the five columns: STUDENT NUMBER and COURSE NUMBER and COURSE SECTION NUMBER and COURSE YEAR and SEMESTER, the columns COURSE NUMBER, COURSE SECTION NUMBER, COURSE YEAR, and SEMESTER additionally take on the role of a compound foreign key. That is, COURSE NUMBER, COURSE SECTION NUMBER, COURSE YEAR, and SEMESTER together, are the primary or candidate key of the row COURSE SECTION. Further, within the COURSE SECTION table, COURSE NUMBER is a foreign key to the COURSE row in which it is the primary key.

It is obvious from the foregoing presentation that a column can take on multiple roles. The IRDS must be able to reflect them all.

### 3.2.3.5 Physical Implications of Table structures

If a row has all fixed length columns assigned to it, the physical effects are minimal. However, if a row is complex and/or has variable length character columns, then the DBMS has to be quite sophisticated to handle the complexity in an acceptable manner.

A number of benefits result from having complex tables. For example, if an EMPLOYEE row is to be defined, most of the columns that relate only to the EMPLOYEE can be defined and stored in one logical unit. Along with the EMPLOYEE's basic biographical information, it is important to collect and store pre-employment interview results, prior company work experiences, education, courses attended, dependents, benefits selected, and so on. To require the definition of completely separate tables, replete with their own primary keys and referential integrity constraints for each of these data groupings, over and above the definition of the main EMPLOYEE table, is to make work where none should be.

It is a perfectly good idea and discipline to specify tables in third normal form, but it is absurd to require that tables be implemented that way in the face of sophisticated complex row capabilities. Not only does the process of definition take longer, but computer performance often suffers dramatically. In a benchmark between a relationally implemented database and an equivalent one implemented with complex rows, the DBMS with complex tables performed 300% better.

In dynamic relationship DBMSs, rows are normally stored contiguously. Because of this, most dynamic relationship DBMS tables are simple. Some are complex, but dynamic relationship DBMS vendors typically recommend only one level of repeating groups in order to have rows that can be stored in a computer-efficient manner. Dynamic relationship DBMSs thus have either simple rows or complex rows with only one nested level.

In some static relationship DBMSs, the tables can have complex structures such that the segments are stored noncontiguously from the main part of the row. These noncontiguous segments are connected by DBMS-generated addresses. Consequently, a complex table, which implies a significant amount of data (maybe 5000 characters), can be stored in a computer-efficient manner.



### 3.2.3.6 Table Summary

A fully defined row consists of table semantics, and columns and associated clauses containing directly and indirectly the appropriate combination of the following categories of information:

- Table name
- Table description
- Column name
- Column description
- Data type, length, and picture clauses
- Column role identifiers (primary, candidate, etc.)
- Valid value entries
- Invalid value entries
- Range value entries
- Encode/decode entries
- Referenced domain name
- Referenced domain description
- Referenced domain's valid value entries
- Referenced domain's invalid value entries
- Referenced domain's value entries

Completely defining tables and their associated columns is not a simple task. To support the definitions there must be a sophisticated IRDS to store all these business semantics.

### 3.2.4 Relationships

A relationship is the manifestation of a business rule between two or more rows from the same or different tables. As an example, a company might require that all employees be assigned to a project, or that an employee be assigned to a department, or (in the case of a University) that all teachers must belong to a college and teach at least one course. Such business rules, manifest in a database as relationships, can be either arbitrary or computed. A relationship is arbitrary when the set of rules that binds rows together is not reflected in the set of values stored in one or more columns contained in the rows. A relationship is computed when the set of rules is manifest as commonly shared column values.

The following example illustrates both arbitrary and computed relationships. A company's regional manager determines, once each quarter, that a certain set of salespersons are the BEST SALESPERSONS.

If the relationship between the sales manager and the salespersons selected for the honor is arbitrary, then the rules determining the set of BEST SALESPERSONS might only be known to the sales manager. Once the decisions had been made, the sales manager would create the list of SALESPERSONS belonging to that category. The mechanism that relates the sales manager to all these SALESPERSONS is the relationship.



Alternatively, if the relationship between the sales manager and the SALESPERSONs selected is to be computed, then all the factors that determine the BEST SALESPERSONS category must be known to all and stored in the rows, so that a mathematical formula involving all the columns representing the factors can be created and used to find the SALESPERSONs satisfying the BEST SALESPERSONS category.

Arbitrary and computed relationships differ in one very important respect: explicit semantics. In the case of arbitrary relationships, the semantics that bind the rows are known to the sales manager, personally. In the case of computed relationships, the semantics must be known to all, democratically.

Relationships are an extremely critical component of database, and cannot be fully understood until the following topics have been examined:

- Relationship types
- Relationship implementation mechanisms
- Referential integrity

#### **3.2.4.1 Relationship Types**

There are eight types of relationships that can be defined among rows. These are

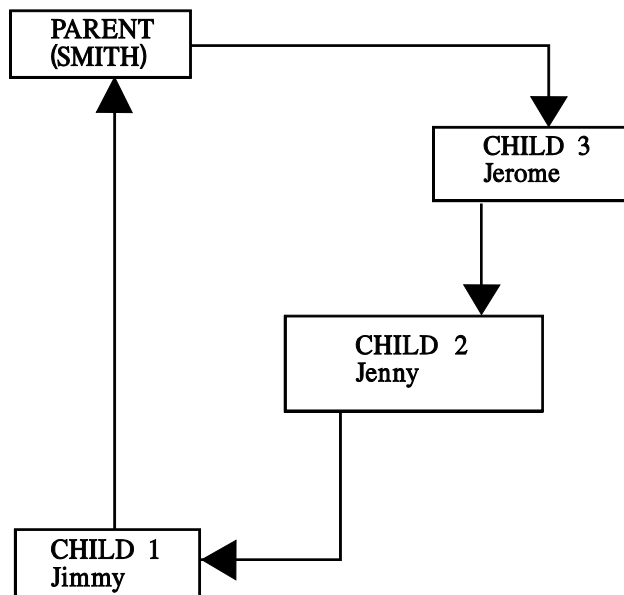
- One-to-many
- Owner-multiple-member
- Singular-one-member
- Singular-multiple-member
- Recursive
- Many-to-many
- One-to-one
- Inferential



### 3.2.4.1.1 One-to-Many Relationships

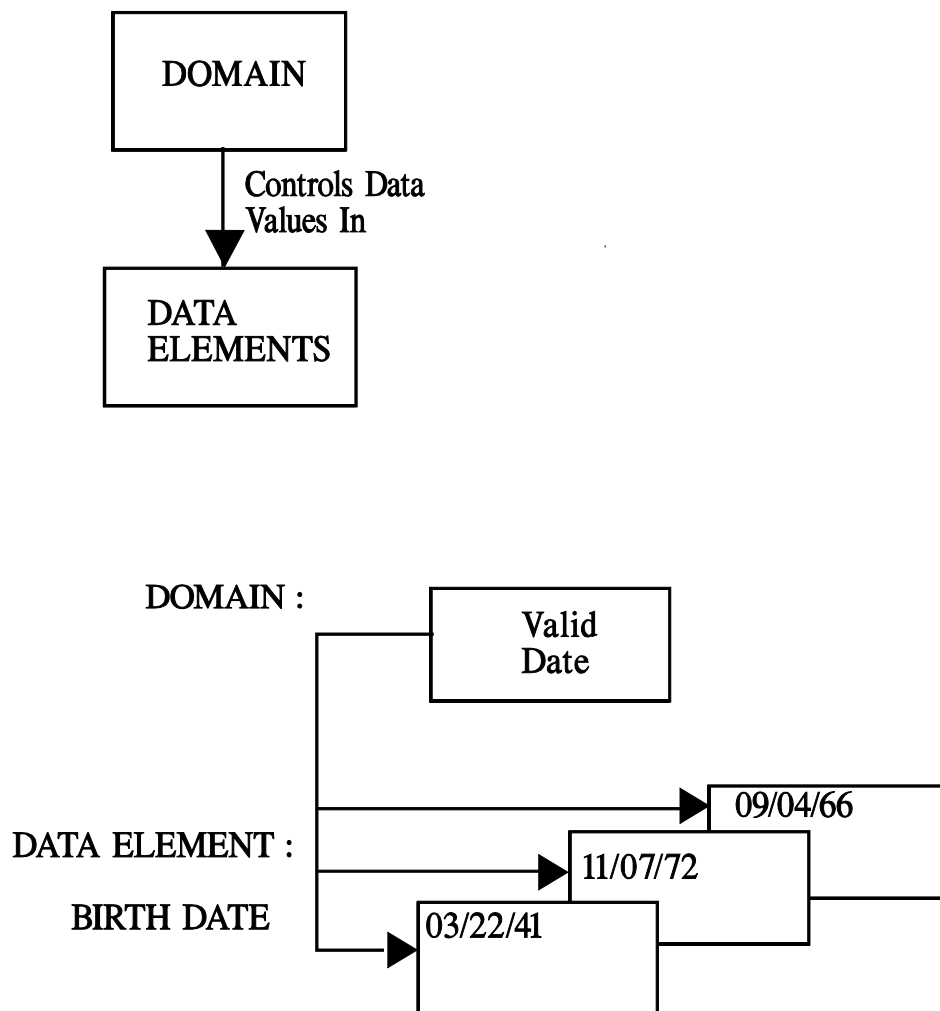
The most common relationship is one-to-many, also known as owner-member. An example of this relationship type is: *COMPANY has many EMPLOYEES*. In the case of an IRDS, an example is: *Row has many columns*. The relationship is hierarchical in nature. That is, for each owner instance, there can be one or more member instances. The relationship is graphically represented with an arrow that has two heads on one end (  $\text{---}\triangleright$  ). The point of the arrow touches the member and the shaft end touches the owner (owner  $\text{---}\triangleright$  member). In Figure 3.9, all the relationships, starting at the top of the figure and preceding to the bottom, are owner-member relationships.

Figure 3.10 illustrates a one-to-many relationship that exists between the two tables PARENT and CHILD. An instance of the relationship is illustrated. Note that the relationship is not the rows, but the mechanism of connection between the rows. Thus, in the relationship CHILDREN, there are four links that are the relationship, as well as four rows. Figure 3.11 depicts an IRDS owner-member relationship that exists in an IRDS between a column's domain and the column. For each column there can be only one domain, while there may be many columns governed by one domain.



**Figure 3.10.** Regular Owner-member set. Set type: regular, single member





**Figure 3.11.** One-to-many relationship



### 3.2.4.1.2 Owner-Multiple-Member Relationships

An owner-multiple-member relationship is an owner-member relationship in which there are multiple-member tables. This type of relationship is necessary when a single relationship is expressed across multiple types of diverse data. For example, a company has different classes of employees and keeps different quantities of data for each class. To interrelate all the employees from the different classes, a single relationship is needed from COMPANY through each of the different classes. Such a relationship has COMPANY as the OWNER, and PART-TIME-EMPLOYEES, FULL-TIME-EMPLOYEES, and RETIRED-EMPLOYEES as the members.

Another use of the owner-multiple-member relationship is found in sales and marketing. Companies typically have their sales regions divided into territories, where each territory has one salesman and multiple customers. The owner-multiple-member relationship would thus have REGION as the owner, and SALESMAN and CUSTOMER as the members. This relationship would be processed by accessing an owner row instance, for example, the San Francisco region, then accessing the salesman and each of the customers assigned to the territory.

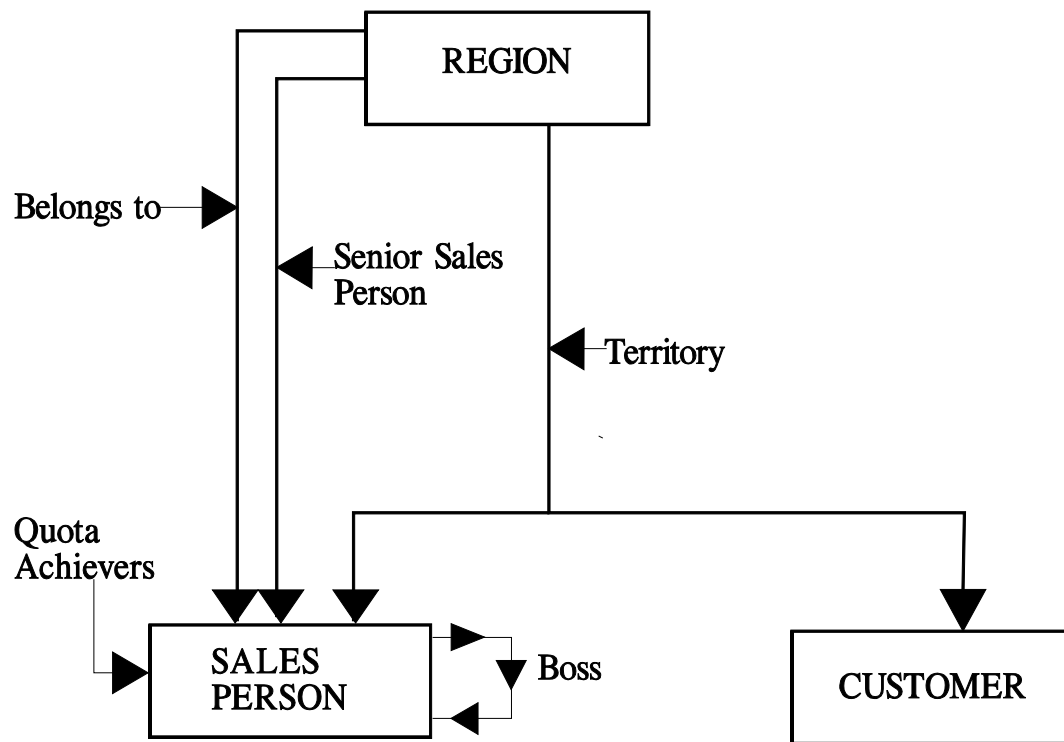
Figure 3.12 illustrates the definition of a multiple-member relationship TERRITORY. The owner is REGION and the members are SALESPERSON and CUSTOMER.

Figure 3.13 illustrates an instance of another multiple-member relationship in which the owner table is MACHINE TYPE, and the member tables are SALESPERSON and STORE. An owner row is Vending Machine, and the STORES in which vending machines are placed are the Arrowsmith Archery, the Hi Rise Bakery, and the Hard Knock Hardware Store. Finally, the SALESPERSON is Jones.

Here is a final example of multiple-member relationships. If there are separate tables for each marketing program due to different columns, and if they are *owned* by a marketing campaign row, then the multiple-member relationship permits the review of all rows from all members through the traversal of one relationship, rather than requiring a different relationship for each table. This capability saves relationship definition time and programming time, and greatly eases report writing syntax.

Each of the rows in the multiple-member relationship must be capable of standing on its own. It must have its own primary key, column definitions, column and row check clauses, row lengths, row sort clauses, ability to participate in other relationships, and ability to be stored within the same or different O/S files.



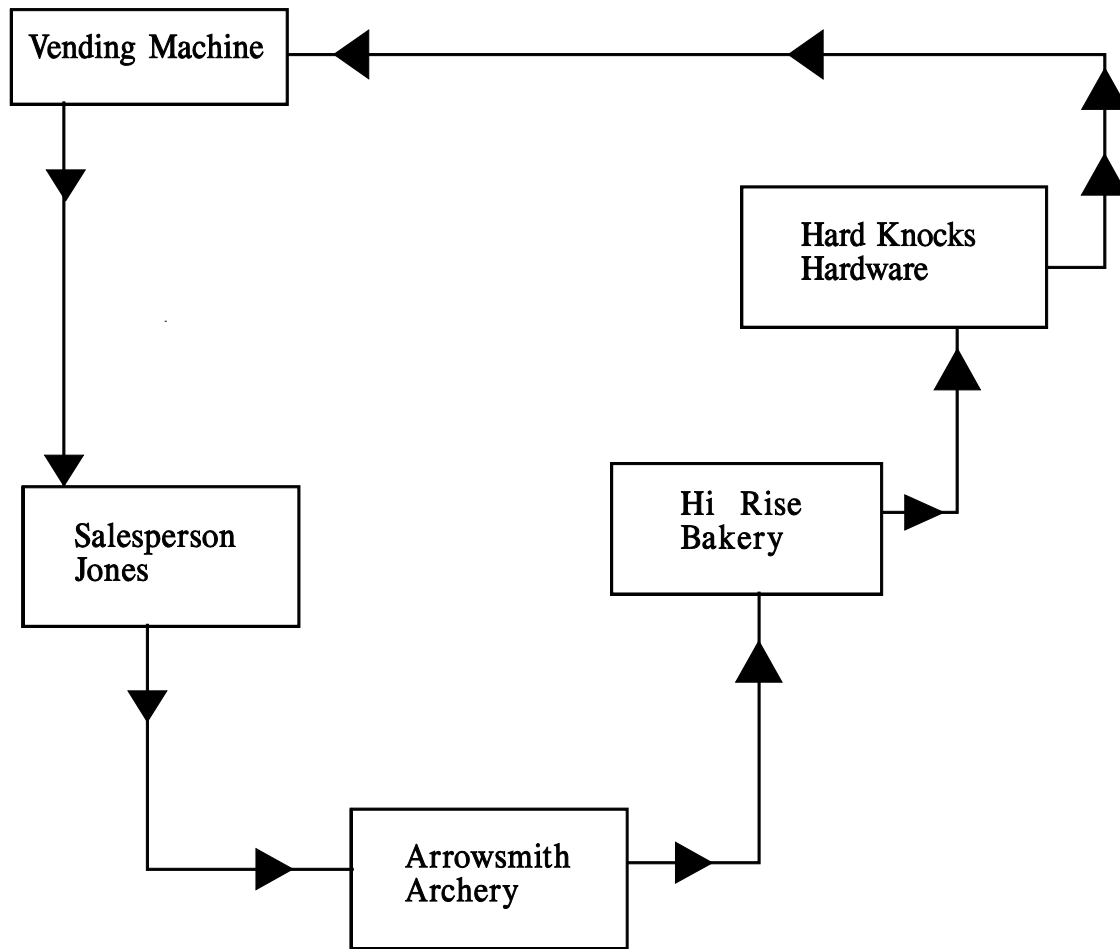


**Relationship Types :**

Singular - Quota Achievers  
 Single - Member - Senior Sales Person  
 Multiple - Member - Territory  
 Recursive - Boss

**Figure 3.12.** ANSI Network data model relationships.





Set : Territory

Owner : Machine Type

Member : Salesperson, Store

**Figure 3.13.** Relationship Type: Multi-member set



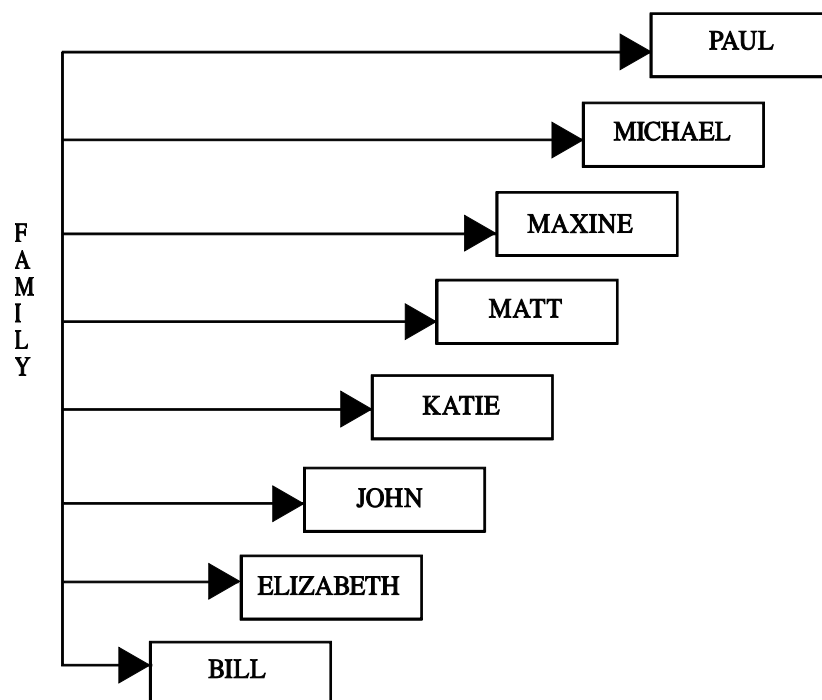


### 3.2.4.1.3 Singular-Single-Member Relationships

A singular-single-member relationship is so named because it is defined to contain only one table, a member. There is no owner table. The need for an ownerless relationship arises when it becomes necessary to relate rows from the same table without there being a naturally existing--different--owner table. For example, in Figure 3.9 there is no naturally existing table that is the owner of the COMPANY table instances. The singular-single-member relationship interconnects all COMPANYS that have a profit margin above a certain percentage.

Another example of a singular relationship is all the employees who are HIGH-ACHIEVERS, where FULL-TIME-EMPLOYEES is the member table, and the rows that belong to the relationship are those determined to be high achievers

Figure 3.14 illustrates a singular relationship of one member. The member is FAMILY MEMBER, and the relationship name is FAMILY. There is no owner. The instance of the relationship shows that the family members are shown in alphabetical order. From the information given in the figure, it is not possible to know whether the relationship binding mechanism is static or dynamic. If the relationship binding is static, then the rows were sorted by name before loading. It is also possible that the family named the children in alphabetical order in the order of their BIRTHDATES. If that were true, then an instance of the FAMILY



Set Type : Singular, single member

**Figure 3.14.** Relationship type: Singular set.



relationship from another family would most likely not show the children's names in alphabetical order.

If the relationship binding mechanism is dynamic, then the presentation order of the rows could be alphabetic by coincidence, or else there must be a sort clause in the program that retrieves the set of rows belonging to the specific family.

One of the important differences between static and dynamic relationship binding is sorting. If the relationship binding is static, then the sort rules can be contained in the data definition language (DDL). As rows are stored in the database, the relationships that thread through the rows are an implementation of these ordering rules. It is important to note that a DDL-based sorting clause between an owner and its members creates a logical ordering of the rows, not a physical ordering.

Figure 3.15 illustrates this point. The EMPLOYEE rows are stored in EMPLOYEE-ID order. This is depicted by the increasing EMPLOYEE-ID starting with Baker to Able to Richards to Clark. There is, however, a relationship between REGION and EMPLOYEE that is ordered on LAST-NAME. Figure 3.15 illustrates this alphabetic ordering, which begins with Able, and continues through to Richards.

If the relationship is dynamic, then the ordering among the rows is completely controlled on the selected rows at retrieval time.

#### **3.2.4.1.4 Singular-Multiple-Member Relationships**

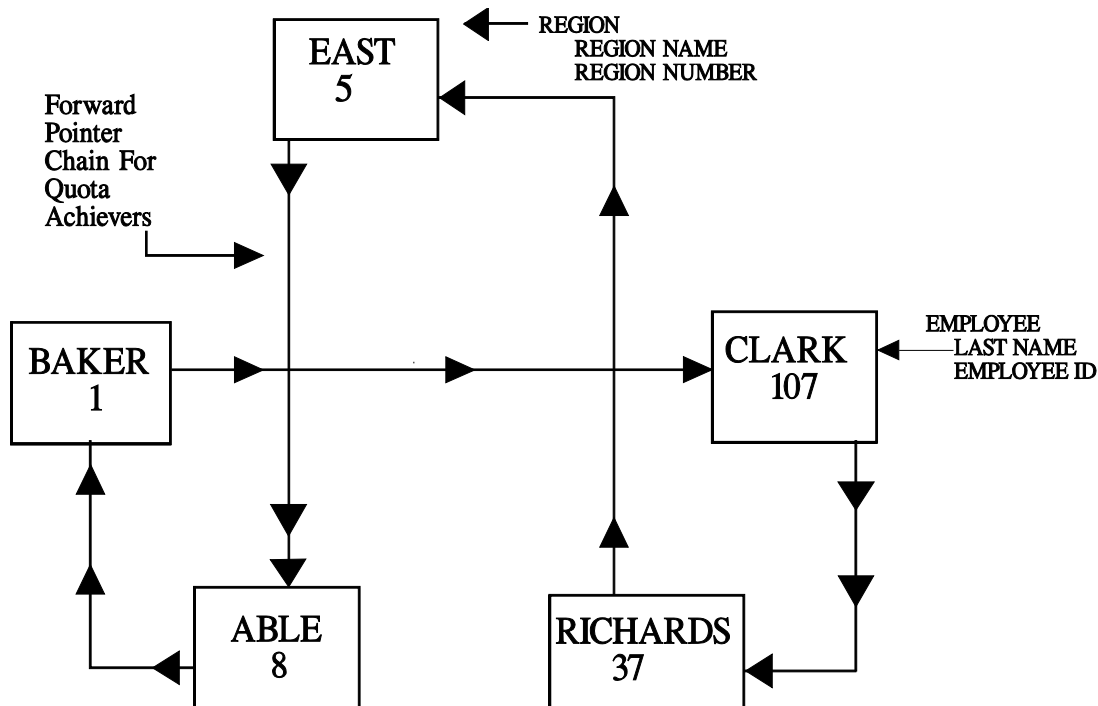
A singular-multiple-member relationship has no owner table, but more than one member table. There may be one or more member instances from each of the member tables for each relationship occurrence. A typical example for the relationship is HIGH-ACHIEVERS, but unlike a singular relationship, all the different types of employees can participate. That is, PART-TIME-EMPLOYEES, FULL-TIME-EMPLOYEES, and RETIRED-EMPLOYEES.

A singular-multiple-member relationship is essentially multiple lists of different table instances. There might be different types of customer tables, each having a different set of columns, but all participating in a single relationship BEST ACCOUNTS. The singular-multiple-member relationship BEST ACCOUNTS would be the list of relationship links from all the instances of the different tables according to the criteria that made them best accounts. The definition of a *best account* does not have to be value-based. That is, there does not have to be a value upon which the relationship is based, such as TOTAL-ORDERS. If singular-multiple-member relationships were required to be value-based, then the DBMS would not conform to ANSI/NDL.

#### **3.2.4.1.5 Recursive Relationships**

Recursive relationships are a special class of owner-member relationship in which the owner table and the member table are the same. A recursive relationship exists between one instance of a table and other instances from the same table. Over the years, recursive relationships have been known as nested relationships, looped relationships, and bill-of-materials relationships. When





## MEMBERS

Chain Represents Forward Alphabetical Order  
Stored in EMPLOYEE-ID ORDER

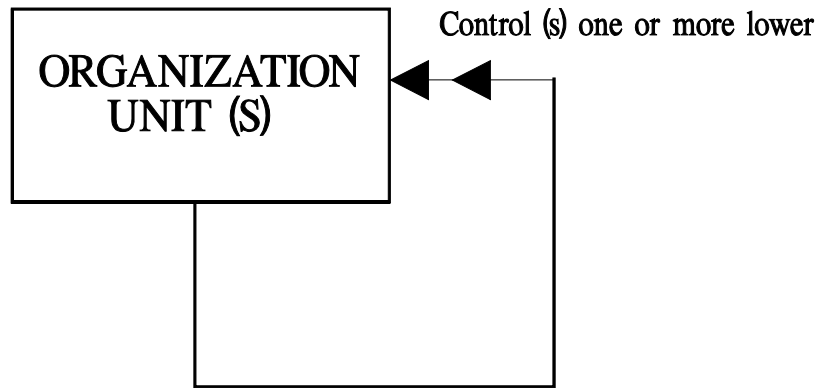
**Figure 3.15.** Physical relationship illustration.

rows of a given table are associated with other rows from the same table, the graphic symbol used to represent the relationship is a loop. Figure 3.16 presents the recursive relationship. Recursive relationships represent hierarchies of varying depths within one structure. For example, within a company's chart there can be different levels, depending on the suborganizations. Figure 3.17 illustrates that very situation. A company is divided into two divisions. Each division is divided into subordinate areas. San Francisco is subdivided, and within San Francisco, the Silicon Valley is subdivided.

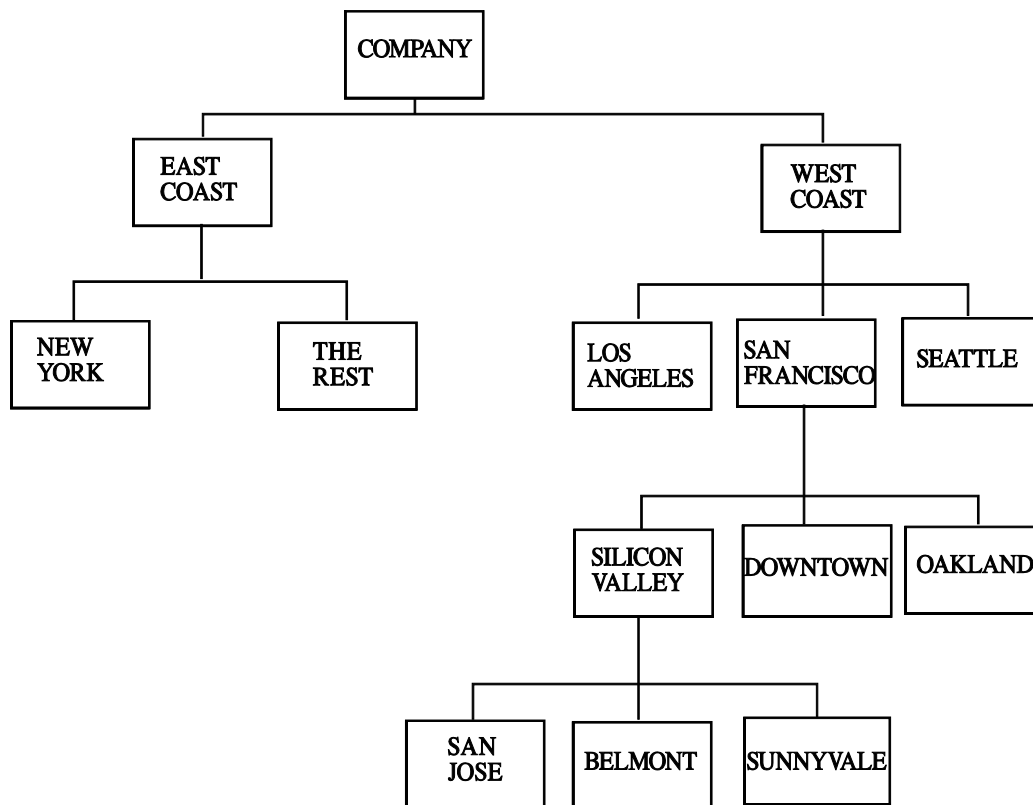
Prior to the *invention* of recursive relationships, a database might be hierarchically structured to represent the five levels. Each level would have to be assigned a name, indicative of its level. In the example we have been using, the levels might be as follows:

- Company
- Division
- Region
- District
- Territory





**Figure 3.16.** Recursive relationship



**Figure 3.17.** Hierarchical layout of company organization chart.

The four names fit just fine: West Coast division, San Francisco Region, Silicon Valley District, and Belmont territory. But what about other cases? Is New York a region, a district, or a territory? If it is a territory, are there then phantom regions and districts between the East Coast division and New York?



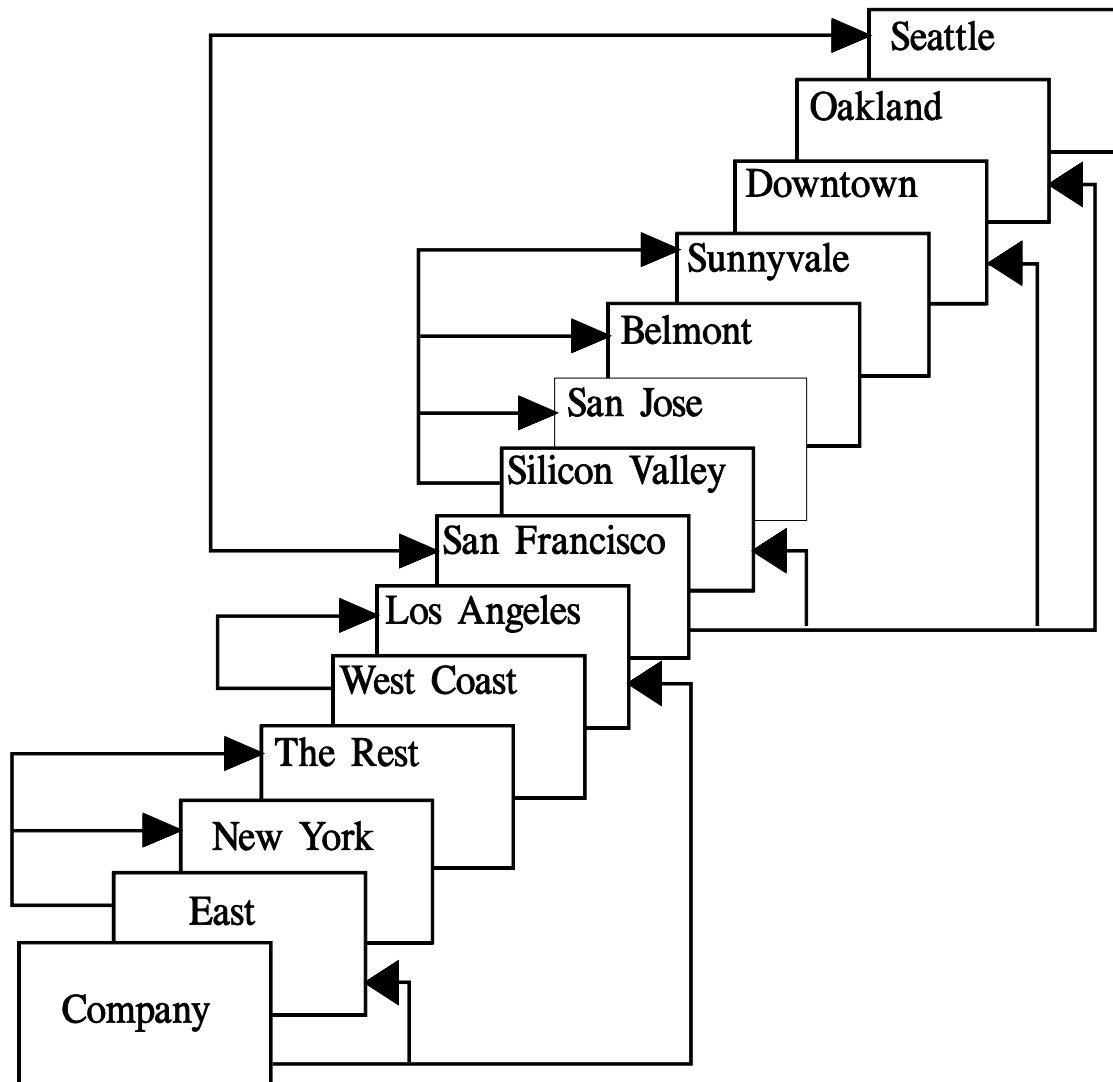
In addition to these kinds of problems, there is also the negative impact of having to add an additional level. For example, suppose there was a need to divide San Jose into three subordinate units. The database would have to be restructured into six levels to reflect the organizational change. Along with the database design changes there would also be programming changes for update and retrieval.

Recursive relationships are a way to avoid this difficult situation. A DBMS that supports recursive relationships allows a relationship to be defined with the owner row and the member row belonging to the same table. In the example, the owner row would be ORGANIZATIONAL UNIT, and the member row would also be ORGANIZATIONAL UNIT. Some DBMSs that support recursive relationships keep a *hidden* set of pointers, maintaining the order of the organization chart, all in one table. Figure 3.18 illustrates the net effect of the one table approach to recursive relationships. Any number of offices can be added at any level, and any number of additional levels can also be added, all without any database redesign or programming changes. The ANSI NDL fully defines the recursive relationship and the data manipulation operations necessary to process the row set.

Not all network DBMSs can explicitly define and process a recursive relationship. IDMS/R, for example, simulates the recursive relationship by defining and implementing two tables. One table is ORGANIZATIONAL UNIT and the other is SUBORDINATE ORGANIZATIONAL UNIT. Figure 3.19a shows these two tables and the relationship types between them. The rows that are represented by this two-level hierarchy are shown in Figure 3.19b. IDMS/R requires the decomposition of the recursive relationship into two different relationships: a one-to-many and a one-to-one. The one-to-many relationship is reflected in Figure 3.19a as *COMPANY is divided into EAST COAST and WEST COAST*. The one-to-one relationship exists between the SUBORDINATE ORGANIZATIONAL UNIT and ORGANIZATIONAL UNIT.

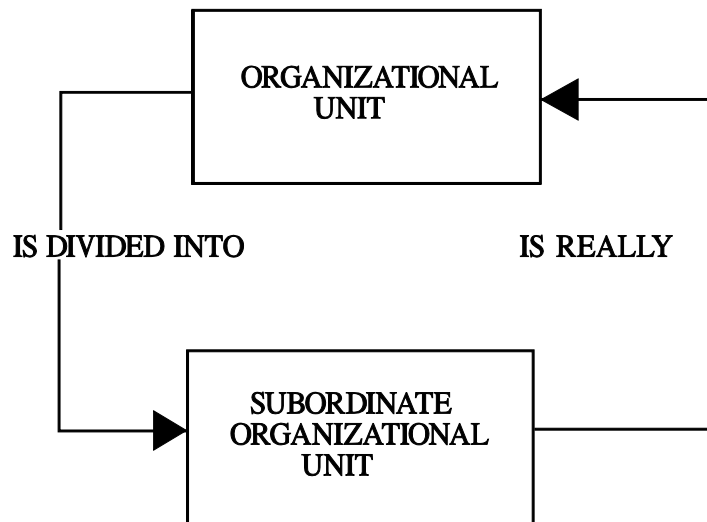
For example, WEST COAST (subordinate organizational unit) is related to WEST COAST (organizational unit).





**Figure 3.18.** Recursive layout of company organization chart.





**Figure 3.19a.** IDMS/R Data model for representing recursive relationships.

ORGANIZATIONAL UNIT	SUBORDINATE ORGANIZATIONAL UNIT
Company	East Coast, West Coast
East Coast	New York, the rest
West Coast	Los Angeles, San Francisco, Seattle
San Francisco	Silicon Valley, Downtown, Oakland
Silicon Valley	San Jose, Belmont, Sunnyvale

**Figure 3.19b.** Sets of rows loading an IDMS/R recursive relationship.



### 3.2.4.1.6 Many-to-Many Relationships

A common relationship type in a database is the many-to-many relationship. This relationship, like the one-to-many relationship, involves two different rows. The relationship, however, is one-to-many in BOTH directions. The first of the tables is defined as the owner of the second, and the second of the tables is defined as the owner of the first. A common example of a many-to-many relationship is:

- 1) each STUDENT is taught by many TEACHERS and
- 2) each TEACHER teaches many STUDENTs.

An instance of this many-to-many relationship appears in Figure 3.20.

STUDENT	COURSE		
	MATH 203	ENGLISH 701	BIOLOGY 107
Jones		Yes	Yes
Smith	Yes	Yes	Yes
Jackson	Yes		

**Figure 3.20.** Many to many relationships

Another example of a many-to-many relationship is registered owners of automobiles. Several persons can be the registered owners of one automobile, and several automobiles can be owned by one person. Figure 3.21 illustrates instances of this many-to-many relationship.

In an IRDS database application, the many-to-many relationship is very important. Figure 3.22a presents a many-to-many relationship. Instances of this many-to-many relationship are presented in Figure 3.22b and they show that the ZIP CODE column is contained in the EMPLOYEE and COMPANY tables, and the COMPANY table contains the CITY and ZIP CODE columns.



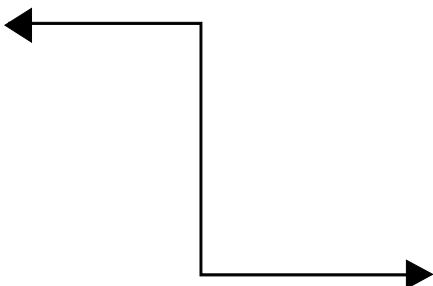


CAR OWNER FIELDS

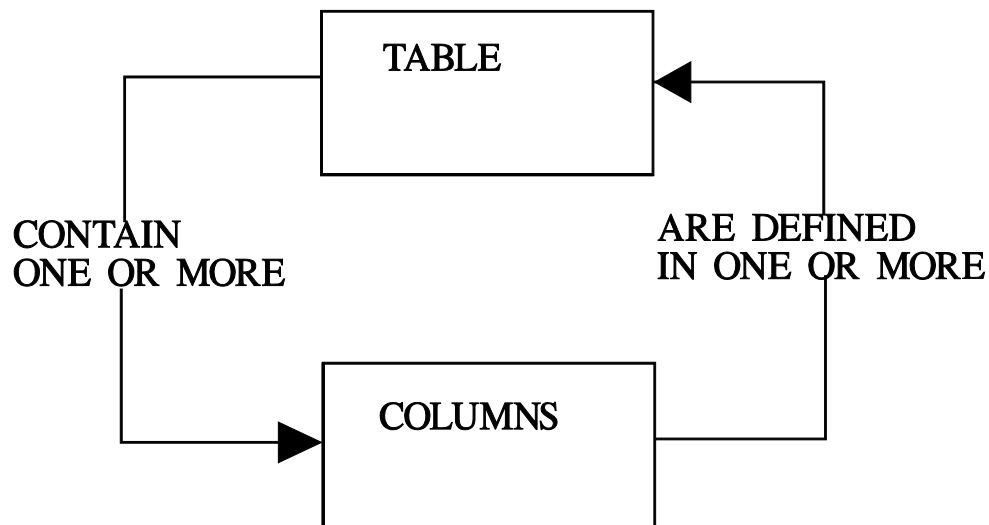
SOCIAL SECURITY NUMBER  
NAME  
STREET ADDRESS  
CITY  
COUNTY  
ZIP CODE  
AGE  
SEX  
DATE OF LICENSE

CAR FIELDS

MANUFACTURER  
MODEL  
TYPE  
ENGINE  
CHASSIS NUMBER  
TAG NUMBER  
YEAR  
COLOR  
OWNER SOC SEC NUMBER  
####1  
####2  
####3



**Figure 3.21.** Many to many relationships illustration.



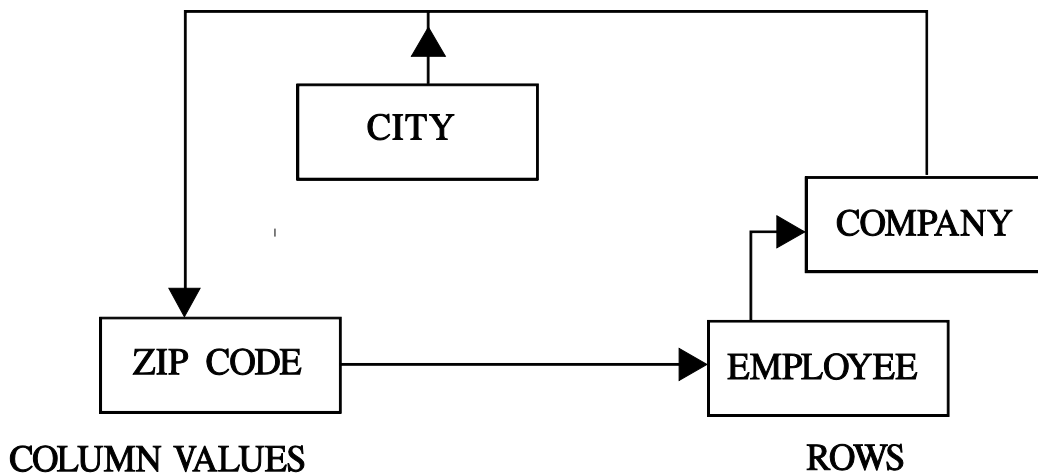
**Figure 3.22a.** Many to many relationship (direct)



Examples of many-to-many relationships in an IRDS include:

- Policy governs many objects, and an object reflects many policies.
- A column is represented in many tables, and a table represents the inclusion of many columns.

In the event that a DBMS cannot directly support a many-to-many relationship, the many-to-many relationship must be decomposed into two one-to-many subrelationships. To accomplish the decomposition, a third table is created to act as the *member* that both *owners* point to. Figure 3.23a illustrates the three table decomposition of the two table many-to-many relationship illustrated in Figure 3.22. Figure 3.23b presents the instances from that three table decomposition.



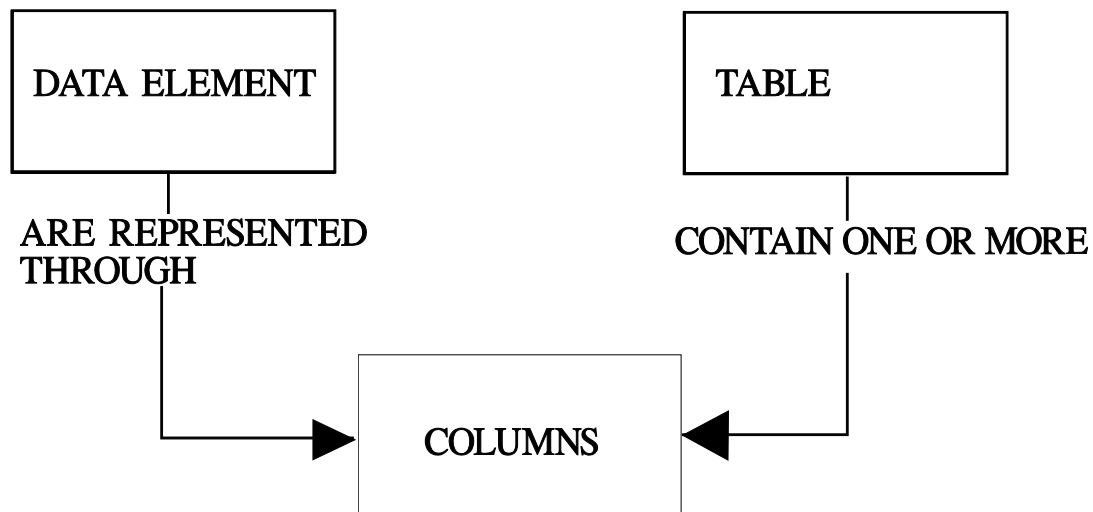
**Figure 3.22b.** Many to many relationship (direct) – instances.



### 3.2.4.1.7 One-to-One Relationships

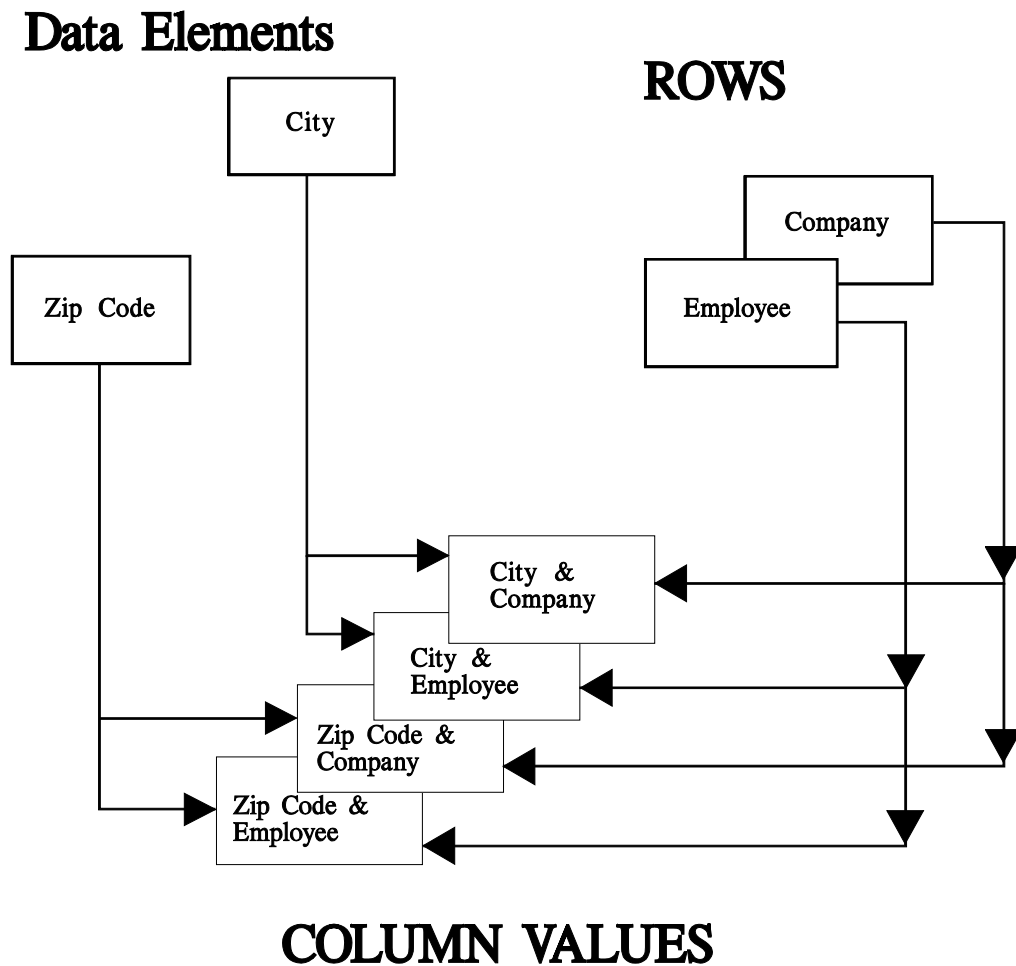
The one-to-one relationship is not employed often. It binds a single row from one table to a single row of another table. The most common use of this relationship type is to segment the columns from one table into two or more tables. Suppose a table had four hundred columns, and that 90% of the time only 10% of the columns were accessed, and that the other 90% of the columns were accessed only 10% of the time. If a one-to-one relationship type were used to segment the table into two tables, 360 columns could be in the little accessed table and the other, often accessed, 40 columns could be in the other table. Dramatic performance improvements would be achieved. While this is process analysis driving database design, such techniques are sometimes required to get a day's processing accomplished in just one day.

A one-to-one relationship specifies that there is one owner instance and one member instance without any program intervention for support. To say that the DBMS supports the relationship means that the DBMS, upon receiving a request to store a second member, would return an error message indicating that one member instance is already present. If the DBMS does not support one-to-one relationships, the application programmer is required to develop the logic to discover that there is already a member instance and to refuse to install an additional member instance.



**Figure 3.23a.** Many to many relationship (decomposed)





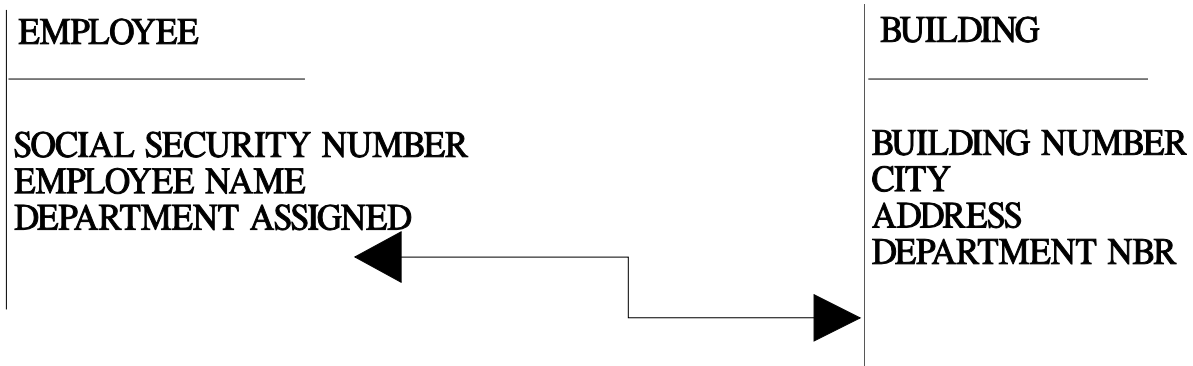
**Figure 3.23b.** Many to many relationship (decomposed) –instances.



### 3.2.4.1.8 Inferential Relationships

The inferential relationship relates rows from two tables. This relationship, however, does not involve the primary key of either table. A typical example is PRODUCT-CONTAINER. The basis of the relationship is the dimensions of the product and the dimensions of the container. The relationship states that the product is able to fit in the container, but this does not mean that it is actually stored there.

Figure 3.24 illustrates another example of an inferential relationship. In this example, an employee is assigned to a department. In the other table, a department is located in a building. The department may also be located in other buildings. The relationship between employee and building is thus inferential: it can be inferred that the employee is located in one of the buildings, but not specifically which building. If there are four buildings that house that department, then there is a 100% probability that the employee is in one of the four buildings, but only a 25% probability that the employee is in any randomly selected building.



**Figure 3.24.** Inferential relationship illustration.



### 3.2.4.1.9 Relationship Type Summary

Figure 3.12 illustrates a very important point. Even though it depicts only three different tables, REGION, SALESPERSON, and CUSTOMER, there are five different relationships representing four different types. To presume that a valid and substantial database application can be served with one or a few types of relationships is to be operating database from Alice's Wonderland.

### 3.2.4.2 Relationship Implementation Mechanisms

Relationships are bound into rows through either a static or dynamic relationship mechanism. A static relationship mechanism is one controlled by the DBMS as the user performs row additions or deletions. A dynamic relationship mechanism is one controlled by the end user.

In a DBMS with static relationship mechanisms, data manipulation operations both ADD and DELETE rows to and from the database and also modify relationship instances between the rows. For example, the DML ADD operation not only adds a row to a database, but also triggers a DBMS action that creates a relationship instance that relates the just loaded row to its owner(s), its member(s), or its sibling(s). Conversely, the DML DELETE operation not only deletes a row from the database but also triggers a DBMS action that adjusts the relationship mechanisms among the rows that previously pointed to the just deleted row. The row relationship mechanisms affected would be the deleted row's former owner(s), its former member(s), and its former sibling(s). Thus, in a DBMS with static relationship mechanisms, both the quantity of rows that participate in the relationship and the identities of the actual rows are known both before and after the ADD/DELETE operation. During a retrieval operation, these rows, already bound into relationships, are simply retrieved. Because the rows that participate in a relationship are already known prior to the execution of a retrieval statement, the relationship binding is said to be done statically. DBMS facilities that support static relationship mechanisms offer special relationship processing verbs like CONNECT and DISCONNECT that affect only the relationships that exist between rows. They do not affect the column values contained in the rows.

A dynamic relationship mechanism is always controlled by the end-user, as the relationship is manifest only through the value of a column within a row. As a user ADDs or DELETes a row, not only is the row added or deleted, the quantity of rows implicitly represented by the column's value (relationship) is also affected. If a CUSTOMER belongs to a specific TERRITORY, then both must be tables, and the TERRITORY-IDENTIFIER must appear in the CUSTOMER's row. The quantity of rows both before and after the ADD or DELETE can only be known by a SELECT clause (retrieval). Thus, the number of CUSTOMERs in a TERRITORY can only be known after the execution of a SELECT statement containing the condition . . . WHERE TERRITORY EQ <territory identifier>. Because the rows are known only upon the execution of a retrieval statement (SELECT), the relationship binding is said to be done dynamically. As a user performs a MODIFY that affects the value of the relationship bearing column, the relationship instance (materialized through a SELECT) in which the row participates also changes. That means that a change to the TERRITORY-IDENTIFIER value changes the territory of the customer.



In both static and dynamic relationship environments, end user actions can affect relationship instances among rows. With static relationships, relationship membership changes take place because end user table actions--such as ADD and DELETE--trigger additional DBMS operations such as CONNECT and DISCONNECT. A MODIFY table operation cannot affect relationships among rows as column values do not directly participate in relationship operations. With dynamic relationships, the end user actions such as ADD, DELETE, and MODIFY directly affect the relationships.

Relationships that are required to be value-based are always dynamic. That is because the value that is the basis of a row membership to a relationship instance is a contained column value.

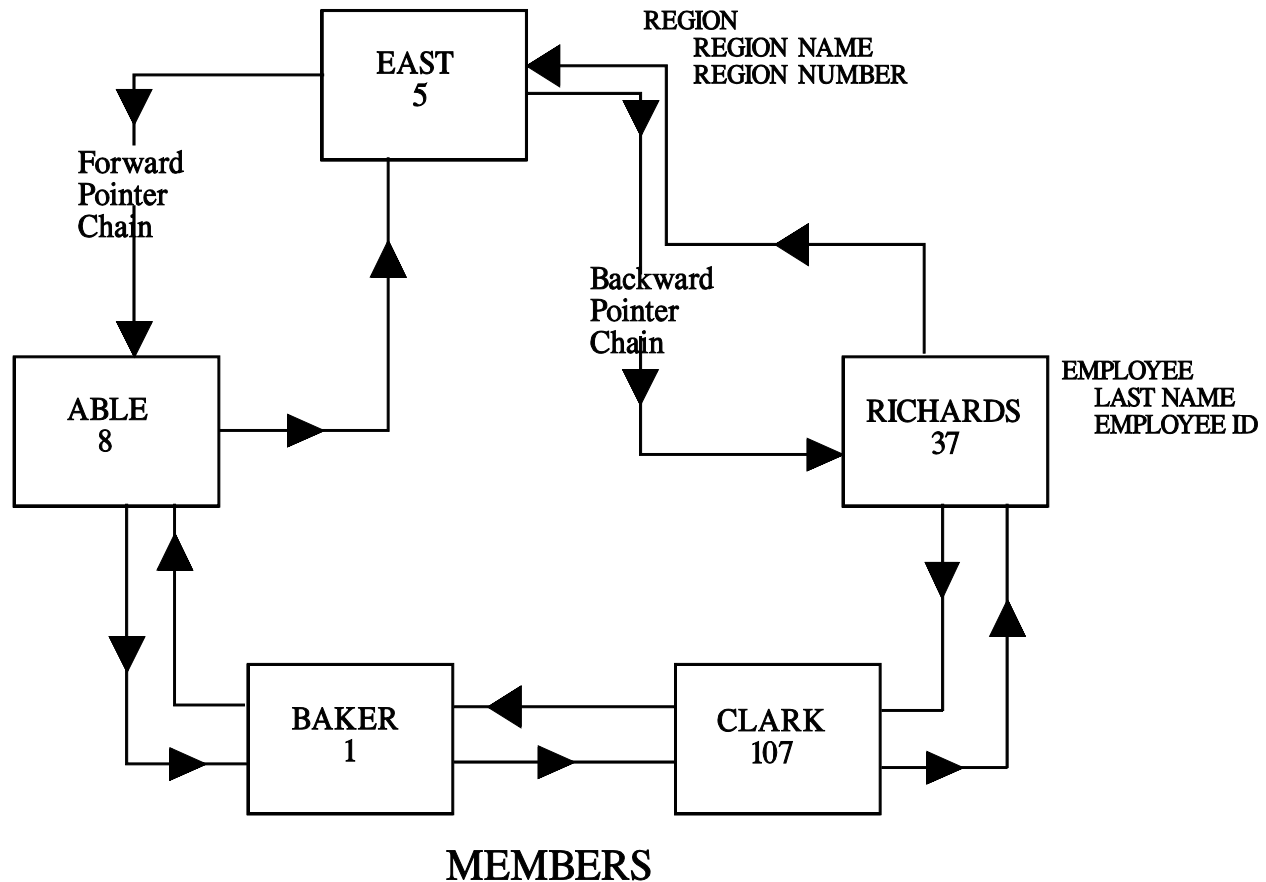
Arbitrary relationships can be either static or dynamic. If there is a column in the SALESMEN row called BEST SALESMEN and it contains the data value YES, then that arbitrary relationship is value-based (BEST=YES) and the relationship would be dynamic. If the arbitrary relationship is static, however, then the relationship identifier of the first *BEST SALESMAN* is stored in the regional manager's row, and the relationship identifier of the second *BEST SALESMAN* is stored in the first salesman's row, and so on. The most common form of the relationship identifier used in static relationships is the relative row address. Figure 3.25 identifies the critical differences between static and dynamic relationships.

It is important to understand that a relationship instance that binds rows together may or may not also reflect the row's physical order. For example, if EMPLOYEE rows are physically stored in the database in the order of birth, then a sequential access of the database also represents a chronological access. However, if EMPLOYEE row are stored in a physical sequence that is EMPLOYEE-ID, as illustrated in Figure 3.15, then it is very unlikely that a sequential access will produce rows in FULL-NAME alphabetical order. To accomplish the alphabetical order report, the rows must be copied from the database and sorted. Alternatively, there may be a separately maintained association among the rows that is used for row access. Such an association (ANSI/NDL SET) is illustrated in Figure 3.26

<b>DBMS TYPE</b> <b>(Inter-row Relationships)</b>	
<b>STATIC</b>	<b>DYNAMIC</b>
Defined Through DDL	Optionally Defined
Created Through DBMS	Created Through User Defined Field Values
Bound at Load/Update	Bound at Retrieval
Changed Through Delete and Reload	Changed Through Field Update

**Figure 3.25.** Inter-row relationships. Static and Dynamic DBMS Comparison.





**Figure 3.26.** Chair represents forward and backward alphabetical order. Logical relationship illustration.

This alphabetical association can be represented as relative row address pointers stored in the rows. The association is easiest to create when the DBMS initially stores the rows. Upon storage, the row's address and the alphabetical value of FULL-NAME on which the sort is to be based are stored by the DBMS in a temporary work file. As the last row is stored, the temporary work file is sorted by the DBMS by the FULL-NAME alphabetical key order. The stored rows can then be accessed by the DBMS in alphabetical order via the sorted set of row addresses. The pointer space within the row that is used for chaining to the *next* row is filled with the address of the *next* row and then the row is written back to the database. This DBMS process of row access and *next* row address storage continues until the DBMS accesses and stores the alphabetical sequence of pointers in all the rows. The final row, depending on the DBMS, usually contains a special flag to indicate *end-of-chain*, or points back to the owner row.

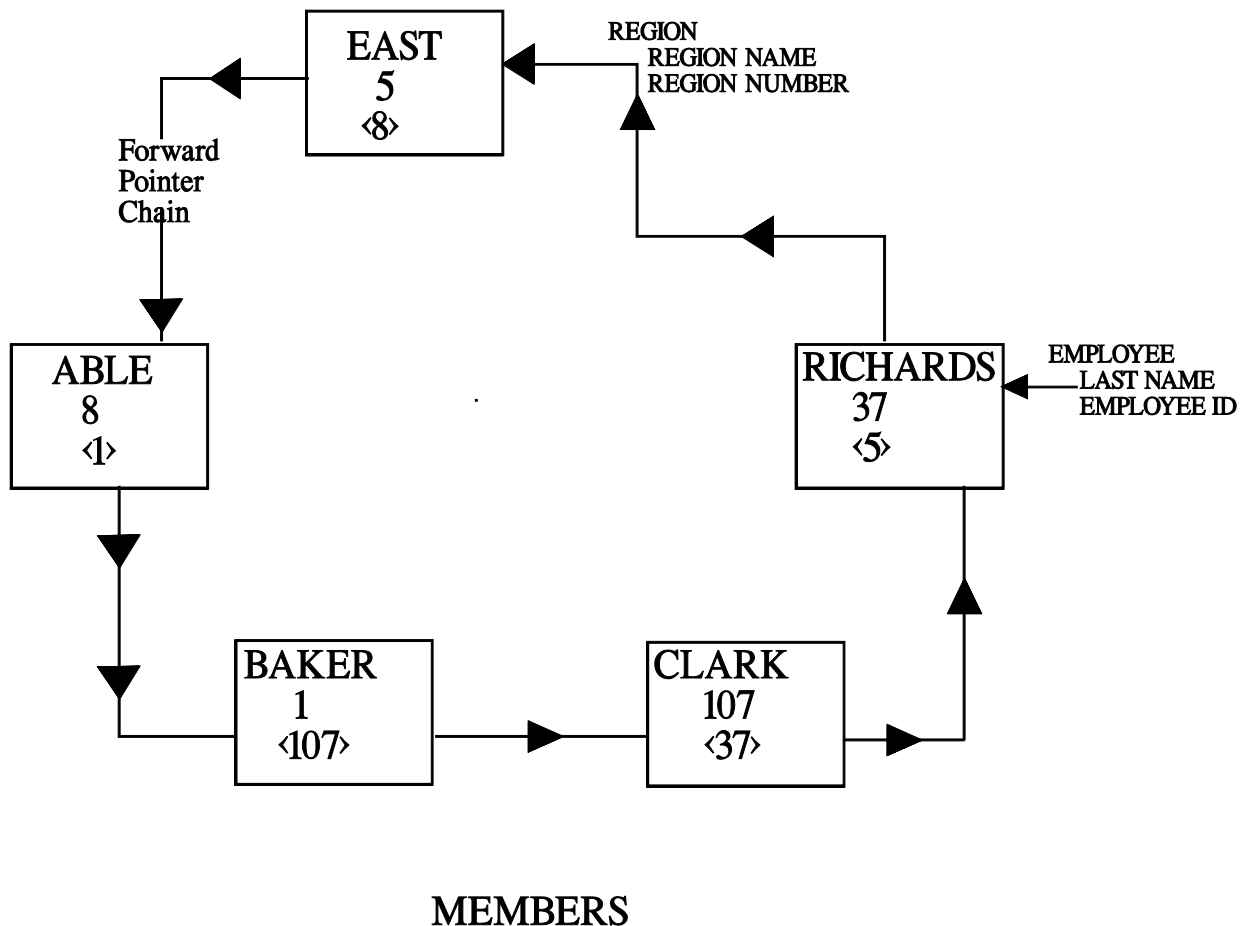
After initial loading, the alphabetical order can be maintained by first storing the newly entered row, then accessing the existing rows in alphabetical chain order until the row is found that represents the immediately preceding alphabetical row. Then, the address of the newly stored row replaces the address of the existing *next* pointer in the alphabetically preceding row. Finally, the *old* next pointer is stored in the new row.





Traditionally, these pointers are stored in the rows. However, they might be stored in structures separated from the rows, commonly called pointer arrays. Regardless of how these pointers are stored, relationships are association mechanisms among rows that may have little or nothing to do with the actual storage addresses of the rows. This is certainly true if the *pointers* that point from one row to another in the relationship chain are primary-key values such as EMPLOYEE-ID. If all the EMPLOYEE-IDs are stored in an array, having been first sorted by the alphabetical key, then when an EMPLOYEE-ID is accessed, it is used to locate and access the row.

The advantage of having data values as relationship mechanisms instead of row addresses is that rows can be relocated without having to modify relationship pointer addresses. The disadvantage is that, since the data value does not point directly to the location of the row, extra database accesses are needed to locate and retrieve the row. Figure 3.27 illustrates the use of primary keys as *next-pointers* in a relationship chain. The primary key value is the REGION-NUMBER column. The use of the EMPLOYEE-NUMBER as a next-pointer is represented in the rows as the value contained in angle brackets. The owner row, REGION = East, contains the member-pointer value of <8>, which is the primary key value of the first member row



**Figure 3.27.** Chain represents forward alphabetical order. Logical relationship illustration.



(EMPLOYEE = Able). The first member contains the next-pointer value of <1>, which is the primary key value of the next member row (EMPLOYEE = Baker).

It is possible, however, for the mechanism of relationship to be a data value and the relationship still not be dynamic. In the case described above, the relationship is clearly static, for three reasons:

- The EMPLOYEE-ID value is intended solely to represent the alphabetical association of one row to the next.
- When a row is removed, the alphabetical association among the rows breaks and an alphabetical scan is no longer possible: the row whose *next* pointer was the primary key of the removed row now points to nothing, and the row whose *prior* pointer was the primary key of the removed row also points to nothing.
- When a new row is added, a dynamic relationship DBMS has no mechanism to add the row in the proper location within the alphabetically sorted list of rows.

If the DBMS possesses the mechanisms to add, delete, and repair relationship links (whether pointer or data value), then the DBMS is static. Without these mechanisms, the DBMS is dynamic and must possess syntactic constructions within the interrogation languages to command the DBMS to discover which rows participate in relationships. A static relationship DBMS allows for the establishment of named relationships among two or more tables. Additionally, a static relationship DBMS allows multiple table relationships to connect one owner table with one or more multiple-member tables.

As stated many times above, there are two methods by which a DBMS knows of a relationship: static and dynamic. A static relationship is created by the DBMS and is normally stored in the *pointed-from* row. A dynamic relationship is created by the user and is exercised by the DBMS through shared data value, one in the owner, and one in each of the *pointed-to* members.

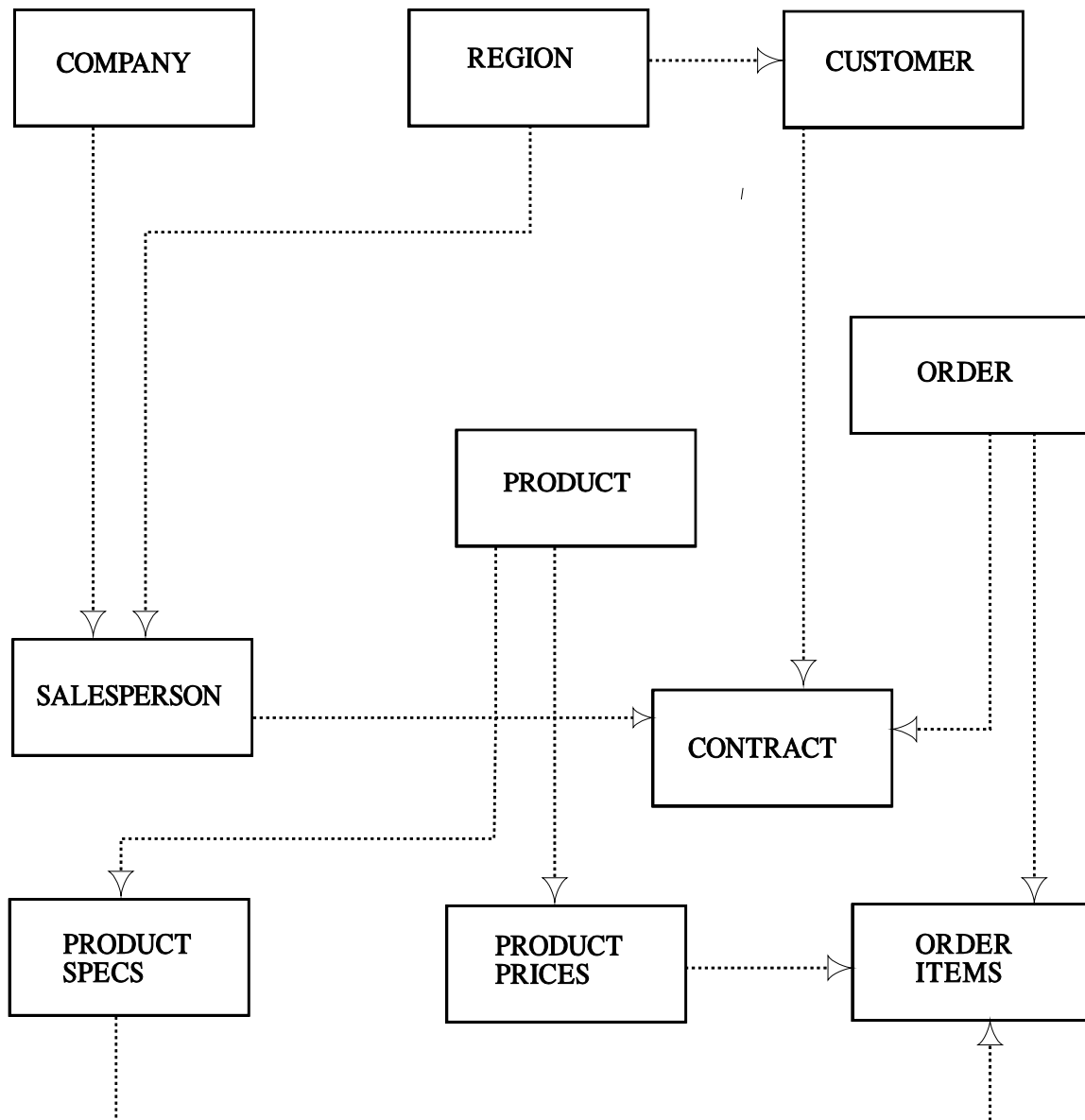
#### 3.2.4.2.1 Dynamic Relationships

The value-based relationship exists between two tables when the column upon which the relationship is based contains values acceptable to the rules stated in the relationship.

For example, in Figure 3.28, the ORDER table with its column ORDER CONTRACT ID is related to the CONTRACT table with its column CONTRACT ID. A relationship exists between instances of these two tables when a CONTRACT table row's CONTRACT ID column contains the same value as the ORDER table row's ORDER CONTRACT ID column. In this example, the relationship is one-to-many.

In a database system, if two rows are related to each other with two one-to-many relationships, one in either direction, and if the semantics of both relationships are the same, then the relationship between the two tables has common semantics, and is termed a many-to-many relationship. Figure 3.29 shows a many-to-many relationship.

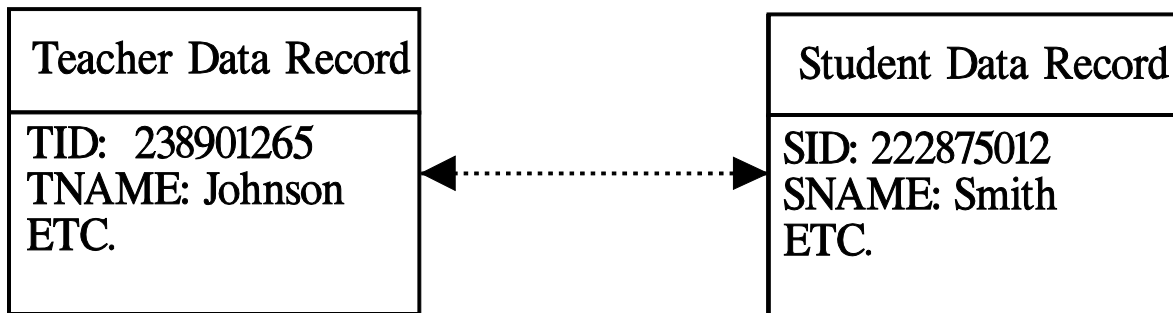




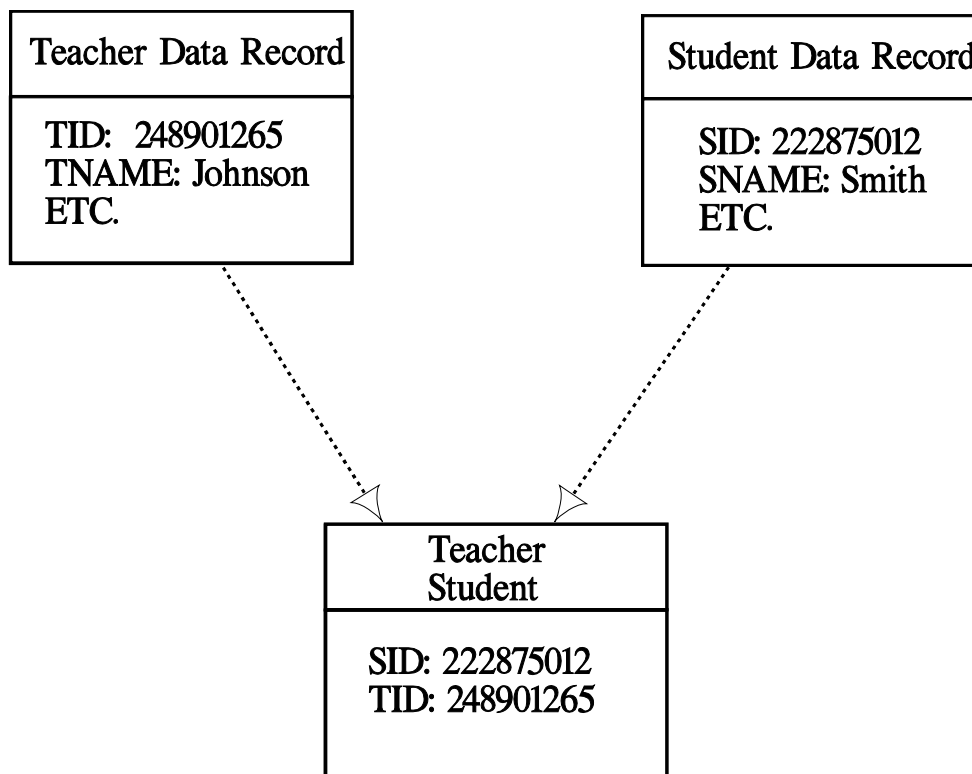
**Figure 3.28.** Relational data model: Ten simple tables

In a dynamic relationship DBMS, there are two ways to represent many-to-many relationships. Figure 3.30 illustrates the triple table representation. This technique is appropriate for ANSI/SQL and relational data model DBMSs. Figure 3.31 illustrates the double table representation, which is typically accomplished by independent logical file data model DBMSs. Of these two representations of a many-to-many relationship, only the double table representation is explicit. The triple table representation is, at best, a simulation of the many-to-many relationship type.



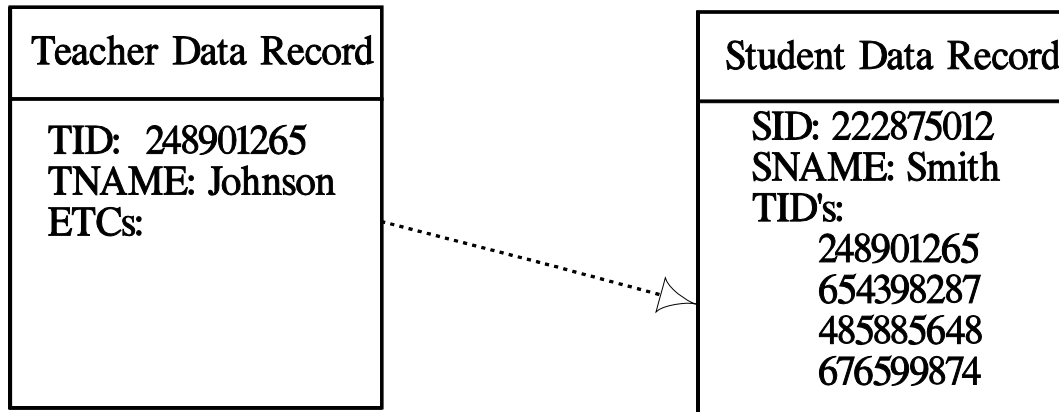


**Figure 3.29.** Teaches Many to many relationship between student and teacher.



**Figure 3.30.** A many to many relationship: Taught By between student and teacher. Tree table approach.





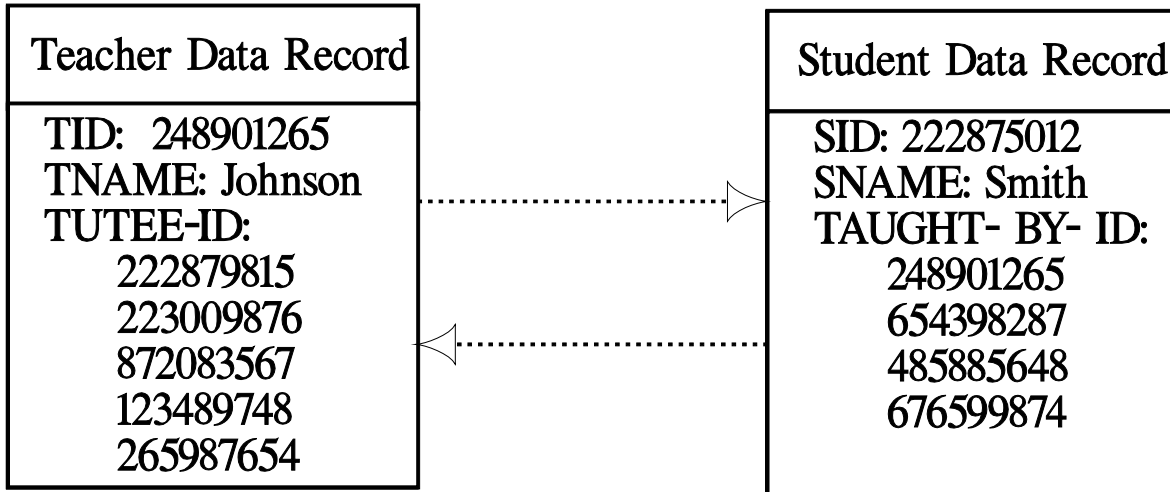
**Figure 3.31.** A many to many relationship: Taught By between student and teacher. A two table approach.

If the two rows are related to each other in two many-to-many relationships, one in either direction, and if the semantics of the relationship are the same, then the relationship between the two rows is really only one relationship, and only one relationship should be created and represented. If, however, the semantics of the two relationships are different, then the two relationships must be separately named, defined, created, and maintained. Figure 3.32 represents two many-to-many relationships between STUDENT and TEACHER, one for TAUGHT BY and the other for TUTORS.

Figure 3.27 shows a static, ordered relationship between REGION and EMPLOYEE. To accomplish that with dynamic relationships requires two steps. First there must be the REGION-NUMBER column in the EMPLOYEE table to store the value EAST. Second, the user must extract the appropriate rows (SELECT . . . WHERE REGION EQ EAST), and then sort the row keys into the right order (BY LAST-NAME). In short, what is accomplished in a single step with static relationships requires two steps with dynamic relationships.

Most dynamic relationship DBMSs do not allow for the formal definition of relationships through DDL. Rather, the burden of knowing which relationships are defined and the burden of knowing how to select rows according to a relationship implied through commonly named columns is left to the user.





**Figure 3.32.** Two many to many relationships: Taught By between student to teacher, and Tutors between teacher and student.

#### 3.2.4.2.2 Static Relationships

Static relationships are sometimes called information-bearing relationships because the relationships themselves convey information that normally cannot be verified through data values. For example, when the first ORDER row is retrieved for a given CONTRACT, it might be inferred that the first ORDER row represents the first ORDER placed under the contract, the second row represents the second ORDER, and so forth. The sequence of the rows corresponds to the date when the ORDER was placed.

The static relationship mechanism is normally controlled by the software system that inserts and retrieves rows. The form of a relationship instance is normally a row's relative row's address within the O/S file. This address is stored in the row's owner or prior member's row. This location is commonly referred to as a row's *relative row address*, which is the starting address of the row relative to the start of the O/S file.

Figure 3.33, shows two sets of data, one for relationship array instances (left) and another for rows (right). For the purposes of this example, all the relationship array instances belong to one O/S file and the rows belong to another O/S file. Within the row file, the EMPLOYEE rows are each 1000 bytes long, and the starting address of each row is the byte address of the start of the row. Thus, in the example, the relative row address of the EMPLOYEE row is 5000. The relationship arrays are similarly constructed, each being 10 bytes long. The second array is shown, starting at address 10. It ultimately points to its corresponding row, EMPLOYEE, which has the data file relative row address of 5000. The technique and benefits of separating relationships from the actual rows are presented in Chapter 4.

As another example of relative row addresses, if a table's length is 100 bytes, and if each row is stored in a DBMS physical row that is 1000 bytes, there are 10 rows per physical row. If the relative row address of a row is 37, the row is the seventh row of the fourth DBMS row. The



## RELATIONSHIP POINTERS

## DATA RECORDS

10						
RTI	NEXT	MEMBER	OWNER	ADDRESS	5000	
EMPLOYEE	???	20	???	5000	EMPLOYEE	

20						
RTI	NEXT	MEMBER	OWNER	ADDRESS	7500	
Course	30	???	10	7500	Course	

30						
RTI	NEXT	MEMBER	OWNER	ADDRESS	8000	
Course	40	???	10	8000	Course	

40						
RTI	NEXT	MEMBER	OWNER	ADDRESS	9700	
Degree	???	???	10	9700	Degree	

RTI means record type identifier

**Figure 3.33.** Examples of relative record addresses.

relative row addresses of a row are created and stored by the DBMS during data loading or data update.

If a mistake is made regarding the order of the database rows for ORDERS, the pointers that logically represent the order of the two rows must be physically reversed. For example, if the third ORDER is really the second, the second ORDER must be retrieved from the database into a computer program, its database representation deleted through a DBMS delete row



command, and then the computer program's representation of the row stored back into the database *after* the next ORDER.

The control that must be exercised by the user over static relationship facilities through commands such as CONNECT, DISCONNECT, and sort orders is in marked contrast to the user's casual exercise of control over the dynamic relationship facilities through simple column value updates. To change the dynamic relationship merely requires a change to the column's value that is used as a basis of a relationship. For example, changing the current value of a CONTRACT ID in an ORDER row to another value automatically changes the ORDER instance's CONTRACT.

Static relationship DBMS relationships are fast to traverse, but slow to update. There are three types of updates: add, delete, and modify. To add a relationship instance to a row that is already stored, but not yet a member of a relationship, requires three steps:

- The user must obtain the row that is to be a member.
- The user must invoke DML verbs to locate the proper position within the relationship chain, usually a GET NEXT command.
- The user must then invoke a DML verb like CONNECT that modifies the relationship references in the row's prior and next rows to reflect the addition of the row's relationship reference.

The relationship deletion operation is the inverse of the add operation. The modify operation is the combination of the delete operation and the add operation.

To review briefly, Figures 3.15 and 3.26 illustrate the collection of static relationships that bind rows. This collection forms a chain. The head of the chain--that is, the start of the linked relationship--is called the owner. In the owner row there is a pointer that refers to the first member. In the first member there is a pointer that refers to the next member, and so forth. If the last member contains a pointer that refers back to the owner, the chain forms a ring. Some relationships also have prior and owner pointers.

The DBMS's relationship specification clauses state whether new rows are stored automatically at the front of the chain, at the end of the chain, or in sorted order. When the DBMS automatically sorts the relationships that represent the rows, the DDL sort clause contains one or more columns that serve as sort keys. The sort clauses clearly state how duplicate values for the combined sort keys are handled, that is, whether they are stored at the head or the tail of the chain, or rejected altogether.

Because a database built through a static relationship DBMS has a large collection of relationship and integrity clauses, the data is highly organized. All the rows are stored in a very specific order and can be retrieved only in that order. If there is to be any other order, the rows must be removed, placed in an external file, and sorted.

When the application's processing logic is the same as the organization of the rows in the database, the application runs very efficiently. When the application's processing logic is significantly different from the row organizations in the database design, either the application cannot be carried out, or it performs very slowly, or the database has to be completely





reorganized and reloaded before the application operates efficiently. When another new set of reports comes along, this database reorganization process must begin again.

### 3.2.4.3 Relationship Integrity

Relationship integrity, commonly called referential integrity, is a shorthand name for a defined set of rules and actions that apply to identified relationships. The rules relate to the type of relationships that must be maintained, and the actions are the identified activities that must take place to enforce a rule or respond to a change in the database in order to maintain the database's referential integrity.

Referential integrity rules govern relationships that involve both an owner and member table, for example, one-to-many or one-to-one. Referential integrity actions concern two basic row operations ADD (insertion) and DELETE (retention). Referential integrity is affected by the type of relationship implementation mechanism that governs the ADD (insertion) and DELETE (retention) operations, in other words, by whether the DBMS uses static or dynamic facilities.

Specific actions, for example, reject, are taken when referential integrity conditions are not met. For example, there can be a referential integrity rule that rejects a DELETE row command whenever that specific row has other rows related to it as members. If this rule applied to CONTRACT and its corresponding ORDERs, the command DELETE CONTRACT WHERE CONTRACT ID EQ 1234 would be rejected if there were any orders related to that contract. If the action is the inverse, that is, to store a new ORDER row, then there must be a corresponding parent CONTRACT row or else the *child* row is not allowed to be stored.

Another referential integrity action, NULL, means that when a *parent* row deletion occurs all the member rows are specially modified to indicate a *no owner* condition. If a CONTRACT row is deleted, all its corresponding ORDERs belong to the NULL contract set (static), or the CONTRACT ID column in ORDER is set to NULL (dynamic). If the action is the inverse, that is, to add a new ORDER row, if the corresponding CONTRACT row is not already present, then the ORDER row belongs to the NULL owner set (static), or the CONTRACT ID column in ORDER is set to the NULL value (dynamic).

A final referential action is CASCADE DELETE. It is the same as DELETE except that all the row members of a deleted member are also deleted. The CASCADE DELETE operation is most appropriate for hierarchies. For example, if the ORDER row had ORDER-LINE-ITEMs member rows, it makes no sense to keep these when the ORDER itself is deleted.

#### 3.2.4.3.1 Static Insertion Referential Integrity

Insertion referential integrity affects only the addition of new *member* rows. Generally, a member row is allowed to be inserted (added) only if there is an appropriate *owner* row already stored.

For static relationships, the *owner* row is defined through the data definition language's relationship clause. Static referential integrity can be either value-based or arbitrary. In ANSI/NDL the insertion options are AUTOMATIC and MANUAL.



If the NDL relationship integrity is AUTOMATIC, then the row can be inserted in the database only if it can also belong to at least one relationship instance. If the NDL relationship integrity is MANUAL, then the member row is stored in the database without connecting it to a relationship instance.

As an example, if a company wants to pay only valid invoices, it would reject an invoice that did not relate to an already approved vendor and was not drawn against an approved open purchase order. APPROVED VENDOR and the PURCHASE ORDER are owners of the INVOICE, and the insertion options attached to the relationships between these two tables and the INVOICE table are identified as AUTOMATIC. If both owner rows are not present, the insertion is rejected.

If the NDL insertion option is MANUAL, however, the INVOICE is accepted into the database without having to pass either of those two DBMS imposed tests. But, the INVOICE is not a part of any relationship instance between VENDOR and INVOICE, nor between PURCHASE ORDER and INVOICE.

#### 3.2.4.3.2 Dynamic Insertion Referential Integrity

Dynamic insertion referential integrity also affects only the addition of new *member* rows. Again, a member row is allowed to be inserted (added) only if there is an appropriate *owner* row already stored.

For dynamic relationships, the DBMS attempts to select the appropriate owner rows (VENDOR and PURCHASE ORDER) and, if they are present, accepts the invoice for payment. The selection is through the use of the VENDOR ID and the PURCHASE ORDER ID that must be supplied with the invoice row. If the owner instances are not present, then the test fails.

In this example, the basis of dynamic referential integrity is the VENDOR ID and the PURCHASE ORDER ID on the invoice. These are the primary keys of the VENDOR and PURCHASE ORDER tables.

Dynamic insertion referential integrity has a capability analogous to the static manual insertion option. It is accomplished by the DBMS's referential integrity rule stating that either the PURCHASE ORDER ID or the VENDOR ID columns can contain the NULL value in the INVOICE row. In such a case, the validity of the invoice can not be proven as there are no appropriate owner rows present.

#### 3.2.4.3.3 Static Retention Referential Integrity

Row retention referential integrity means that rows are retained in the database only under certain conditions, or until certain conditions are no longer fulfilled.

For an ANSI/NDL static relationship DBMS specification, there are three variations to retention: FIXED, MANDATORY, and OPTIONAL.

A row that is governed by the FIXED retention option remains a member of the specific relationship instance to which it is initially connected until it is deleted from the database. For example, if a company has a database that stores invoices only until they are paid (open), then its



retention is FIXED. Once an application program *cuts a check*, it would delete the invoice row, which in turn, would cause the DBMS to delete the row from the open invoice class.

If the company modified the system so that invoices remained in the database after they were paid, then one choice for the invoice row's retention option would be MANDATORY. The invoice remains part of the database, so long as it belongs to some other relationship instance, for example, paid.

The company might further modify the bill paying system to allow the retention option of OPTIONAL. This allows the invoice to remain part of the database even though it belongs to neither of the two approved categories: open or paid.

The retention option dramatically affects delete statements. For example, if a deletion occurs to an owner row in which there are members connected by a FIXED retention option, then all the members are also deleted. If the retention option is OPTIONAL, then the rows become disconnected, but still remain even if they do not belong to another relationship. Finally, if the retention option is MANDATORY, then the rows remain only if they are part of another relationship. For example, if the invoices are from an approved vendor, they are probably related to the vendor through a FIXED option. These same invoices probably belong to the INVOICE STATUS relationship with a MANDATORY option, or possibly an OPTIONAL option.

#### **3.2.4.3.4      Dynamic Retention Referential Integrity**

Dynamic retention referential integrity is analogous to the corresponding static facility. If the CASCADE option is specified, then the row selected from the owner table and all the matching rows of the member are deleted. If the option is SET NULL, then the foreign key value in the member row is set to NULL. If the option is SET DEFAULT, then the value is set to whatever value was established as the default.

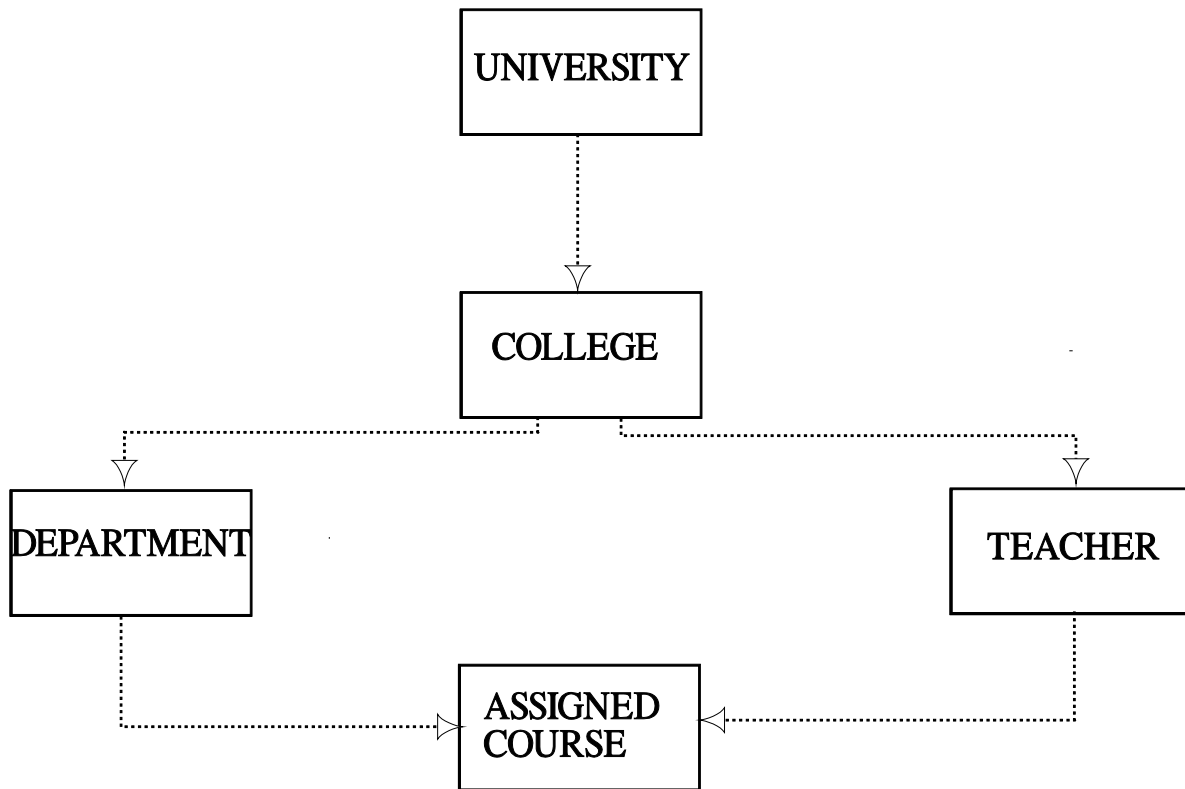
#### **3.2.4.3.5      Referential Integrity Surprises**

Referential integrity initially is a simple concept. When fully specified in a production database, however, ambiguities and anomalies arise that produce surprising results. For the ANSI/SQL, referential integrity was not included in the first version of the standard (October, 1986). Since that time, H2 worked diligently to develop a special referential integrity addendum to the SQL standard, which combined with SQL 1986, is now called SQL 1989. During the development of the referential integrity addendum, H2 found that about one-third of nine different meetings (18 months) was spent reviewing members' papers dealing with referential integrity anomalies. At each instance of an anomaly, additional rules were placed in the addendum. One member remarked that the rules section on referential integrity was likely to be larger than the original standard itself!

One H2 member proposed that to determine inconsistent referential integrity, the vendor's SQL schema processor digest all the referential integrity statements and determine if there are ambiguous paths. A path is a candidate for ambiguity if, for example, there is more than one way to get from one row to another. The path is ambiguous if different results are produced when the



referential actions occur along the different paths. In this situation, the software that checks for referential integrity consistency must not accept definitions of referential integrity actions that



**Figure 3.34a.** Ambiguous referential integrity access path among tables.

result in ambiguity.

Figure 3.34a and 3.34b illustrates the types of problems that can arise with referential integrity. In these Figures, there are five tables: UNIVERSITY, COLLEGE, DEPARTMENT, TEACHER, COURSE. There is also an appropriate set of rows for these tables. Because the tables are all connected, it is possible to get different results when an update occurs on the COURSE table, based on column selection criteria in COLLEGE table.

For example, `DELETE COURSE WHERE COLLEGE EQ 'ARTS & SCIENCES'`. If the DEPARTMENT side of the diagram is traversed, the COURSEs deleted are those connected to the DEPARTMENTS, which in turn are connected to the selected COLLEGES. If however, the TEACHER side of the diagram is traversed, the COURSEs deleted are those taught by TEACHERs of the COLLEGES selected. As Figure 3.34b illustrates, the rows deleted depend on which side of the diagram is traversed. Hence, the ambiguity.



### 3.2.5 Operations

There are three types of database operations: row, relationship, and combination. The row retrieval operations are: FIND, GET, and PROJECT. The row update operations are ADD, DELETE, and MODIFY. The eleven relationship operations are: CONNECT, DISCONNECT, JOIN, DIVIDE, GET OWNER, GET MEMBER, GET NEXT, INTERSECTION, DIFFERENCE, PRODUCT, and UNION. Some of the operations are actually a combination of several operations, for example, FETCH, MODIFY, and INSERT.

To understand all these operations, a set of relational tables and rows are provided. They appear in Figure 3.35.

#### 3.2.5.1 Row Operations

Fundamentally, row operations are independent of data model in that rows are either stored or deleted. The exact behavior of the operations, however, depends on whether there are static or dynamic relationships involved in the database structure. If there are no relationships involved, then the operation's behavior is the same for static and dynamic. If there are static relationships then it is necessary for the DBMS to connect the row to one or more relationships, or to delete the row from one or more relationships. In such a case, the insertion option would be AUTOMATIC.

If the DBMS that provides static relationship binding is sophisticated, then the DBMS optionally defers the immediate processing of relationship changes to a later time when it may be more cost effective. In such a case, the insertion option has to be MANUAL.

If the DBMS's relationships are dynamic, then no relationship processing occurs whenever a row is stored or deleted, except an action that may be imposed by referential integrity.

Figure 3.36 contrasts the actions resulting from the different row operations for both static and dynamic relationship bindings on rows. Figure 3.37 and 3.38 illustrate the PROJECT operation that has the DISTINCT operator in effect. Figures 3.39 and 3.40 show a simple SELECT operation.



Relation : Team

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>
T1	Black Birds	NE	Bowie
T2	Capitols	SE	Laurel
T3	Metros	NW	Laurel
T3	Bears	NE	Bowie
T5	Maulers	NW	Croom

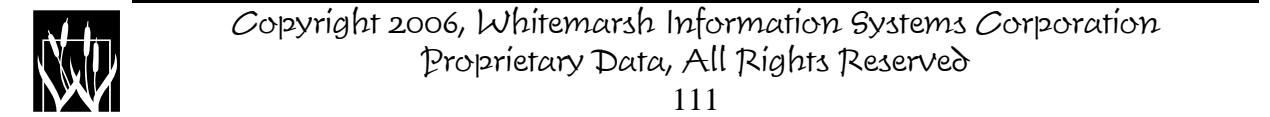
Relation : Player

<u>P#</u>	<u>PNAME</u>	<u>COLLEGE</u>	<u>WEIGHT</u>	<u>CITY</u>
P1	Jackson	Colby	180	Bowie
P2	Beaner	Hamline	165	Laurel
P3	Tufts	St. Mikes	165	Savage
P4	Tufts	Colby	205	Bowie
P5	Larson	St. Mikes	180	Laurel
P6	Bowes	Colby	190	Bowie

Relation : Team - Player

<u>T#</u>	<u>P#</u>	<u>GOALS</u>
T1	P1	50
T1	P2	37
T1	P3	40
T1	P4	37
T1	P5	22
T1	P6	22
T2	P1	50
T2	P2	40
T3	P2	37
T4	P2	37
T4	P4	50
T4	P5	40

**Figure 3.35.** Three relational tables: player, team and team-player.



SQL Syntax :

```
Select Distinct COLLEGE, CITY
From   Player
```

GIVEN: Relation = PLAYER

Relation : PLAYER

<u>P#</u>	<u>PNAME</u>	<u>COLLEGE</u>	<u>WEIGHT</u>	<u>CITY</u>
P1	Jackson	Colby	180	Bowie
P2	Beaner	Hamline	165	Laurel
P3	Tufts	St. Mikes	165	Savage
P4	Tufts	Colby	205	Bowie
P5	Larson	St. Mikes	180	Laurel
P6	Bowes	Colby	190	Bowie

RESULT:

<u>COLLEGE</u>	<u>CITY</u>
Colby	Bowie
Hamline	Laurel
St. Mikes	Savage
St. Mikes	Laurel

**Figure 3.38.** Relational Operation: Project example.





Table Foo				
Col 1	Col 2	Col 3	Col 4	Col 5
a1	ww	xx	yy	zz
a2	ww	xx	yy	zz
a3	ww	xx	yy	zz
a4	ww	xx	yy	zz
a5	ww	xx	yy	zz
a6	ww	xx	yy	zz
a7	ww	xx	yy	zz
a8	ww	xx	yy	zz
a9	ww	xx	yy	zz

Figure 3.39. Relational Operation: Select.

SQL Syntax:

```
Select * From TEAM
Where CITY = "Bowie"
```

GIVEN: Relation = Team

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>
T1	Black Birds	NE	Bowie
T2	Capitals	SE	Laurel
T3	Metros	NW	Laurel
T4	Bears	NE	Bowie
T5	Maulers	NW	Croom

RESULT

<u>T1</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>
T1	Black Birds	NE	Bowie
T4	Bears	NE	Bowie

Figure 3.40. Relational Operation: Select example.



### 3.2.5.2 Relationship Operations

Relationship operations are quite different for static and dynamic relationship DBMSs. The CONNECT and DISCONNECT operations and the GET OWNER, GET NEXT, and GET MEMBER operations are included exclusively in the static data model DBMSs. The INTERSECTION, DIFFERENCE, JOIN, DIVIDE, PRODUCT, and UNION operations are exclusively in the dynamic data model DBMSs. Figure 3.41 enumerates all eleven relationship operations and briefly defines the results.

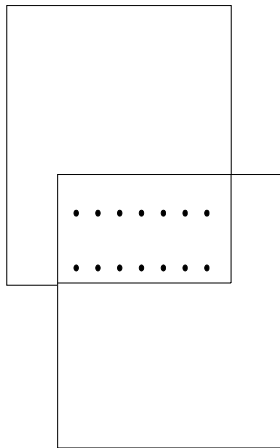
The static CONNECT operation *connects* an already stored row to a relationship. The DISCONNECT operation *disconnects* an already connected row from a relationship.

OPERATION	STATIC	DYNAMIC
CONNECT	Add to a Named RELATIONSHIP in Specific Oder	N/A
DISCONNECT	Delete From RELATIONSHIP	N/A
GET OWNER	Obtains The Parent of the Row That is Current	N/A
GET MEMBER	Obtains the First Child of the Owner For the Named Relationship	N/A
GET NEXT	Obtains the Next Row Within The Named Relationship	N/A
INTERSECT	N/A	Find and Keep Only the Common
DIFFERENCE	N/A	Find and Keep Only the Not Common
JOIN	N/A	“Append” Relations to Each Other
DIVIDE	N/A	Subset
PRODUCT	N/A	Cross-Product
UNION	N/A	Merge and Drop Duplicates

**Figure 3.41.** Relationship Operations



The dynamic operations are often shorthand references to actual programs that the user must create and execute. The INTERSECTION, DIFFERENCE, JOIN, DIVIDE, PRODUCT, and UNION relationship operations are graphically and programmatically presented in Figures 3.42 through 3.53.



**Figure 3.42.** Relational operation: Intersection.

#### REQUIREMENTS

2 RELATIONS  
SAME DEGREE

2 ATTRIBUTES  
SAME DOMAIN

#### SQL Syntax

```
Select T# From A
Where Not Exists
(Select T# From B
Where B.T# = A.T#)
```

GIVEN RELATION = A (Bowie Teams)

```
T1 Black Birds NE Bowie
T4 Bears NE Bowie
```

AND RELATION = B ( Teams of Player P1 )

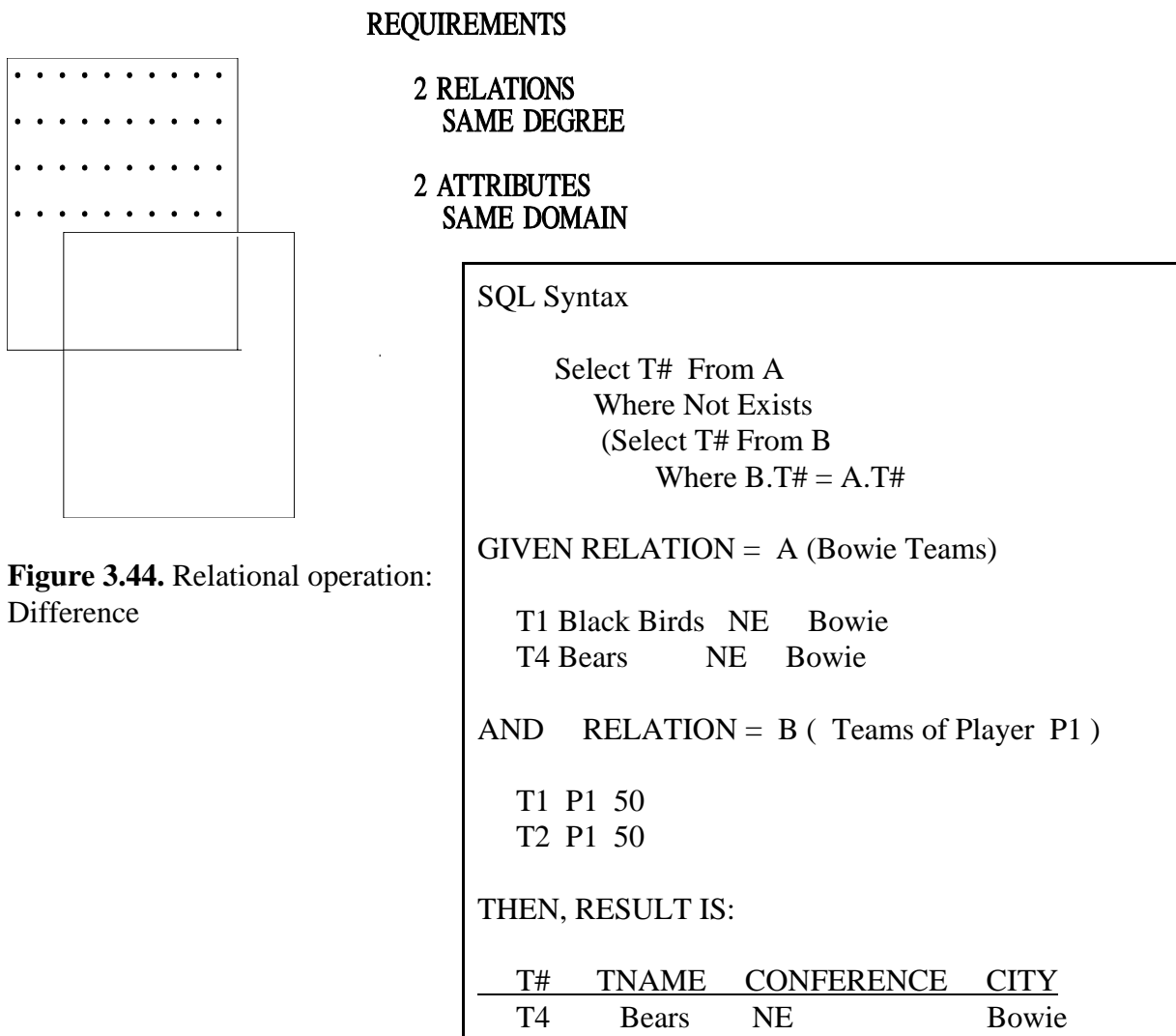
```
T1 P1 50
T2 P1 50
```

THEN, RESULT IS:

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>
T4	Bears	NE	Bowie

**Figure 3.43.** Relational operation: Intersection example.





**Figure 3.44.** Relational operation: Difference

**Figure 3.45.** Relational Operation: Difference example



a1	b1
a2	b1
a3	b2

(1)

b1	c1
b2	c2
b3	c3

(2)

a1	b1	c1
a2	b1	c1
a3	b2	c2

(3)

**Figure 3.46.** Relational Operation: Join T1 with T2, producing T3



SQL Syntax:

```
Select TEAM. *, PLAYER. *
From TEAM, PLAYER
Where TEAM. CITY = PLAYER. CITY
```

GIVEN Relation = Team

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>
T1	Black Birds	NE	Bowie
T2	Capitals	SE	Laurel
T3	Metros	NW	Laurel
T4	Bears	NE	Bowie
T5	Maulers	NW	Croom

AND Relation = Player

<u>P#</u>	<u>PNAME</u>	<u>COLLEGE</u>	<u>WEIGHT</u>	<u>CITY</u>
P1	Jackson	Colby	180	Bowie
P2	Beaner	Hamline	165	Laurel
P3	Tufts	St. Mikes	165	Savage
P4	Tufts	Colby	205	Bowie
P5	Larson	St. Mikes	180	Laurel
P6	Bowes	Colby	190	Bowie

RESULTS IN:

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>P#</u>	<u>PNAME</u>	<u>COLLEGE</u>	<u>WEIGHT</u>	<u>CITY</u>
T1	Black Birds	NE	p6	Bowes	Colby	190	Bowie
T1	Black Birds	NE	p1	Jackson	Colby	180	Bowie
T1	Black Birds	NE	p6	Tufts	Colby	205	Bowie
T2	Capitals	SE	p2	Beaner	Hamlin	165	Laurel
T2	Capitals	SE	p5	Larson	St. Mikes	180	Laurel
T3	Metros	SE	p2	Beaner	Hamline	165	Laurel
T3	Metros	SE	p5	Larson	St. Mikes	180	Laurel
T4	Bears	NE	p1	Jackson	Colby	180	Bowie
T4	Bears	NE	p6	Bowes	Colby	190	Bowie
T4	Bears	NE	p6	Tufts	Colby	205	Bowie

**Figure 3.47.** Relational operation: Join example.



a	x
a	y
a	z
b	x
c	y

(1)

X
Z

(2)

=

a
---

(3)

**Figure 3.48.** Relational operation: Divide. T1 divided by T2 produces T3



SQL Syntax:

Select Distinct P# From TEAM-PLAYER

Where Not Exists

(Select P# From TEAM-PLAYER

Where Not Exists

(Select T# P# From TEAM-PLAYER

Where TEAM-PLAYER. T# = HOT-TEAM.T# ) )

GIVEN: Relation : Team-Player

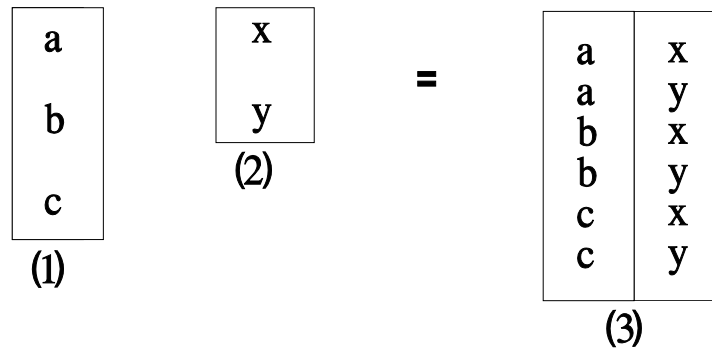
T# P# GOALS And Relation: HOT-TEAMS

T1	P1	50	(SELECT T# FROM TEAM-PLAYER where GOALS > 37
T1	P2	37	
T1	P3	40	
T1	P4	37	
T1	P5	22	DIVIDED EQUALS
T1	P6	22	BY <u>P#</u>
T2	P1	50	T1 P2
T2	P2	40	T2
T3	P2	37	T4
T4	P2	37	
T4	P4	50	
T4	P5	40	

**Figure 3.49.** Relational operation: Divide example.







**Figure 3.50.** Relational operation Product T1 on T2 produces T3.



SQL Syntax:

```
Select TEAM. *, PLAYER. *
From TEAM, PLAYER
```

GIVEN:

Relation: Team

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>
T1	Black Birds	NE	Bowie
T2	Capitals	SE	Laurel
T3	Metros	NW	Laurel
T4	Bears	NE	Bowie
T5	Maulers	NW	Croom

Relation = Team - Player

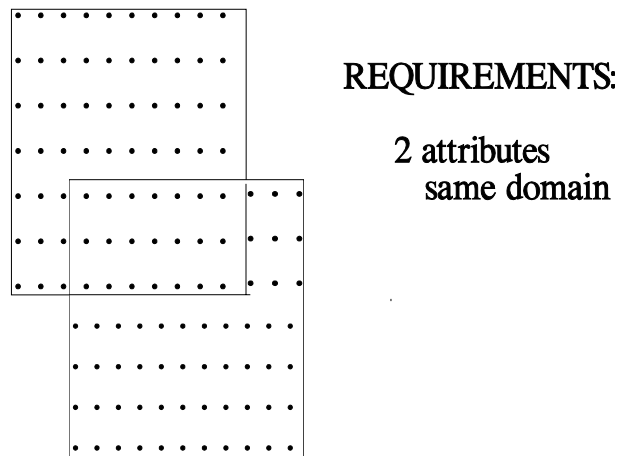
<u>T#</u>	<u>P#</u>	<u>GOALS</u>
T1	P1	50
T1	P2	37
T1	P3	40
T1	P4	37
T1	P5	22
T1	P6	22
T2	P1	50
T2	P2	40
T3	P2	37
T4	P2	37
T4	P4	50
T4	P5	40

RESULT:

<u>T#</u>	<u>TNAME</u>	<u>CONFERENCE</u>	<u>CITY</u>	<u>P#</u>	<u>GOALS</u>
T1	Black Birds	NE	Bowie	P1	50
T1	Black Birds	NE	Bowie	P2	37
T1	Black Birds	NE	Bowie	P3	40
T1	Black Birds	NE	Bowie	P4	37
T1	Black Birds	NE	Bowie	P5	22
T1	Black Birds	NE	Bowie	P6	22
					ETC.
					ETC.

**Figure 3.51.** Relational operation. Product example.





**Figure 3.52.** Relational operation: Union.



SQL Syntax:

```
Select P# From PLAYER
where WEIGHT > 180
Union
Select T# From TEAM-PLAYER
where T# = "T2"
```

Relation : Player

<u>P#</u>	<u>PNAME</u>	<u>COLLEGE</u>	<u>WEIGHT</u>	<u>CITY</u>
P1	Jackson	Colby	180	Bowie
P2	Beaner	Hamline	165	Laurel
P3	Tufts	St. Mikes	165	Savage
P4	Tufts	Colby	205	Bowie
P5	Larson	St. Mikes	180	Laurel
P6	Bowes	Colby	190	Bowie

AND

Relation : Team-Player

<u>T#</u>	<u>P#</u>	<u>GOALS</u>
T1	P1	50
T1	P2	37
T1	P3	40
T1	P4	37
T1	P5	22
T1	P6	22
T2	P1	50
T2	P2	40
T3	P2	37
T4	P2	37
T4	P4	50
T4	P5	40

RESULT:

P#  
p1  
p2  
p4  
p6

**Figure 3.53.** Relational operation: Union example.



### 3.2.5.3 Combination Operations

Combination operations are two or more row and/or relationship operations. Figure 3.54 identifies the more common combination operations and their component operations.

Fetch = FIND + GET

MODIFY = FIND + GET + STORE

INSERT = STORE + CORRECT

**Figure 3.54.** Combination operations.

### 3.2.6 Logical Database Components Summary

The logical database consists of components. These are domains, tables, columns, relationships, and operations on the rows.

Domains, the first component, contain the specifications of the legal values for classes of columns. Domains also specify the legal combinations of various columns within DBMS operations such as select, join, and so on.

Tables, the third component, contain their own naming clauses, any applicable integrity conditions, and an enumeration of the member columns. The table's definition also identifies the roles played by the various columns, namely: primary key, candidate key, secondary key, and foreign keys.

Relationships are the fourth component of the logical database. There are eight types of relationships:

- One-to-many
- Owner-multiple member
- Singular-one member
- Singular-multiple-member
- Recursive
- any-to-many
- One-to-one
- Inferential

Operations are the final component of the logical database. Whether a particular operation is permitted or not depends on whether the relationship binding method is static or dynamic.

The next section brings together four popular combinations of these five logical database components in order to define the four data models common to most DBMSs.



### 3.3 Data Models

Every DBMS's logical database is controlled by a set of rules. The combination of the rules that allow network or hierarchical relationships, that bind rows through either static or dynamic relationships, and that permit distinct operations is called the DBMS's data model.

Because data models have different table structures and relationship binding methods, each data model also permits specific operations to manipulate these structures. A data model's definition is thus in three parts:

- The formal definition of the permitted table structure, *AND*
- The formal definition of the number, types and kind of relationships that can be defined between rows of the same and different types, *AND*
- The formal definition of the operations that can take place on the resultant data structure of row and relationship instances

The study of data models is important, because a data model determines the data structure, operations, and relationships that are applied to the tables. Any data structures and operations beyond those set by the DBMS must be created by the application's designers and programmers on the basis of shared column values. Programs are needed to navigate among the rows to effect the operations required by the application's designers. Finally, in most cases, the application designer is also responsible for the maintenance of the non-supported data structures and operations. In short, the data model of the DBMS determines the default set of allowed data structures and operations. Anything beyond these defaults has to be required by the application designer and then programmed by the application programmers.

For example, suppose one DBMS contains an operation that found the occurrences of one table that did not have a shared column's value in another table, while another DBMS supports that operation. The first DBMS explicitly supports answering the question: FIND STUDENTS NOT ASSIGNED TO ANY CLASS. The second DBMS requires a programmer to write a program to access a student row and then search the class rows to see if the student is assigned to any classes; if not, then that student is reported. The program then has to pick up the next student row and perform the search again, until all the student rows are exhausted.

The point of this example is not whether the first DBMS can perform the operation faster than the programmer-written program can, but that the first DBMS explicitly supports the operation, while the second requires that the user program it. Because the first DBMS explicitly supports this operation, there is no need for program design, development, or debugging, as these have already been accomplished by the DBMS developers.



While the five components of the logical database can be brought together in any number of combinations to create many different data models, there are four combinations that are most popular. These are:

- Network
- Hierarchical
- Independent logical file
- Relational

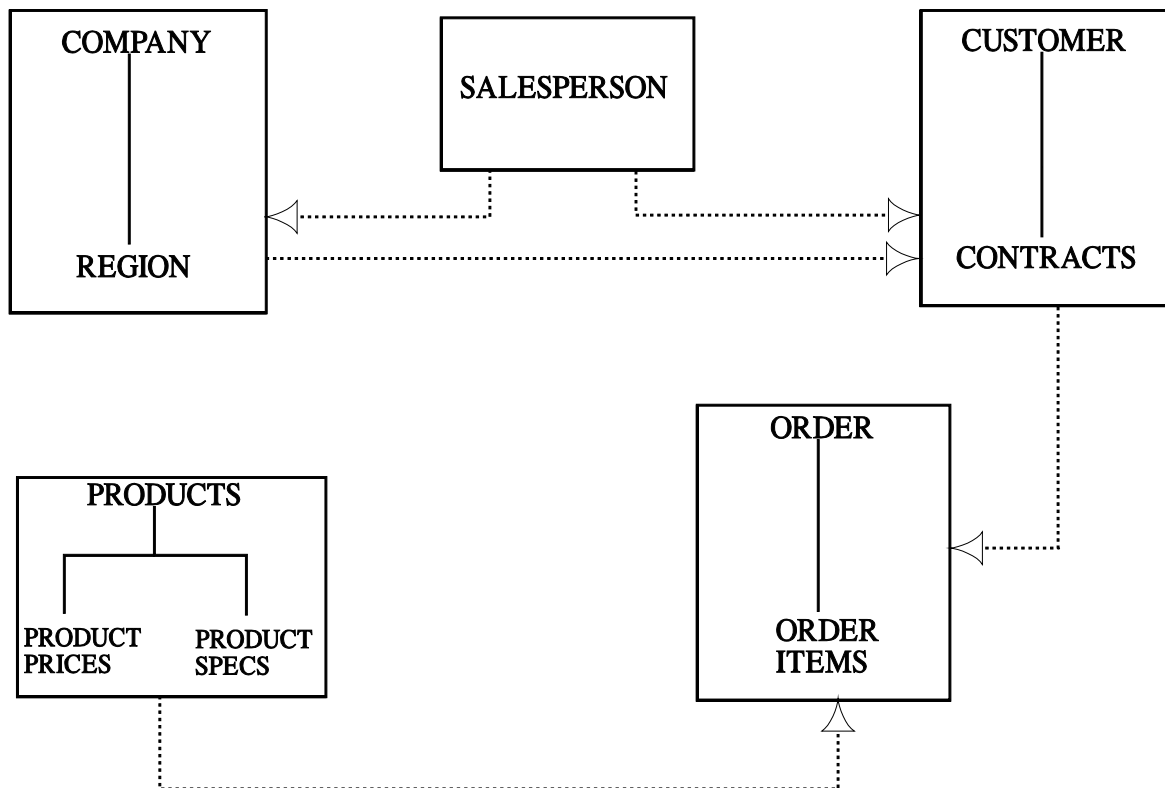
Most of the popular DBMS literature acknowledges the existence of only the network, hierarchical, and relational data models. To ignore the existence, however, of the independent logical file data model is to refuse to acknowledge the existence of some of the most popular DBMSs, e.g., ADABAS, FOCUS, Inquire, Nomad, Model 204, and RAMIS. To be both complete and correct, this book describes the independent logical file data model and its capabilities.

Since the first edition of this book, the SQL language has grown to include complex table structures, relationships and operations from the network, hierarchical, and independent logical file data models. Consequently, this edition of the book asserts that SQL/1999 defines its own unique data model. In SQL/1999 some relationships are value-based while others clearly imply embedded system generated pointers. Because of these two “features,” SQL/1999 is clearly not relational.

The relationships built by the network and hierarchical data models are static, and the relationships used by the independent logical file and relational data models are dynamic. A network data model database permits multiple complex rows and allows a member row to be owned by multiple rows from different tables (see Figure 3.9). A hierarchical data model database permits a single hierarchy of segments, each of which is simple. The entire collection of segments is actually one complex table. Each segment in the hierarchy is owned by only one owner segment (see Figure 3.8). The columns in the segments can only be single valued. An independent logical file (ILF) data model database permits multiple complex tables (see Figure 3.55). A relational data model database permits multiple rows (see Figure 3.28). However, unlike the ILF data model, all relational tables can only contain single-valued columns.

In general, the emphasis in the network data model is complex database and table structures which then only require simple operations, while the emphasis in the relational data model is simple database and table structures which then require complex operations. The network data model provides facilities for strong, central, schema-based control over table structures. Thus, the network data model has a complex table structure and many types of relationships. Because of this strong structure, network data model operations are defined in terms of navigating these structures. In short, the number and variety of explicit DDL-defined data structures is greatest in the network data model. Conversely, the relational data model provides only simple table structures but complex operations. Figure 3.56 expresses the approximate proportions of components for these two data models.





..... = Dynamic Relationships  
 ————— = Static Relationships

**Figure 3.55.** Independent logical file data model. Five “files” with complex and simple record types.

FUNCTIONS OF NETWORK	FUNCTIONS OF RELATIONAL
Data Structure (.8) And Operational (.2)	Data Structure (.2) And Operational (.8)

**Figure 3.56.** Location of static and dynamic functionality.





The hierarchical data model has simpler table structures than the network data model and similar operations. The hierarchical data model is actually a subset of the network data model.

The ILF data model has complex table structures, thus it has some operations similar to the network data model. The ILF data model also has dynamic relationships between tables, thus the ILF supports operations similar to relational. The ILF data model is a superset of the relational data model.

The manner in which relationships are created and maintained by the DBMS is also very important. It is generally true that the network data model DBMSs are built to utilize static relationships. There are a few network data model DBMSs, however, that have dynamic relationships.

The static relationship is declared explicitly by the database designer--through the DDL--and then its instances are automatically generated by the DBMS whenever an owner row is created and one or more member rows of the relationship are inserted. The relationship is usually a relative row address. Traditionally, this address is stored in the owner row for its first member row, and each member row has the address of the next member row. Some static relationship DBMSs additionally allow both the owner row's address and the prior member row's address to be stored in each member row. These relationships are built as the rows are loaded into the database, and are maintained through updates.

An advantage of a static relationship is the speed with which it can be traversed. A disadvantage is that whenever the row's relationship needs to be changed, the row must first be *disconnected* from its owner and then *reconnected* to the new owner.

In most dynamic relationship DBMSs, the relationships between tables are not declared in any way. As stated above, the basis of the relationship is a defined column in both the owner and member table. Since a dynamic relationship is represented through a column's value, the relationship that one row has with another may be changed through a simple update. While changing a relationship is very simple, the amount of computer resources required to traverse a dynamic relationship (from owner to all members) is normally greater than it would have been in a static relationship.

The reason for the performance difference is as follows. In the vast majority of static relationship DBMSs, the relationship mechanism is a relative row address of the row. Thus, when there is a request to traverse from an owner row to its first member, the member's relative row address (the location of the row) is already known and stored, usually in the owner.

Once the address has been obtained, the only step left in the relationship traversal process is to get the row from its location. Getting the *next* member row requires only accessing the *next* relative row address and going to that address to get the row. In contrast, in the vast majority of dynamic relationship DBMSs, the relationship mechanism is a shared column value, which most often is the value of the owner's primary key, replicated in each of the member rows. To access the set of member rows, the typical dynamic relationship DBMS has to first use the owner row's primary key in a secondary key search for the member rows. Such a search often results in a list of member row primary keys. Then each of these keys must be used in a primary key search to locate and retrieve each member row.

The table structure is formally defined through the data definition language (DDL), and the operations occur through the expression of the various data manipulation languages (DML) that are supported by the DBMS. This division of function can also be used to distinguish static



from dynamic. In a static relationship DBMS, the DDL contains both the definition of table structures and relationships, while the DML contains only the operations. In a dynamic relationship DBMS, the DDL contains only the table structures, while the DML contains the relationship definitions and operations. Figures 3.57 and 3.58 depict these differences. Figures 3.59a and 3.59b tabulate the combinations of facilities that are present in the four data models.

When the concept of data model is combined with the concept of static and dynamic, a 2 x 4 matrix results. The names of DBMSs along with their principal data model are identified in Figure 3.60. It should be noted that a number of DBMSs support more than one data model. Today's popular combinations are network and relational (IDMS/R and SUPRA), hierarchical and relational (FOCUS and RAMIS), and independent logical file and relational (ADABAS, Datacom/DB, FOCUS, Inquire, Model 204, and RAMIS). Note that in Figure 3.60, DBMSs implementing SQL/1999 and beyond can include features from all four data models.

Figure 3.61 tabulates the effects the four data models have on database applications. Independent of this comparison is the effort to create a quality database design, which, regardless of data model, is the same. That is because all the design work necessary for a quality database must be done before any transformation steps are begun.

Data Model Type	Data Definition Language Components	Data Manipulation Language Components
STATIC	Table structures Relationships	Operation
DYNAMIC	Table structures	Relationships Operations

**Figure 3.57.** Comparing static and dynamic data model type components

Data Model = {Row Organization} + {Inter-row Relationships} + {Operations}
Data Model = {Data Definition Language} + {Data Manipulation Language}
Dynamic Data Model = { DDL (RO) } + { DML (REL + OPS) }
Static Data Model = { DDL (RO + REL ) } + { DML (OPS) }
WHERE:
DDL = Data Definition Model
DML = Data Manipulation Language
RO = Row Organization
REL = Relationships
OPS = Operations

**Figure 3.58.** Data Model Components.



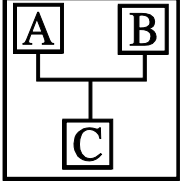
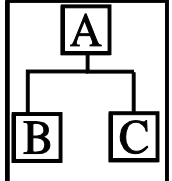
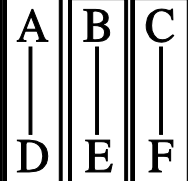
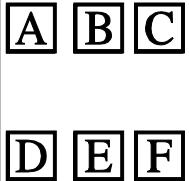
		Data Model			
Characteristics		Network	Hierarchy	Independent Logical File	Relational
Record Organization		SV, MV, MD, G, RG	SV segments	SV, MV, RG	SV
Relationship (REL)					
Operations (OPS)	Record	A, D, F, M	A, D, F, M	A, D, F, M	A, D, F, M, P
	Relationships	C, D, GO, GM, GN	GO, GM, GN	J, INT, DIV, UN, DIF	J, INT, DIV, UN, PR, DIV
DDL		REL, RO	REL, RO	RO	RO
DML		OPS	OPS	REL & OPS	REL & OPS

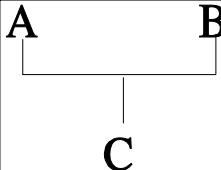
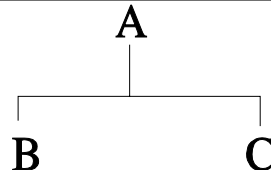
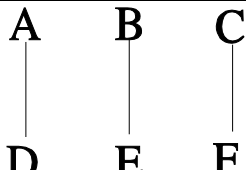
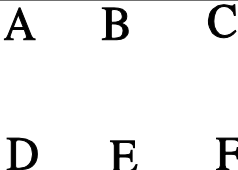


Figure 3.59a. The four data models



Row Organization	SV = Single Value		
	MV = Multiple Value		
	MD = Multiple Dimension		
	G = Group		
Operations	Row	A = Add	
		D = Delete	
		M = Modify	
		P = Project	
		F = Find	
	Relation-ship	C = Connect	Dis = Disconnect
		P = Project	J = Join
		DIV = Divide	GO = Get Owner
		GM = Get Member	GN = Get Next
		INT = Intersection	PR = Product
		UN = Union	DIF = Difference

**Figure 3.59b.** The four data models: Legend.



DBMS TYPE			
STATIC		DYNAMIC	
NETWORK	HIERARCHICAL	ILF	RELATIONAL
			
Supra IDMS/R* IDS* DMS-2 DMS 1100 IMAGE VAX-DBMS* PRIME-DB*	IMS System 2000	Manage Model 204 Inquire Adabas Nomad Focus Ramis DMS 170 Datacom/DB	Ingres* Oracle* DB/2 Sybase Informix SQL Server
*ANSI-NDL			*ANSI-SQL/1986, 1989, and 1992
		*ANSI-SQL/1999 and 2003	

Note: By ANSI/NDL or ANSI/SQL, it is meant that the DBMSs at least generally conform to the data model and the process models specified by these two standards. The exact syntax and semantics may vary slightly, making each DBMS more or less compatible.

**Figure 3.60.** DBMS data models.



Characteristic	Data model			
	Network	Hierarchy	ILF	Relational
Structure	Complex	Moderate	Complex	Simple
Row Order Control	Update	Update Retrieval	Retrieval	Run-Unit
Relationship Binding	Load or Update	Load or Update	Only at Retrieval	Only at Retrieval
Mandatory Design Work	High	Moderate	Low	Lowest
DBMS Control Over Programmer	High	Moderate	Low	Lowest
Ease of Database Design Change	Poor	Less Poor	Better	Best
Effort to Design a Quality Data Database	Great	Great	Great	Great

**Figure 3.61.** Data model effects on application development.



DATA MODEL				
Relationship Type	DDL Declaration		DML Simulation	
	ANSI Network	Hierarchy	ILF	ANSI-SQL Relational
Owner 1 Member >1 Member	DS	DS	DD	DD
	DS	NO	NO	NO
Owner 1 Member >1 Member	DS	NO	NO	NO
	DS	NO	NO	NO
Recursive	DS	ID	DD	ID
Many to Many	IS	ID	DD	ID
Inferential	ID	NO	DD	DD
One to One	DS	ID	DD	DD

DS means direct, static relationships

IS means indirect, static relationships

DD means direct, dynamic relationships

ID means indirect, dynamic relationships

No means no practical method

**Figure 3.62.** Method of relationship implementation by data model.



### 3.3.1 Data Models and DBMSs

As stated above, a data model is a formalized approach to the organization of data. By permitting some data structures and forbidding others, a data model determines the types of operations that can be performed against the data structures. Within specific DBMSs, relationships are either explicitly declarable or they can be simulated through special data manipulation operators. When a relationship is explicitly declared and data is loaded according to these explicitly declared relationships, the pre-executed relationships exist for all to use without regeneration.

A relationship can be simulated through the use of a special data manipulation language operator. Then, if twenty different users need to select rows from two different tables, the DBMS must determine the set of rows belonging to those relationships--twenty different times.

Relationships are explicitly declarable only in the network (NDL) and hierarchical data models. A relationship is a formalized expression of the nature of the allowed interaction between two tables. In the network model, the relationship is defined through the use of the SET clause. In the hierarchical data model, for example, in SYSTEM 2000, the relationship is explicitly stated through the use of a RG IN <parent repeating group id> clause. In the independent logical file data model, the relationships are implicitly defined only within tables through the use of multiple-valued columns, or through the use of repeating group columns. In the relational model, most notably ANSI/SQL, relationships cannot be explicitly or implicitly defined. The relational model can only simulate relationships through the use of specially defined operators such as JOIN and UNION that interrelate rows at execution time.

Figure 3.62 identifies the different types of relationships that can be explicitly defined in the four data models. A relationship can be implemented either statically (S) or dynamically (D), and can be accomplished directly (D) without having to define additional tables, or can be accomplished indirectly (I) once additional tables are defined. A table is considered to be *additional* whenever the DBMS cannot execute the relationship directly without the database's designer first defining another table. For example, the many-to-many relationship depicted in Figure 3.31 can be executed directly by ADABAS with only two tables. To accomplish that relationship in DB2, however, first requires the definition of the three table structure represented in Figure 3.30. To accomplish an indirect (I) relationship type requires the definition of additional tables. Thus, the relationship type is not really accomplished; rather it is transformed into a different relationship type, and then executed through custom programming in the various application programs. Finally, Figure 3.62 declares a relationship-data model intersection as a *no* whenever there is no practical method of accomplishing the relationship.

DBMSs implementing data models not standardized by ANSI are only generally equivalent even though they might be identified as having the same data model. An example of this general comparability is IBM's Information Management System (IMS) and SAS's SYSTEM 2000.

As stated above, there are four commonly employed data models; two are static--network and hierarchical, and two are dynamic--independent logical file and relational.

Because of the ANSI network data model standard, some static network DBMSs compare favorably to the ANSI/NDL. These are: Computer Associate's IDMS/R, Unisys' DMS-1100, Digital's VAX/DBMS, and Honeywell's IDS-2. While Unisys supports another network data model DBMS, DM-2, it does not compare favorably to the ANSI requirements. CINCOM's





SUPRA network data model is the same as it was in the early 1960s, thus it does not conform to either the ANSI/NDL nor this book's more general specification of the network model.

There are two hierarchical DBMSs, SAS's SYSTEM 2000 and IBM's IMS. There is, however, no ANSI standard data model for hierarchies, and no standard is contemplated.

There are a number of independent logical file DBMSs. Since there is no ANSI standard for this data model, however, their similarities, while significant, cannot be depended upon for database and application portability.

A version of the relational data model has been standardized through the specification of ANSI/SQL. A number of DBMSs either already conform to the ANSI/SQL standard or are in the process of delivering SQL facilities. DBMSs in the former category include DB2, ORACLE, INFORMIX-SQL, and INGRESS. DBMSs in the latter category include IDMS/R, Model 204, ADABAS, and SUPRA.

IDMS/R and SUPRA are identified as both network and relational because they support both network and relational processing. IDMS/R is generally a superset of ANSI/NDL and provides support to the ANSI/SQL language for accessing IDMS/R's relationally defined tables. While SUPRA's network model is non-standard, SUPRA offers ANSI/SQL support against its *master* tables.

Notwithstanding their differences, all DBMSs supporting the data models defined in this book are fundamentally the same--from the perspective of data models. Further, all static relationship DBMSs possess a common major characteristic--static relationships between rows of the same or different type. All dynamic relationship DBMSs possess a common major characteristic--dynamic relationships between rows of the same or different types. What this implies is that the most suitable of each DBMS type should be procured. Having several static, or several dynamic, relationship DBMSs, is of little value.

If an organization judges that a static-hierarchy DBMS is the most appropriate for its applications, then it would be appropriate to perform a detailed comparison of IMS and SYSTEM 2000 to determine which one contains the features most suitable for the organization.

Similarly, if a dynamic relationship DBMS is also desired, then an examination of Datacom/DB, DB2, ADABAS, Inquire, Model 204, and Oracle would be appropriate.

What would not be appropriate would be, for example, a comparison of IMS and ADABAS, or of SYSTEM 2000 and INQUIRE. The desire to make such comparisons indicates that the evaluators do not yet understand either DBMS or the needs of their applications, or both.

### 3.3.2 Static Data Models

The network and hierarchical data models are founded mainly on static relationships. Within a database the relationships are static. Relationships between databases are dynamic. Fewer dynamic relationships are needed in the network data model than in the hierarchical data model, simply because the network model supports more relationship-modeling capabilities than does the hierarchical model.



### 3.3.2.1 Network Data Models

The data structure characteristic unique to the network data model is the ability of a row to be a member of two different relationships. This is illustrated in Figure 3.63, where COURSE-SECTION is a common member to both COURSE and TEACHER. Figure 3.63 contains five tables. The owners of the COURSE-SECTION table are COURSE and TEACHER. The dependents of the COURSE-SECTION table are the student's ENROLLMENT and the EQUIPMENT needed for the COURSE-SECTION.

As illustrated in Figure 3.62, the network model can accomplish all the relationship types. Six are accomplished directly. Two are accomplished indirectly, and only one requires dynamic relationships.

The network data model has several major versions. Notable among these is the Conference on Data System Languages (CODASYL) model that has been standardized by ANSI as NDL and implemented by Honeywell (IDS-2), Unisys (DMS-1100), VAX/DBMS, and Computer Associates (IDMS/R). Since the ANSI/NDL is a subset of the CODASYL network model, most subsets of all these DBMSs comply to ANSI/NDL.

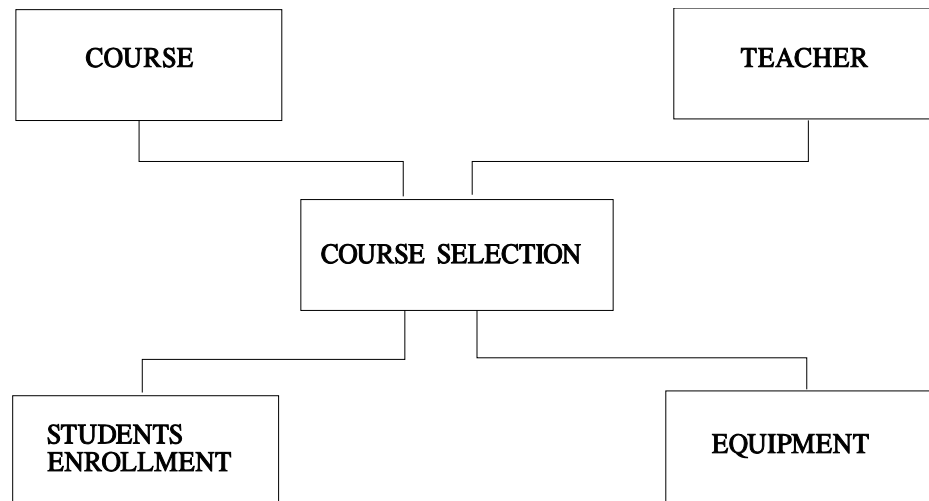
The table structure allowed by ANSI/NDL is identified in Figure 3.64. In this same figure is a comparison between the table structure allowed in both the CODASYL network model and ANSI/NDL network model.

In general, the ANSI network data model allows multiple relationships between an owner and its members, among members of the same and different types with and without an owner, and between owners and members of the same type. Relationships that do not have another table as an owner are called singular relationships. Relationships that have the same table as both owner and member are called recursive. Figures 3.10, 3.11, 3.13, 3.14, and 3.16 illustrate the different types of relationships explicitly definable by the NDL's data definition language. The many-to-many relationship can only be implemented indirectly within the NDL. Thus, to model the many-to-many relationship (Figure 3.20) involving two tables (COURSE and STUDENT) a triple table structure (COURSE, STUDENT, and ENROLLMENT) has to be created. This three table structure is illustrated in Figure 3.65.

Figure 3.12 illustrates a composite of different relationships that can all be defined at the same time on the identified tables within the ANSI/NDL.

Another type of network data model is SUPRA (formerly TIS (formerly TOTAL)). It is a very restricted subset of the capabilities of the ANSI/NDL model. SUPRA's network data model has remained essentially unchanged since the middle 1960s. It only permits single-value fields within the table. It does not contain singular, multi-member, or recursive sets. Further, the owner table of SUPRA's single member relationships cannot be defined to be the member table of another relationship. Because of this restriction, intermediate tables have to be invented, which requires extra user written programs.



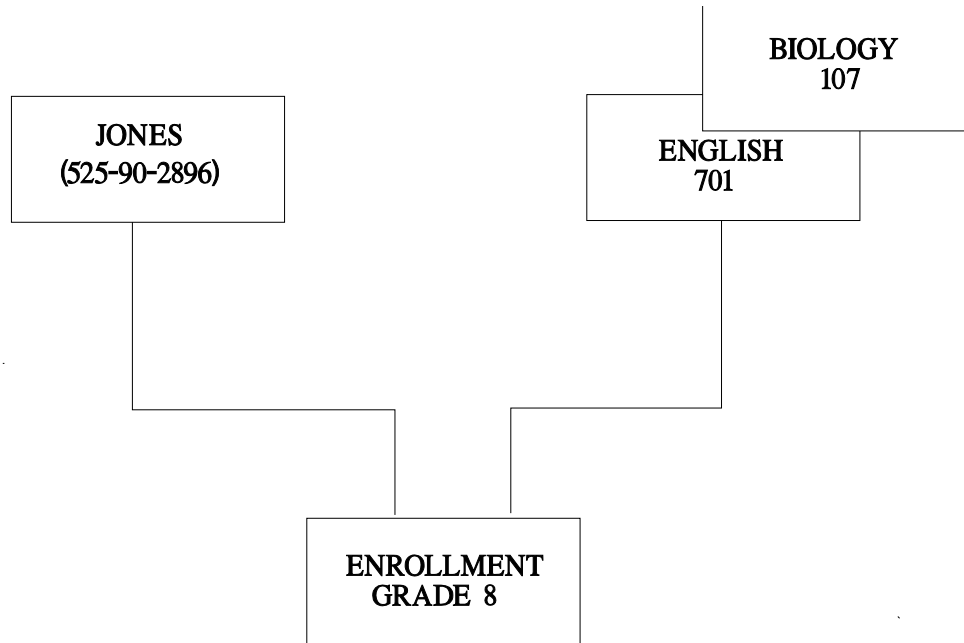


**Figure 3.63.** Network data model.

MODELS			
CODASYL	ANSI	STRUCTURE	Column EXAMPLE
Yes	Yes	Primary Key	SSN
Yes	Yes	Non-Repeated Fields	Name Address Sex
Yes	Yes	Vectors	Nicknames
Yes	No	Matrix	Monthly -Division Sale
Yes	No	Repeating Group	Job Title Start Date Stop Date
Yes	No	Nested Repeating Group	Family Kids Hobbies

**Figure 3.64.** Network row organization.



**TWO SET TYPES****STUDENT WITH GRADE RECORDS****COURSE SECTION WITH GRADES****Figure 3.65.** Indirect network relationships.

The operations allowed in the network model are identified in Figure 3.66. As stated above, relationships in NDL must not be required to be value based. That is, the relationship definition must not require that the primary key value of the owner row nor any other owner column values be present in the member row. This feature insulates member row occurrences from having to be accessed and modified if the primary key of the owner is changed.

NDL DBMSs permit great sophistication in placing new rows within the context of a relationship, that is, always at the head, always at the end, or in sorted order. This way, if rows are stored in time sequence, and if the store order is always HEAD, then the oldest rows are always towards the end, and the newest rows are always at the front.

Storing rows in sorted order within the context of a relationship is certainly less expensive than storing an entire table in a sorted order. However, the DBMS's maintenance of the sorted order is more expensive than retrieving rows and then sorting afterwards. Sorting after retrieval also requires that all the natural languages contain sort verbs.



<b>Retrieval operations</b>
Retrieval
Find
Get
Update
Store
Delete
<b>Relationship operations</b>
Get owner
Get number
Get next
Connect
Disconnect
<b>Combination operations</b>
Insert
Fetch

**Figure 3.66.** Network Operations.

### 3.3.2.2 Hierarchy

The hierarchical data model represented by IBM's IMS and SAS's SYSTEM 2000 is simpler than the network data model because each hierarchical database only allows one complex table. Each segment in the table can have only one owner, and there can be only one relationship between an owner segment and its member segments. The only relationship type allowed is owner-single-member. Whenever a connection between hierarchical databases is required, a dynamic relationship must be created (see Figure 3.8). For example, the relationship between PRODUCT and PRODUCT SPECIFICATION segments is static, but the relationship between the PRODUCT PRICE and ORDER ITEM segments is dynamic.

Each segment within a hierarchical data model row has traditionally been restricted to single-valued columns (see Figure 3.67). Each member segment type can only have one owner.

As illustrated in Figure 3.62, the hierarchical model directly accomplishes only one relationship type. Two others are accomplished indirectly and dynamically. One is accomplished directly but dynamically. Four relationship types are identified as *no*.

Figure 3.68 illustrates a typical hierarchical structure. Normally there is no practical limit to the number of levels in the hierarchy, nor to the number of segments on each level. In SYSTEM 2000, for example, if each segment (called a repeating group) contains about 9 columns, SYSTEM 2000 would then allow up to 100 segments to be configured. Each repeating group (segment) and column definition counts towards a maximum of 1000.



PRIMARY KEY ---- SSN	
Single-Valued-Columns	BIRTH DATE SEX CURRENT JOB DEGREE MAJOR MARITAL STATUS

**Figure 3.67.** Hierarchical column examples.

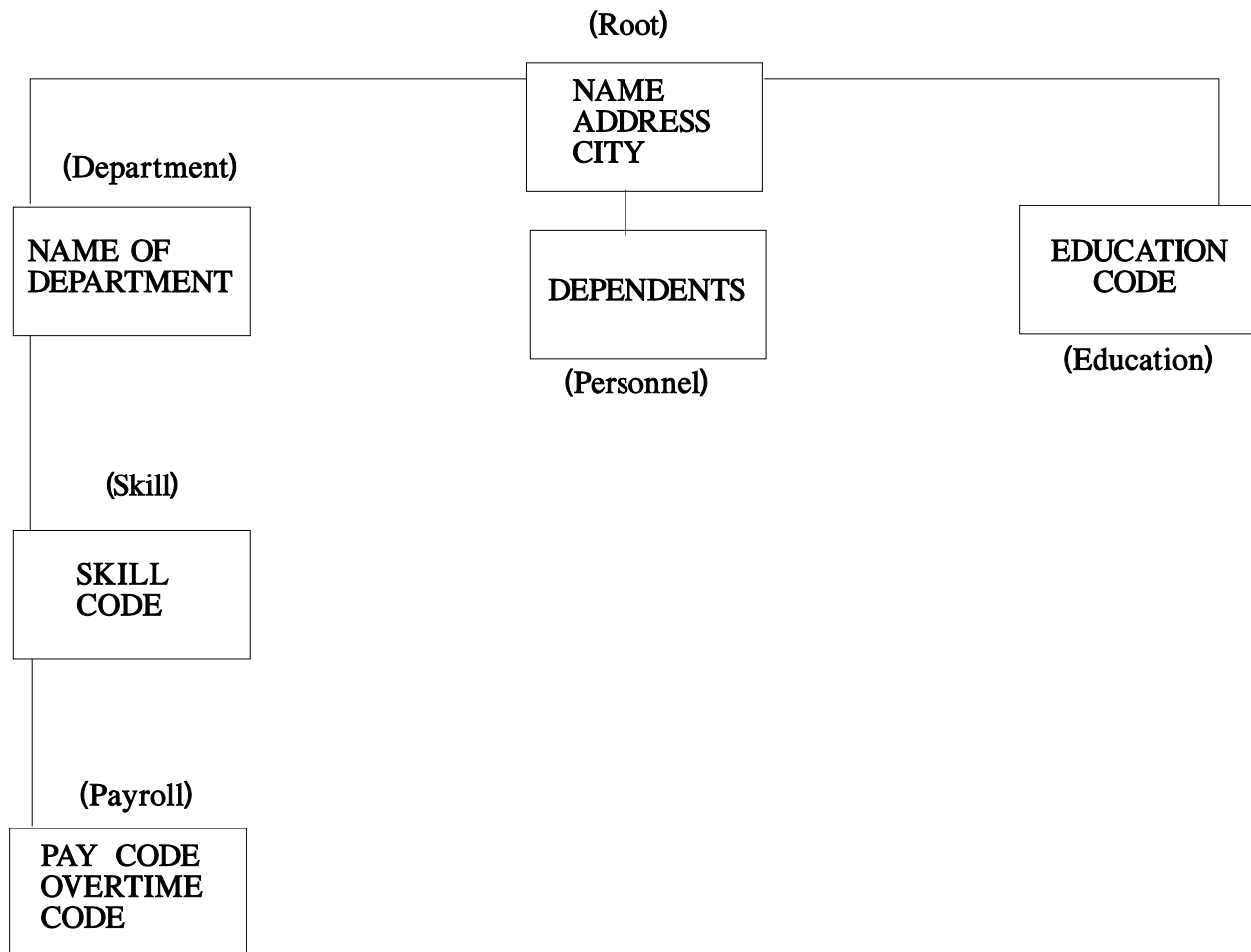
A SYSTEM 2000 hierarchy can be 32 levels deep and can contain as many repeating groups and columns as may be necessary so long as the 1000 count is not exceeded. In IMS, the depth of the hierarchy is restricted to nine levels.

Most production applications that use hierarchical data model databases require that multiple hierarchies be related to avoid unnecessary data redundancy. For example, if a STUDENT is taught by multiple TEACHERs, it is redundant to represent each TEACHER's data under each student or each STUDENT's data under each TEACHER.

To avoid redundant data definition and storage, the multiple hierarchies are tied together through dynamic relationships. The two major hierarchical data model systems, SYSTEM 2000 and IMS, differ significantly in this regard. IMS allows the explicit definition of dynamic relationships (twins) between tables within the same or different hierarchies. In SYSTEM 2000, intra- or inter-hierarchy relationships are specified linguistically by the application designer-programmer and executed by SYSTEM 2000. Notwithstanding its lack of explicit networking, SYSTEM 2000 is far simpler to use than IMS. Additionally, SYSTEM 2000's storage structure, although static, supports efficient ad hoc inquiry within the design constraints of the defined hierarchical structure, while IMS's does not.

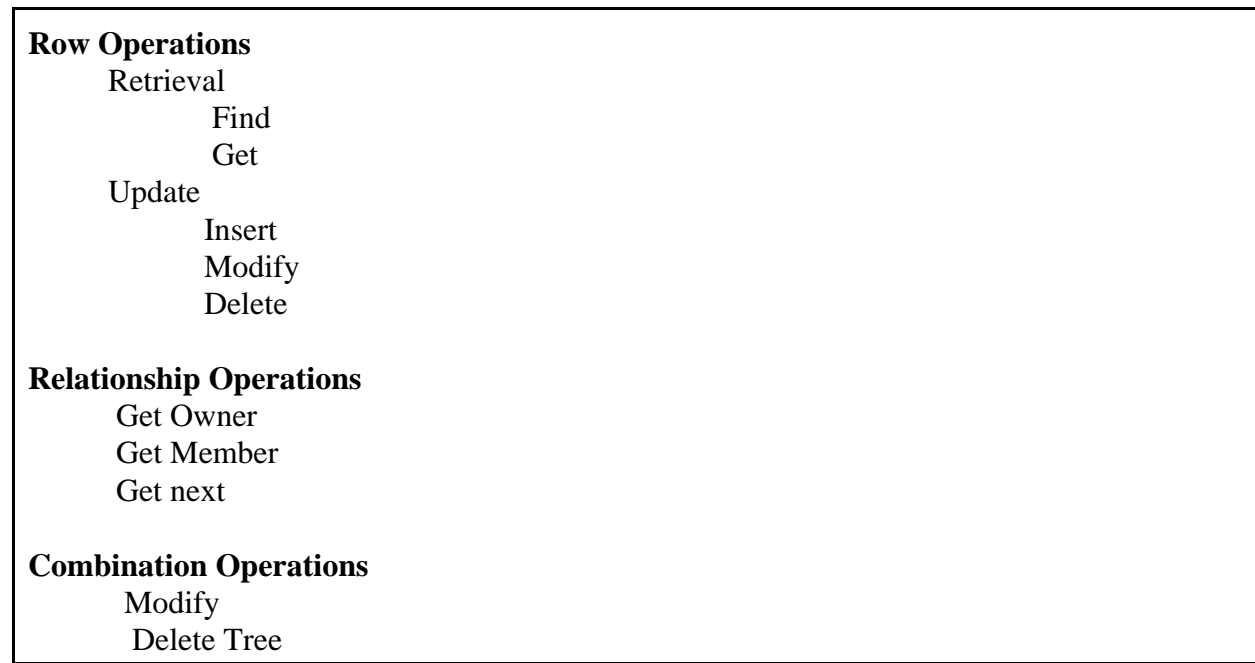
The allowable operations for the hierarchical data model appear in Figure 3.69.





**Figure 3.68.** Hierarchical relationships.





**Figure 3.69.** Hierarchical operations.

### 3.3.3 Dynamic Data Models

The independent logical file (ILF) and the relational data models, illustrated in Figures 3.55 and 3.28, are based on dynamic relationships. The ILF data model contains complex tables. The relationships between tables are represented through shared data values. Figure 3.60 lists some of the popular ILF systems. A relational data model has only simple rows and, like the ILF data model, represents relationships between rows through data values.

#### 3.3.3.1 Independent Logical File Data Model

An independent logical file (ILF) database allows tables to be complex. But the complexity is sometimes restricted to just two levels--a CUSTOMER and its CONTRACTs (see Figure 3.55). For some ILF DBMSs, the additional level of ORDERs for the CONTRACTs is represented by adding another table to the database and then connecting the first table to it through a dynamic relationship. For example, in Figure 3.55, the COMPANY table contains a repeating group, REGION. The relationship between the SALESPERSON and the REGION tables is dynamic. That is, the REGION ID from the REGION row is also present in the SALESPERSON row. The types of columns allowed in an ILF table are represented in Figure 3.70.





TABLE STRUCTURE	COLUMN EXAMPLE
Primary Key	SSN
Non-Repeated Fields	NAME ADDRESS SEX
Vectors	Nicknames
Matrix	Monthly by Division by Sales
Repeating Group	Job Title Start Date Stop Date
Nested Repeating Group	Project Project-ID Hours Description

**Figure 3.70.** Independent logical file column types.

FOCUS allows the definition of tables as complex as SYSTEM 2000. FOCUS also permits the definition of simple tables like relational DBMSs. Relationships between tables in FOCUS can be across the *tops* of tables, and also between lower level segments within the hierarchy.

As illustrated in Figure 3.62, the ILF data model requires all the relationships to be dynamic. Five of the relationships are direct, and three are *no*. A sophisticated ILF DBMS explicitly permits:

- One-to-many
- One-to-one
- Inferential
- Many-to-many
- Recursion

Figure 3.71 illustrates an ILF one-to-many relationship. A many-to-many relationship is depicted in Figure 3.21, an inferential relationship is depicted in Figure 3.24, and a recursive relationship is presented in Figures 3.16 through 3.18.

The operations allowed in the ILF data model are listed in Figure 3.72.



**CAR OWNER FILE FIELDS**

SOCIAL SECURITY NUMBER

NAME

STREET ADDRESS

CITY

COUNTY

ZIP CODE

SEX

BIRTH PLACE

**CAR REGISTRATION FILE FIELDS**

MANUFACTURER

MODEL

TYPE

ENGINE

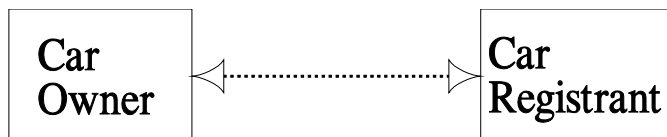
CHASSIS NUMBER

LICENSE NUMBER

YEAR

COLOR

OWNER SOCIAL SECURITY NUMBER (ARRAY)



**Figure 3.71.** Independent logical file. Many to Many relationship.



### **Row Operations**

Retrieval

Find

Add

Delete

### **Relationship Operations**

Connect <file - Name> to <File - Name>

Via <File - Name -1 Column> equal

File - Name - 2 - Column>

### **Combination Operations**

Select <Row - Name>

Ordered by <Field - Name>(s)

Where <Field - Name> Ro....

Modify

**Figure 3.72.** Independent logical file operations.



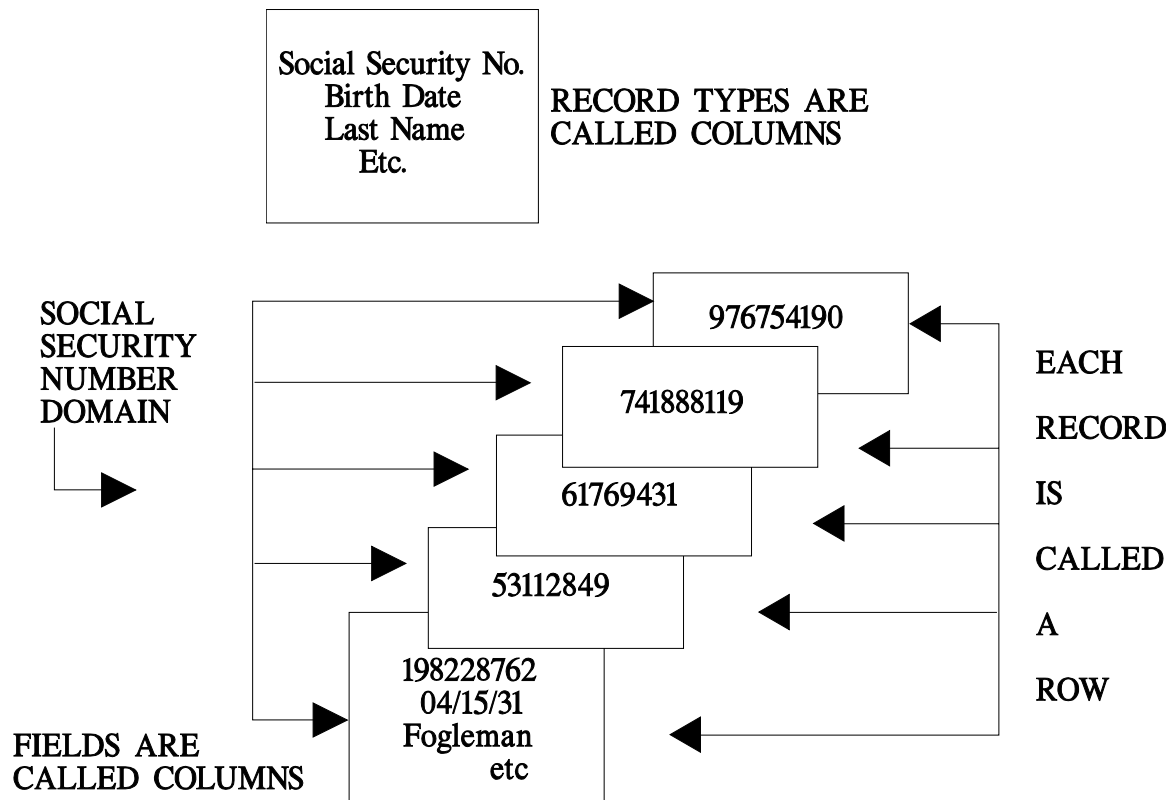
### 3.3.3.2 Relational Data Model

The fourth data model, relational, is the simplest. A relational database is restricted to simple tables called tables. All relationships between tables must be dynamic. For example, in Figure 3.28, the SALESPERSON table is related to the CONTRACT table by the SALESPERSON-ID stored in the CONTRACT data table.

Figure 3.62 shows that the relational model accomplishes all relationship types dynamically. Three relationship types are directly modeled. Two can only be modeled indirectly, and three cannot be modeled.

The terminology commonly utilized in relational databases is identified in Figure 3.73. Figure 3.74 provides an example that supports the use of these different terms.

The operations that are defined for the relational model are listed in Figure 3.75. Examples of each of these operations are provided in Figures 3.42 through 3.53.



**Figure 3.73.** Relational data model terminology.



### 3.3.3.3 Dynamic Data Model Summary

Of the two dynamic data models, the more sophisticated is the ILF model. It has the richest table structure and the largest number of explicit relationships to model real world problems.

Row ID	COLUMNS		
	SSN	BIRTHDATE	LASTNAME
1	123	04/15/31	Fogleman
2	23	.....	.....
3	643	.....	.....

**Figure 3.74.** Relational table columns and rows.

<b>Row Operations</b>
Add
Delete
Modify
Select
Project
<b>Relationship Operations</b>
Product
Union
Intersection
Difference
Join
Divide
<b>Critical Requirement for Some Operations</b>
Union Compatible
Same Degree (Number of Columns)
2 Attributes from Same Domain

**Figure 3.75.** Relational data model operations



### 3.3.4 Data Model Summary

The network data model is far richer than the relational data model. Of the eight different types of relationships that can be defined among tables and between rows, the SQL data model can simulate only three of these relationships, while the NDL data model can directly declare six. The network model can accomplish the other two relationship types by defining extra tables and by programming. Similarly, the ANSI/SQL data model can accomplish the other five relationship types after first defining extra tables, executing views, and through end-user programming. Figures 3.8, 3.9, 3.28 and 3.55 depict the four data models and identify the static and dynamic relationships supported by each.

NDL describes quite sophisticated referential integrity. Because NDL referential integrity is not value-based like SQL, but is based on sets, it is actually more effective. There may be multiple sets between tables of different types, and thus there may be multiple types of NDL referential integrity. DBMSs supporting NDL can thus indicate whether integrity constraints are to be placed on the connection and disconnection of rows to or from relationships in the database. This type of capability prevents deletions posted by a run-unit from actually happening. For example, suppose that a view row is constructed of columns from the CUSTOMER row, the ORDER row, and the ORDER-LINE-ITEM row. If the command to delete occurs, then these DDL-based restrictions prevent the ORDER and CUSTOMER row from being deleted if there are any other relationships between these rows and other database rows.

NDL, unlike SQL, can store rows and defer connecting them to relationships. For example, if there is an on-line order entry system, and if the orders are collected during the day, but the database's financial statistics have to remain unchanged during the whole day to permit consistent financial reporting, the deferral capability prevents the day's order rows from affecting these financial statistics. Not only is this capability useful for retaining consistency, it also lessens I/O and CPU resources consumed during the day, postponing them until a night batch run.

In keeping with NDL's version of referential integrity, NDL can indicate whether the DBMS has cascade delete authority. The default is no cascade delete, and if the option is turned on, then cascade delete is active.

### 3.3.5 ANSI Standard SQL Data Model

SQL 1999 is not just a simple language for defining, accessing and managing tables consisting only of single valued columns of data. With respect to the basic data model capabilities, the SQL language more closely supports the independent logical file data model from the 1960s. It is therefore true to say that SQL standards, starting from SQL 1999 is more of an implementation of the independent logical file data model (e.g., Adabas, Inquire, Datacom/DB, and Sybase) than of the 1970 relational data model.

SQL 1999 has, however, gone way beyond the capabilities of the independent logical file data model by incorporating facilities such as user-defined types, embedded programming language, and libraries of SQL 1999 defined routines for areas like full text management and



spatial data. To say that these SQL 1999 extensions are mere “extended interpretations” of the relational data model is like saying that an intercontinental ballistic missile is merely an “extended interpretation” of a spear.

SQL 1999's impact on network and hierarchical data model DBMSs is significant. Network data model DBMSs have traditionally allowed complex table structures with arrays, groups, repeating groups and nested repeating groups. A very unique characteristic of the SQL 1999 data model is that it now allows arrays. In addition, the columns of the array are able to be outward references to other data. Since the order of the columns in an SQL 1999 array is maintained by the SQL 1999 DBMS, then the array, with its outward references, is essentially a CODASYL set. This is a dramatic departure from the relational data model.

The only remaining and viable network DBMSs are IDMS by Computer Associates and Oracle DBMS (formerly the VAX DBMS). Both have had an SQL language interface for about 10 years. How Computer Associates plans to take advantage of the existing IDMS facilities with SQL 1999 is not known. A significant customer of Oracle's DBMS (formerly Vax DBMS from DEC) is Intel who uses the Consilium manufacturing package to manage computer chip manufacturing. How the Oracle Corporation plans to take advantage of the existing VAX DBMS facilities with SQL 1999 is also not known.

The only two hierarchical DBMSs, System 2000 and IBM's IMS will likely not be impacted at all. System 2000 is no longer being advanced by SAS, and IBM has a full implementation of DB/2 on many different operating systems.

SQL 1999's impact on independent logical file DBMSs, for example, Adabas, Focus, and Datacom/DB is significant. These DBMSs already support many of the SQL 1999 data model facilities. It would seem that these DBMSs could rapidly conform to the new SQL 1999 standard. If these vendors embrace the SQL 1999 model, then these DBMSs could claim conformance sooner than the existing set of relational DBMSs.

Simply stated, the SQL 1999 language defines a unique data model. It contains:

- The ability to model CODASYL sets,
- Many of the natural data clustering features of the hierarchical data model,
- Explicit many-to-many and inferential relationships like the independent logical file data model, and finally,
- The unique ability to directly model recursive relationships.

It therefore can only be said that the SQL 1999 data model is unique unto itself. Clearly, it is not the relational data model, CODASYL network, hierarchical, or independent logical file data models. Simply, SQL 1999 is a data model unto itself.

The figures that follow show the SQL 1999 and beyond (that is, the SQL 2003) data model in terms of its:

- Data Structures



- Relationships
- Operations

These are then compared to the facilities that exist in the network, hierarchical, and independent logical file data models.

### 3.3.5.1 Data Model: Data Structures

Column Data Type	Definition	SQL 1999
Single Value	Each component represents a single value such as <u>Birthdate</u> with the value 11/11/1987	✓
Multi-value	Each component represents multiple values such as <u>Nicknames</u> with values “Buddy, Guy, Mac”	✓-Note 1
Groups	Each component has subcomponents to represent single-set of values such as <u>Address</u> with Street-1, Street-2, City, State, Zip	✓-Note 2
Repeating Groups	Each component has subcomponents to represent multi-sets of values such as <u>Dependents</u> that contains subcomponents, Dependent Name, Dependent Birth date, Dependent SSN.	✓-Note 3
Nested Repeating Groups	Employee (Dependents (Hobbies))	✓-Note 4

**Notes:**

- 1 Arrays as a data type for a column
- 2 ROW data structure of a column
- 3 ROW data structure for a column wherein each Table structure field has the data type, ARRAY
- 4 ROW data structure for a column with contained ARRAYs where each ARRAY column is a ROW data structure with contained ARRAYs where each ARRAY column , etc....





### 3.3.5.2 Data Model: Relationships Background

#### SQL 1999 Relationship Types

Name	Example	SQL 1999
One-to-many	Employee to dependents	✓-Note 1
Owner-multiple-member	Territory contains salesmen and customers	✓-Note 2
Singular-one-member	Top performing employees	✓-Note 1
Singular-multiple-member	Top performing current, former, part-time, and retired employees	✓-Note 3
Recursive	Organization contains organization	✓-Note 4
Many-to-many	Automobiles and owners	✓-Note 5
One-to-one	Table and its primary key	✓-Note 6
Inferential	Many houses each with a location, and then buyer with desired location	✓-Note 7

#### Notes:

- 1 Traditional relational data model (SQL 1986, 1989, & 1992)
- 2 Implemented as a ROW(TerritoryId, Salesman REF (SalesmanId),
- 3 Developed using Subtables where Employees are partitioned off into their common columns (employee) and their unique columns (current, former, part-time, and retired)
- 4 Recursion operations built into the language (WITH RECURSION...)
- 5 Cross joins from within columns of ARRAYS contained as data types of column in different tables
- 6 Effectively as tables and subtables. Most directly with UNIQUE Fkey
- 7 A single valued non-primary key Location within House and same for Buyer



### 3.3.5.3 Data Model: Operations Background

#### SQL 1999 Operations

- **Row Operations** – traditional insert, delete, and modify of rows and columns within rows
- **Relationship Operations** – traditional operations that affect the relationships between rows of the same or different tables

Row Operations			
Operation	Static	Dynamic	SQL 1999
<b>Find</b>	SELECT According to STORED Order	SELECT and PUT into DML Specified Order	✓
<b>Get</b>	Obtain Row From Find	Ditto	✓
<b>Add</b>	Install a New Row Into Database	Ditto	✓
<b>Delete</b>	Remove an Existing Row From Database	Ditto	✓
<b>Modify</b>	Change Some Column Values in Existing Row	Ditto	✓

Relationship Operations			
Operation	Static	Dynamic	SQL 1999
<b>Connect</b>	Add to a Named RELATIONSHIP in Specific Order	N/A	✓-Note 1
<b>Disconnect</b>	Delete From RELATIONSHIP	N/A	✓-Note 2
<b>Get Owner</b>	Obtains The Parent of the Row That is Current	N/A	No
<b>Get Member</b>	Obtains the First Child of the Owner For the Named Relationship	N/A	✓-Note 3
<b>Get next</b>	Obtains the Next Row Within The Named Relationship	N/A	✓-Note 4



<b>Intersect</b>	N/A	Find and Keep Only the Common	✓
<b>Difference</b>	N/A	Find and Keep Only the Not Common	✓
<b>Join</b>	N/A	“Append” Relations to Each Other	✓
<b>Divide</b>	N/A	Subset	✓
<b>Product</b>	N/A	Cross-Product	✓
<b>Union</b>	N/A	Merge and Drop Duplicates	✓

### SQL 1999 Relationship Operations(cont.)

Notes:

1	Value an Column in an ARRAY. Data type of Array is REF type pointing elsewhere.
2	Set value of column in array to null
3	Get the first column of an array. Data type of Array is REF type pointing elsewhere
4	Get next column in array. Data type of Array is REF type pointing elsewhere

## 3.4 Data Definition Language

The language that communicates the logical database to the DBMS is called the data definition language (DDL). Once the DBMS successfully processes these language clauses, it produces the database's schema. The definition process itself and the specification of the tables and the columns are the same for both static and dynamic relationship DBMSs. The DDL tends to be tailored to the DBMS's data model, regardless of whether its tables are simple or complex, or whether its relationship mechanism is static or dynamic.

A static relationship DBMS's DDL typically contains more editing and validation clauses that affect multiple tables than would a dynamic relationship DBMS's DDL. In a dynamic relationship DBMS these editing and validation clauses relate only to a single table. Some dynamic relationship DBMSs allow for a central schema of multiple, dynamically connected



tables. In such a case, the dynamic relationship DBMS DDL becomes more like its static relationship DBMS counterpart.

The data definition language has traditionally been syntax oriented. There is no reason, however, why it can not be represented through interactive screen displays that accomplish the correct syntax. Such a facility exists for FOCUS, and is called FILETALK. A DDL typically contains the syntactic specifications to support:

- Schemas
- Domains
- Rows
- Columns
- Relationships

Figure 3.76 illustrates the DDL for an ANSI NDL database. Figures 3.77 and 3.78 show DDLs for IMS and SYSTEM 2000. Figure 3.79 contains the DDL for a FOCUS schema. Figure 3.80 contains the DDL for an SQL database.

The data definition language interface enables those who define database structures to communicate those structures to the DBMS. The database structure consists of clauses for tables, columns, and relationships (static only). The table clauses include the table's name, primary keys, foreign keys, and column clauses. The column clauses indicate whether the columns are single valued, multiple valued, groups, or repeating groups. The column clauses also state the data types, and any edit and validation rules. The final set of clauses define the relationships, that is, single or multiple member, one to one, many to many, recursive, inferential, etc. SQL referential integrity clauses are defined within the CREATE TABLE statements. In NDL, referential integrity is defined in the NDL SET clauses, which define NDL relationships.

The quantity and sophistication the data definition language clauses reflect are inverse to the amount of effort required to develop database applications. The fewer and less sophisticated the clauses, the larger the design and programming effort. The greater the quantity and sophistication of the clauses of the data definition language, the smaller the design and programming effort. Given that COBOL programs can have a life cycle cost of 100,000 to 300,000 dollars, it is easy to see that increased capabilities can save hundreds of thousands of dollars in a very short time.

Alongside definitions of tables, columns, and relationships must be the data integrity clauses. Data integrity clauses are rules and regulations just like other clauses in that they control operations on the database. Integrity clauses are of two types: procedure-oriented and value-oriented.

Procedure-oriented clauses enable the execution of preprogrammed logic that serves to accept or reject column values, row adds/deletes/modifications, or invoke other programs that cause actions to happen such as reorders. Value-oriented clauses are used to verify that only allowable values are stored in the database.

It is very important for a DBMS to have integrity facilities. Integrity facilities increase programmer productivity by permitting the use of natural languages over COBOL for updating. Even if COBOL were as easy to use as a natural language, integrity facilities would still be important, because they permit all the clauses to be defined and maintained in the database's



schema rather than in each update program. This makes it unnecessary to find, modify, recompile, and retest many COBOL programs every time a data integrity rule is changed.

```
SCHEMA NAME IS FACULTY - DATA

RECORD NAME IS FACULTY
  DUPLICATES ARE NOT ALLOWED
  NAME TYPE IS CHARACTER 25
  ADDRESS TYPE IS CHARACTER 40
  SSN TYPE IS CHARACTER 11

RECORD NAME IS JOBHIST
  JOBTITLE TYPE IS CHARACTER 25
  JOBCODE TYPE IS FIXED DECIMAL 2

SET NAME IS FAC - JOBHIST
  ORDER IS SORTED
  OWNER IS FACULTY
  MEMBER IS HOBHISST
    AUTOMATIC MANDATORY
    ASCENDING KEY IS JOBCODE
    DUPLICATES ARE ALLOWED
```

**Figure 3.76.** ANSI/NDL data model data definition language



```
1  dbd      name=ecudpdbd
2  segm     name=course, bytes=256
3      field name= (coursenbr, seq), bytes=3, start=1
4      field name= title, bytes=33, state =4
5      field name=descripn, bytes=220, start=37
6  segm name=prereq, parent=course, bytes=36
7      field name=(coursenbr,seq), bytes=3, start=1
8      field name=title, bytes=33, start=4
9  segm name=offering, Parent=course, bytes=20
10     field name=(date,seq,m), bytes=6, start=1
11     field name=location, bytes=2, start=19
12     field name=format, bytes=2, start=19
13  segm name=teacher, parent=offering, bytes=24
14     field name=(empnbr, seq), bytes=6, start=1
15     field name=name, bytes=18, start=7
16  segm name=student, parent=offering, bytes=25
17     field name=(empnbr,seq), bytes=6, start=1
18     field name=name, bytes=18, start=7
19     field name=grade, bytes=1, start=25
```

**Figure 3.77.** Hierarchical data model data definition language (IBM's IMS)



## STATIC HIERARCHY (SYSTEM 2000)

- 1\* COURSE TITLE (INT 9(6))
- 2\* COURSE DESCRIPTION (NAME X (25))
- 3\* PREREQUISITES (RG)
  - 4\* PREREQ COURSE NBR (INT 9 (6) IN 3 )
  - 5\* PREREQ COURSE TITLE (INT 9 (6) IN 3)
- 5\* OFFERING (RG)
  - 6\* SECTION NBR (INT 99 IN 5)
  - 7\* DATE (DAYS OF WEEK (NAME X (12) IN 5)
- 8\* TEACHERS (RG IN 5)
  - 9\* TEACHER NBR (INT 9 (9) IN 8)
- 10\* STUDENTS (RG IN 5)
  - 11\* STUDENT ID (INT X (9) IN 10)
  - 12\* FINAL GRADE (NAME XX IN 10)

**Figure 3.78.** Hierarchical data model data definition language (System 2000)

```

FILENAME= TEACHER, SUFFIX=FOC
SEGNAME=TEACHER, SEGTYPE=S1
FIELDNAME=T-SSN, ALIAS=FORMAT=I9, FIELDTYPE=I,$
FIELDNAME=T-NAME,ALIAS=, FORMAT=A30,$

FILENAME=COURSE,SUFFIX=FOC
SEGNAME=COURSE, SEGTYPE=S1
FIELDNAME=COURSE-NBR, ALIAS=, FORMAT=I5,FIELD=I,$
FIELDNAME=COURSE-NAME, ALIAS=, FORMAT=A30, $

FILENAME=COURSECT, SUFFIX=FOC
SEGNAME=COURSECT, SEGTYPE=S3
FIELDNAME=COURSE-NBR, ALIAS=, FORMAT=I5, FIELDTYPE=I, $
FIELDNAME=SECTION-NB, ALIAS=, FORMAT=I2, $
FIELDNAME=T-SSN, ALIAS=, FORMAT=I9, FIELDTYPE=I, $

```

**Figure 3.79.** Independent logical file data model DDL, IBI's Focus.

```
CREATE TABLE STUDENT
  ( STUDENT ID  CHAR (5) NOT NULL
    SNAME CHAR (20),
    STATUS SMALLINT,
    MAJOR CHAR (15) );

CREATE TABLE TEACHER
  ( TEACHER# CHAR (6) NOT NULL
    TEACHER_NAME CHAR (20),
    TEACHER_DEGREE CHAR (5) );

CREATE TABLE COURSESECTION
  ( COURSE# SMALLINT NOT NULL
    SECTION# SMALLINT NOT NULL
    YEAR SMALLINT NOT NULL
    SEMESTER SMALLINT NOT NULL
    TEACHER# CHAR (6) );

CREATE TABLE ENROLLMENT
  ( COURSE# SMALLINT NOT NULL,
    SECTION# SMALLINT NOT NULL,
    STUDENT# SMALLINT NOT NULL,
    YEAR SMALLINT NOT NULL
    SEMESTER SMALLINT NOT NULL
    GRADE_RECEIVED CHAR (1) );
```

**Figure 3.80.** SQL Data definition language.

### 3.5 Logical Database Summary

The logical database is the expression of the database's data organization. The language that communicates this data organization to the DBMS is called the data definition language (DDL). The two different relationship binding mechanisms are static and dynamic. Within the static mechanism, the two data models are network and hierarchical. Within the dynamic mechanism, the two data models are independent logical file and relational.

Ideally, every DBMS should support the maximum number of relationships allowable within its data model. As stated above, these relationships are:

- One-to-many
- Owner-multiple member
- Singular-one member
- Singular-multiple-member
- Recursive
- Many-to-many





- One-to-one
- Inferential

There should be no restriction on the inherent properties of tables. Any table should be capable of being both an owner of one table, and a member of another. If a table can be only an owner or a member, then additional tables must be created to make up for the DBMS's inability to have a table serve both roles. In addition to the additional tables, there will also be additional disk space, additional I/Os for reading and writing, and in the case of a DBMS or database failure, additional resources for recovery.

Integrity constraints are critical to quality database. These constraints can be defined for the single column, between columns within the same table, and between columns in different tables. Even though an integrity constraint between columns from different tables is an interrow relationship, it must be noted that the value is not the basis of the relationship. It serves as an additional integrity constraint to make sure, for example, that districts are related to their regions.

Just as there must be integrity procedures for columns and tables, there must be integrity procedures for relationships. These procedures or formulas are executed whenever a relationship operation is executed, and they determine the success or failure of the relationship operation. Further, they reduce the time spent on creating applications programs, and eliminate the need to make updates when users perform illegal operations.

All the integrity procedures should be written in either a compiler language or a natural language. Further, the invocation of a procedure should be for a specific purpose, so that the procedure matching the specific purpose can be changed as needed.

In summary, the network (ANSI/NDL) data model is a richer data model than is hierarchical, and the ILF data model is a richer model than is the relational (ANSI/SQL) model. However, with the SQL/1999 standard, the SQL language defines its own data model which is clearly not relational. It includes features from the network, hierarchical, and independent logical file data models. In its most trivial form, SQL/1999's data model is relational..

If an organization wishes to procure DBMSs that enhance portability, it should procure DBMSs that have languages conforming to ANSI/NDL and ANSI/SQL. Most ILF DBMSs can be restricted to operate like relational DBMSs. Thus, procuring an ILF DBMS and restricting its databases to relational structures for portable applications, but allowing complex tables for non-portable applications, is a way to have portability when it is needed and data model richness otherwise. In short, buying an ANSI/NDL data model DBMS, as well as an ILF data model DBMS that can optionally conform to ANSI/SQL, is clearly the best choice.





## 4

# THE PHYSICAL DATABASE

### 4.1 Physical Database Components

The physical database, the second component of a DBMS, involves the creation and maintenance of the actual database. The preceding chapter, the Logical Database is about the database's logical structure, that is, its tables, columns, and relationships. In essence, the *logical database* is the database's *blueprint* and the *physical database* is the database's *construction*.

This chapter addresses the DBMS's and the computer's view of these components, and the techniques the DBMS employs to minimize both access times and mass storage use. This chapter then is a study in architecture and techniques, none of which have been--nor should ever be--standardized by ANSI. While the ANSI data models identify DBMS commonality, that is, data model and data manipulation language interfaces, physical database is a DBMS discriminator that sets apart the DBMS vendor's investment in DBMS performance and mass storage efficiency.

Generally speaking, the physical database involves much more than just the layout of the operating system files that store rows. It also covers row access strategy, data loading, row update effects and capabilities, and database maintenance (backup).

The physical database coincides with many of the aspects of the ANSI/SPARC internal schema. The edited version of the ANSI/SPARC report describes it as

specifications of the internal objects, indices, pointers, and other implementation mechanisms, other parameters affecting (optimizing) the economics of internal data storage, integrity, security, recovery, and administrative matters. (1)

Because this book was originally written 15 years after the ANSI/SPARC draft report, it has a more refined view of the technology of database. Today, 2006, this continues to be true. Thus certain aspects of the internal schema, e.g., integrity, security, and recovery are addressed in the system control chapter.

### 4.2 Storage Structure

The storage structure of a database consists of four parts: dictionary, indexes, relationships, and data. This architecture, like the logical database's architecture, is communicated to the DBMS through a language that is often called the data storage definition language (DSDL). The critical differences among static and dynamic storage structures are presented in Figure 4.1

The dictionary is the repository of the information represented by the compiled DDL. Dictionaries contain many types of information, such as validation rules and extra space lists.



STORAGE STRUCTURE		
Storage Structure Component	STATIC	DYNAMIC
	Often multiple component physical files Typically dictionary, indexes, relationships, and data	Often single component physical files  Typically dictionary, indexes, and data
ACCESS STRATEGY	Primary key and relationship searching via row processing	High number of indexes Dynamic matching/ merging of row extracts via column values
DATA LOADING	Complete logical rows Careful Planning Exact Placement Large Volumes All or Nothing	Table by table Load what you have Incremental building
DATA UPDATE	Very careful far reaching effects HLI only Periodic database reorganization	Casual, add new rows, columns at will Seldom needs reorganization
DATABASE MANAGEMENT	Usually one or a few databases Global save/restore at high level	Many storage structures Careful planning Many commands at low levels

**Figure 4.1.** Physical database: static and dynamic relationship comparison.

The DBMS creates the dictionary upon successful processing of the DDL. It is automatically maintained by the DBMS during various processes. Some DBMSs store the dictionary data for each database in a common *super* dictionary, that is, an IRDS.

Indexes are mechanisms for fast access. They exist either as primary indexes (unique value guaranteed) or secondary indexes (repeated values allowed). Indexes usually point either to the relationship instances that exist distinct and separate from the rows (static only), or to the actual row (both static and dynamic). In most DBMSs, defined indexes are automatically created during database loading, and are automatically maintained during update.

Relationships are the connections among rows. If a relationship is static, then it is usually a relative row address that is stored in the owner row (pointing to the member) or the member row (pointing back to the owner row and/or to the next and/or prior row). If the relationship is dynamic, then the data value that is uniquely contained in the owner row is replicated in many



member rows. In a static relationship DBMS, relationships are created automatically according to row loading sequences and special data loading verbs. Subsequent to database loading, the static relationship DBMS relationships are maintained through special user language verbs. In a dynamic relationship DBMS, relationships among rows exist implicitly as shared data values, and are maintained through column value changes.

The last part of the storage structure is the row. Its storage structure design can have several tables for each operating system file, or only one table in each operating system file. This component is created and maintained as rows are stored during loads, updates, or deletes.

The sophistication of a DBMS's storage structure indicates the power of the DBMS for certain applications. Complex storage structures boost the efficiency of one particular kind of report processing over another. The simple storage structure, on the other hand, is ready for any interrogation that comes along; that is, it has no structural orientation toward any particular type of report. For applications in which no type of report predominates, simplicity is highly desirable. But if an application has a large number of tables and a great volume of data, a simple storage structure will probably not be able to handle efficiently a report that involves complex relationships and report ordering. For an application in which certain reports are executed frequently, a storage structure favoring these reports is highly desirable.

#### 4.2.1 Dictionary Component

The component that separates an ordinary data file from a database is the dictionary. In traditional applications, the knowledge of the organization of a file is stored in the application programs. If there are fifty such programs, then all fifty have to have the same information. A natural outgrowth of this was to push all this information into a single routine and to then have all the other routines invoke it. When this information was removed from the routine and stored in a stand alone file, database was born. Since this information addressed only one file, it was referred to as file management. When databases grew to be able to handle many such files, the dictionary became both more complex and more useful. Data files and the various data types and relationships that existed between columns could be reviewed at a glance.

Over time, the dictionary has grown to encompass much more than just column names. Included now are column editing and validation rules. Included also are triggers that call other programs to accomplish DBMS-controlled processing on a row before it is stored, modified, deleted, or retrieved.

An organization that has a DBMS installed for several years is likely to have developed fifty or more databases, each with its own dictionary. And, just as the file information was redundant when stored in many programs, so too is the individual database dictionary information redundant when stored in many databases. A sophisticated DBMS allows this information to be *lifted* from each individual database and stored in another repository that functions as a super-dictionary. ANSI's H4 committee has standardized the super dictionary, calling it the Information Resource Dictionary System (IRDS).

When data dictionaries were first developed, distinct systems arose. Each had to be specially defined, loaded, reported, and updated. Because this work was redundant to DBMS data definition work, these separate data dictionary systems have faded. The scope of today's



DBMS integrated data dictionary systems has enlarged from being merely a passive repository of data structures to being a *command and control* center for multiple databases. Because of this scope change, each DBMS vendor created its own version of a data dictionary and integrated it into the DBMS software. Because it is integrated, it is able to be active, automatically recording in the IRDS any changes to the individual databases.

The standardization of the data dictionary has been completed by the ANSI committee H4, and was confirmed as an ANSI standard in early 1989. The facility is not restricted in its contents. The H4 IRDS can store data about database's design, the DBMS, and database applications. Data about a database's design is called metadata.

There are really no clear distinctions between static relationship DBMS data dictionaries and dynamic relationship DBMS data dictionaries. Each type of dictionary and each DBMS vendor's dictionary seem to have the same common fundamental information, such as column and table names, expository definitions, and their usage in various programs, providing, of course, that the programs have been properly compiled. In addition to that basic set of common data, all data dictionaries contain other facilities required by the specific data model, or required for fully defining and controlling database (not DBMS) environments.

To have a complete and fully functional database environment requires both a passive IRDS that contains all the application's specification metadata, and one or more active DBMS-IRDS that contains all the applications' implementation, operational, and maintenance metadata.

#### 4.2.2 Index Component

An index is a data value based access path to one or more rows. Some indexes are primary and others are secondary. A primary index is most often used to locate a single row. A secondary index locates one or more rows.

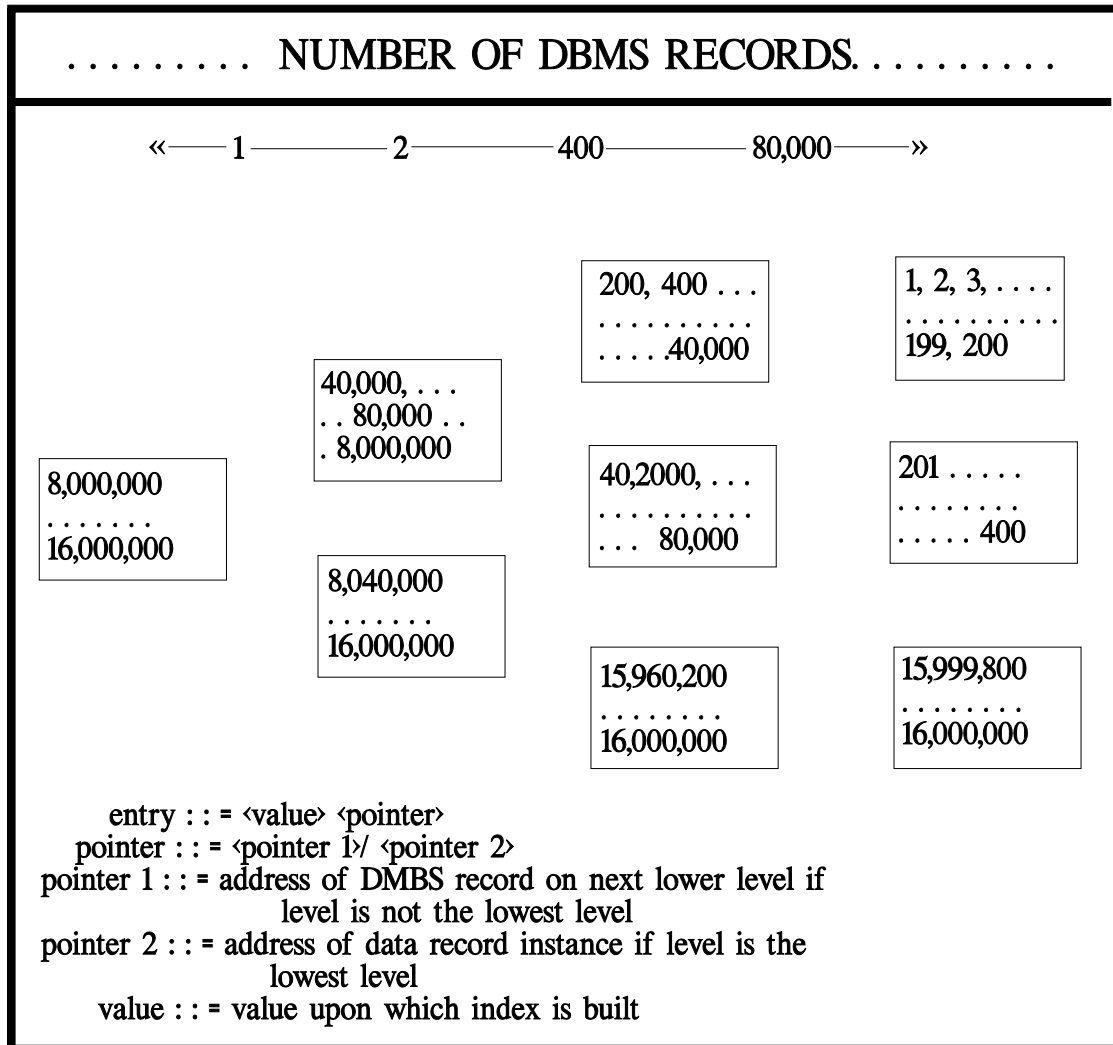
It is commonly thought that primary indexes must have unique values and that secondary indexes must have non-unique values. A company might, however, have its own employee identification number that is used to identify employees and to store and find rows in a database. The fact that the employee also has a social security number (SSN) and that the SSN is used to access an employee whenever it is the only number known, does not make the SSN a primary key. In the terminology of the previous chapter, the SSN is called a candidate key. Since most DBMSs do not have a candidate key type, the only way to achieve its effects is to declare it an index with a unique value integrity constraint.

Primary indexes are also sometimes used along with secondary indexes to provide row *addresses* whenever a secondary key search needs to locate rows for a column value that is not unique. For example, the query `FIND EMPLOYEES WHERE JOB CODE EQ PROGRAMMER` is likely to turn up several employees, and the list of row addresses resulting from the query would usually consist of the primary key of each employee's table. Two typical ways of organizing primary key indexes are hierarchies and hashing (see Figures 4.2 and 4.3).

A problem arises whenever the DBMS actually uses a row's primary key value to compute the storage address assigned to the row by the database. The problem is that, whenever the row's primary key value has to change, then the DBMS must perform the following actions:



- The old primary key value must be used to retrieve the row from its old location.
- The row must be deleted from its old storage location.
- The new primary key value must be used to discover the new physical location for the row.
- The row must be stored in the database at its new location.



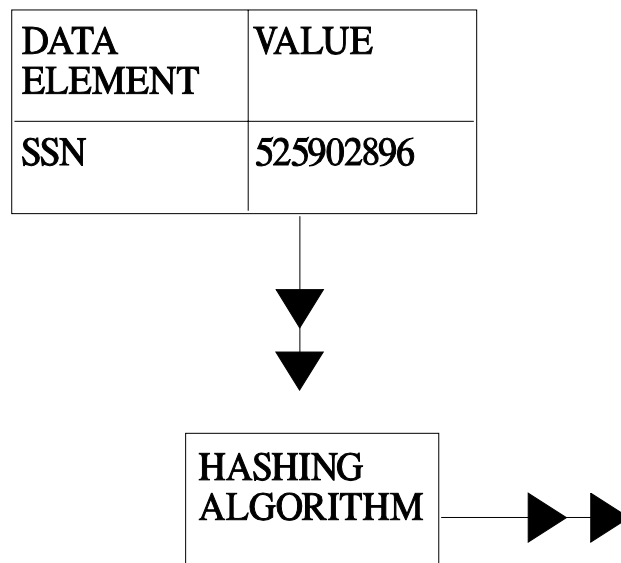
Hierarchical Index Organization  
 200 index component entries per DBMS record  
 16,000,000 Records Represented in Four Levels

**Figure 4.2** Hierarchical Index Organization



Figure 4.3 illustrates this problem when the actual value of the primary key determines the storage location of the rows. If the value of the primary key changes, then the row has to be located, retrieved, deleted, relocated, and then re-stored.

Furthermore, if that row contains several columns that are indexed, and if the secondary key value entry in the index table is composed of a concatenation of the secondary key value and the primary key value, then whenever the primary key value is changed, the secondary key value entries in the index would also have to be adjusted. While the DBMS performs all these locate, delete, relocate, and re-store operations automatically, computer resources are still consumed, and since rows are *moving* around in the database, there is an increased window for database damage should some DBMS, system software (O/S), or hardware failure occur.



**Note:** RRA means relative record address

**Figure 4.3.** Hash based primary key index organization.





Figure 4.4 illustrates the construction of an index. The column column identifies the name of the indexed column. The unique value column identifies the unique values associated with each of the two columns. In the case of SKILL, the unique values are CODER, DBA, . . . , SYSTEMS ANALYST. The multiple occurrence column contains an array of primary key column values (e.g., PK1, . . . , PK109). In this example, the primary key values are EMPLOYEE-IDs. If the value of an EMPLOYEE-ID is changed, then not only must the primary key value of the actual EMPLOYEE row change, but also the index entries for the row's SEX and SKILL column values.

DATA ELEMENT	INDEX		DATA
	UNIQUE VALUE	MULTIPLE OCCURRENCES	
SEX	M	PK1, . . PK10, . . , PK109	PK 107
	F	PK100, PK111, . . . PK107	MARY LIBRARIAN F
SKILL	CODER	PK1, PK101, PK184 ..... PK2, PK107, PK384 ..... PK248, PK109, . . . , 218 PK1008, PK9982	MORE DATA
	DBA		
	LIBRARIAN		
	MANAGER		PK 109
	PROGRAMMER		JOHN PROGRAMMER M
	SYSTEMS ANALYST		

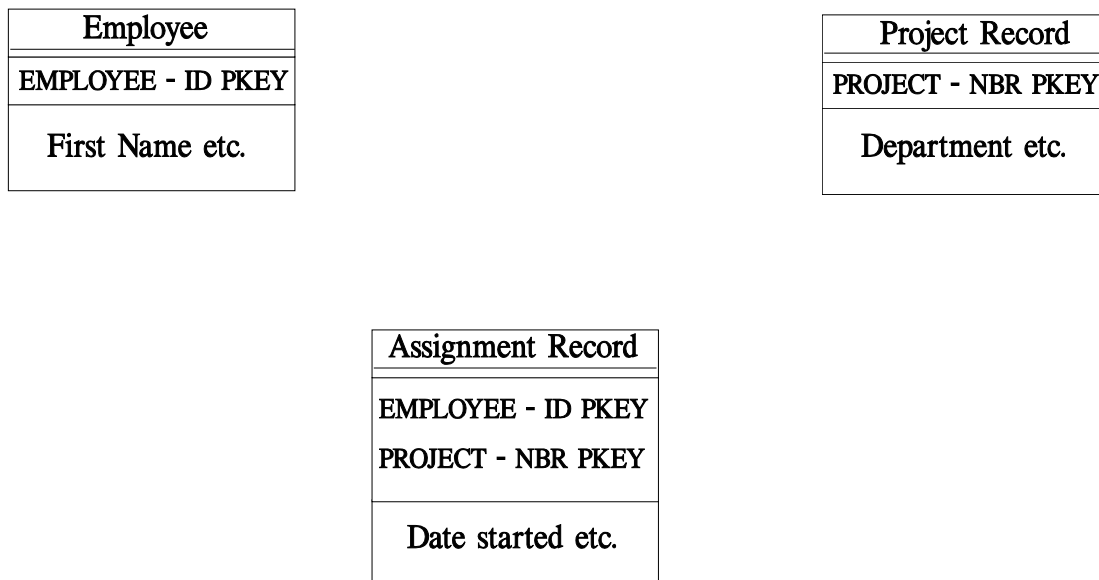
Note: PK means primary key, and PK1 means the first primary key.

**Figure 4.4** Multiple Key Access (static or dynamic)



As a further complication of this problem, suppose there is a table called PROJECT-ASSIGNMENT (see Figure 4.5) that is used to identify the employees assigned to specific projects. The primary key of this table is likely the combination of the EMPLOYEE-ID and the PROJECT-NUMBER. If, as in the example above, the column value for EMPLOYEE-ID is changed, then in addition to the changes cited above, every PROJECT-ASSIGNMENT row involving the employee has to be selected, retrieved, deleted, updated, re-selected, and then re-stored. Of course, if any of the columns in the PROJECT-ASSIGNMENT are indexed, then these index structures also have to be changed.

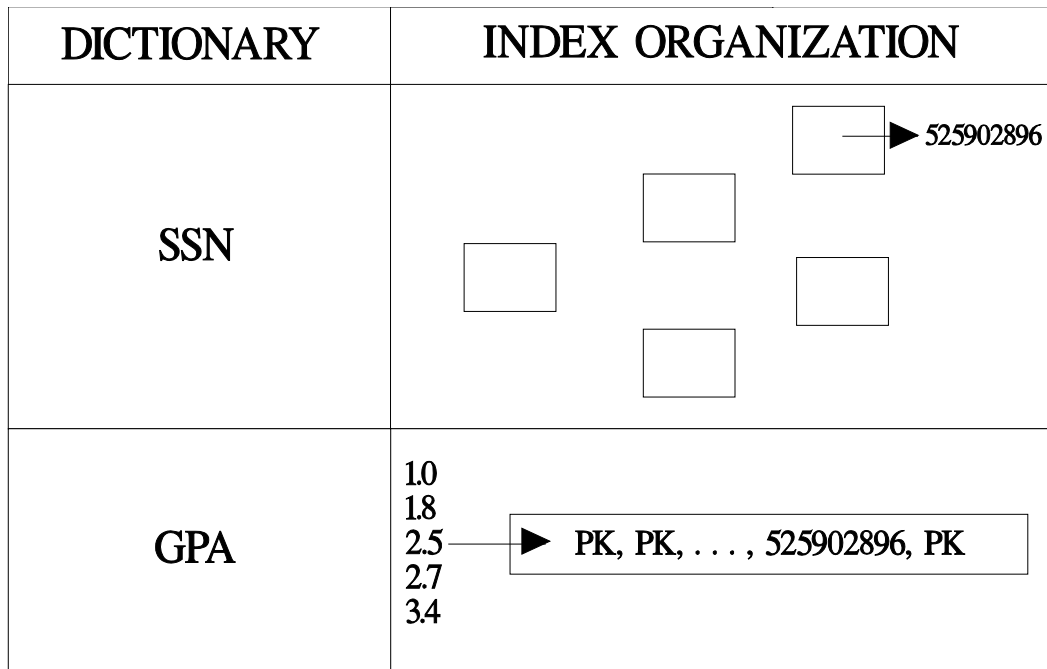
To avoid such a situation, some DBMSs use the user-defined primary key only as a logical identifier, and translating it through a table look-up into a permanent DBMS-generated physical identifier. This physical identifier is then independent of user value changes. Figure 4.6 illustrates the use of this indirect technique. The indexes on the SSN and GPA columns both point to the primary key value of <525902896>. In the translator part of the storage structure (lower left), the <525902896> value points to a DBMS row address of <1000>. This in turn points to the actual internal row location <200> (within DBMS row 1000) of the actual row. The other problem of having EMPLOYEE rows that have the wrong EMPLOYEE-ID value still has to be addressed, but at least none of the rows have to be deleted and re-stored. An additional



**Figure 4.5.** Project assignment.



benefit resulting from this logical-to-physical key lookup table is that none of the column index structures associated with any affected rows have to be updated just because the EMPLOYEE-ID primary key of the employee table is changed. Furthermore, DBMSs designed in this way do not require primary keys at all.



Associator: Organized By Primary Key Value	
RECORD	
KEY	ADDRESS
149248819	123
219194187	512
325902896	250
423134781	183
525902890	1000

1000 DATA AREA			
			200 525902896
KEY AREA			
⌊PK⌋ ⌊RA⌋		⌊PK⌋ ⌊RA⌋	
⌊PK⌋ ⌊RA⌋		⌊PK⌋ ⌊RA⌋	
⌊525902896⌋ ⌊200⌋			

**Figure 4.6.** Indirect index organization for a student database.



Clearly, indexes are far more complicated than they first appear. The quality of the design of a DBMS's indexes and the manner in which they are used by the DBMS, regardless of the DBMS's data model, dramatically affects the types of applications that can be handled satisfactorily.

#### 4.2.2.1 Index Alternatives

This section presents the essential components of a case study from an organization that has a real need for sophisticated indexing. For the sake of simplicity, this case study assumes that each computer access consumes one I/O, and that each DBMS row is 6000 bytes long. Any improvements due to blocking, in-memory hits, and the like are not factored in, except where noted. Also, for the sake of simplicity, all calculations are approximate and serve only to illustrate differences in index constructions and strategies.

The company services both central office and field offices. The field offices obtain orders and enter them into a centralized system. Daily reports are produced for each field office, its sales staff, and for various headquarters sales support organizations. Additionally, the order entry data is allocated to the various types of product categories so that the product managers can keep track of their products. Comparisons are needed by different time periods, namely, by seasons, quarters, and days. For the sales management staff, comparisons are needed by different sales regions, by districts, by territories, and by sales staff members. For the marketing research staff, demographic comparisons such as the average sales by product to persons over 18 who are living in rural areas, or small towns, or large cities are needed so that product penetration can be determined and marketing dollars can be wisely spent. Clearly, the problem involves much more than generating invoices.

To handle such a complicated problem, DBMS vendors trumpet indexes to facilitate quick access of different sets of rows based on a column's value.

An index is a value-based relationship among a set of rows. When a WHERE clause is constructed for use in any of a DBMS's languages (for example, host language interface or query), the programmer must determine the relative benefits derived from including only one column indexed condition, several such conditions, or a mixture of indexed and non-indexed conditions.

Almost all DBMSs can improve processing speed by using indexes. However, some DBMSs can only use one index within a search clause to improve performance while other DBMSs can use multiple indexes and achieve even greater processing speeds. The first type of DBMS is termed a row processor, and the second type is termed a list processor.

In row processing DBMSs, if a WHERE clause contains two conditions, both involving indexed columns, the DBMS only employs one index to help find a set of rows. The DBMS then searches those rows to find the net set of rows possessing the value from the other condition. In the case study, there are indexes for REGION and another for PRODUCT, and a query is to determine the sales for a given product within a specific region. A row processing DBMS can only take advantage one of the indexed columns (for example, REGION) to retrieve all the invoice rows for orders within that REGION. The row processing DBMS then searches these rows for the specific PRODUCT. The row processing DBMS functions just as if PRODUCT is



not indexed at all. A list of some of the popular row processing, *IBM* mainframe DBMSs by data model is provided in Figure 4.7.

List processing DBMSs, in contrast, improve processing speed by employing more than one indexed column in the *WHERE* clause. Typically, when several indexed conditions are included, each is processed separately, for each selection condition. These lists are then combined appropriately (*AND* or *OR*), resulting in a final list of row identifiers. In the example, the list processing DBMS determines one list of rows for *REGION* and another list of rows for *PRODUCT*. These two lists are then *ANDed* together to determine the list of row identifiers that represent the set of invoices meeting both conditions. A list of some of the popular list processing, *IBM* DBMSs is presented in Figure 4.7.

Regardless of whether a DBMS is a list processor or a row processor, if one or more conditions included in the *WHERE* clause involve a non-indexed column, the rows acceptable to these non-indexed conditions are determined as follows. If the non-indexed condition is joined to the other conditions with an *AND*, then the selection is against only the index-selected rows. If the non-indexed condition is joined to the other conditions with an *OR*, then the rows not passing the index tests also have to be searched; but because this means searching all rows, all index benefits are lost.

The organization in the case study needs to perform interrogations on the order data from three different perspectives:

- Marketing research
- Headquarters sales
- Field sales

The marketing researcher needs to compare the sales of various items of different brands and their four different promotion variants, all of which were offered in different areas of the country for the past five Easters. Of special interest is whether various trade promotions and mass advertising campaigns affected the sales in three of the five areas. Further, since two of these promotions were price discount coupons for different products, did these coupons affect the sale of these products? Did all of these promotions affect the sales of similar items of the competitors? And did the promotion affect the sales of the organization's existing items?

Type of Processing	STATIC		DYNAMIC	
	Network	Hierarchical	ILF	Relational
Row	IDMS/R SUPRA	IMS	FOCUS	DB-2 IDMS/R SUPRA
List		SYSTEM 2000	ADABAS M-204	

**Figure 4.7.** Popular list of row processing DBMSs.



In contrast, a headquarters sales staffer wants to identify the various regions, districts, and territories that are not performing as well as others. Additionally, the staffer performs intense research on the best and worst districts to determine differences in sales managers, salespersons, and in the demographics between these best and worst sales districts. Further, he wants an alert report each Monday of all districts that are performing at a rate significantly different from last month's, or last year's, or whenever.

As for the third perspective, the salesperson in the field wants to plan the next week's sales calls. Clearly, if there is a push from the home office to make sales on a particular item, the salesperson needs to know the ranked order of the customers for that item during this time period for the last few weeks, months, and years. Armed with that information, the salesperson can visit the high probability customers first. And, before visiting these clients, the salesperson has to know which orders were placed recently, which orders were being delivered next week, and which orders are back ordered to assess the client's disposition.

#### **4.2.2.2 Problem Specification**

Clearly, the data needs of the case study company are quite complex. The marketing researcher needs data for different items for different areas of the country during different calendar weeks (Easter is not on the same day every year). The researcher also needs competitive data for the same periods to determine whether any gain or loss of sales is different from the competition's gains or losses. Sales data is needed for the areas of the country that both did and did not offer promotions so that the results can be compared. Promotional cost data is needed to determine the real gains or losses. Finally, the sales data shows whether the destinations of the items are within the areas that are measured for gain or loss.

The second user, the headquarters sales manager, needs data by the various sales organizational units across time. Furthermore, norms need to be developed so that the sales units that deviate from the norms can be easily identified and reported to both headquarters.

The individual salesperson, finally, needs to be able to review the order data for distinct clients in order to plan sales trips. This data is needed at the order level, customer level, and item level (for special sales and promotions).

The amount of data available is enormous. The granularity of the data is at the order and line item level, and is organized by client, by the various types of selling organizations, by various product configurations, and so on. The data is longitudinal to accurately track various types of promotions, brands, and customers over a number of years. Of course, any other changes in the product have to be considered, such as price changes or weight changes. And lastly, it is necessary to consider outside factors such as fluctuations in the economy.

The problems cited above are just some of the many types of problems that need to be solved in this typical business application. Furthermore, the data resulting from solving any one of these problems is sure to lead to other new problems. Questions produce answers that uncover new questions. In any analysis, once the basic data is found, it should be easy to feed it to statistical processors, graphics packages, and the like for analysis and refinement.

If the company in the case study had only about 25 items, one or two hundred customers, several orders for each customer, and needed only to research the past several month's data, then



the solution to a problem like this would fall within any DBMS's capabilities. There are DBMSs that can store and retrieve the data, and follow-on processors that can accomplish the statistical analysis. Finding solutions to these problems should take no longer than a few minutes to, at most, a few hours. The case study company, however, has about 278,000 line items from within 27,000 orders per month and the company has determined that it needs five years (60 months) of this data. Multiplied, that is about 16.7 million line item rows. The space required is 1.1 billion characters.

It is often said that 10% of the data is used 90% of the time and 90% of the data is used 10% of the time. In terms of report needs, this means that only a small amount of the data from the line items is needed for most interrogations. The set of often-used data is depicted in Figure 4.8. The first columns are part of the primary key (PK(1) to PK(3)). Appendix A contains a more detailed explanation of this key notation. Each column is also a foreign key (FK(n)) to another table. To help understand the data distribution, the quantity of unique values over the 16.7 million rows is provided. Finally, a picture clause indicates the type of data for each column.

To assist in appreciating the relative performance differences in the various indexing alternatives, a simple computer performance model was constructed. The computer used for the calculations is a mid-1980s, large IBM mainframe, with a throughput rate of about 81 rows per wall clock second for serial searches, and about 50 rows per second for random searches. These through put rates were determined over a number of months of actual observations during the times of day that the case study computer system was operating, and are therefore quite realistic given the computer's configuration and all the other applications competing for service. Since this is now 20 years later, all these velocities have changed. Notwithstanding, the overall process and methodology holds.

#### **4.2.2.3 Initial Solution**

The company began its first attempt to solve the problem by calculating how long it would take to serially sweep all the order-item rows in order to select a specific subset (see Figure 4.8). The calculation was performed as follows: 16,700,000 logical rows / 81 logical rows per second = about 206,172 seconds, which comes to 57.27 hours. Since the blocking factor on the file was about 12, the rate for reading DBMS (physical) rows was determined to be about 7 per second. Not surprisingly, the organization decided that a 57-hour response time to an on-line interrogation was unacceptable. In response, they tried to develop a manageable number of statistical summarizations of the line item table, and to generate these summaries every month. This meant that the 16.7 million invoice line items did not have to be searched for every interrogation. Without this summarization approach, the organization determined the application would not be possible. An example of a summary is the total product sales for a particular division for a particular program for a particular sales organization for a particular company month (DIV-PROG-ORG-SLS-MON-AMT). Another example is a summary of the total product sales of an item-version for a particular sales region for a particular month (ITEMVER-REGN-MON-AMT).

All in all, there were about 100-200 such summaries. All the summaries were created and formatted as rows for summary level tables, which were loaded into a sales and marketing



database. These summary level tables were used to meet the needs of the marketing, sales, and research departments. So, for any particular problem, if there was a summary that fit the data needs, then everything was fine. But if not, researchers had to solve the problem by getting reports, sifting and selecting data, and recompiling new statistics--all manually. As an alternative to manual work, the data processing department created an additional data extract, summarized the data into a new summary level table, modified the sales and marketing database by loading the rows from the new summary table, and then allowed the staff to access it. The creation of each new summary table required about one staff month. In short,

- If the summary data was used, queries could run much faster, but the improvement would apply only to problems that fit a particular summary.
- If the raw data was used, any kind of problem could be researched, but each query would take 57.27 hours.

**LINE ITEM**

LENGTH=66

MAXIMUM COUNT=16,700,000

COUNT PER MONTH=278,000

TOTAL SPACE=1.102 Billion Characters

CUSTOMER-NUMBER (PK (1), FK (1), UNIQUE VALUES=47,000, PIC 9 (11) )

ORDER-NUM (PK (2), FK(2), UNIQUE VALUES=1,500,000, PIC YYYYMMDDNNNN

LINE-NUM (PK (3), PICTURE 99)

MANUFACTURER-ID (FK (3), UNIQUE VALUES = 1, PIC X (5) )

ITEM - NUM (FK (4), UNIQUE VALUES = 500, PIC X (5) )

VERSION-NUM (FK (5), UNIQUE VALUES=10, PIC 99)

MARKETING-PROGRAM-ID (FK (6), UNIQUE VALUES=1000, PIC YYYYNNN)

ITEM-ORDER-QTY (PIC 9 (7) )

REGION (FK (7), UNIQUE VALUES=24, PIC 99)

DISTRICT (FK (8), UNIQUE VALUES=150, PIC 9999)

TERRITORY (FK (9), UNIQUE VALUES=750, PIC 999999)

DIVISION (FK (10), UNIQUE VALUES=10, PIC 99)

SELLING-ORGANIZATION (FK (11), UNIQUE VALUES =99, PIC 99)

Note: See appendix A for notation definition

**Figure 4.8.** Logical database design.





The summary approach was implemented at the client site. After several years three problems emerged:

- The demand for new summaries kept growing making it increasingly difficult to keep current.
- As existing summaries became outmoded, it was necessary either to drop an old summary by reorganizing the database at great expense, or to keep paying for generating and updating unused data.
- The end user, that is, the researcher, was dependent on the data processing department to develop the necessary computer programs to select rows, combine them with others, and produce a report.

Two conclusions were drawn:

- The data processing department was unable to keep up with the development of summaries that were to feed the database.
- The data processing department was also unable to generate new reports in a timely manner.

In short, because of the method of solution implementation, the backlog in data processing kept costs increasing and progress slowing.

For these reasons, a search to replace the summary approach began. At the beginning of the search, specific criteria were established so that alternatives could be properly evaluated. To be acceptable, a solution had to:

- Store data efficiently
- Enable users to directly formulate interrogations of any kind
- Have computer response times that were reasonable

#### **4.2.2.4 The Company's DBMS**

An obvious way to begin the search for a new solution was to examine the client's DBMS to see if its indexing facility could assist. The DBMS's data model installed at the client site was network. The DBMS did have an indexing capability to speed searches, and this capability could be applied to one or more columns. However, if multiple indexed columns within a row were employed, the DBMS used only one of them. This meant that to achieve the effect of a multiple-column index, the column values had to be concatenated, with the resultant value being stored as a single string in the index file. In short, the DBMS was a row processor, not a list processor.



For an illustration of how indexing is beneficial, assume that it is necessary to total all the sales for one of the 24 regions. The DBMS would, of course, suggest that the column REGION be indexed. Using this method, the access time is twenty-four times less, or about 2.39 hours. Obviously this is an improvement. The improvement for totaling all the sales for one month is even greater if month is indexed. The elapsed time using a MONTH index is 57.27 hours / 60, or about 1 hour.

The corporation's DBMS also allowed indexed columns to be combined to make compound indexes. For example, the MONTH and REGION columns can be combined into a single index. This has the effect of shortening the query time to about 2.39 minutes  $((57.27 / 60 / 24) * 60)$ .

<region>	<customer number>	<order number>	<ln>
8	249241142	198901221223	01
8	249495200	198802017524	01
8	262725949	198802017524	02
8	566669405	198802017534	03
8	548645913	198802017534	04
8	657671952	198812150001	01
8	73373817	198812150001	02
8	939399791	198811130765	01
8	939934155	198811130765	02
8	939992240	198811130765	03

**Figure 4.9.** Secondary index organized as a primary key index.

To build indexes, the client's DBMS appends, for example, the primary key (25 characters) of the row to the unique value of the REGION, producing the index shown in Figure 4.9. The primary key is identified as the concatenation of the first three columns. The columns that comprised the key are identified in Figure 4.8. Summing the character space required by their combined picture clauses results in 27 characters. The size of such an index is the sum of the secondary key's length plus the primary key's length (see Figure 4.8) multiplied by the number of rows. For example, the size of a REGION index on the line-item file is: (2 bytes + 27 bytes) \* 16.7 million rows, or 450.9 million bytes. If all the indexes are two bytes long, then for 13 such indexes, the total space required for the single-column indexes is 13 \* 450.9 million characters, or about 5.861 billion characters.

Now, for the compound indexes, the space has to be expanded for the additional columns that are concatenated with the indexed column. For example, if an indexed column is created that consists of the primary key (27 characters) and the REGION (2 characters), SELLING-ORGANIZATION (2 characters), and DISTRICT columns (3 characters), then the index size is:  $(27 + 2 + 2 + 3) * 16.7$  million, or about 534.4 million characters. To index the 20 most popular compound indexes, the total index space is 10.688 billion characters. Adding that to the index space required for the single indexed columns, the total index space comes to 16.54 billion characters.



To implement indexes, the company's DBMS creates large files of small index row. While index processing to find the list of line-item rows is faster than reading the entire set of rows in the line-item file, the indexing done by the organization's DBMS has drawbacks in two areas:

- It requires concatenated indexes to achieve multiple index processing.
- It is large. In fact, the total required space for indexes is 15 times more than for the data.

The reason the DBMS requires concatenated indexes is that the DBMS does not use list processing, that is, ANDing or ORing multiple lists of keys to arrive at the target set of row identifiers. If the DBMS used list processing, each list of keys could be created by processing an index to find all the keys for a specific index value. To simulate list processing, the company's DBMS offered compound indexes. These compound keys, however, only increased the problem, as each takes up an increasingly larger amount of space, and each has to be updated by the DBMS. The DBMS's indexing strategy thus seems to be nothing more than replacing threaded lists of pointers in the rows with large files of small rows of specially constructed primary keys.

At first blush, indexing seemed to be a solution to the sales and marketing problem. The indexing strategy employed by the company's DBMS, however, does not include list processing, and this results in the definition, creation and constant maintenance of compound indexes. Furthermore, the DBMS's indexes, singly or as compound indexes, consume large amounts of disk space. Having 16.54 billion characters of DBMS index space for only about 1.102 billion characters of data seriously reduces its value as a solution.

What is the solution, or is the problem impossible to solve?

#### **4.2.2.5 Looking Beyond a Popular Myth**

The key to finding the right solution is to look beyond the widespread myth that indexes are always an add-on feature to database management systems, and not a fundamental component of a DBMS's access strategy. This myth leads one to believe that index mechanisms necessarily consume large amounts of space and require considerable resources for updating. But, in fact, there are a number of DBMSs that have indexing facilities designed especially to handle problems similar to the one in the case study.

#### **4.2.2.6 Index Effects on Access Strategies**

While the data model of a DBMS is clearly an important way of distinguishing one DBMS from another, access strategy is the DBMS component that provides real speed. For example, there is nothing in the relational data model that prohibits a DBMS from relying completely upon magnetic tape for storage and serial access for processing and calling itself relational. In fact, a DBMS called SIRS, developed in the 1960s by the National Aeronautics and Space Administration (NASA), was



completely based on tape files, one for each table. Thus, conforming to one data model or another has nothing to do with performance.

What primarily determines the DBMS's speed is the sophistication of its access strategy. The two most important components of a DBMS's access strategy are indexes and relationships. The third most important component of a DBMS's access strategy is data storage formats. Both relationships and data storage are covered later in this chapter.

Although many people know that the fastest mechanism for accessing data is an index, few know that there are several different types of index organizations, and that some of these organizations can be very efficient in storage and very fast in processing. Figure 4.10 identifies a number of the more popular index structures and indicates their relative size and performance rankings.

#### 4.2.2.7 Index Structures

Indexes can be built on single columns or on groups of columns. The first is simply called an index, while the latter is called a compound index. An index may also be built on a column that represents multiple values within a row. For example, an employee may have a multiple-valued column: TELEPHONE NUMBER. ILF and some network DBMSs allow this case, while hierarchical and relational DBMSs prohibit it outright (because the data model does not allow multiple-valued columns). Most ILF DBMSs handle the index values by pretending that a *subordinate* table exists to contain the indexed column. This enables each value from a multiple-valued column to be distinctly addressable.

Every index entry contains two components: the unique value component and the location component. The location component can either be a single location component or a multiple location component. The unique value component represents the column values extracted from the rows. The location component contains, in some form, the address of the row from which the column value was obtained.

The unique value component of a column's index portrays the set of data values from one or more columns across the entire set of rows for a specific table. This set of data values can be either all unique or can initially contain some duplicates. When the value set is unique, then the quantity of instances in the unique value set is the same as the number of rows. These index types are often called primary keys and are useful for invoice numbers. Thus, a primary key index instance has a unique value component and single location component instance.

When the unique value component's value set is not unique, that is, it contains some duplicate values, then more than one row must have the same index value. In this case, the index is called a secondary key type index. As the duplicates are removed the quantity of unique values becomes less than the quantity of rows. Each removed duplicate leaves behind its location component. These *orphaned* location components are all collected around their common unique value and are contained in the index's multiple occurrence component.



INDEX TYPE	Unique Values	Multiple Occurrence	Within Group Rank	Overall Speed Ranking
PRIMARY	Hash/Calc			1
	Hierarchical			2

Times: Slower from fastest	Unique Parts	Multiple Occurrence Part	Row Access	Multiple Occurrence Table Size	Logical Accesses to Obtain Rows.	Access Time
24	NONE	NONE	Serial	NONE	16.7 mil.	57 hrs
4	Hierarchical	Combined	Hier	450.9 mil.	2,803,504	9.6 hrs
2	Hierarchical	Combined	Hashed	450.9 mil.	1,403,504	4.8 hrs
4	Hierarchical	Primary Key List	Hier	17.4 mil.	2,802,901	9.6 hrs
2	Hierarchical	Primary Key List	Hashed	17.4 mil	1,401,901	4.8 hrs
4	Hashed	Primary Key List	Hier	17.4 mil.	2,802,900	9.6 hrs
2	Hashed	Primary Key List	Hashed	17.4 mil.	1,401,900	4.6 hrs
1	Hierarchical	DBKEY	Hier.	2.1 mil.	700,354	2.4 hrs
1	Hashed	DBKEY	Hashed	2.1 mil.	700,350	2.4 hrs
1	Hierarchical	Bit Map	Hier.	600 k	700,018	2.4 hrs
1	Hashed	Bit Map	Hashed	600k	700,017	2.4 hrs

**Figure 4.10.** Primary and secondary index structures performance comparisons for single column select statements.

For each unique value there exists a multiple occurrence component which contains one or more location components. Each location component instance contains, in some form, the address of the row from which the column value was obtained. Normally, the quantity of all location component instances contained in all the multiple occurrence components for an index is the same as the quantity of rows. If the DBMS contains the NULL concept and includes that special value (NULL = *do not know*) in the index then the quantity is the same as the row count.

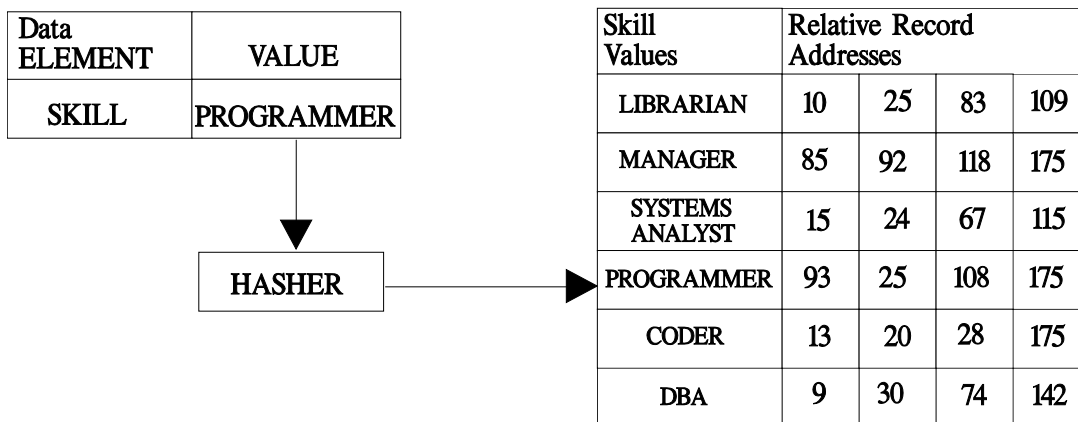


If NULL is not represented in the index then the quantity is equal or less depending upon the existence of NULLs in the set of column values. Secondary (key) indexes are useful for finding whole classes of rows such as REGION, TERRITORY, and the like.

As stated above, when an index represents only unique value references, it contains the unique value component and is called a primary (key) index. When an index is a secondary index, it contains both the unique value component and the multiple occurrence component (a set of location components). Figure 4.2 illustrates a hierarchically structured index organization that contains only the unique value portion. Figure 4.3 illustrates a hashed structured index organization that contains the unique value portion. Figure 4.11 illustrates a hash-organized unique value component combined with a multiple occurrence component making it a secondary key. Figure 4.12 illustrates a hierarchically organized unique value component combined with a multiple occurrence component making it a secondary key. From Figure 4.11, the select clause `SELECT SSN, NAME FROM EMPLOYEE WHERE SKILL EQ PROGRAMMER` would return the EMPLOYEEs whose RRAs are 93, 25, 108, and 175. From Figure 4.12, the select clause `SELECT CUSTOMER NUMBER WHERE REGION EQ 8` would return CUSTOMERs meeting that condition including customer 249241142.

#### 4.2.2.7.1 Unique Value Component of an Index

The unique value component of an index contains the set of all unique values. For the column REGION, the case study contains only twenty-four values, 1 through to 24. The unique value portions of these indexes are typically organized in one of two ways--hierarchical or hash/calc--each having its own advantages and disadvantages.



**Figure 4.11.** Hash-based secondary index organization.



#### 4.2.2.7.1.1 Hierarchically Organized Unique Value Component

A hierarchically organized unique value component of an index is typically constructed of a series of DBMS rows containing the collection of unique values in a hierarchically organized structure. The lowest level contains DBMS rows with unique values in increasing value order. When a DBMS row of these values is filled up, another DBMS row is constructed. Eventually all DBMS rows on the lowest level are filled. The DBMS then determines whether the index structure has two or more DBMS rows with unique values at the lowest level. If so, the DBMS builds a higher level DBMS row. It places in the first DBMS row of this second higher level the greatest value of the keys that were stored in the lower-level DBMS row. When the first higher level DBMS row fills, then a second DBMS row is built. When all these second level DBMS rows are built, the DBMS determines whether there are more than two DBMS rows of unique values within this second level of index. If so, then the DBMS begins to build a third level. The DBMS continues building these additional levels until there is only one DBMS row at the top.

Figure 4.2 illustrates the result of a hierarchical index. It can be seen that with about 80,000 DBMS index rows, 16,000,000 rows can be represented. Representing 16.7 million rows (the case study requirement) requires slightly more space. The space required by such an index is the sum of the space required at each index level. In the example, the lowest level contains 80,000 pages. Each index entry consists of the primary key value and one of either of the two pointer types. Usually each pointer requires three bytes. This lowest level requires 467.6 million bytes ( $16.7 \text{ million} * (25 + 3)$ ). The next higher level, assuming that 200 higher level pointers (about 6000 characters) are stored in each DBMS row, requires about 2.4 million characters (400 DBMS rows). The third level requires only 12,000 bytes (2 DBMS rows). The top level contains one DBMS row and requires 6000 characters. The total index space is about 470 million characters. To find the location component of any row requires accessing no more than four DBMS row index instances in this hierarchically organized index.

To represent the unique value part of a secondary key, the hierarchical component only has to store the unique values that exist within the complete set of rows. In the case of CUSTOMER-NUMBER and LINE-ITEM, there are 47,000. This requires less than 240 DBMS rows of index space and about 1.45 million characters.

An advantage of a hierarchical index is that it enables range searching. So, if a query needs to find all invoices with INVOICE TOTAL between \$20,000 and \$25,000, the index processor finds the corresponding row identifiers for invoices where the indexed column INVOICE TOTAL has values between those two values.

A disadvantage of the hierarchical index is that in this example it can take up to four accesses to find the row's location component. Further, updates can affect multiple levels of the index. Typically however, 95% of the unique values are found within the first 20% of the rows that are loaded. Thus, the real effect is to the lower level DBMS rows whenever a new value has to be included. Inclusions in the lowest level seldom affect the upper levels as values newly included in the lower levels are seldom outside the ranges of values in the upper levels.



#### **4.2.2.7.1.2 Hash/Calc Organized Unique Value Component**

A hash/calc unique value component is organized very differently from its hierarchical counterpart. Simply, a data value is entered, and through a formula, an address is produced that points to the DBMS/data/index row location. At the location of the address, what is found is either a row or the address of the row. In the first case, the addressing scheme is direct, and in the second, indirect. An illustration of a hash/calc unique value component of an index is contained in Figure 4.3.

If the addressing is direct, then the rows must, in turn, be stored in random order. The space required for hash/calc direct index is inconsequential as the formula produces the address of the DBMS row on which the row is stored. Alternatively, if the addressing is indirect, then the rows can be stored in the order in which they are entered, in a sorted order by primary key, or in some other order.

The value of indirect addressing is that a change to the primary key value does not require that the row be moved. Such a benefit is not free, however. The space consumed by the indirect addressing is the space required by the cross reference table. This is the space required by each unique value and the row address. Not surprisingly, this is typically the same amount of space required by the lowest level of hierarchical index. As in the previous example, for a hierarchically organized primary key, this is about 467.6 million bytes. The address produced by the hash/calc/scheme is likely to be the DBMS row containing the primary key value. The space required for this cross reference table for just the customer number secondary key is about 240 DBMS rows, or 1.45 million characters.

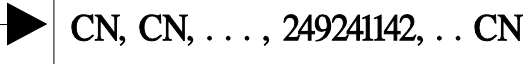
In summary, if a hierarchical unique value component is used, its advantage is that it enables ranges searches, but in terms of the case study, it takes four accesses to find one row in 16.7 million. If a hash/calc unique value component organization is used, then while range searching cannot be performed, it takes only one or two accesses to find the one row.

#### **4.2.2.7.2 Multiple Occurrence Component of an Index**

As stated earlier, an index consists of a unique value component and a multiple occurrence component. If the index is a primary key type index then the multiple occurrence component is not present. If, however, the index is to be used to find a class of rows, for example, all line-item rows within REGION 8, then the index consists of both the unique value component and a multiple occurrence component. Each instance of the multiple occurrence component contains some form of the row's address. Again, when the index consists of both components it is often called a secondary index.





DICTIONARY	INDEX ORGANIZATION	
DATA ELEMENT	UNIQUE VALUES	Mutliple Occurrences Arrays
REGION	1 ... 8 ... 24	

**Figure 4.12.** Multiple occurrence component of an index structure for customer numbers.

Fundamentally, there are four different ways to organize the components of a secondary key index:

- Combined secondary key and primary key
- Separated secondary key and primary key
- Separated secondary key and DBKEYs
- Separated secondary key and bit maps

#### 4.2.2.7.2.1 Combined Secondary Key and Primary Key

The first type of multiple occurrence component is not really a separate structure, but a hybrid of the hierarchically organized unique value component. It is formed by concatenating the row's secondary key value to the row's primary key value. If, for example, there are 2000 customers in the San Francisco region (REGION 8), the value <8> is the prefix of each customer line item's row's primary key (see Figure 4.9). This list is shown sorted and organized hierarchically. To add or delete from this type of secondary index, the proper value is located and either added or deleted. Higher levels of DBMS rows might have to be adjusted as well during updates. The size of this index type for REGION is 450.9 million characters  $((25 + 2) * 16.7)$ .

The number of accesses to acquire the REGION 8 customer's line items in a hierarchically organized primary key index is 2,803,504: four to work through the index to the lowest first value within the range, 3500 for obtaining the line item primary keys for the customers within the range, and then four for each of the 700,000 REGION 8 customer line items. If the primary key index is hash/calc then there are only two accesses for each of the 700,000 REGION 8 customer line items, giving a total number of access of 1,403,504.



#### 4.2.2.7.2.2 Separated Secondary Key and Primary Key

The second type of multiple occurrence component, separated secondary key and primary key, separates the multiple occurrence component of the index from the unique value component of the index. The value of the primary key, the customer number, is stored in an array representing all customers in REGION 8. An array containing the customer numbers REGION 8 is illustrated in Figure 4.12. A significant disadvantage of having the actual value of the customer number stored in the multiple occurrence array is that it is not the address of the customer's row. The DBMS has to take the customer number and use some primary key access strategy to find the locations of the row(s) representing the customers within REGION 8. As a consequence of separating the unique value portion and the multiple occurrence portion of the index there are four subcases to consider:

- 1) Hierarchical unique and hierarchical row access
- 2) Hierarchical unique and hashed row access
- 3) Hashed unique and hierarchical row access
- 4) Hashed unique and hashed row access

If the unique part of the secondary key is a hierarchically organized index (1), and the rows are organized through a hierarchical primary key access, the space is the sum of the space required for the hierarchical unique portion and the hierarchical row access portion. The hierarchical unique portion is one DBMS row (6000 bytes) as all 24 regions' column values surely fit on one DBMS row. The multiple occurrence portion is the space required for all the lists (24) containing all the primary keys. That is, the sum of the space required for all the primary keys, or about 417.5 million characters ( $16.7 \text{ million} * 25$ ). Assuming the accounts are evenly distributed, each list is 2900 DBMS rows long ( $417.5 \text{ million} / 24 / 6000 \text{ bytes per DBMS row}$ ). Once the primary key values are found, the DBMS proceeds to perform a hierarchical row access. The total number is 2,802,901: one to work through the hierarchical index, 2900 to obtain the complete list of all 700,000 REGION 8 customer numbers, and 2.8 million to obtain the 700,000 REGION 8 customer line items.

If the unique part of the secondary key is a hierarchically organized index (2), and the rows are organized through hashing, the space is the sum of the space required for the hierarchical unique portion and the hashed row access portion as defined in subcase (1). The multiple occurrence portion too is defined as in subcase (1). The total number of accesses is 1,402,901: one to work through the hierarchical index, 2900 to obtain the complete list of all 700,000 REGION 8 customer numbers, and 1.4 million to obtain the 700,000 REGION 8 customer line items.

If the unique part of the secondary key is hash organized (3), and the rows are organized through a hierarchical primary key access, the space is what is required for all the lists as computed in subcase (1) above. The total number of accesses is 2,802,900: one to work through the hierarchical index, 2699 more to obtain the remaining REGION 8 customer numbers, and 2.8 million to obtain the 700,000 REGION 8 customer line items.

If the unique part of the secondary key is hash organized (4), and the rows are organized through a hash organized primary key access, then the index space is only the amount required



for the lists as computed in subcase (1). The total number of accesses is 1,402,900: one to work through the hierarchical index, 2699 to obtain the remaining REGION 8 customer numbers, and 1.4 million to obtain the 700,000 REGION 8 customer line-items.

It should be noted that the first two cases allow the secondary key selection clause to express a range clause while the last two cases do not.

#### **4.2.2.7.2.3 Separated Secondary Key and DBKEYs**

The third type of multiple occurrence component, separated secondary key and DBKEYs, relates the secondary key value to the actual address of the row. A DBKEY is a DBMS generated row address. A DBKEY is typically three bytes and thus 2000 DBKEYs can fit on each DBMS row. If the multiple occurrence part of the index contains, not customer numbers, but the actual addresses of the rows, then the number of accesses for hierarchically organized indexes drops to 700,351: one to work through the hierarchical index, 350 to obtain the complete list of all 700,000 DBKEYs, and 700,000 to obtain the REGION 8 customer line-item rows. The saving results from using DBKEYs, thus eliminating the additional index search operation to find the row's address using the customer number. The space required by this index is considerably smaller than the prior cases. All 700,000 DBKEYs can fit on 350 DBMS rows and take up only 2.1 million characters. Since every row has one of the 24 region values, then the total space for all the REGION secondary index is 50.1 million characters ( $16.7 * 3$  bytes).

For hash/calc index organizations, the number of accesses is 700,350: 350 to access the array of DBKEYs for the REGION 8 customers, and 700,000 to obtain the customers.

Again, if the secondary key value part of the index is hierarchically organized then range searching is allowed. If the secondary key value part is hash/calc organized, range searching is not possible.

#### **4.2.2.7.2.4 Separated Secondary Key and Bit Maps**

The fourth type of multiple occurrence component, separated secondary key and bit maps, relates the secondary key value to a bit map representation of DBKEYs. Traditionally, the notion of a DBKEY means relative row address. The *row* in relative row access often refers to the DBMS row number, and then the logical row's position within the DBMS row. Thus, if there is a blocking factor of 10, a DBKEY value of 85 means the fifth logical row within the ninth DBMS row. If, however, the DBKEY value itself is not stored in the multiple occurrence array, but a bit string of 85 bits is stored, then a bit map can be used to determine whether a row has the value of the secondary key value as follows: if the *n*-th bit is on (value = 1) then the *n*-th row contains the indexed column's value. If the *n*-th bit is off (value = 0) then the *n*-th row does not contain the indexed column's value.

Bit maps save considerable storage space. A traditional DBKEY, for example, is three bytes long. Thus, to store 2000 DBKEYs, 6000 bytes are needed. With bit maps, the same number of bytes could store references to 48,000 rows, a twenty-four-fold density increase. If 20% of the 16,700,000 rows are for sales to the West Division, then the length of the traditional DBKEY array will be  $.2 * 16,700,000 * 3$  bytes, about 10,020,000 bytes. If bit maps are used,



the total number of bits is 16,700,000 (one for each row), or about 2.087 million bytes, about five times less than with the traditional approach.

If the blocking factor of secondary key multiple occurrence arrays is 6000 bytes per DBMS row, then about 1670 DBMS rows ( $10,020,000 / 6000$ ) are needed to store the traditional DBKEYs. If the same blocking factor is utilized for the bit map, then the storage requirement is  $(16,700,000 / (8 * 6,000))$ , or about 348 DBMS rows, again about five times less. For our REGION 8 example, only about 4% of the rows are valued with an 8. Using DBKEYs, the space requirement for the multiple occurrence list is 2.087 million characters ( $((16.7 \text{ million} / 24) * 3 \text{ bytes})$ ), or about 350 DBMS rows. The bit map approach for that multiple occurrence list requires 2.087 million characters ( $16.7 \text{ million} / 8 \text{ bits per byte}$ ). The total space for all the regions for the bit map approach is 24 times larger, or 50.1 million characters. For this sparse valuation example, the index space requirement for the multiple occurrence list for DBKEYs and bit maps is coincidentally the same.

As illustrated, the same unique value is not in every row. If that were the case, all the rows would belong to REGION 8 and all the bits for that index's multiple occurrence array would be set to *on*. The other 23 multiple occurrence lists would be set to *off*. If the DBMS can determine that whole DBMS rows of index bits for that unique value are set to *off*, then there is no need to read that DBMS row of bits for either an AND or an OR operation. In the case study, that situation is likely as only 4% of the rows have a value of 8. In such a case, a higher level array of bits, analogous to the higher level in a hierarchical index, can serve to indicate which DBMS rows of bits possess rows with the data value that is being looked for. As there are 16,700,000 rows, the array at this higher level is 348 bits. This higher level bit array is used to accomplish a preliminary screening of multiple condition WHERE clauses before actually searching through the lower level DBMS rows of bits. This technique also reduces space since the lower level DBMS rows that do not contain *on* bits do not have to exist. In the case study, the space is thus reduced by at least 66% to about 700,000 bytes. Thus, for this sparse example, the bit map requirement is not the same as DBKEYs, but one-third that of DBKEYs. The number of DBMS pages that have to be scanned for one list is thus reduced to about 117. The total space for all 24 of the bit map organized multiple occurrence arrays is 24 times larger, or 16.8 million. That is 66% smaller than the space required for DBKEYs.

DBMS row counts provide a direct link to performance. For the Western Division example, it takes 238 seconds ( $1670 / 7 \text{ I/Os per second}$ ) to obtain all the DBKEYs of the customers. Since there are only 348 DBMS rows for the bit maps, the time is about 49 seconds ( $348 / 7$ ) to find the rows. That is about 5 times faster than the traditional three-byte index addressing scheme ( $238 / 49$ ). It is even faster if any DBMS rows of bits could be screened out at the higher level bit map. In practice, at least one-half of the DBMS rows are screened in this way, and thus the time is 24 seconds. That means that the bit map index search performs about 10 times faster than the DBKEY index. For the REGION 8 case study example, the DBKEY list contains 350 DBMS rows and takes 50 ( $350 / 7$ ) seconds to acquire. The bit map approach only requires about 17 ( $117 / 7$ ) seconds. Thus, the bit map approach is three times faster.

Now, to actually obtain the rows takes additional time. This row access time is independent of whether the DBKEY is a traditional three-byte address or a bit map address. While the theoretical maximum number of I/Os is 1 per row access, the experienced rate on the case study's computer, considering blocking factors and in-memory hit rate, is about 50 rows per



second. That means that the time to access the index selected (bit map or DBKEY) rows is  $(16,700,000 * 20\%) / 50$  rows per second, or about 18 hours. Utilizing the non-indexed method, the time was calculated above to be about 57 hours. The indexing produces a savings in excess of 66%.

Some sets of single-index select criteria result in selected row populations closer to 0.1% than 20%. For example, to pick the sales data for only one of the 47,000 customers produces only 356 rows  $(16,700,000/47,000)$ , which is only 0.00215%. The space required by the DBKEYS is 1020 bytes, or less than one DBMS page. For the bit map approach the space is about 700,000 bytes (already reduced because of all 0 index DBMS rows) or about 117 DBMS rows. In this very sparse example, the bit map approach takes more space and takes longer to process than the DBKEY approach. This clearly shows that the physical database designer has to know the characteristics of the data before choosing the type of index construction.

The top part of Figure 4.10 ranks the speed of the unique value component of an index. From the calculations in previous sections, the hash/calc alternative is faster, but it cannot accommodate range searching.

The second part of Figure 4.10 presents a summary of secondary index structure analyses. The first row, no indexes, clearly takes the longest to process a single column select statement. The next two rows, combined secondary and primary keys, consume a large amount of space, but do produce an 800 percent improvement. The next four rows provide a distinct improvement in index space size, but provide no improvement. The last four rows cut the access time in half and also significantly reduce index space size. The clear choice then is DBKEYS for bit maps, with either hierarchical or hashed row access. If hashed is chosen, however, range searching is not possible. The access time column is determined by dividing the number of accesses by 81 to give seconds. That result is divided by 60 minutes, and that result divided by 60 for hours.

The *times slower* column indicates the ratio of the fastest index type to the slowest. Four index structures perform the same (1.0), three others perform two times slower, three more perform four times slower, and one (no indexes) performs 24 times slower.

Because the four fastest index types (last four rows) are available in popular DBMSs, only these index types are further pursued in the next section. DBMSs that only support index types from the first six rows should be avoided if at all possible.

#### 4.2.2.8 Multiple Condition WHERE Clauses

The presentation of different index structures and the estimates of the access resources consumed are all based on having only one condition in the WHERE clause. For the case study, and also in almost all real world applications, WHERE clauses contain multiple conditions. If the DBMS can only increase the performance of operations through the inclusion of at most one indexed column, then it is termed a row-processing DBMS. It retrieves the rows selected and then searches the selected rows--one at a time--for the other conditions in the WHERE clause.

If the row-processing DBMS is able to automatically optimize WHERE clauses, then the amount of resources consumed in a query depends on whether the DBMS stores information about its own indexes, and whether the conditions in the WHERE clause are connected by ANDs



or ORs. For example, if the DBMS stores the number of rows for each unique value at the head of each of the multiple occurrence lists, the DBMS knows the counts that are present in column 2 of Figure 4.13.

SELECT CONDITION	NUMBER OF RECORDS SELECTED	TRADITIONA L DBKEY READS	HI LEVEL BIT MAP ANDS	LO LEVEL BIT MAP READS
1	50,000	25	333	3
2	100,000	50	167	3
3	500,000	250	83	3
4	1,500,000	750	42	3
5	2,000,000	1000	21	3
6	500,000	250	11	3
7	100,000	50	6	3
8	2,000,000	1000	3	3

**Figure 4.13** Multiple Occurrence DBMS Record Reads

If the eight conditions are all connected through ANDs, then the DBMS picks the shortest list (number 1), retrieves the rows, and searches for the column values represented by the other conditions. The total accesses takes about 1000 seconds (50,000/50) or about 17 minutes. While not fast enough to be interactive, it is much faster than performing a serial sweep of 16.7 million rows (about 57 hours).

If, however, the select conditions are connected with ORs, and conditions 1 and 5 are in the select list, then both sets of rows have to be retrieved and the lists joined, dropping duplicates. The time this takes is determined by adding the time required to access the 50,000 rows and the 2,000,000 rows. That is about 17 minutes for the 50,000 rows, and about 11 hours for the 2 million rows. The total time probably approaches 12 hours considering the effort required to drop the duplicates. If all the eight conditions are included, then the time to find the rows approaches 57 hours. When the number of rows searched approaches 40%, whole file sweeps are often a more cost effective choice.

The DBMSs in Figure 4.7 optimize the multiple condition WHERE clause through a technique called list processing. List processing involves taking the DBKEYs, or primary keys found from each of the select conditions, and ANDing or ORing these keys together to produce the final set of identifiers of the rows.

In Figure 4.13, there are eight lists of selected rows. The length of each list is provided in the second column. If the operation between two lists is an AND, the two lists are merged and the DBKEYs retained are only those that are in both lists. If the operation is an OR, then after the two lists are merged all the DBKEYs are retained except for duplicates.



A common technique for merging sorted lists of DBKEYs is based on a numbering sequence invented by the 13th century mathematician, Fibonacci. The technique for ANDing and ORing bit maps is based on Boolean algebra.

As the number of search criteria within a WHERE clause increases, the performance advantage of bit maps over traditional DBKEYs increases even more. Figure 4.13 illustrates the number of DBMS row reads necessary to read lists of multiple occurrence pointers and bit maps. Each item in the third column is derived by dividing the number of rows selected (50,000) by the blocking factor of DBKEYs to the DBMS row (2000). Column four of the first row is the number of bits (333) in the higher level bit map. The number in each succeeding row is halved since the probability of finding a desired DBMS row is halved with each higher level bit map AND.

The bit map processing is accomplished in three steps. The first is the identification of the desired higher level bit map arrays. This is done with a hash/calc index. The required value for the indexed column is combined with the column's name code to form a unique column-name unique-value pair that is used in a hash/calc index to determine if the value is present in any tables. If not, the search is terminated. If present, the particular high level bit map array is identified.

Next, the identified high level bit map array is examined to determine if there are any desired lower level bit map DBMS rows. This is accomplished through an examination of the higher level bit map array cells. If a cell is on, then the corresponding lower level DBMS row of bits contains at least one *on* bit. When the lower level DBMS row is examined, if a bit is *on*, then the corresponding row contains a column with the key value.

When individual conditions in the WHERE clause are all connected with ANDs, the processing of the conditions can proceed in parallel. That is, all the higher level arrays are determined first, then they are ANDed together, probably dropping a good number of arrays with each AND. If 50% are dropped with each AND, then only three DBMS rows of each of the 50,000 bit arrays are finally reviewed. That means that a total of 24 accesses are performed, and merging the bit maps takes about ½ second. Added to this time is the time to actually retrieve the rows, which is about 100 seconds (5000 rows / 50 rows per second), or a total time of about 1.68 minutes. The ratio of bit map index processing to row retrieval is 1:50, or fifty times faster to identify a row through the indexes than to retrieve it from the database.

With the traditional DBKEY solution, not only do the rows have to be read, they must be merged. The overhead of the Fibonacci merging technique doubles the I/Os (sorting, writing, and so forth) with each four select conditions. Thus, since there are eight select conditions, the I/Os for the traditional DBKEYs are not 3375, but 6750 for the first four and then 13,500 for the second four, for a total of about 20,250 accesses. That means that merging the lists takes about 7 minutes. Added to this time is the time to actually retrieve the rows, which is about 100 seconds (5000 rows / 50 rows per second), for a total time of about 8.67 minutes, about 5 times slower than the bit map solution. The ratio of DBKEY index processing to row is 4.2:1, or four times slower to identify the row through the indexes than to retrieve it from the database.

The overall ratio of bit map indexes to DBKEYs in the case study is about 200:1. That is, the bit map indexes are 200 times faster than the DBKEY indexes. But when row time is incorporated, the overall performance ratio between bit maps and DBKEY is 5:1 in favor of bit maps.



#### 4.2.2.9 Case Study Summary

To successfully solve the problem posed in the case study, three considerations have to be addressed:

- Queries have to use line-item detail data stored for five years to answer the needs of headquarters sales and marketing, and to respond to the needs of field sales. Such queries, however, were estimated to take 57 hours to perform a serial search.
- While summarized data was initially used to achieve acceptable response times, the only problems that were easily solved were those fitting within the scope of the summaries. It was not long after the initial set of summaries were built that dissatisfaction with the choice of summaries started to build. Creating additional summaries was seen to be a never ending project.
- If indexes are to be used to achieve acceptable response times, the indexes have to solve more problems than they create. First, the underlying DBMS has to perform list processing with the conditions present in the WHERE clause, rather than row processing. Second, the actual space required for indexes has to be small. And third, the amount of time necessary to update the indexes has to be short enough to be practical.

DBMSs that offer choices among sophisticated index structures can truly perform interactive queries with 8 or more conditions that select about 5,000 rows from databases with 16.7 million or more rows in a single table. Certainly, that solves the problem of query response time (see Figure 4.14). DBMSs that support bit map list processing index strategies operate five times faster than DBMSs that support the best list processing traditional index access strategy.

With respect to having the correct set of summaries, if the raw data is stored, then there is never a summary that can not be derived, unless, of course, the data required by the summary has not been collected at all!

With respect to index sizes, if a DBMS offers an indexing strategy that is built into its access strategy, and further, if these indexes are very sophisticated in design, then not only can the required processing speeds be achieved (eliminating the need for summary level data), but also the size of the indexes need not be excessive. Figure 4.10 clearly shows that fast indexes do not have to consume a large amount of disk space.

As for solving the needs of the three user groups, marketing's needs are met because their data needs are founded upon weekly level, line-item data. Headquarter's needs are met because they can access data by sales unit across specific times. If the line-item data is loaded daily, then the reports are almost in real time. Finally, the field's needs are met because salespersons can find the delivered orders, near term deliveries, and those that are back ordered. They can use past performance data to plan the next week's sales strategy.





Number of Records:	16,700,00
Number of Records Retrieved:	5,000
Number of Conditions:	8
Time to select and retrieve:	
No Indexes (serial sweep)	57 hours
Traditional Index Structure	8.67 minutes
Bit Map Index Structure	1.68 minutes

**Figure 4.14.** Summary of index strategy times.

#### 4.2.2.10 Index Summary

From the presentation of the case study, it is obvious that sophisticated indexes are quite different from ordinary indexes. An index serves not only as a fast access to a set of rows, but also as a way to completely avoid the writing of complicated program logic to narrow a set of rows initially selected on the basis of one indexed column. Because indexes save so much time and effort, sophisticated ones make possible entire applications that are not practical by any other means.

This section on indexes has focused on the selection of rows of the same type. For the vast majority of real business applications, selection of rows from just one type is too simplistic. For example, while it is certainly important to find all the orders that have a TOTAL-ORDER-AMOUNT over a certain value, it is probably more important to know the name of the CUSTOMERs placing the order. Another very reasonable query is to find all the CUSTOMERs located in the WESTERN REGION with TOTAL-ORDER-AMOUNT over a certain value. This query not only involves relationship processing (ORDER and CUSTOMER), but also index processing from two different tables.

The next section deals with relationships, which are not only important in their own right, but also contribute to the cost-effective solution of the above-mentioned query types.

#### 4.2.3 Relationship Component

Relationships are the third component in the storage structure of any database. If the DBMS supports building static relationships, then the form of the relationship is normally a relative row address established during loading. If the DBMS has dynamic facilities, then the form of the relationship is always a shared value stored in the source and target row of the relationship. Notwithstanding these differences, both static and dynamic relationship DBMS facilities must support referential integrity. Finally, relationships dramatically affect the ability of the DBMS to handle interrogations involving multiple tables and the indexed columns within them.



#### 4.2.3.1 Relationship Basics

The manner in which relationships are bound into rows constitutes a fundamental difference between static and dynamic relationship DBMSs. In a static relationship DBMS, the relationship stated in the DDL is bound into the rows by the DBMS during data loading or update. In a dynamic relationship DBMS, relationships exist in the rows as shared values. Dynamic relationships between tables are presumed by the user and then uncovered by the DBMS when it actually finds the rows during an interrogation.

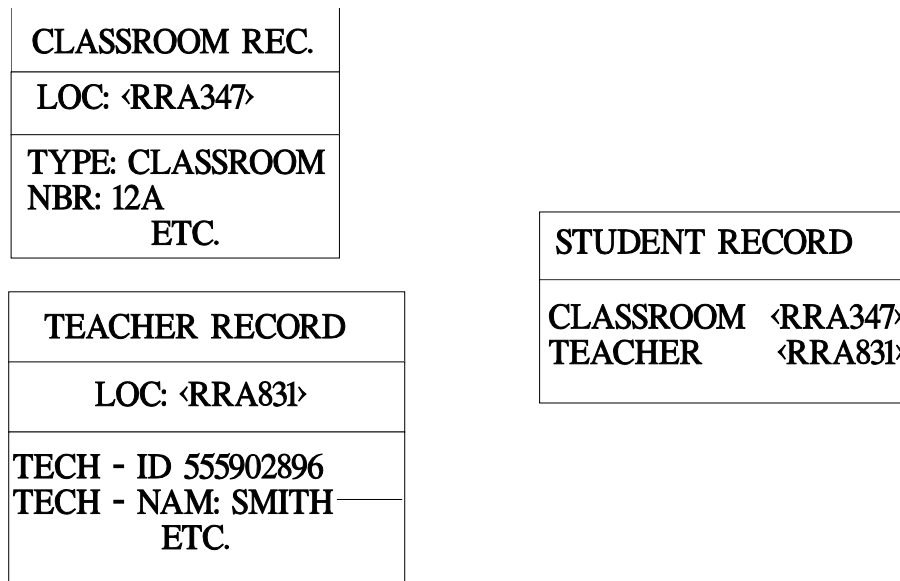
##### 4.2.3.1.1 Static Relationship Basics

A static relationship is most often a relative row address. The address of the *next* row is stored in the *prior* row. The address of the *prior* row is stored in the *next* row, and the address of the *owner* row is stored in all the member rows. Figure 4.15 illustrates the use of embedded relative row addresses. In the STUDENT row, the relative row address <RRA347> points to the actual CLASSROOM row stored at relative location <RRA347>. The STUDENT row also contains the relative row address, <RRA831>, which points to the actual TEACHER row stored at <RRA831>. Figure 4.16 illustrates the embedded pointers linking one owner ADMINISTRATION row to the four EMPLOYEE rows, ABLE, BAKER, JONES, and SMITH. The EMPLOYEE rows are shown sorted, while the actual physical storage order (relative row addresses) is quite different from the illustrated sorted order. These sets of pointers linking owner to member, member to member, and member to owner are commonly referred to as chains.



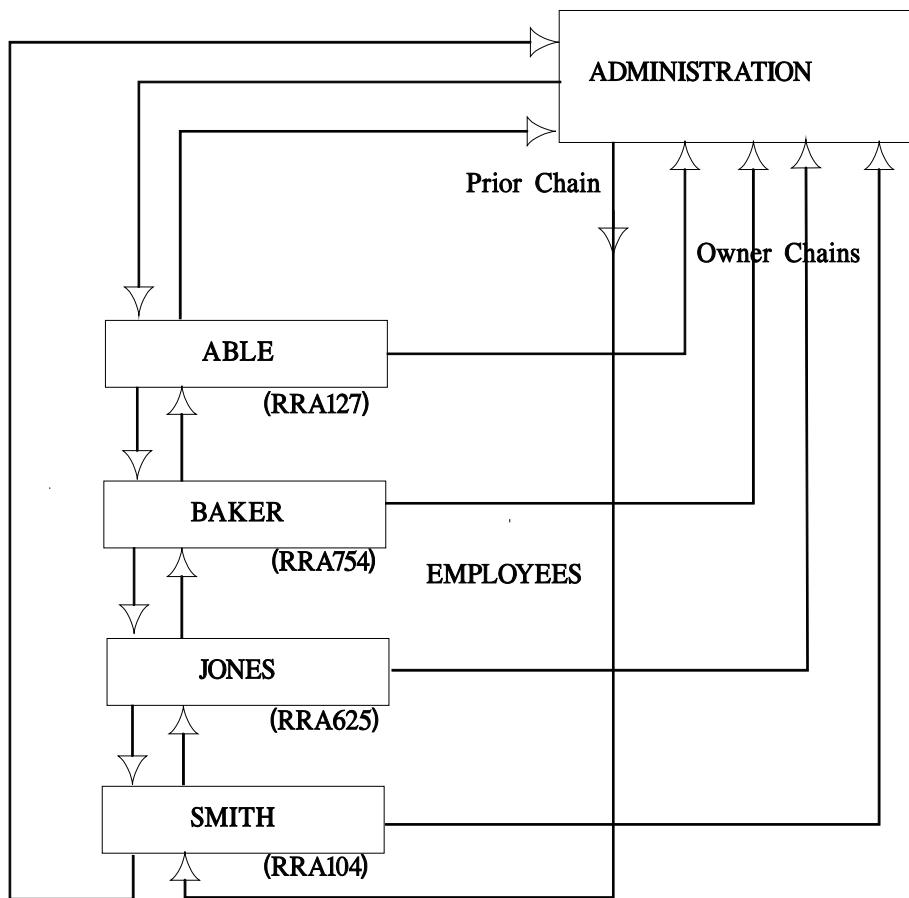
The three types of chains are next, prior, and owner. The next chain connects an owner to the first member, then connects the first member through the members to the last member and back to the owner. The prior chain connects the owner to the last member, then connects the last member through the prior members to the first member and back to the owner. The owner chain directly connects each member to its owner. These three chain techniques are illustrated in Figure 4.16.

There are a number of variations to this basic static relationship scenario. Three are: pointer arrays, primary key as owner relationship values, and separated relationship arrays.



**Figure 4.15.** Embedded static relationships.





SET: DEPT-EMPL  
 OWNER: DEPARTMENT  
 MEMBER: EMPLOYEES  
 ORDER: SORTED ON LAST NAME

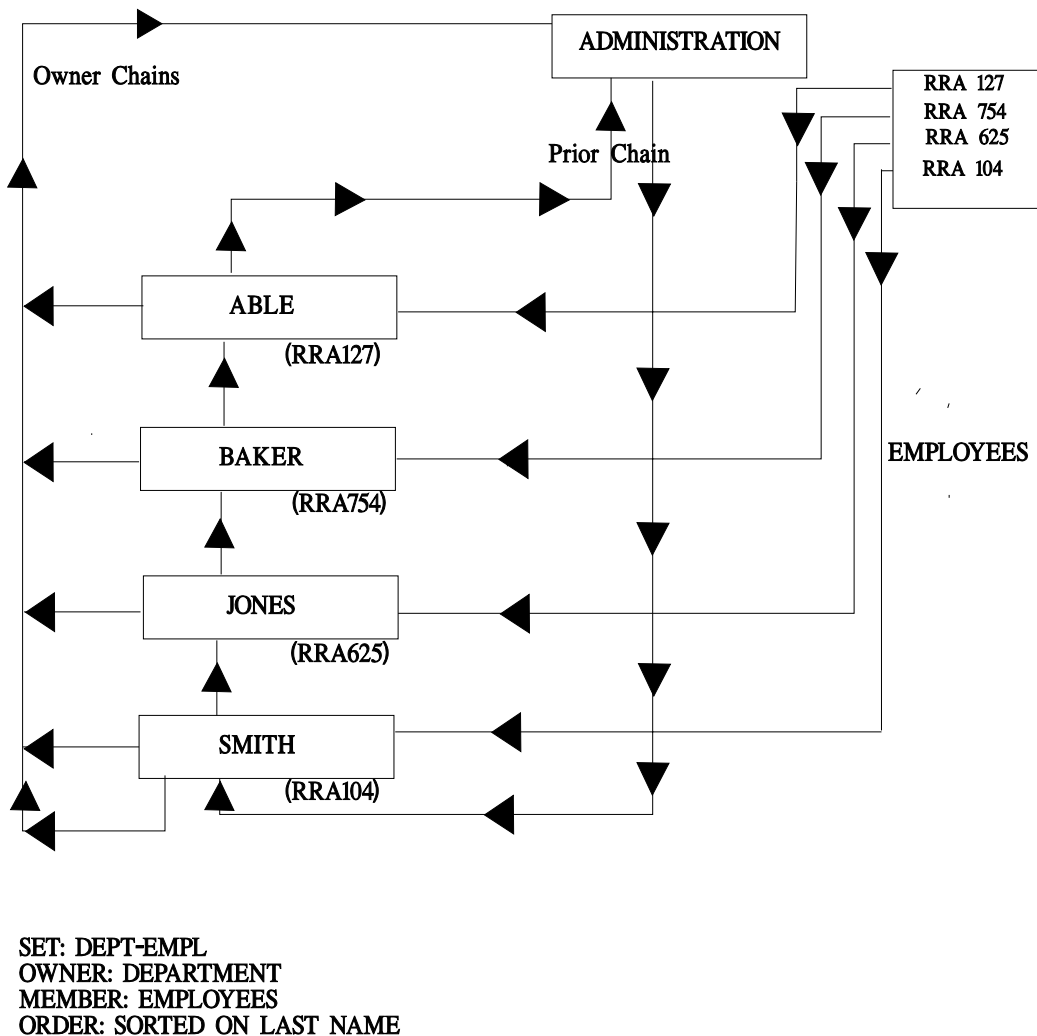
**Figure 4.16.** Traditional next, prior, and owner chains.



#### 4.2.3.1.1.1 Pointer Arrays

Some DBMSs place an array of member relative row addresses in the owner row. This makes it possible to select  $n$ -th member without having first to access all the preceding members in the chain (see Figure 4.17). Another advantage is that, if a query is constructed to determine the number of employees in the ADMINISTRATION department, the DBMS only has to count the links in the list that is stored in the owner. It does not have to access all the individual rows for ABLE, BAKER, JONES, and SMITH and then count them.

If, however, the query must count the rows at a level below employees, for example the



**Figure 4.17** Pointer Array: Next, Prior, and Owner Pointers



number of courses taken by all the employees, then all the employee rows have to be accessed to get access to the array of RRAs for employee courses.

A real drawback of this scheme is that the row has to be variable length. For each row, its length not only consists of the actual data, which also may be variable length, but also must consist of the total number of RRA lists of each set of descendent tables.

#### **4.2.3.1.1.2 Primary Key as Owner Relationship Value**

Some DBMSs place the primary key value in the member row as the owner address. This has both an advantage and a drawback. The advantage is that if the owner is moved, then its address, which also changes, does not have to be changed in each of its members. The drawbacks are two:

- Access to the owner can only be through a primary key search.
- To efficiently access the owner, the owner's primary key must be indexed.

#### **4.2.3.1.1.3 Separated Relationship Arrays**

SYSTEM 2000, for example, extracts all the relationship mechanisms from the rows and places them in a separate component of the storage structure. Figure 4.18 illustrates this technique. The ??? marks in Figure 4.18 are this book's convention to show that the illustration is partial. Because SYSTEM 2000 is hierarchical, each database contains only one complex table consisting of multiple repeating group columns that are also called segments. The data structure illustrating the relationship among the EMPLOYEE, COURSE, and DEGREE segment instances is illustrated in Figure 4.19. Each relationship array from Figure 4.18 represents the hierarchical relationships that a specific segment instance has with its related segment instances. In this particular example, taken from SYSTEM 2000, there are four relationship arrays, one for each of the four segment instances. The segment instances are <RRA5000> (EMPLOYEE), <RRA7500> (COURSE), <RRA8000> (COURSE), and <RRA9700> (DEGREE).

Each relationship array instance contains five parts: segment type, next, member, owner, and address. The segment type component indicates the type of segment type represented by the array (COURSE or DEGREE or EMPLOYEE). For example, in <RRA10> the segment type indicator states that the relationship array represents an EMPLOYEE segment instance.



RELATIONSHIP POINTERS

DATA RECORDS

10

RTI	NEXT	MEMBER	OWNER	ADDRESS
EMPLOYEE	????	20	????	5000

5000
EMPLOYEE

20

RTI	NEXT	MEMBER	OWNER	ADDRESS
Course	30	????	10	7500

7500
COURSE

30

RTI	NEXT	MEMBER	OWNER	ADDRESS
Course	40	????	10	8000

8000
COURSE

40

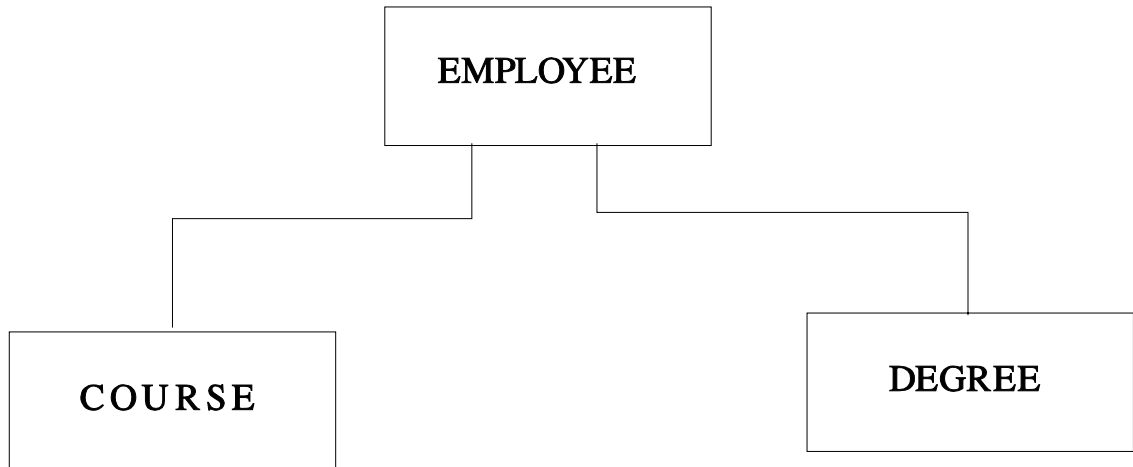
RTI	NEXT	MEMBER	OWNER	ADDRESS
Course	????	????	10	9700

9700
DEGREE

RTI means record type identifier

**Figure 4.18** Separated Relationship Pointers





**COUNTS: EMPLOYEES: 50,000**

**COURSES: 3,000,000 OR 60 PER EMPLOYEE**

**DEGREE: 75,000 OR 1.5 PER EMPLOYEE**

**Figure 4.19** Employee Data Structure Data Record Instance Counts and Population Ratios

The next component contains the RRA of the next relationship array instance along a next chain. For example, in the <RRA20>, the next component contains the value <RRA30> indicating that the relationship array at <RRA30> represents the next segment instance along a courses chain.

The member component contains the RRA of the first member in a child (member) chain. For example, <RRA10> contains an entry <RRA20> that is the address of the first *child* for the employee.

The owner component contains the RRA of the relationship array that represents the segment instance of the owner. For example, <RRA30> contains the address <RRA10>, which points to the EMPLOYEE relationship array.

Take note of the fact that in SYSTEM 2000 there is only one member component in each relationship array. That would seem to imply that any table could only have one descendent table. SYSTEM 2000, to accommodate multiple descendent tables, links all member rows, regardless of table (COURSE or DEGREE). The segment type indicator (STI) component of the relationship array indicates which table is represented. That is, whether it is an EDUCATION, COURSE, DEGREE, or EMPLOYEE relationship array. That is the reason why DEGREE <RRA40> is a NEXT relationship instance to COURSE.

When a particular segment instance is actually required, for example an EMPLOYEE segment instance, access is accomplished by determining the actual segment instance address, <5000>.





This technique of separating relationships from the actual segment instances has its advantages and disadvantages. The principal advantages are three:

- To count member segment instances without having to access them
- To obtain a member segment instance on the fourth level without first having to pass through three levels of rows
- To AND and OR index created lists that belong to different segment types

The principal disadvantage is:

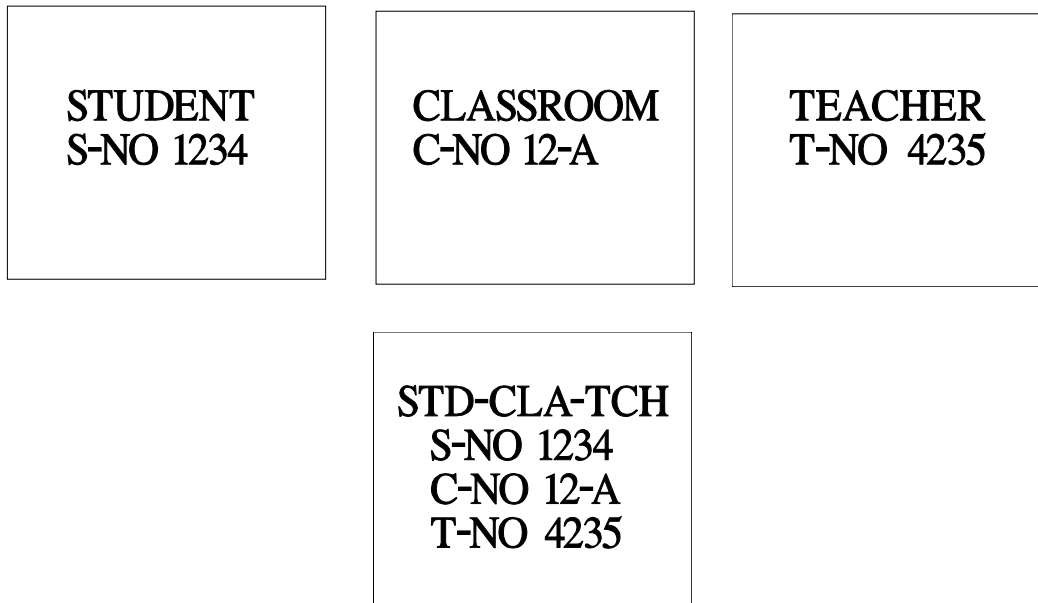
- If the segment instances themselves are required for the interrogation, then accessing relationships separately from the segment instances consumes slightly more resources than accessing segment instances alone with the relationships buried in them.
- Maintenance is slightly more expensive as two different storage structure components have to be updated during row adds or deletes.

#### **4.2.3.1.2 Dynamic Relationships**

If a DBMS's relationships are pointers, its relationship mechanism is static. If there are no separate relationship structures, and if the only relationship mechanism between two rows is a shared data value, then the DBMS's relationship mechanism is dynamic. It is false to state that static relationship DBMSs have relationship mechanisms and dynamic relationship DBMSs do not. If that were true, then the dynamic relationship DBMS that performed relational operations such as JOIN is doing so on the basis of magic. The relationship is based on values present in both of the related rows. For example, in Figure 4.20, the STUDENT can be JOINed with the TEACHER because TEACHER-NUMBER and the STUDENT-NUMBER are both columns that are defined as part of the table STUDENT-TEACHER-CLASSROOM. And the STUDENT row can be JOINed with the CLASSROOM row because the columns STUDENT-NUMBER and CLASSROOM-NUMBER are defined in the STUDENT-TEACHER-CLASSROOM table.

An owner-member relationship is accomplished in a dynamic relationship DBMS by defining the primary key (or candidate key) column from the owner table as a column in the member table. For example, in Figure 4.21 the primary key EMPLOYEE-ID from EMPLOYEE is defined as part of a compound column COURSE-ID and EMPLOYEE-ID in COURSE. An owner-member relationship between two or more owner rows is defined similarly except that their common member table might contain only the concatenation of the primary keys of the owners. Figure 3.30 represents such an example.





**Figure 4.20.** Value based relationships.

#### 4.2.3.2 Sophisticated Relationships

In the section on indexes, the case study contrasts the different types of index structures and compares list processing with row processing. The key difference is that sophisticated indexes and list processing indexes make certain applications possible that are otherwise impractical.

The index case study centers on selecting rows from the same type. If the WHERE clause involves conditions from different but connected tables, the DBMS not only must find the rows from each table, but also must figure how to intersect the rows from the different tables. The methods of resolution differ if the DBMS processes relationships in a static or dynamic manner.



Figure 4.21 presents the data definition language for three personnel tables. The data definition language is constructed along value-based relationship guidelines so that the relationships are obvious. If the DBMS is NDL then the foreign keys would be replaced by SET clauses after the definition of all the tables. The population of rows for each table is shown along with its graphic structure in Figure 4.19. An EMPLOYEE may be taking COURSES and may have also received one or more DEGREEs. Take the interrogation requirement such as:

```
PRINT EMPLOYEE-ID, EMPLOYEE-NAME WHERE
      EMPLOYEE-CLASS EQ FULL-TIME AND
      EMPLOYEE HAS (COURSE-NAME EQ ENGLISH AND
      COURSE-GRADE EQ A AND
      STATE-OF-SCHOOL EQ PENNSYLVANIA)
```

#### Schema is Personnel

##### Record is Employee

Primary key is Employee-Id  
 Secondary key is Employee-class  
 Columns are  
     Employee-Id  
     Employee-Class  
     Employee-Name

##### Record is Course

Primary key is Employee-Id, Course Id  
 Secondary key is Course-Name  
 Foreign key is Employee-Id references Employee  
     On update set null  
 Columns are  
     Employee-Id  
     Course-Id  
     Course-Name  
     Course-Grade

##### Record is Degree

Primary key is Employee-Id, Degree-Name  
 Secondary key is State-of-School  
 Foreign key is Employee-Id references Employee  
     On update set null

**Figure 4.21.** Data definition language for Personnel schema.



When this is executed against the database, the method of DBMS processing differs with the form of relationship (static or dynamic) and the type of indexing (list processing or row processing). The six cases, two dynamic and four static, are:

- Dynamic, row processing
- Dynamic, list processing
- Static, embedded pointers, row processing
- Static, embedded pointers, list processing
- Static, separated pointers, row processing
- Static, separated pointers, list processing

To help quantify the relative performance differences among these six cases, assume that 50% of the employees are full-time (25,000); 10% of all courses are English (300,000); 10% of all course grades are A (300,000); and 15% of all degrees (11,250) are from schools in PENNSYLVANIA. These percentages are uniformly distributed through the structure. Thus, of the 25,000 full-time employees, 150,000 of the courses taken by the employees are English (six per employee); 10% of the employees in these English courses receive an A (15,000). Assume further that 5000 of the full-time employees receive at least one A in an English course; and 5625 of the employee's graduation schools are from PENNSYLVANIA. Finally, assume that a total of 750 of the full-time employees pass all the selection criteria.

Also assume all indexes have hierarchically organized unique value portions, with a separated secondary multiple occurrence DBKEYs. It takes 4 accesses to find one employee, degree, or course. For secondary key accesses, it takes one additional access to obtain each 2000 row identifiers. For actual row accessing, it takes one access for each row. For embedded relationship processing, it takes only six accesses to obtain all COURSE rows, as there are 10 COURSE rows per DBMS row. Figure 4.19 indicates 60 courses per employee. For the same type of processing against the DEGREE member rows, it only takes one access (Figure 4.19 indicates 1 or 2 per employee). For separated relationship array processing it takes .25 accesses to obtain the relationship array instance implied by an index. The .25 access is due to the high density of arrays per DBMS row. For separated relationship arrays, it takes one access to process the entire set of relationships for each employee.

The steps in each of the six cases illustrate the type of processing performed. The actual program syntax to accomplish these steps varies from that depicted in the requirement. This interrogation requirement syntax was adapted from the SYSTEM 2000 DBMS. While not the same interrogation requirement, a network row processing DBMS syntax is constructed to conform to that depicted in Figure 4.22. Also, while not the same interrogation requirement, a dynamic list processing DBMS utilizes a syntax similar to that depicted in Figure 4.23. It is presumed that the programmer constructing the interrogation is supported by a language that permits optimum processing of the database's storage structure. For all these reasons, the steps in each case must differ. Finally, since every DBMS has a different access strategy and storage structure, the exact steps are likely to differ in some respects from those listed here. The point to this case study is not to compare the quantity of the steps, but to compare the efficiencies of the relationship processing alternatives.



```
10  SELECT SALESPERSON, ON ERROR GOTO 100, AT END GOTO 110

20  GET MEMBER CONTRACT, ON ERROR GOTO 100, AT END GOTO 10

30  GET MEMBER ORDER, ON ERROR GOTO 100, AT END GOTO 20

40  GET MEMBER ORDER - ITEMS, ON ERROR GOTO 100, AT END GOTO 30

    GET OWNER PRODUCT - SPECIFICATIONS, OR ERROR GOTO 100

    GET OWNER PRODUCT, ON ERROR GOTO 100

    IF PRODUCT - MFG - CITY EQ SALESPERSON - BIRTH - CITY AND

        PRODUCT - MFG STATE EQ SALESPERSON - BIRTH - STATE

        THEN PRINT SALESPERSON - NAME, PRODUCT - NAME,

            SALESPERSON BIRTH - CITY, SALESPERSON - BIRTH - STATE

    ELSE GOTO 40

100  PRINT "ERROR"

110  END
```

**Figure 4.22.** Network (static) program illustration.



```
CONNECT PRODUCT TO ORDER - ITEM

  VIA PRODUCT - NBR EQ ORDER - ITEM - PRODUCT -NBR

FIND MATCH AND KEEP PRODUCT - NAME, ORDER - ITEM,

  PRODUCT - MFG - CITY, PRODUCT - MFG - STATE IN LIST - 1

CONNECT SALESPERSON TO CONTRACT

  VIA SALESPERSON - NBR EQ CONTRACT - SALESPERSON - ID,

FIND MATCH AND KEEP SALESPERSON - NAME, CONTRACT - ID,

  SALESPERSON - BIRTH - CITY, SALESPERSON - BIRTH - STATE

  IN LIST - 2

CONNECT LIST - 2 TO ORDER VIA CONTRACT - ID OF LIST - 2

  IQ ORDER - CONTRACT - ID

FIND MATCH AND KEEP ORDER - ID, SALES - PERSON - NAME,

  SALESPERSON - BIRTH - CITY, SALESPERSON - BIRTH - STATE

  IN LIST - 3

CONNECT LIST - 1 TO LIST - 3

  VIA ORDER - ID OF LIST - 1 EQ ORDER - ID OF LIST - 3 AND

  SALESPERSON - BIRTH - CITY EQ PRODUCT - MFG - CITY AND

  SALESPERSON - BIRTH - STATE EQ PRODUCT - MFG - STATE

FIND MATCH AND PRINT SALESPERSON - NAME, PRODUCT - NAME,

  SALESPERSON - BIRTH - CITY, SALESPERSON - BIRTH - STATE
```

**Figure 4.23.** ILF or relational (dynamic program illustration).



If the interrogation is presented to a dynamic relationship DBMS and if the DBMS' index strategy is row processing (first case), the steps:

- 17 accesses to obtain the DBKEYs of all full-time employees.
- 25,000 accesses to obtain the full-time employee rows for their primary key values.
- 300,000 accesses to obtain the course rows for the ENGLISH courses.
- Some amount of processing to check each course row to see if the grade received is an A, and then more processing to see if the EMPLOYEE-ID for that course matches one of the employees already selected. If so, then the employee is kept. All employees not matched are dropped.
- 10 accesses to obtain the primary keys of all degree rows (STATE-OF-SCHOOL EQ PENNSYLVANIA).
- 11,250 accesses to obtain the degree rows.
- Some amount of processing to check each degree to see if the EMPLOYEE-ID on each selected row the employees that remain; if so, then the employee is kept. All employees not matched are dropped.

The total accesses for this case are 336,277.

If the interrogation is presented to a dynamic relationship DBMS (second case), and if the DBMS' index strategy is list processing, the steps are:

- 335 accesses to build four lists, one for full-time EMPLOYEEs (17 accesses), one for DEGREEs from PENNSYLVANIA schools (10 accesses), one for ENGLISH COURSEs (154 accesses), and one for courses receiving an A (154 accesses).
- Some amount of processing to AND the two course table identifier lists to produce just one list of DBKEYs for the two course select conditions.
- 25,000 accesses to obtain the employee tables to obtain their primary key values.
- 150,000 accesses to select the employee's COURSE rows and to determine whether the primary keys of the selected COURSE rows match the primary keys of the rows on the COURSE list. Employees not having an A grade are dropped from the EMPLOYEE list.
- 7,500 accesses to obtain the employee's DEGREE rows and to determine whether the primary keys of the selected DEGREE rows match the primary keys of the



rows on the DEGREE list. Any employee not having a PENNSYLVANIA degree is dropped, leaving a list of only those employees meeting all four criteria.

The total accesses is 182,835.

If the interrogation is presented to a static relationship DBMS, the relationship mechanism is embedded pointers, and the DBMS' index strategy is row processing (third case), then the steps:

- 17 accesses to obtain the DBKEYs of all full-time employees.
- 25,000 accesses to obtain the full-time employee rows.
- 75,000 accesses to process each employee's COURSE chains to determine if the course is ENGLISH and the grade is A. If yes, then the employee is initially identified as satisfying the second criterion. Employees not tagged after processing all selected employees are dropped from further processing. It should be noted that if all the course rows were checked then 150,000 accesses are needed. 75,000 is estimated on the assumption that on average only half the rows need to be accessed to satisfy the select condition that an employee has an A on one or more of the ENGLISH courses.
- 5000 accesses to process each employee's degree chain to determine if it is from a PENNSYLVANIA school. If yes, then the employee also satisfies the last condition. Employees not tagged after processing all selected employees are dropped from further processing.

The total accesses are 105,017.

If the interrogation is presented to a static relationship DBMS, the relationship mechanism is embedded pointers, and the DBMS' index strategy is list processing (fourth case), then the steps are:

- 308 accesses to build two lists, one for ENGLISH COURSEs (154 accesses), and one for courses receiving an A (154 accesses). Only two lists are built as the strategy starts at the COURSE table, moves up to the EMPLOYEE table and then accesses the DEGREE tables.
- Some amount of processing to AND the two course table identifier lists to produce just one list of DBKEYs for the two course select conditions.
- 15,000 accesses to obtain the A grade ENGLISH courses.
- 15,000 accesses to obtain the set of all employees that are the parents of the selected courses. Upon retrieval the 5000 full-time employees are found.





- 5,000 accesses to process the employees' degree chains to determine if the degree is from a PENNSYLVANIA school. If yes, then the employee is identified as satisfying the last criterion. Employees not tagged after processing all selected employees are dropped from further processing.

The total accesses are 35,308.

If the interrogation is presented to a static relationship DBMS, the relationship mechanisms are separated, and the DBMS' index strategy is row processing (fifth case), then the steps are:

- 17 accesses to perform an index search that obtains the addresses of the relationship arrays for all the full-time employees.
- 6250 accesses to obtain the relationship arrays from the addresses in step 1 for the full-time employees.
- No accesses to find the course relationship arrays for the employee.
- 75,000 accesses to process the employee's COURSE chains to determine if the course is ENGLISH and the grade is 'A'. If yes, then the employee is initially identified as satisfying the first AND. Employees not tagged after processing all selected employees are dropped from further processing. It should be noted that if all the course rows are checked, then 150,000 accesses are needed. 75,000 is estimated on the assumption that only half the rows are accessed to satisfy the select condition that an employee has an A on one or more of the ENGLISH courses.
- 5,000 accesses to process the employee's degree chains to determine if it is from a PENNSYLVANIA school. If yes, then the employee also satisfies the last condition. Employees not tagged after processing all selected employees are dropped from further processing.

The total accesses are 86,267.

If the interrogation is presented to a static relationship DBMS, the relationship mechanisms are separated, and the DBMS' index strategy is list processing (sixth case), then the steps are:

- 335 accesses to build four lists of relationship array addresses, one for full-time EMPLOYEES (17 accesses), one for DEGREES from PENNSYLVANIA schools (10 accesses), one for ENGLISH COURSES (154 accesses), and one for courses receiving an 'A' (154 accesses).
- Some amount of processing to AND the two course table identifier lists of relationship array addresses to produce just one list for the two course select conditions.



- 6250 accesses to obtain the relationship arrays for the full-time employees.
- 3,750 accesses to obtain the relationship arrays for the A ENGLISH courses.
- 2813 accesses to obtain the relationship arrays for the PENNSYLVANIA schools.
- Each course relationship array instance is processed by finding its employee owner relationship array instance. If the owner's relationship array address is the address of an already selected employee, the employee relationship array is tagged as satisfying both the full-time and course conditions. After all the course relationship array instances are processed, all the untagged employee arrays are dropped.
- Each PENNSYLVANIA school array instance is processed by finding its employee owner relationship array instance. If the owner's relationship array address is the address of an already selected employee, the employee relationship array is tagged as satisfying the last criterion. After all the PENNSYLVANIA relationship array instances are processed, all the untagged employee arrays are dropped.

The total accesses are 13,148.

As can be seen from the different cases, the effect of the storage structure's indexes, list processing, and types of relationships is quite significant.

#### **4.2.3.3 Relationship Summary**

All databases having multiple tables have relationships between the rows. If the relationships are dynamic, the DBMS will always incur the extra processing overhead associated with primary and secondary key access as the method of processing relationships. If the relationships are static, the DBMS will always know exactly where the target of the relationship is located, thus avoiding the extra processing incurred by the dynamic relationship DBMS.

The major value of having static relationships separated from the rows is achieved when indexes point to these separated relationships rather than to the rows. When this in turn is coupled with sophisticated indexes, applications such as the one described in the case study become practical.

The order of sophistication of relationship constructions and their relative performance degradations from best to worst are presented in Figure 4.24

#### **4.2.4 Data Component**

DBMSs vary greatly in the storage of rows. The variations range from storing all the rows from each table in separate O/S files to storing all the rows for all the tables in one O/S file. Broadly,



the three major sets of distinctions relate to O/S file storage formats, row storage formats, and column storage formats. For O/S file storage formats the most common alternatives are:

- One table per file
- One file for all tables
- Multiple tables per file
- Multiple files per table
- Rows and key formatted files

For the table storage formats, the alternatives are:

- Fixed format, fixed length tables
- Fixed format, variable length tables
- Variable format tables

CASE	TIMES SLOWER THAN BEST	RELATIONSHIP TYPE	NUMBER OF ACCESSES	ACCESS TIME
1	25	Dynamic, row processing	336,277	69.00 min
2	14	Dynamic, list processing	182,835	37.62 min.
3	8	Static, embedded pointers, row processing	105,017	21.61 min
5	6	Static, separated pointers, row processing	85,267	17.54 min
4	3	Static, embedded pointers list processing	35,038	7.27 min
6	1	Static, separated pointers, list processing	13,148	2.71 min

**Figure 4.24.** Summary comparison of the relative performance of relationship mechanisms.



RELATIONSHIP BINDING CLASSIFICATION				
TYPE OF FILE FORMAT	STATIC		DYNAMIC	
	NETWORK	HIERARCHICAL	ILF	RELATIONAL
1 Table per file	IDMS/R SUPRA		M-204 Inquire	DB-2 SUPRA
All table in 1 file	IDMS/R	SYSTEM 2000	ADABAS M-204	IDMS/R
Less than one table per file	IDMS/R	IMS	M-204  Focus	
Less than one file per table				

**Figure 4.25.** Data file storage formats by relationship type, data model, and DBMS for various popular IBM mainframe DBMSs.



RELATIONSHIP BINDING CLASSIFICATION				
TYPE OF DATA RECORD FORMAT	STATIC		DYNAMIC	
	NETWORK	HIERARCHICAL	ILF	RELATIONAL
Data record and key format files	IDMS/R		M-204	
Fixed format fixed length data records	IDMS/R SUPRA		M-204 Inquire	DB-2 SUPRA
Fixed Format variable length data records	IDMS/R	SYSTEM 2000	ADABAS M-204	IDMS/R
Variable format data records			M-204	

**Figure 4.26** Data Record Instance Storage Formats by Relationship Type, Data Model, and DBMS for Various Popular IBM Mainframe DBMSs

RELATIONSHIP BINDING CLASSIFICATION				
TYPE OF DATA ELEMENT FORMAT	STATIC		DYNAMIC	
	NETWORK	HIERARCHICAL	ILF	RELATIONAL
Simple data element formats	IDMS/R	IMS SYSTEM 2000	M-204 ADABAS FOCUS	IDMS/R SUPRA
Complex data element formats	IDMS/R		M-204 ADABAS FOCUS	

**Figure 4.27** Data Element Storage Formats by Relationship Type, Data Model, and DBMS for Various Popular IBM Mainframe DBMSs



For column storage formats, the alternatives are:

- Simple column formats
- Complex column formats

As indicated in Figures 4.25, 4.26, and 4.27 the data model of the DBMS provides no definite preset pattern among file, row, and column formats. The DBMSs that are cited more than once offer multiple format choices. In general, Figures 4.29 through 4.37 contain a definition part (e.g., Figure 4.29a), and an example part (e.g., Figure 4.29b). References in this section are only to the general figure number (i.e., 4.29) rather than to the individual part (i.e., Figure 4.29a). Each figure contains a BNF (Backus Naur Form) format, which is defined in Appendix A.

.

#### **4.2.4.1 File Storage Formats**

Figures 4.28 through 4.32 illustrate and provide examples of the different file storage formats. A file is an operating system file. These storage formats are used in DBMSs as indicated in Figure 4.25. In general, the more options a DBMS offers, the more it can be optimized. More options, however, increase a DBMS's complexity and training requirements.

##### **4.2.4.1.1 One Table per File**

The simplest storage structure of all is one table per file. Each file contains all the rows from only one table. Figures 4.28a and 28b illustrate this file storage format. The BNF indicates that the database has one or more files. Each file is defined to have one or more rows for the same table. These rows can be sorted or loaded in some primary key order. These tables are easy to reorganize; all that is required is to modify the table's format by adding or deleting a column.

For simple databases performing simple functions, the single file per table design is appropriate. If, however, the application requires sophisticated performance and other types of flexibility, the design is just too simple. For example, if a report is desired that pulls together many rows from different tables, the number of accesses can be prohibitive.



BNF

Database ::= <file X> [ <file Y> ...]

<file> ::= <Table T> [ <Table T>...]

DDL

DATABASE IS personnel

TABLE is department

TABLE is employee

RELATIONSHIP is deptempl

OWNER is department

MEMBER is employee

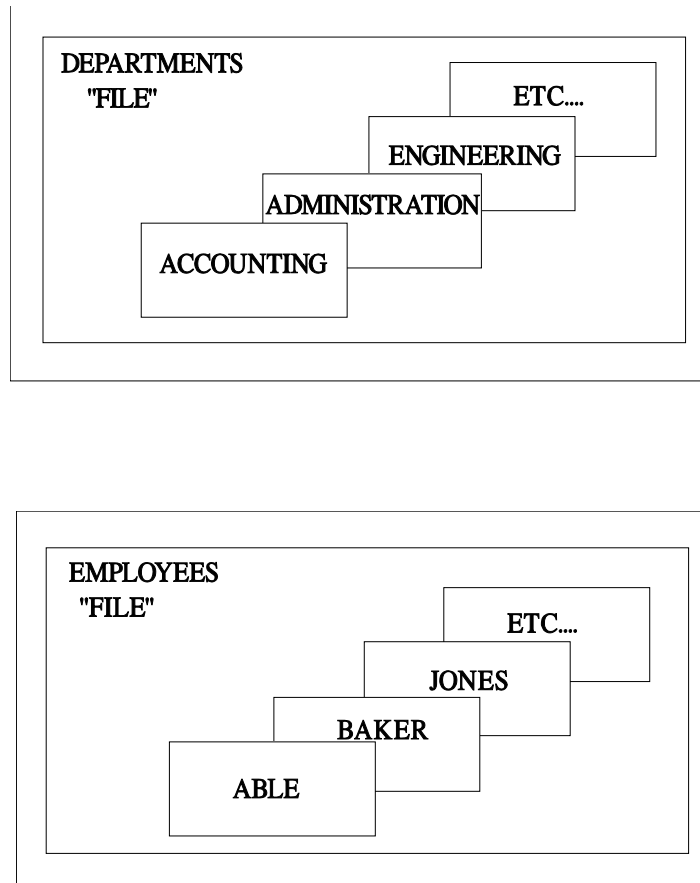
DSDL

FILE department CONTAINS department

FILE employee CONTAINS employee

**Figure 4.28a.** One table per file organization.





**Figure 4.28b.** One table per file organization.





#### 4.2.4.1.2 One File for All Tables

The one file for all rows format is the inverse of the first format. Each file is structured to contain all the rows from all tables. In the BNF from Figure 4.29a, the data portion of the database consists of one file. That file contains one or more rows of one or more different tables. Figure 29b illustrates the actual file organization.

Since the file is to contain rows of different tables, the DBMS must have a way to locate a specific row within each DBMS row. This is sometimes accomplished by storing the actual address of the start of the row in whatever row access mechanisms exist, for example, primary or secondary indexes. The actual length of a row is either determined through computations of the table's column lengths or through computations of the actual column value instance lengths. In the case of fixed format fixed length rows, the defined column lengths are defined in the DDL. In the case of variable length rows, the column lengths are computed through information stored in the actual row.

Another method of locating rows is to store the address of the DBMS row (sometimes called the relative row address) in the row access mechanism. When a DBMS row instance is accessed, the DBMS determines the actual starting address of the row by finding the row's primary key value (and its paired start address) in a special section of the DBMS row instance.

In general, the second method of row location is better than the first as the DBMS is free to *shuffle* rows around within the DBMS row to accommodate row expansions, deletions, or additions without first having to modify any part of the row access mechanism. Under either of these formats, load engineering is possible (see Section 4.4).



BNF

Database ::= <file X>

<file X> ::= <data record S> [ <data record S>...]  
[ <data record T> [ <data record T>...] ...]

DDL

DATABASE IS personnel  
RECORD is department  
RECORD is employee  
RELATIONSHIP is deptempl  
OWNER is department  
MEMBER is employee  
MODE is via

DSDL

FILE db CONTAINS department, employee

**Figure 4.29a** One File for all Tables.

PHYSICAL Row 1		
ACCOUNTING	ABLE	BAKER
GRODMAN	KERR	PERKINSON
ADMINISTRATION	EDMONDS	CRITENDEN

PHYSICAL Row 2		
FRANCIS	GIMBLE	HARRISON
JONES	ENGINEERING	APPLETON
QUINCY	RICHARDSON	ZIEGFELD

**Figure 4.29b** Example of One File For All Tables



#### 4.2.4.1.3 Multiple Tables per File

Another way of organizing rows is to allow rows of different types to be stored in the same file. The BNF from Figure 4.30 indicates that the data portion of a database consists of one or more data files. A file contains one or more rows of one or more tables. Note from the BNF that all the rows from one table are restricted to just one file. This capability should be in addition to the ability to store all the rows from all the tables in a single file. That is, a database designer should be able to define the tables separately, and then decide, on the basis of performance and the like, which files are assigned which tables.

If multiple tables are able to have their rows stored in the same file, then efficiency can be improved in some situations. For example, Figure 4.30 illustrates the storage of two tables, DEPARTMENT and EMPLOYEE. A row for the ACCOUNTING table is stored in DBMS row 1. Next to it are the employees in that department, ABLE, BAKER, GRODMAN, KERR, and PERKINSON. The next department, ADMINISTRATION, is stored along with its employees EDMONDS and CRITENDEN. The main benefit of this type of structuring is that member rows can be as physically close to their owners as possible. Reports of DEPARTMENTS and their EMPLOYEEs only require one access to obtain all the related rows. Of course, the disadvantage of such a scheme is that reorganization becomes more complicated, as more than one type of row has to be accommodated during the reorganization process. Additionally, once rows of one type are spread over a large area, concentrated reporting of data from just one table consumes more resources.

#### 4.2.4.1.4 Multiple Files per Table

If an application has millions of rows in a single table, it is often useful to spread the set of rows over multiple files. The BNF in Figure 4.31 illustrates this file storage format. In this BNF, the data portion of the database is contained in one or more files. Each file contains rows from one or more tables. The BNF for both file definitions is exactly the same, indicating that the rows from the same table can be stored in one or more data files. The DBMS knows which set of rows are in which file through the data storage definition language (DSDL) intermediate allocation table. In this example, the database administrator must state, for example, that the department rows AA . . . DZ are stored in file 1 while the remaining rows are stored in file 2. This capability permits some sets of the rows to be moved off-line until needed. For example, if the EMPLOYEE table stored full-time, part-time, and former employees, it might be wise to remove the former employees from on-line status, as they are not likely to be used often. Another reason for splitting up the rows might be to spread all the ORDERS for a 24 hour order-entry system into different files so that maintenance, backup, and consolidated reporting could be performed on the *closed* (i.e., off-line) sections of the ORDER table. Without the ability to split a table into multiple files, this is impossible.



BNF

Database ::= <file X> [<file Y>...]

<file X> ::= <table S> [ <table S>...]  
 [ { { <Table T> [ <table T>... ] } ... }

<file Y> ::= <table U> [ <table U>...]  
 [ { { <table V> [ <table V>... ] }...} ]

DDL

DATABASE IS personnel  
 RECORD is department  
 RECORD is employee  
 RELATIONSHIP is deptempl  
 OWNER is department  
 MEMBER is employee  
 MODE is via

DSDL

FILE db CONTAINS department, employee

**Figure 4.30a.** Multiple files per table.

FILE 1, DBMS row 1		
ACCOUNTING	ABLE	BAKER
ADMINISTRATION	EDMONDS	CRITENDEN

FILE 1, DBMS row 2		
JONES	ENGINEERING	APPLETON
QUINCY	RICHARDSON	ZIEGFELD

**Figure 4.30b.** Example of multiple tables per file.



BNF

Database ::= <file X> [<file Y>...]

<file X> ::= <table S> [ <table S>...]  
 [ { { <table T> [ <table T>... ] } ... } ]

<file Y> ::= <table S> [ <table S>...]  
 [ { { <table T> [ <table T>... ] } ... } ]

DDL

DATABASE IS personnel  
 TABLE is department  
 TABLE is employee  
 RELATIONSHIP is deptempl  
 OWNER is department  
 MEMBER is employee  
 MODE is via

DSDL

FILE one CONTAINS department WITH department - id FROM aa to dz  
 FILE two CONTAINS department WITH department - id FROM ea to zz

**Figure 4.31a.** Multiple files per table.

FILE 1, DBMS row 1		
ACCOUNTING	ABLE	BAKER
ADMINISTRATION	EDMONDS	CRITENDEN

FILE 2, DBMS row 1		
JONES	ENGINEERING	APPLETON
QUINCY	RICHARDSON	ZIEGFELD

**Figure 4.31b** Example of Multiple Files Per Table



#### 4.2.4.1.5 Rows and Key Formatted Files

Some DBMSs have DBMS rows that contain *intelligence* about the contents of the DBMS row itself. The BNF in Figure 4.32 illustrates the format of DBMS rows in any database file. A DBMS row is either a simple format or a key formatted row. In the former case, the DBMS row consists of a series of rows, from *top to bottom*. The exact starting address of the row must be stored elsewhere in the database's storage structure.

In the later case, the exact starting address of the row is determined only after the DBMS row is obtained. After the DBMS scans the set of row keys to find the correct key, it then knows the exact starting address of the row.

As shown in Figure 4.32, the actual DBMS row format typically has a DBMS row broken into two parts. The first part is for the storage of the actual rows. The second part is for the storage of primary key values and addresses of the start of the row. This permits the contraction and expansion of rows within the DBMS row. It also permits the expansion of the file size resulting from increasing DBMS row size without the rows leaving their home DBMS row. This type of DBMS row construction is most often used with the multiple table file storage format and with the following table storage formats:

- Fixed format, fixed length rows
- Fixed format, variable length rows
- Complex row formats
- Variable format row formats



BNF

<file>::= { <simple DBMS row>... } / { <key formatted DBMS row>... }

<simple DBMS row >::= { <table A> [ <table B>... ] }

<key formatted DBMS row>::= { <table A> [<table B>... ]  
   { <table address key A>  
   [ <table address key B>... ] }

<table address key>::= { <table start address>  
                                   <table key value> }

DDL

DATABASE is personnel  
 TABLE is departments  
 TABLE is employee  
 RELATIONSHIP is deptempl  
 OWNER is department  
 MEMBER is employee  
 MODE is via

DSDL

Any DSDLs from figures 4.28 through 4.31

**Figure 4.32a.** Rows and key formatted files.



PHYSICAL RECORD 1		
RA=100 PK=ACCOUNTING ETC	RA=200 PK=ABLE ETC	RA=210 PK=BAKER ETC
RA=220 PK=GRODMAN ETC	RA=230 PK=KERR ETC	RA=240 PK=PERKINSON ETC
RA=250 PK=ADMINISTRATION ETC	RA=350 PK=EDMONDS ETC	RA=360 PK=CRITENDEN ETC
KEY-----RELATIVE ADDRESS MATCH		
100 ACCOUNTING 220 GRODMAN 250 ADMINISTRATION	200 ABLE 230 KERR 350 EDMONDS	210 BAKER 240 PERKINSON 360 CRITENDEN

**Figure 4.32b** Example of Data Record Instances and Key Formatted Files

#### 4.2.4.2 Table Storage Formats

Figures 4.33 through 4.35 illustrate the different row storage formats. These row storage formats are used in DBMSs as indicated in Figure 4.26. In general, the more options a DBMS offers, the better its performance. Again, more options, however, increase a DBMS's complexity and training requirements.

The table format depends upon whether the table is fixed or variable length, and fixed or variable format.

##### 4.2.4.2.1 Fixed Format, Fixed Length Tables

The most traditional format for a row is fixed format and fixed length. In Figure 4.33, the BNF illustrates that the row consists of a series of simple column value instances. The length of each column's value is defined in the DDL, and space is allocated by the DBMS for all the column value instances within the row whether the column's value exists or not. The space for a value representation for every column defined is contained in every row. The value representation takes on one of two states: a fixed number of NULLs or a fixed length value. In the example contained in Figure 4.33, the LAST NAME column has a fixed length of 11 characters. If a last name of more than 11 characters is stored, all characters to the right of the 11th are chopped off. The total length of every fixed format, fixed length row is the sum of the defined column lengths.





In this example, the row length is 39 characters. Some DBMSs allow numeric data to be stored in compressed formats such as packed decimal or binary integers. In such cases the actual space consumed is less than the sum of the defined lengths.

A benefit of fixed length and fixed format rows is that the DBMS does not need to spend any time decoding lengths and formats. A drawback is that a very detailed understanding of the application's use of the data is needed so that these fixed length parameters can be set up in a way that results in no data loss.

#### BNF

```
<row> ::= <fixed format row>
        / <variable format row>

<fixed format row> ::= <simple row>

<simple row> ::= <simple column A>

                [ <simple column B>...]
```

#### DDL

LAST - NAME, TYPE IS CHARACTER 11

FIRST - NAME, TYPE IS CHARACTER 12

BIRTHDATE, TYPE IS DATE, FORMAT IS YYYYMMDD

SX, TYPE IS CHARACTER 1

**Figure 4.33a.** Fixed format and fixed length rows (simple).

SSN	LAST	FIRST	BIRTHDATE	SEX
525902896	GORMAN	MICHAEL	19410322	M

**Figure 4.33b** Example of Fixed Format and Fixed Length Records



#### 4.2.4.2.2 Fixed Format, Variable Length Tables

Many database applications today require variable length tables. In the BNF notation from Figure 4.34a, a row is composed of zero or more simple column values and zero or more complex column values. The length of each column value is defined in the DDL for the simple columns (e.g., SSN) and is stored (in some way) with the column's value for complex columns (e.g., EVALUATION SUMMARY).

As with the fixed format, fixed length rows, every column in the table is contained in every row and takes on either of two states: a variable/fixed number of NULLs or a variable/fixed length value. This capability is most useful for storing textual data and repeated instances of values. For example, if the table illustrated in Figure 4.34b represents the evaluations of EMPLOYEES, there should be room for comments of varying lengths. If variable length rows were not available, either the comments have to be truncated, or room for the largest possible comment has to be allocated. Neither is acceptable. The evaluation comment from Figure 4.34 is 47 characters long. The length indicator is the first part of the column's instance. Other rows could contain evaluation comments of any length.

Variable length columns, however, have drawbacks. The DBMS must read the DBMS rows, and must decode the actual format of the table, which almost always requires building a fixed-length row for use by compiler languages such as COBOL or FORTRAN.



**BNF**

```
<row>::= {<fixed format row>
          / <variable format row> }
```

```
<fixed format row>::= [<simple row>]
                      / [<complex row>]
```

```
<complex row> ::= [ <simple column >...]
                  [ <complex column >...]
```

**DDL**

SSN, TYPE IS INTEGER 9(9)

LAST - NAME, TYPE IS VARCHAR

FIRST - NAME, TYPE IS VARCHAR

SX, TYPE IS CHAR 1

BIRTHDATE, TYPE IS DATE , FORMAT IS YYYYMMDD

EVALDATE, TYPE IS DATE, FORMAT IS YYYYMMDD

EVALUATION SUMMARY, TYPE IS VARCHAR

**Figure 4.34a** Fixed Format and Variable Length Rows (Complex)

SOC. SEC. NUM.	LAST	FIRST	B - DATE	SEX	...
525902896	5SIBLO	3BOB	19350822	M	...

...	EVALDATE	EVALUATION SUMMARY
...	19871027	47 VERY TALENTED HOCKEY COACH, AND FINE INDIVIDUAL
...		

**Figure 4.34b.** Example of fixed format and fixed length rows.



#### 4.2.4.2.3 Variable Format Tables

Variable format rows permit storage of data in different formats. The BNF in Figure 4.35 gives an example of the content of a variable format row. It consists of a series of column value instances just like the fixed format row illustrated in Figure 4.33. In the fixed format row, every column is contained in each row either as a value or as a NULL. In the variable format row, the appearance of any column value is entirely optional. A variable format row's column value can appear in any order within the stored row and can occur any number of times. Unless the DBMS had clauses to enforce single values and uniqueness on its columns, a particular employee could end up with two or more Social Security Numbers of different values within the same row!

As illustrated in Figure 4.35, each column is assigned a specific code. Column FC is the code for FIRST NAME, and FG is for NICKNAME. Column values can be either fixed length or variable length. In the first line of the example, the SKILL column values appear twice. In the second line there is a fixed length column for SSN; two variable length columns, LAST NAME and FIRST NAME; a date type column; and a single-digit SEX column. In the third line, another SKILL column value appears, and then two NICKNAMES.

The variable format table has both advantages and drawbacks. The only real drawback is the amount of decoding the DBMS has to perform to translate the DBMS's data back into a *regular* form, which is required by all programming languages. The big advantage is reorganization. The addition of a column type to a row has no effect on the database until its column values are added. When added, the column value is merely appended to the end of the row.

When a column type is removed from the database, the DBMS would simply not map the discarded column value from the existing rows to any program requesting it. Furthermore, each time a row is updated, the discarded column value is dropped from the existing row when the row is returned to the database.



BNF

```

<variable format row>::= { {<column indicator><column>}
                               [{<column indicator><column> }...] }

<column>::= { <simple column> / <complex column. }

```

DDL	
Column Code	DDL Column's NAME AND TYPE
FA	SSN, SVI, NUM 9
FB	LAST NAME, SVI, VARCHAR
FC	FIRST NAME, SVI, VARCHAR
FD	BIRTH DATE, SVI, DATE
FE	SEX, SVI, CHAR 1
FF	SKILL, MVI, VARCHAR
FG	NICKNAME, MVI, VARCHAR

**Figure 4.35a** Variable Format Row

## RECORD INSTANCE

FF10PROGRAMMER	FF15SYSTEMS ANALYST
----------------	---------------------

FA07543182	FB9BRONSTEIN	FC4MANUEL	FD08/28/1963	FEM
------------	--------------	-----------	--------------	-----

FF07MANAGER	FG6MIMOSA	FG3MNU
-------------	-----------	--------

**Figure 4.35b.** Example of variable format row.

#### 4.2.4.3 Column Storage Formats

Figures 4.36a, 4.36b, and 4.37a and 4.37b illustrate the different column storage formats. These storage formats are used in DBMSs as indicated in Figure 4.27. Again, the more options a DBMS offers, the better its potential performance at the cost of increased complexity and training requirements.

In addition to the column instances within a row being either fixed or variable length, the columns may represent either a single column value instance or multiple column value instances. In the former case, the column is termed simple, and in the latter case the column is termed complex.

When a row is defined through a DDL to contain only simple columns, the row is known as a simple row. When a row is defined through the DDL to contain at least one complex column, the row is known as a complex row.

##### BNF

```
<simple column>::= <fixed length column value>
                  / <variable length column value>

<fixed length column value>::= [ <value> [<blank>...] / <null>...]

<variable length column>::= { [ <length><value> ] / <null>...}...
```

**Figure 4.36a.** Simple column formats.

SOCIAL - SECURITY - NUMBER, TYPE IS INTEGER 9 (9)  
FULLNAME, TYPE IS VARCHAR

SSN	FULLNAME
648711400	9BOB SIBLO

**Figure 4.36b** Example of Simple Column Formats



#### 4.2.4.3.1 Simple Column Formats

Simple columns represent the storage of single value instances of a column. The BNF from Figure 4.36 illustrates that a simple column can be either fixed or variable length. For fixed length columns, the space defined in the DDL for the column value contains either values, blanks, or NULLs. As the BNF illustrates, no subordinate column structures are allowed, and a column is not allowed to represent repeated values.

In the examples thus far presented, all the column types have been simple, for example, SSN, FULL NAME, and EMPLOYEE EVALUATION.

#### 4.2.4.3.2 Complex Column Formats

Some DBMSs permit complex rows. The BNF from Figure 4.37a illustrates complex columns, which are required for a complex row. There are three different types of complex columns: vectors, groups, and repeating groups. A vector is a column that would otherwise be simple except that it represents multiple value occurrences. An example of a vector is TELEPHONE NUMBER, where an organization or person is allowed to have several telephone numbers.

##### BNF

```

<complex column>::= <vector column>
                    / <group>
                    / <repeating group>

<vector column>::= { <qty> <simple column value>... }

<quantity>::= the number of occurrences for <simple column value>

<group column>::= { <simple column A>
                    [ <simple column B>... ]
                    [ vector column>... ]

<repeating group>::= { <complex column A>
                      [ <complex column B>... ] }

```

**Figure 4.37a** Complex Column Storage Formats: BNF



A group column is a column that is defined to have subordinate columns, each of which, in turn, may be a simple or vector column. A very common example of a group column is ADDRESS. The subordinate columns are DWELLING STREET NUMBER, STREET NAME, CITY, STATE, and ZIP. While this example only contains simple columns, some DBMSs allow a group to contain vector columns as well, for example DWELLING TELEPHONE NUMBERS.

A repeating group column is a column that is defined to have subordinate columns. A repeating group is different from a group column in that a group column can only have one set of values for its defined set of subordinate columns, while a repeating group can have multiple sets of values.

Figure 4.37b illustrates the difference between a group and a repeating group. Some DBMSs, for example FOCUS and SYSTEM 2000, allow repeating groups to be nested. The DDL in Figure 4.37b illustrates the situation in which an EMPLOYEE has DEPENDENTS, which in turn have the repeating group column HOBBIES and the repeating group column MEDICAL CLAIMS, as well as the two columns FIRST NAME and BIRTHDATE. The nested repeating group column HOBBIES is defined to contain two columns, HOBBY NAME and HOBBY ANNUAL COST.

In addition to having one or more nested repeating group columns and one or more simple columns, a repeating group may also contain vector columns.

The flexibility added to a table by allowing complex structures far outweighs the disadvantages. For example, if an EMPLOYEE needed to store multiple TELEPHONE NUMBERS, DEGREES, or the like, the only way a DBMS adhering to the relational data model can accommodate that requirement is to add an additional table, which would require defining, loading, and maintaining the additional table for just one column, DEGREES.

In the ILF or network data models, the complex columns might be stored as illustrated in Figure 4.37. The first example in this figure shows variable-length multiple-valued columns with an indicator of the number of values at the head of the array. The second example shows fixed length values with an indicator of the number of values at the head. The third example shows a repeating group. Of course, within each occurrence of the group there is either a fixed format or a variable format for the columns contained in the group.

The main reason for allowing storage of complex tables is to reflect the way data exists in real business organizations. To require subordinate repeating group instances and/or multiple valued column instances to be tables (tables) in their own right is to distort reality. A drawback, however, is that the DBMS will have to digest, decode, and then present these to the users. However, the effort pays off, given the main reasons for allowing storage of complex tables.





DDL

TELEPHONE, TYPE IS INTEGER 9(9), OCCURS UP TO 10 TIMES

ADDRESS TYPE IS GROUP

HOUSE NUMBER, TYPE IS CHARACTER 5

STREET - NAME, TYPE IS CHARACTER 25

CITY, TYPE IS CHARACTER 25

STATE, TYPE IS CHARACTER 2

ZIP, TYPE IS INTEGER 9 (9)

RECORD NAME IS EMPLOYEE

PRIMARY KEY IS EMPLOYEE - ID

ColumnS ARE

EMPLOYEE - ID, TYPE IN INTEGER 9 (9)

EMPLOYEE - FULL - NAME, TYPE IS CHARACTER 15

DEPENDENTS, TYPE IS REPEATING GROUP, OWNER IS EMPLOYEE

FIRST - NAME, TYPE IS CHARACTER 10

BIRTH - DATE, TYPE IS DATE, FORMAT IS YYYYMMDD

HOBBIES, TYPE IS REPEATING GROUP, OWNER IS DEPENDENTS

HOBBY - NAME, TYPE IS CHARACTER 15

HOBBY - ANNUAL - COST, TYPE IS MONEY, FORMAT IS 99,999.99

MEDICAL - CLAIMS, TYPE IS REPEATING GROUP, OWNER IS DEPENDENTS

Etc.

**Figure 4.37b.** Complex column storage formats: DDL



VECTOR:			
Telephone numbers , up to 10 occurrences			
3012229876	2019981284	.....	.....

GROUP:				
House number	Street	City	State	Zip
1234	Maple Street	Sadlebrook	NJ.	029879999

REPEATING GROUPS: DEPENDENTS	
First Name	Birthdate
Bobby	19780817

HOBBIES	
HOBBY NAME	HOBBY-ANNUAL-COST
Ice Hockey	2000.00
Model Boats	500.00
Soccer	250.00

**Figure 4.37c.** Examples of complex column storage formats.

#### 4.2.4.4 Data Summary

As can be seen from Figures 4.25, 4.26, and 4.27 and from the presentation of the different types of storage formats for files, tables, and columns, knowledge of the DBMS's data model is not complete knowledge of a database's internal structure and design.

A sophisticated DBMS can store rows from different tables in the same O/S file. If rows of several tables can be stored on the same O/S file, I/O efficiency will be better than if all instances from each table must be stored in one O/S file. This optimization is due to the fact that applications often require collections of related rows of different types, for example, a DEPARTMENT, its EMPLOYEEs, and the PROJECTs that are assigned to EMPLOYEEs. If these rows are contained on three different O/S files, then there must be I/Os to each of those files to obtain the necessary data for the interrogation. If these rows are all located in the same O/S file, and stored in the same or in nearby DBMS rows, then the interrogation will be much faster.



A DBMS should also have the ability to store different sets of rows from the same table on different O/S files. For example, if all the invoices sent to customers are stored on O/S files by month, that is, if there is one O/S file for January's rows, one for February's rows, and so on, then when a new year begins, the prior January's O/S file containing its rows could be moved off-line, clearing disk space for the new January's rows, without having first to unload the *old* rows. These off-line O/S files must be able to be brought back on-line through a RESTORE operation.

In addition to being able to store rows of different types in the same O/S file, a DBMS should be able to store these rows in the same DBMS row. This makes possible an even more sophisticated type of optimization. The specification of this capability is usually contained in the relationships section of the DBMS's data storage definition language (DSDL), whereby member rows are targeted to be stored *close* to their owner. Such clauses may affect the access strategy that is chosen for the member table. For example, if an owner row's location is determined by hash/calc, that is, randomly, and if a member's rows are all targeted to be stored within the DBMS row that contains the owner row, then the member's access strategy typically cannot be hash/calc.

To aid in the optimization of these complex file structures, the DBMS should be able to change the blocking factor of an O/S file. In some DBMSs, the size of the DBMS row instance can be either the same size as the O/S row that is actually read by the O/S's software routines, or it can be smaller so that, for example, five DBMS rows fit in one O/S row. In such a case, the blocking factor is five. The DBMS must be able to spread rows across DBMS row boundaries. If this is possible, then all the space contained in the O/S row will be used. If not, then either the O/S row size must be an exact multiple of the DBMS row size, or there will be unused space.

Sophisticated DBMSs should be able to store rows of varying length to accommodate either complex tables or variable length columns. How the DBMS handles variable length rows is important. For example, as rows shrink, the space should be made available for new or adjacent expanded rows. In the event the DBMS moves a row from one location to another, then a sophisticated DBMS will not have to reorganize indexes, relationship addressing schemes, and the like. If these have to change as well, the update operations will certainly take longer. An additional benefit of complex row storage schemes is fast reporting. When a report request is executed, a single DBMS instigated file I/O is likely to obtain most of the requested DBMS rows, and therefore most of the rows from one or more types that is needed for a single logical sequence of owner and members.

In addition to the different formats described above, some DBMSs allow free space in the DBMS row for expansion of individual rows. Once the space is used up, the DBMS usually divides the contents of the DBMS row in half, with one half of the rows and their associated key values staying in the original DBMS row, and a pointer to the new DBMS row storing the remainder of the rows and their keys.

DBMSs that allow the many different data formatting options described above permit great flexibility in database construction and maintenance.



### 4.2.5 Data Storage Definition Language

A sophisticated, high performance DBMS separates the definition of logical data structures (columns, rows, and relationships) from the definition of the mapping of the rows onto O/S files. The components of a data storage definition language (DSDL), which accomplishes the mapping of logical definitions to physical structures, must include clauses to define at least these six components:

- Files
- Areas
- Tables & instance allocations
- Column physical characteristics
- NDL sets (relationship definitions)
- Indexes

There is generally a sequence to the definition of these DSDL components.

- Files are defined, including declaring the blocking factor for O/S rows.
- Areas are defined, including the allocation of areas to files.
- Tables are then defined and their instances are allocated to areas.
- Column physical characteristics are defined, that is, fixed or variable length, etc.
- Indexes are defined. The type of index is specified and, if appropriate, so is the allocation of index structures to specific areas (and in turn, files).
- If ANSI/NDL, then sets are defined, specifying principally whether the member rows are to be collocated with the owner instance of the set in a specific DBMS row.

#### 4.2.5.1 File Clauses

A file is a commonly understood operating system component. DBMS rows are added to, modified in, and deleted from files. File specifications typically include the type of access method used by the operating system to access the file's rows, and the name, type, and location of the actual physical device (disk drive spindle) on which the file is to reside. For an IBM VSAM file, the various access methods that could be defined are key sequence data sets, entry sequence data sets, or relative row data sets. A most important subclause in the file's specification is the O/S record clause. It typically specifies the file's size and other information, such as the blocking factor. The blocking factor defines the number of DBMS rows present in each O/S record.



Some DBMSs clauses indicate what happens when a DBMS row becomes full of component instances from indexes, relationships, or tables. A common practice is that the DBMS retains half the contents in the current DBMS row and then creates new DBMS row.

#### 4.2.5.2 Area Clauses

An area is a DBMS *creation* used to identify a specific collection of rows from one or more tables. Because the operating system, rather than the DBMS, actually places rows on the disk, an area is only a logical specification of the space, and not a physical specification of space. The area clause would thus contain specifications for its initial size and allowable expansion sizes, and the mapping between it and O/S files.

The size subclause of an area is typically in the form of how many DBMS rows can be stored in the area, e.g., 100,000.

The mapping subclause typically specifies the relationship between the AREA and one or more O/S files. Some DBMSs only allow an AREA to be contained in one file; other DBMSs allow AREAs to be contained in multiple O/S files; and still other DBMSs allow multiple AREAs to be contained in one or more O/S files. If more than one AREA is assigned to a file, then the portion of the file that the AREA is to occupy is also identified.

#### 4.2.5.3 Rows

Row clauses specify the type of access strategy to be used for row storage and selection, and also the mapping of the instances of tables and areas.

Storage specification subclauses typically indicate whether the rows are to be randomly scattered over the entire space identified for row storage, sequenced by entry or by one or more columns, or whether the rows are to be placed in the same DBMS row as their owner.

In some DBMSs, all the instances of a row must belong to only one area. In such a case, the area mapping clause only requires the identification of the area name. In DBMSs that allow the instances from one table to belong to multiple areas, the mapping clause must include both the area name and the identification of the DBMS row instance range for the rows.

#### 4.2.5.4 Columns Clauses

If a DBMS has only a general specification of the type of data represented, for example, character or numeric, then the DSDL must complete the specification. In the case of CHARACTER data, it specifies whether the length is fixed or variable. If variable, then the specification states whether the column has a maximum length or an indeterminate length. For numeric columns, the specification indicates whether the column is integer or decimal. If integer, then the length is specified. If decimal, then the length and number of decimal places is specified.



#### 4.2.5.5 NDL Set Clauses

NDL set specifications specify both the owner and the member table(s) that participate in the relationship and also the access strategy for the member rows of the set. Set organizations are typically random, chained, or sorted. For a random set, the DBMS uses the values of one or more columns to calculate the location of the row within the space identified for the storage of rows belonging to the set. For chained set organizations, rows are stored and accessed in loaded order. In sorted sets, the rows are ordered according to the value(s) of the column(s) defined as the set's sort key(s).

When there are multiple tables as members, then the set clause indicates whether the rows are to be accessed by table, or by the order of a commonly existing sort key (one or more defined columns) across all the tables. An organization might have, for example, four tables: FULL-TIME-EMPLOYEES, PART-TIME-EMPLOYEES, RETIRED-EMPLOYEES, and FORMER-EMPLOYEES. To create a list of all employees, regardless of type in an NDL (or CODASYL) DBMS requires the definition of a set that has COMPANY as the owner and all four table as members. The set additionally indicates that the members are sorted by EMPLOYEE-NAME. The set could additionally indicate that in the case of a duplicate name, the FULL-TIME employees appear before the PART-TIME, who appear before the RETIRED, who appear before the FORMER.

In a non-NDL or non-CODASYL system, this effect could certainly be achieved by having all employees in one table with an employment indicator column which states that *A* means full-time, *B* means part-time, *C* means retired, and *D* means former. A retrieval sort clause could then be generated that retrieves all employees and then sorts them by name and then by employment indicator. While such a scheme is possible, it requires that all employees be on-line all the time. Since many companies have a significant employee turn-over every 36 to 48 months, a company of 7000 current employees might have a file of about 21,000 employees. That is a waste of space if the combined sorted list is needed only quarterly. In the case of an NDL or CODASYL DBMS, the three other tables could be placed in a separate area and then moved off-line until needed, making for a much more cost-effective solution. Additionally, all programs accessing only the full-time employees would not then have to include a select condition that stated . . . AND NOT (PART-TIME OR RETIRED OR FORMER). The reader merely has to review the cost and complexity of the indexing section to know the cost savings of NOT having to use indexes when they are otherwise unnecessary.

#### 4.2.5.6 Index Clauses

Index specifications indicate the type of index organization that is operative for a particular table's columns. Since every index type consists of a unique value component and a multiple occurrence component, the selection is from among alternatives within each component.

As to the unique value component, the two common alternatives in the selection clauses are either hash/calc or hierarchical. Hash/Calc is more appropriate for primary or candidate keys, and hierarchical is more appropriate for secondary keys in which range searching is a common requirement.



As to the multiple occurrence component, the alternatives are listed in Figure 4.10. Most DBMSs do not offer a choice. The design of the multiple occurrence component that exists, however, typically indicates whether indexes are an integral component of a DBMS's design or an afterthought. From Figure 4.10, the secondary index types that are the four fastest are generally an indication of the fact that indexes are an integral component of the DBMS's design.

#### **4.2.5.7 DSDL Summary**

There are six types of clauses in any data storage definition language: files, areas, tables and instance allocations, column physical characteristics, NDL sets (relationship definitions), and indexes. Of the six types of clauses within a physical database's specification, the only one that depends on the data model is the one specifying NDL sets. All the other types of clauses are independent of data model. A sophisticated DBMS preserves this separation of physical from logical. A DBMS that does not maintain this separation will probably be too simplistic to be employed on high volume, production applications--regardless of data model.

A very sophisticated DBMS offers the maximum number of alternatives in each of these clause types. Because of the great variety, it is certainly impossible for the novice user to instantly understand them all and put them to use effectively. To overcome this, most DBMS vendors allow these clauses to be defaulted to some practical or common use. Then, as the application and/or database grows in size and complexity, the purpose of these clause alternatives becomes obvious. At that time their value will be appreciated. If all these alternatives are not available, frustrated users and poorly performing applications are the result.

#### **4.2.6 Storage Structure Summary**

As stated above, the minimum required capabilities for the dictionary storage structure component are the same for both static and dynamic relationship DBMSs. A good data dictionary must have fundamental information such as table and column names, expository definitions, and specifications for their use in various programs that have been properly compiled according to DBMS conventions. Beyond the minimum set of information, the capabilities differ widely. To obtain a DBMS that only supports the minimum capabilities is not a good idea, as a fully functional IRDS can be a tremendous assist in the development of fully specified database applications.

The second component, indexes, serves a number of different uses. First, it permits fast access to a set of rows. Second, it also can be a way to avoid writing complicated program logic that narrows a set of rows from the set produced through a WHERE clause that only had one index to the set of rows that would result when the WHERE clause had multiple conditions, each involving different indexed columns. And, in the situation described through the case study, sophisticated indexes enable whole applications to exist that otherwise are impractical.

All databases have relationships of two types: among rows of the same type and among rows from different types. If the relationship mechanisms of the DBMS are dynamic, the DBMS will always incur the extra processing overhead associated with primary and secondary key



access as that is the method of dynamic relationship DBMS relationship processing. If the relationships, however, are static, then the DBMS will always know exactly where the rows are and will avoid this extra processing.

While relationships are important in their own right, separate relationship pointers are critically important for the cost-effective solution of multiple table, index-selected, list processing queries. Indexes are able to point to these separated relationships rather than to the rows. Multiple table queries can then be processed up to the time of determining the final set of rows that comprise the answer set without having to access rows.

There are many different types of data storage structures, but the DBMS's data model alone does not determine whether the DBMS is suited to ad hoc questions, high volume performance applications, both, or neither. A DBMS that only allows a single table per O/S file is certainly the least sophisticated. DBMSs that allow many data formatting options provide for sophisticated database construction and maintenance.

Selecting a DBMS on the basis of its storage structure is probably more important than selecting a DBMS on the basis of its data model. Any DBMS vendor who asserts that selecting a data model is more important than selecting the appropriate storage structure is either presuming extreme naivety on the part of the evaluators, or has never built a large, high volume, production-oriented database application.

### 4.3 Access Strategy

A DBMS' access strategy is the method by which the DBMS accesses rows in the database. In some DBMSs, the access strategy can be controlled by user language verbs, for example, GET-MEMBER or GET-OWNER. In other DBMSs, the access strategy is controlled by the DBMS, for example, FIND SALESPERSON WHERE REGION EQ *EAST*. As might be expected, static and dynamic relationship DBMSs access rows in very different ways.

#### 4.3.1 Static Access Strategy

The access strategy of a static relationship DBMS employs data manipulation language (DML) verbs to navigate among the rows of the data definition language related tables. The programmer must specify the sequence for examining the tables and then follow the inter-row relationship instances that have been created during the data loading or update process.

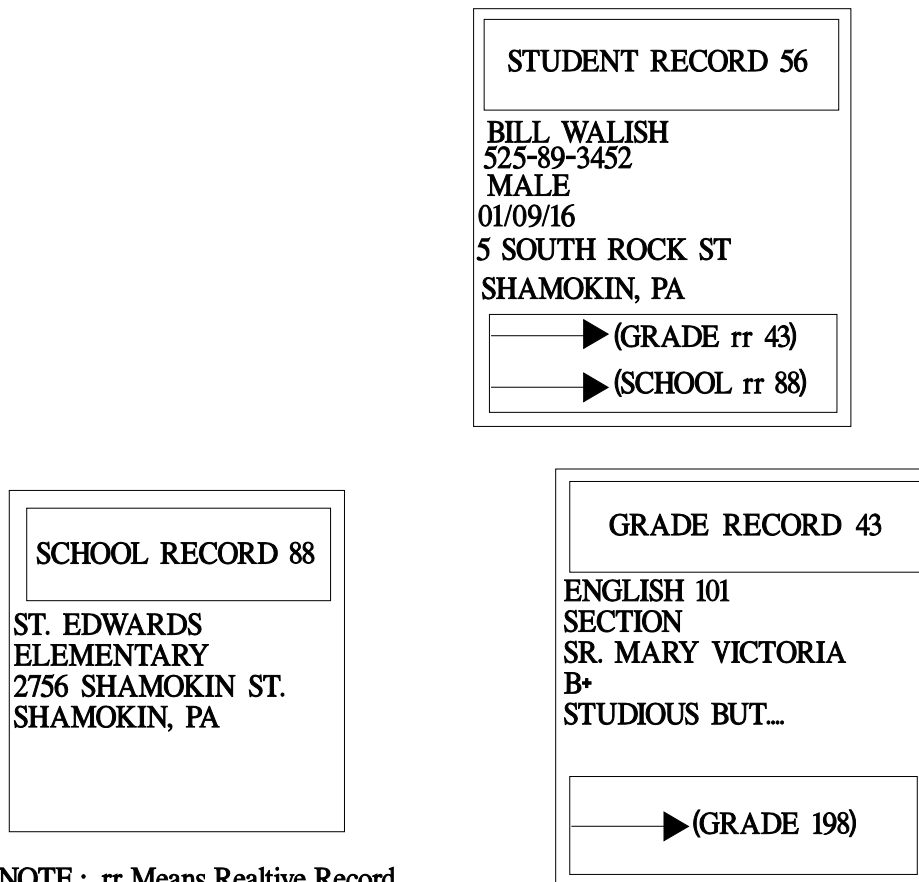
The user has a lot of power because most of the database processing is through a host language. Initial access is usually through a primary key; then, through a series of GET MEMBER, GET NEXT, and GET OWNER commands, the database is traversed until all the data is accumulated for the report, or until all the updates are completed.

In Figure 4.38, for example, a user FETCHes (FIND + GET) the STUDENT row on the basis of the SSN = 525-89-3452, and then utilizes a GET NEXT command to get the first row on the SCHOOL set. Since the DBMS has stored the relative row address of the first school in the student's row, the DBMS knows exactly where to find the school row. A command often used is:

FIND <table> <selection-expression>







NOTE : rr Means Realtime Record

**Figure 4.38** DBMS Based Pointers, Embedded Static

After execution of the <selection expression> a subset of the row identifiers is returned to the run-unit for processing. For example,

FIND STUDENT WHERE COURSE EQ ENGLISH 101.

returns the COURSE row identifiers that are ENGLISH 101. A command sequence to get the student for the course is:

- The COURSE row is brought into memory by the GET COURSE command.
- Then, the GET OWNER COURSE command is issued.
- Then the specific STUDENT row is brought into memory.

As can be seen, the programmer navigates the DBMS's traversal of the database.

The programmer must initially determine the table navigation sequence for each application from a diagram of the database's structure. Figure 3.9 is a database structure diagram. The navigation sequence requirements must be explained by the programmer to the database



administrator so that a judgment can be made on the appropriateness of the access strategy for other user interrogation and update requirements.

Using the database structure contained in Figure 3.9, the access strategy might be as follows: all tables are directly accessed except for ORDER, PRODUCT-SPECIFICATION, PRODUCT-PRICE, and ORDER-ITEM. Direct access means that the row is retrieved directly on the basis of its key. Generally, no rows are scanned to find the target row; it is found directly.

The PRODUCT-SPECIFICATION, PRODUCT-PRICE, and ORDER-ITEM rows are stored VIA. The term VIA means that the rows are stored within the same DBMS row as their owner row or as near to that DBMS row as is possible. If a table is VIA to one table, it typically cannot be VIA to a different table. That means, for example, that the ORDER-ITEM table cannot be VIA within the relationship that goes from the PRODUCT-PRICE table to ORDER-ITEM table. If there is a relationship between two or more direct access tables, then the owner row contains the address of the member row. That address, of course, is a direct address, and most often is a DBMS row address value different from the owner's DBMS row address value.

Hierarchical DBMSs access strategies are much more straightforward. In Figure 3.8, the owners and the members are already set out. Generally, in a hierarchical DBMS, a specific row can be accessed by a primary key such as CUSTOMER-ID. A selection of rows can be found through a secondary key like STATE for a CUSTOMER. This access strategy is used by both IMS and SYSTEM 2000. Beyond this general similarity, these DBMSs differ significantly. For example, in IMS, the pointers among rows are all embedded as in Figure 4.16. In SYSTEM 2000, the pointers are all separated from the rows as illustrated in Figure 4.18. Because of this difference, SYSTEM 2000 supports a list processing access strategy, which is essential to efficient ad hoc query processing. IMS, because it is a row processing DBMS, cannot perform efficient ad hoc queries.

There is a growing need in database environments for on-line access for both ad hoc questions and standard reports. Since a static database is primarily designed to answer the needs of interrogations that mirror the database's organization, many ad hoc questions can cause extensive processing. This is unfortunate as database is supposed to bring diverse data together and make it available to many users with different needs.

The reason this problem arises is because some static relationship DBMSs require rows to be searched whenever:

- The search criteria involve non-primary key columns, and/or
- The search criteria involve more than one indexed column, and/or
- The search criteria involve columns from different tables.

For example, in the query, whose table structure is represented in Figure 4.39,

FIND ALL EMPLOYEES WHERE

CURRENT-JOB-TITLE EQ SYSTEMS ANALYST, AND



SALARY GT \$30,000, AND

FORMER-JOB-TITLE EQ PROGRAMMER,

there is no primary key expression, and the FORMER-JOB-TITLE assignment is within a table that is a descendent of EMPLOYEE.

Three components are required to make such interrogations practical in a static environment:

- Indexes
- List processing
- Efficient relationship normalization

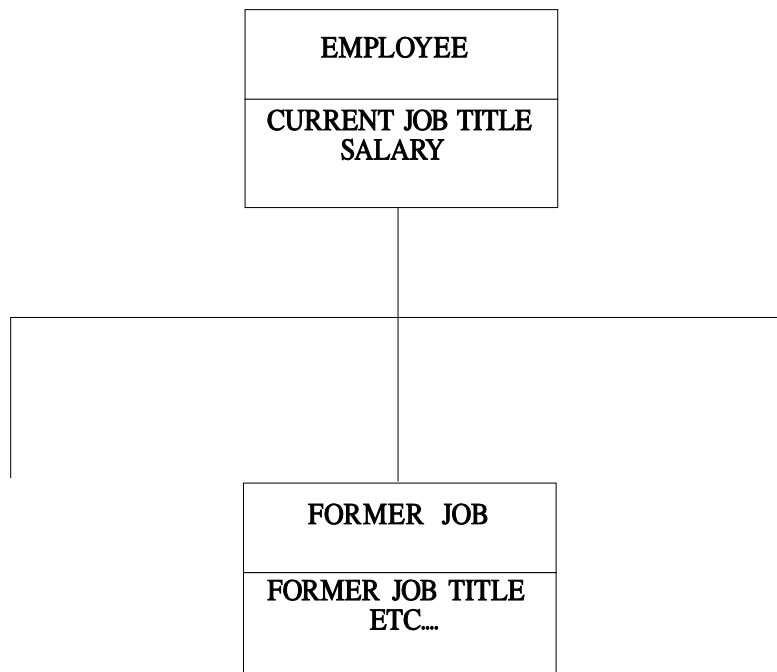
These are detailed in earlier sections.

The first component, indexes, speeds access to the rows on the basis of column values that are not primary keys. In the query described above, CURRENT-JOB-TITLE, SALARY, AND FORMER-JOB-TITLE have to be indexed to avoid row searching.

The second component, list processing, indicates that the DBMS builds row address arrays (lists), one for each indexed condition in the selection clause. It also indicates that the DBMS processes the lists together through AND and OR operations to determine the final set of row addresses that meet the conditions set out in the query. Each of the conditions that come from the same table (SALARY and CURRENT-JOB-TITLE) can have its lists ANDed. Conditions that come from different tables cannot be ANDed together without first having their row addresses (relationships) normalized.

The third component, relationship normalization, requires that all the rows pointed to by the addresses in the lists that are the consequence of processed selection criteria must point to the same table. In the case of the data structure represented in Figure 4.39, two of the select conditions (CURRENT-JOB-TITLE EQ SYSTEMS ANALYST, AND SALARY GT \$30,000) result in lists of row identifiers from the same table, while the third condition, FORMER-JOB-TITLE EQ PROGRAMMER, produces a list of row identifiers of a table from a descendent level. To resolve this disparate level situation, all the addresses of the rows must be brought to a common level. To accomplish that, addresses of rows resulting from the query condition FORMER-JOB-TITLE EQ PROGRAMMER need to be *raised* to the level of the addresses from the EMPLOYEE table. *Raising* is accomplished by traversing the owner relationships contained in the JOB rows to the EMPLOYEE rows. When this is done, there will again be two lists: one for the EMPLOYEES whose FORMER-JOB-TITLE EQ PROGRAMMER, and another for the EMPLOYEES whose CURRENT-JOB-TITLE EQ SYSTEMS ANALYST. Because these two lists are both on the EMPLOYEE table level, they can be ANDed to find the intersection list of row identifiers. Normalization can also be achieved by *lowering*. This is accomplished by traversing the *member* relationships from the *owner* to the *members*.





**Figure 4.39** Relationships Case Study Record Type Diagram

Row address normalization, while achievable when the relationships are embedded in the rows, is only practical when the relationships (see Figure 4.18) have been removed from the actual rows and placed in a separate storage structure component. If a row is about 1500 characters long, and a relationship array is only 15 characters long, then the relationship normalization part of a query has been made 100 times more efficient. That is because under the former case there is one relationship instance every 1500 characters in contrast to the later case where there are 100 relationship instances for each 1500 characters. That is 100 times more dense.

When all three of these facilities (or their functional equivalents) exist in a static relationship DBMS, the static relationship DBMS can handle ad hoc interrogations. SYSTEM 2000 has all three implemented.

Notwithstanding the inclusion of these facilities, a static database is still a static database. That means that regardless of any improvement in the access strategy for servicing ad hoc inquiries, the ad hoc inquiry's data requirements must closely conform to the database's structure. For example, suppose there was a query:

PRINT STUDENT NAMES

WHERE STUDENT-BIRTH-STATE-CODE EQ TEACHERS-DEGREE-CODE.



While such a query borders on the ridiculous, it serves to illustrate the point that queries may arise that have not been designed into the database. If the relationships defined in the database's DDL did not explicitly support this specific query, then no enhanced access processing is going to eliminate the extensive programming and row processing required to answer the interrogation. The only way to efficiently handle such a query is to state the relationship in the query, and then have the DBMS discover whether there were any students who were born in MA (Massachusetts) and who had teachers with an MA (Master of Arts) degree. Such a capability properly exists only in dynamic relationship DBMSs

#### 4.3.2 Dynamic Access Strategy

In contrast to following the static relationships defined in the DDL that are actually created during the data loading or update process, the dynamic relationship DBMS automatically performs most of the navigation and sorting whenever interrogations are made by users. The user constructs an interrogation against a view of data that has the following three components:

```
SELECT <column-list-1>

SORTED BY <column-list-2>

WHERE <conditions list>
```

The <column-list-1> may include columns from one or more tables. The sort clause (<column-list-2>) may include multiple sort keys, in ascending and descending sequences. The WHERE expression (<conditions list>) may include columns from different tables, with conditions involving Boolean operators, relational operators, ranges, and so on.

If the conditions list involves searching different tables, the syntax must also contain sufficient constructs so the DBMS *knows* how to process the relationship among the different tables. The syntax for processing multiple table relationships is either specified from within the conditions list of the interrogation, as in the case of the SQL WHERE clause, or through separately issued relationship binding syntax that establishes the relationship prior to the actual conditions list execution. In the ANSI/SQL language, the following is the essential syntax for selecting the course identifiers and names for full time employees:

```
SELECT COURSES-ID, COURSE-NAME FROM EMPLOYEE, COURSE
WHERE EMPLOYEE.EMPLOYEE-ID EQ COURSE.EMPLOYEE-ID
AND EMPLOYEE-CLASS IS FULL-TIME
```

The following is the syntax required by the DBMS FOCUS that is sufficient for it to know how to process the relationship:



LINK EMPLOYEE VIA EMPLOYEE-ID TO COURSE VIA EMPLOYEE-ID  
PRINT COURSE-ID, COURSE-NAME WHERE EMPLOYEE-CLASS IS FULL-TIME

The actual mechanisms that determine how the DBMS performs multiple table processing vary widely. Some DBMSs determine the lowest node in the selection network and build a completely normalized table at that level, with the resultant rows passing through the various selection criteria to determine which rows fit.

If the query additionally contains one or more SORT clauses, then these selected rows are examined to see if all the data is identified to carry out the sort operation. If not, additional column values are collected so that the rows can be sorted. Once sorted, report formatting is started.

The view under which the interrogation was issued is then examined for any default report formats that may have been specified. If there were none, the interrogation is responsible for supplying all the necessary information. Finally, with all the selecting, sorting, and formatting tasks accomplished, the report is produced. The steps just described all take place automatically without user involvement subsequent to submitting the interrogation.

### 4.3.3 Comparing Access Strategies

The main difference between static and dynamic relationship DBMS access strategies is that in static access strategies, the user has complete navigational control across predefined structures, while in the dynamic access strategy, the user can specify only relationships across completely unconnected tables that have columns sharing common values. In the static relationship DBMS environment, most of the structure definition decisions are made by the database designer on behalf of all users. The languages provide commands for the step-by-step navigation of the database's predefined structures. In the dynamic environment the language contains relationship specification commands in lieu of navigation commands.

For example, in Figure 3.9, if a user wants to know if any salespersons are born in the same town that manufactures a product that the salesperson sells, the static relationship DBMS assists by providing the various links from the SALESPERSON instance to the related PRODUCT instance. The program first has to retrieve a SALESPERSON instance and then traverse the structure by passing through the rows from CONTRACT to ORDER to ORDER-ITEM to PRODUCT-SPECIFICATION and finally to PRODUCT to determine whether the product is manufactured in the same town in which the salesperson was born. If the search is successful, columns from the product and salesperson rows are placed on a list in preparation for the report. Then the next ORDER-ITEM is checked. When the ORDER-ITEMs are exhausted for an ORDER, the next ORDER is checked, which in turn causes the ORDER-ITEM check to begin again. When the ORDERS are exhausted, the next CONTRACT is retrieved and the ORDER check cycle is begun. When all CONTRACTs are exhausted, the next SALESPERSON is retrieved, and the CONTRACT check cycle begins again. Figure 4.22 illustrates an abbreviated and rough pseudo-code program that accomplishes the task.

In contrast, in the dynamic access strategy, the expression of relationships that create inter-table structures during the query's execution is the responsibility of the user. Once these



relationships are expressed by the user, the DBMS translates the relationship expressions into search strategies to accomplish navigation, row selection, and automatic sorting, and presents the user with the results. For example, using the dynamically organized database diagramed in Figure 3.28, the problem described above might be recorded in a pseudo-code like that in Figure 4.23.

It is important to note that even though the number of statements is roughly the same, the process and the mechanism of the search is quite different. In the static example, every row between SALESPERSON and PRODUCT is scanned in search of a match. In the dynamic example, LIST-1 is derived directly, without retrieving rows in PRODUCT-SPECIFICATIONS. LIST-2 and LIST-3 are the result of pre-established connections. The match between LIST-1 and LIST-3 immediately produces the set of matches at one time, rather than iteratively as in the static example.

In the dynamic interrogation example, the generation of interim work files is unique. Once these files are created, some DBMSs allow them to be used completely outside the environment of the database. The DBMS is then free to service other requests such as updates.

#### 4.3.4 Buffer Management

A very critical component of any access strategy, static or dynamic, is buffer management. A buffer is a set of memory-based DBMS rows or high speed disk-based DBMS rows that temporarily hold DBMS rows from the various databases, including sort files, derived AND/OR lists, and DBMS rows from the various storage structure components such as the dictionary, indexes, relationships and data.

Probably the most frequently used storage structure component is the dictionary. It is also usually the smallest, often comprising less than ½ million characters. This storage structure component is not changed often as changes are really logical database reorganization. Consequently, the entire dictionary is often brought into memory when a database is attached and there it stays. The next components that often stay resident in memory are all the edit and validation rules, referential integrity rules and table look-ups.

A *smart* buffer manager *learns* which DBMS rows from the other storage structure components (indexes, relationships, and data) are most often accessed and keeps them in memory in preference to other DBMS rows that are used less often and have the oldest last-accessed time.

An often used capability of the DBMS is the delayed-write feature. With it, the effects of an run-unit update transaction are either written immediately to disk or wait until the DBMS row is regularly written to disk. Waiting usually means waiting until a buffer becomes full of updates, or until the DBMS row has aged as required by the buffer management algorithms.

Buffers can often be different sizes, both in terms of the quantity of DBMS row instances the buffer can hold and in the size of each DBMS row. Frequently, the size of the DBMS row for each of the storage structure components can be determined at DBMS installation time. Sophisticated DBMSs permit DBMS rows from different databases and from different storage structure components to common or specific buffers.



Buffers are a major source of DBMS performance tuning. To effectively take advantage of buffers requires knowledge of their optimum quantities, sizes, and allocations. To acquire this knowledge a great deal must be known about the running database environment. That means that the database administrator must be acutely aware of the various applications. The application characteristics include assessing the update cycles, frequencies, volumes, on- and off-line tendencies, whether it is faster to rerun jobs or to perform database rollbacks and the like. A great deal of this knowledge comes from the statistics generated from each transaction. These statistics can be stored on the journal files, and can undergo analysis to properly determine the appropriate configurations for the buffer sizes, quantities and allocations. Because each write to the journal slows database operations, installations often have these statistics journals recorded only during the first weeks of a new application and only for several days a month to monitor any usage changes.

#### **4.3.5 Access Strategy Summary**

The main difference between the static and dynamic relationship DBMS access strategies is that the static relationship DBMS imposes predefined relationships that are navigated by DML commands. To navigate outside these relationships either is not possible, or requires extensive processing. In short, in the static relationship DBMS environment, most of the structure decisions are already made on behalf of the user. The languages provide commands for step-by-step navigation of the predefined structure.

In the dynamic access strategy, the expression of the relationship is the responsibility of the user. Once it is expressed, the DBMS takes over and performs the navigation, row selection, and sorting on behalf of the user.

#### **4.4 Data Loading**

Data loading is the process of inserting large quantities of data into the database at one time. DBMSs usually have special utilities to accomplish this task.

Some DBMSs allow data to be loaded in special ways to achieve efficiencies during subsequent updating and reporting. This is called load engineering.

##### **4.4.1 Static Data Loading**

Static relationship DBMSs require that the database be loaded according to the dictates of the structure. The owner is loaded first, then its members, and then their members. All the other owners to which a member is to be connected are then loaded. An already-connected member is then accessed and connected into relationships with its other owners. An alternative loading strategy is to load all the owners, then load a member and connect it to all its owners.

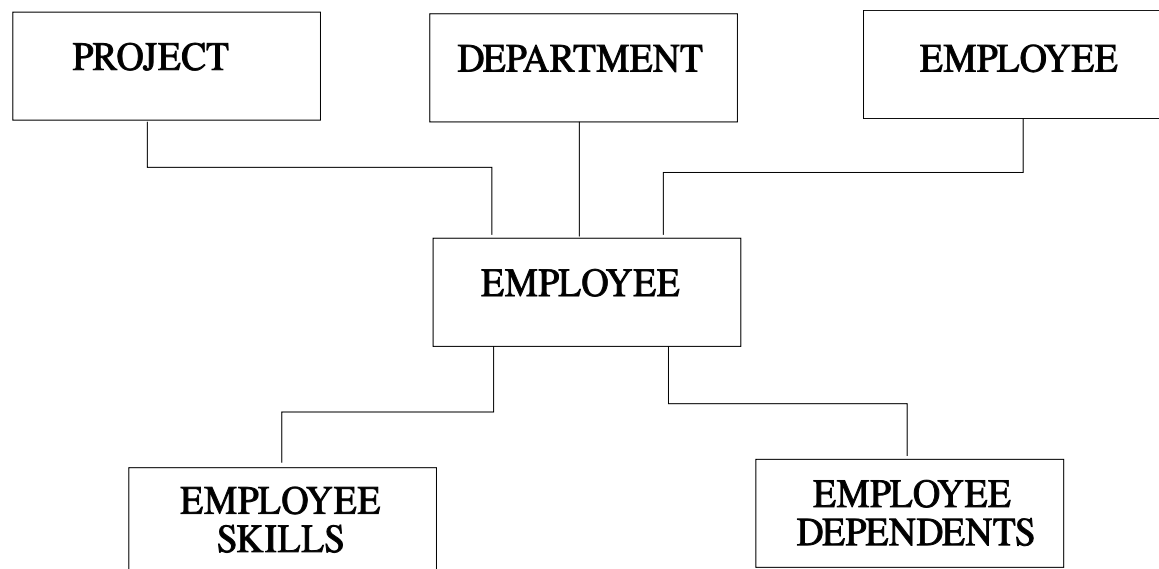
The process of loading a network database can be complex. The complexity is directly related to the number of relationships (sets) between and among tables. Once loaded, however,





the static database is ready for high-velocity reporting and updating, and the complexity and effort required for loading pay off. Row connection to sets can be controlled by an **INSERTION** option on the DDL relationship (**SET**) syntax.

An example of a data loading strategy is presented in Figure 4.40a through 4.40c. In this example, there are six tables: **PROJECT**, **DEPARTMENT**, **EMPLOYEE CLASS**, **EMPLOYEE**, **EMPLOYEE SKILLS**, **EMPLOYEE DEPENDENTS**. The loading strategy presented in Figure 4.40b is based on having their **INSERTION** options defined as **MANUAL**. That means that connecting a member to an owner must be done explicitly. The first two commands are to store rows **<project-1>** and **<employee-1>**. The third command is to connect the **<employee-1>** row into the **<project-employee>** set. The **<department-1>** row is stored, and the **<employee-1>** row is connected to the **<department-employee>** set. The **<employee-class-1>** row is stored, and the **<employee-1>** row is connected to the **<employee-class-employee>** set. The hierarchical process is started again by storing the **<employee-skills-1>** row and connecting the **<employee-skills-1>** row into the **<employee-employee-skills>** set. Finally, the **<employee-dependent-1>** row is stored and connected to the **<employee-employee-1>** set.



**Figure 4.40a.** Tables for static relationship DBMS loading strategy.



```
STORE <project - 1> STORE <employee>
CONNECT <employee - 1> TO <project - employee> SET
STORE <department - 1>
CONNECT <employee - 1> TO <department - employee> SET
STORE <employee - 1>
CONNECT <employee - 1 TO <employee - class - employee> Set
STORE <employee - skill -1>
CONNECT <employee - skills -1> TO <employee - employee - skills> SET
STORE <employee - dependents - 1>
CONNECT <employee - dependents - 1> TO <employee - employee - dependents> SET
```

**Figure 4.40b.** Manual strategy for static relationship DBMS loading.

```
STORE <project - 1> STORE <department - 1> STORE <employee - class - 1>
STORE <employee - 1>
STORE <employee - skills - 1>
STORE <employee - dependents - 1>
```

**Figure 4.40c** Automatic Strategy for Static Relationship DBMS Loading

When the INSERTION option is AUTOMATIC, then all the CONNECT commands are automatically performed by the DBMS. As can be seen in Figure 4.40c, only the STORE commands need to be issued. In Figure 4.40c, the actions are to automatically connect <employee-1> into the <project-employee>, <department-employee>, and <employee-class-employee> sets. Then, after <employee-skills-1> is stored, it is automatically connected into the <employee-employee-skills> set. Finally, after the <employee-dependents-1> row is stored, it is automatically connected into the <employee-employee-dependents> set.

Consider Figure 3.9 for another example of data loading. There are three natural trees:

- COMPANY, REGION, CUSTOMERS, and SALESPERSON
- PRODUCT, PRODUCT-SPECIFICATIONS, and PRODUCT-PRICES
- CONTRACTS, ORDERS, and ORDER ITEM

This example assumes that rows from all tables are available and have been properly loaded into O/S files that are accessible to the data loading program, and that all rows are automatically inserted into their proper relationships when sufficient information is available.

The row loading sequence for the first natural hierarchy is the first COMPANY instance, the first REGION instance, then one or more CUSTOMER instances, followed by one or more SALESPERSON instances. The second tree loads below COMPANY (assumes multiple companies and different products for each) a PRODUCT instance, one or more PRODUCT-SPECIFICATION instances, and one or more PRODUCT PRICE instances. The third tree



locates a CUSTOMER instance and then below the located CUSTOMER, loads one CONTRACT instance, one ORDER instance, and finally one or more ORDER-ITEM instances.

So far, the rows have been loaded in a hierarchical fashion. To create the network connections, for example for CONTRACT and SALESPERSON, there must be a way to determine which salesperson is connected to which contract. If the SALESPERSON-ID is also present in the CONTRACT table, the specific SALESPERSON is identified and retrieved. Upon retrieval, the mechanism that binds the SALESPERSON instance and CONTRACT instances together is also made available for processing. Assuming that the contract is to be stored either as the NEXT, FIRST, or LAST instance within the set, this same process is performed to connect ORDER-ITEMs to PRODUCT-SPECIFICATIONS and PRODUCT-PRICES.

Hierarchical data loading is the same as network loading except that there is no step for network connections, as networks are not supported.

#### **4.4.2 Dynamic Data Loading**

Loading a dynamic relationship DBMS database is very quick and simple. The rows from each type are gathered together and then loaded into the appropriate tables. If the rows are supposed to be related to other rows in a different table, then the data loading program must check--on its own--to determine that matching values exist. If the DBMS allows for the expression of referential integrity clauses, then the DBMS will access the referenced rows to validate the column value and store the row if a match occurs. If there is no match, the load operation is rejected, or else the foreign key column value is set to null.

#### **4.4.3 Load Engineering**

Loading data in special ways to achieve high performance on certain interrogations is called load engineering. Traditionally, load engineering has been a characteristic of static relationship DBMSs. However, due to the efforts by dynamic relationship DBMS vendors to perform at the rate of their static relationship DBMS counterparts, load engineering techniques have been undertaken by all DBMS vendors.

The benefits of load engineering are illustrated by the following example. An organization desires to rapidly retrieve all purchase orders and receive invoices associated with contracts. If over the years the organization is able to determine that there is an average of 15 purchase orders and 2 invoices against each purchase order, then the highest level of retrieval performance is achieved if the disk space for all the rows of these three tables is preallocated (reserved) within the same DBMS row rather than having the space allocated for them as they occurred--that is, scattered over the entire disk space. To achieve this, the total space for the contract, its 15 purchase orders, and the 30 invoices is computed. The load programs create all the necessary blank space for the 15 purchase orders and 30 invoices whenever a new contract is entered in the system. Thereafter, these preloaded blank rows--already stored--are retrieved and updated. The I/Os are kept to a minimum since all the rows are stored in the same DBMS row.



Without these and other physical performance tuning techniques, DBMS, static or dynamic, cannot be used effectively in high performance database applications. None of the four data models have a special claim on any physical performance optimization techniques. Rather than being a characteristic of one particular data model, these techniques are a characteristic of a sophisticated DBMS. Any DBMS not offering these techniques should not be procured whenever alternatives are available.

## 4.5 Data Update

Every database requires updating, which in the context of this book means adding, deleting, and modifying the data within a database that already contains a substantial amount of data. An understanding of how storage structure components react to certain types of updates (adds, deletes, or modifies) is essential to the effective selection, design, and implementation of applications for a particular DBMS.

Critical to application performance is how well the DBMS handles additions of new rows, deletions of existing rows, modification of relationship occurrences (not types: that is reorganization), and modification of column values. It is important to know how the DBMS access strategy accomplishes each update type, so that the database application can be designed in the most efficient manner. It is also important to note the efficiency with which auxiliary structures, such as indexes and relationships, are changed as data values are changed.

### 4.5.1 Table Changes

Figure 4.41 lists the types of effects on rows when ADDs or DELETes are made to static and dynamic databases. In either environment row additions are accomplished either according to a specific plan, or arbitrarily. If a plan is established through the DSDL, then as new rows become available they are stored according to the plan.

Context	Operation	
	Static	Dynamic
Add	Store row, then adjust the next and prior pointers. Adjust all indexes	Check the referential integrity clauses, then store. Adjust all indexes
Delete	Find row then delete. DBMS adjusts prior and next rows appropriately. Adjust all indexes appropriately	Delete row if allowed by referential integrity constraints, then adjust all indexes appropriately

**Figure 4.41.** Static and dynamic effects on row adds and deletes.



One plan developed in the early 1970s for CODASYL (static) network DBMSs is to store the member rows physically close to their owners through a technique called STORED VIA. Today, this technique is called data clustering. Data clustering can only be done for one of a table's owners. Data clustering is naturally practical with hierarchal DBMSs and can sometimes be accomplished with independent logical file DBMSs through the use of multi-valued columns and repeating groups. In relational DBMSs, this technique is only now being rediscovered in order to achieve performances commonly present with static DBMSs. The basis in dynamic DBMSs for clustering is the primary-to-foreign key relationship.

The drawbacks to both static and dynamic DBMSs are the same. As the data organization is optimized to serve the needs of one application, it is de-optimized with respect to other applications.

Whenever space is no longer available for the inclusion of a row in a DBMS row, the DBMS row is either split according to a DSDL plan, or a new DBMS row is started with the new row being stored there.

When rows are deleted, their former space may be available for re-use. With sophisticated DBMSs, space available lists are maintained so that as new rows are entered, the available space is consumed.

In static environment, additions and deletions of rows also affect relationship structures. These changes are presented in Section 4.5.3.

Common to both static and dynamic DBMS is the automatic adjustment of indexes when rows are added or when they are deleted.

#### 4.5.2 Column Changes

In most respects, changing column values is generally the same for static and dynamic relationship DBMSs. The row is located, the column value retrieved and changed, and the row replaced in the database. Figure 4.42 tabulates the types of effects of column value changes depending on the operation and on the context of the change.

Context	Operation		
	Add	Delete	Change
Fixed Length Rows	Change null to blank or value	Change value to null or blank	Change one value to another
Variable Length Rows	Expand row size	Shrink row size	Expand or shrink row size
Indexed Columns	Add new links	Delete links	Delete link Add link



Context	Operation		
	Add	Delete	Change
Primary Keys	Add new row	Remove row	Delete row Add row

**Figure 4.42.** Column change effects.

With fixed length rows, ADDs, DELETEs, or MODIFYs only change the value already stored in the space. If no value is stored, then the change is NULL to a value or vice versa. With variable length rows, the space occupied by the row expands or contracts. If there is no room on the current DBMS row for an expanded row, the DBMS's DSDL plan:

- Moves the row to another DBMS row,
- Leaves (if very sophisticated) one of the expanded rows in the space and then with a forward referencing pointer,
- Moves half of all the remaining rows to a new DBMS row.

In any case, all the appropriate entries for indexes and relationships (static only) are adjusted.

If a column is also indexed, then whenever its value changes the appropriate index structures are changed.

If the column is also a primary key, value change effects range from dramatic to minimal. For some DBMSs the effects are so dramatic that changing the primary key value is prohibited altogether. Some other DBMSs allow the primary key value to change despite the effects. The effects arise whenever a DBMS uses the primary key value to physically locate the row. For those DBMSs, a change in a primary key value is the same as a row delete and a re-add. Further complicating such an operation are the changes required to all the other rows containing the primary key value as a foreign key, and all the rows that are clustered around the existing row.

If the primary key value is part of the secondary index storage structure component then a change to the primary key value automatically invokes changes to these secondary index storage structure components.

If the DBMS supports a logical-to-physical primary key strategy described earlier in this chapter, then none of the column index structures associated with any affected rows have to be updated just because the primary key value of the row is updated. The only restriction is that its value can not be changed to an existing value.

### 4.5.3 Relationship Changes

A significant difference between static and dynamic DBMS approaches pertains to the relationship changes. Inter-row relationship changes for a static-based relationship are very different from the



maintenance to a dynamic-based relationship. In a static relationship DBMS, the relationship change process consists of issuing a series of DML commands that:

- Locates the row,
- Disconnects it from its relationship,
- Finds the new and proper position, and finally
- Inserts it into the new relationship.

To accomplish this task, host language programs are usually written, because the process of correctly maintaining database relationships is complex. The complexity of the process is directly related to the number of tables and the number of relationships among the tables.

For example, in Figure 3.9 and in conjunction with the data loading example above, if a CUSTOMER is to be related to a different SALESPERSON due to a change in territory, then a SALESPERSON is located, and the relationship instances that thread through the CUSTOMER instances are processed until the proper CUSTOMER instance is found and disconnected from the relationship. Since it has to be assumed that the new SALESPERSON-ID is known, the disconnected CUSTOMER row is connected to the new SALESPERSON using the verbs cited earlier. Figure 4.43 identifies the key actions that must take place to ADD, DELETE, and MODIFY relationships in this static network data model environment.

In the ANSI/NDL static DBMS environment, row ADDs and DELETEs always affect relationships. If the relationship (SET) is defined to be AUTOMATIC then the relationship instance to which the row belongs is always affected. If the INSERTION option for a SET is MANUAL then the relationship change is deferred. The row, while belonging to the database, does not belong to any SET. If a row is deleted and the RETENTION option of the involved SET is OPTIONAL, then the row can be deleted. If the RETENTION option is REQUIRED, then the delete request can be rejected.

The hierarchical data model's relationships are always affected when a segment in the hierarchy is added or deleted. When the top (root) segment is deleted there is almost always a cascade delete of all dependent segments. When a dependent segment is deleted then all lower level segments are deleted, the relationships between peer segments are changed, and the relationship to the parent's segment is changed.

When segments are added or deleted within a complex table structure of the independent logical file data model, the effects are the same as those for the hierarchical data model. For relationships across table instances, the effects are the same as for the relational data model.

Relationship modification in a relational data model DBMS is the simplest. The column



Change Type	DBMS	
	Static	Dynamic
<b>Add</b>	Store record, then navigate to proper location and insert into set	Change null or blank to value
<b>Delete</b>	Find record then delete. DBMS adjusts prior and next records appropriately	Change value to blank or null
<b>Change</b>	Navigate to existing location, then disconnect. Then navigate to new location and connect	Change one value to another

**Figure 4.43** Static and Dynamic Effects of Relationship Change

that represents the relationship is located, and its value is changed. That's all (see Figure 4.43). If there are referential integrity rules defined, then a column change may be prohibited because it violates a referential action rule.

## 4.6 Database Maintenance

Database maintenance refers to the creation of a backup copy of a database. While creating a backup is certainly the first step in backup and recovery, the impact of the backup process needs to be understood from the perspective of physical database so that sufficient resources can be allocated for its accomplishment.

The number of storage structure units that constitute an individual database varies widely from DBMS to DBMS. In general, when a DBMS is static, the number is small; when a DBMS is dynamic, the number is large.

Because there is this disparity, and because one application may in fact require multiple physical databases to contain all the logical database data, the techniques employed for multiple database backup can be critical to the application.

For DBMSs like SYSTEM 2000, the process of database backup involves a special utility that locates and combines all the storage structure components of a database into a single file. The amount of time to accomplish this is not much more than the cumulative amount of time to copy each of the components onto individual backup files. While the amount of backup time may be insignificant for a small database, backing up a database of 8 billion characters to tape can take up to 10 hours (60 tapes of 6250 BPI, 135 million characters per tape, 10 minutes per tape to backup).





To accomplish such a backup, the database would certainly have to be segmented and backed up segment by segment. A segment might be a certain collection of O/S files that store specific table instances, as well as their associated inter-row relationships (if static) and indexes.

## 4.7 Physical Database Summary

The physical database, from the DBMS point of view, consists of a database's storage structure, access strategy, data loading, data update, and database maintenance. The four storage structure components are: dictionary, indexes, relationships, and data. The interactions of these four components to store and retrieve DBMS rows are the DBMS's access strategy. Data loading is the process of initial data storage. And data update is the process of adding, changing, or deleting data once a substantial amount of data has been stored. Database maintenance is the process of saving an on-line database onto some off-line media.

From the user's point of view, the database consists of rows. The DBMS sees the database from the same point of view, except that *row* is understood differently. The user OBTAINS and PUTs rows from and to the DBMS. The DBMS accepts these rows and combines them into DBMS rows, which the O/S combines into O/S rows, which are then read from or written to mass storage devices. While a table instance may be coincident with a DBMS row instance, normally it is not. In most DBMSs, a DBMS row instance contains multiple table instances, and a table instance may span multiple DBMS row instances.

As the user reads and writes rows, the DBMS, in support of the user's requests, invokes read and write services from O/S utilities that place DBMS rows onto whatever storage structure components are necessary to support the user's table instances. The DBMS reads and writes DBMS row instances from all of its storage structure components--that is, from its dictionary, its indexes, its relationships (static only), and its data. If a user writes a single table instance to a database, the DBMS may place that row in the appropriate DBMS row buffer and not write it to disk because the DBMS row buffer is not full.

Furthermore, the DBMS (static only), in support of the user's row write, may have to create and store in the database multiple DBMS row instances in support of the table instance's relationships to other rows. The DBMS may also have to update the addresses for indexes that are changed, because the user's row contains indexed columns. In short, a single user's row WRITE might generate no DBMS row I/Os, one DBMS row I/O, or many DBMS row I/Os.

For the static relationship DBMS, once the database is created, it is bound together through the relationships that bind rows to each other. In contrast, once the dynamic database is loaded, the rows are not related until retrieval. Certainly, because so much less work is accomplished during the dynamic loading process, the dynamic database load is going to be significantly faster than the static load.

A critical difference between static and dynamic relationship DBMSs is in the area of data update. The dynamic relationship DBMS update is oriented towards the single table, while the static relationship DBMS handles complex multi-table updates through its comprehensive locking schemes.

Figure 4.1 contains a review of these critical differences in physical database between static and dynamic relationship DBMSs.



## References

1. The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems, Tsichritzis and Klug, University of Toronto, Toronto, Canada, AFIPS Press 210 Summit Ave, Montvale, New Jersey 07645, page 9.



## 5

# INTERROGATION

### 5.1 Interrogation Evolution

The logical database establishes the fundamental structure of the data in terms of ANSI standard or nonstandard data models. The physical database defines the physical structures and defines the basic factors governing performance and space efficiency. The interrogation component provides the linguistic mechanisms to add, delete, and/or modify database data and to conform the data into reports. This chapter identifies the basic language access alternatives and illustrates each within the context of statically and dynamically oriented DBMS.

From the time when DBMSs were first developed in the early 1960s until the early 1970s, access to data was either through primitive natural languages or through host language interfaces (HLIs). A natural language is a special language invented by the DBMS vendor. A host language interface is a specially constructed interface between a compiler language, such as COBOL, and the database.

Today most DBMSs have a variety of access languages. Selection of the right language is important. A database programmer might choose a natural language because writing a program takes only a fraction of the human effort required to write the same program in COBOL. Conversely, one might choose the COBOL interface language for its sophisticated processing and reporting capabilities. Or COBOL might be chosen because the natural language is DBMS-dependent, and any programs written in that natural language have to be re-coded if they are to be moved from one DBMS environment to another.

Determining the proper use of each DBMS language is important. A critical factor in the determination is portability. In general, natural languages should be used for tasks not intended to be ported from one DBMS to another. ANSI standard languages such as COBOL should be used for tasks that require portability.

The static or dynamic nature of a database has a profound effect on language for two reasons: navigation and relationship expression. Navigation is the expression of verbs to traverse already existing relationships. Relationship expression is the statement of new relationships between tables. In any type of DBMS, either navigation or relationships must be included in the end-user program.

In DBMSs with static facilities, navigation must be expressed with verbs such as GET OWNER, GET MEMBER, and so forth. Relationships do not have to be expressed as the relationships are already bound into the rows. In dynamic facilities, the relationships must be expressed with verbs such as:

```
CONNECT STUDENT TO GRADES VIA STUDENT.STUDENT-ID  
EQ GRADE.STUDENT-ID,
```

Or the relationships must be expressed within WHERE clauses such as:

```
... WHERE EXISTS (SELECT GRADE.STUDENT-ID FROM
```



GRADES WHERE GRADE.STUDENT-ID = STUDENT.STUDENT-ID)

The DBMS takes in that relationship expression, and uses it with other components of the language to process selected data for either reporting or updating.

Both static and dynamic facilities require the programmer to know the database's structure. In a static relationship database the structure is predefined in the DDL and the rows are loaded according to the structure's dictates. In the dynamic relationship database the structure exists as shared values between tables which await discovery during database retrievals.

The development of the skills needed to manipulate database structures is not a simple task. In general, small databases with simple structures usually imply a small user community. As databases become larger, the structures become more complicated and the user community grows. A natural consequence of a wider audience is a wider range of skill levels. There is likely to be a small community of very skilled persons who can easily grasp database structures and comprehend how to develop the appropriate navigation or relationship expression statements. There is always a larger audience for the data contained in the database. The corporation cannot always afford the time and resources to develop the needed skills, even if the ability and temperament are present.

In response to this problem, the DBMS vendor community has, starting in the late 1970s, developed an intermediary facility that removes the need for DBMS users to express syntax for database structure navigation or relationship expression. This intermediary facility, the view, was initially a part of the relational data model, but is now an integral interrogation component of all data models. The benefits derived from views are:

- Perceiving related tables as a simple table called a view row
- Shielding users from having to express relationships or navigation
- Enabling the same end-user language to acquire data from multiple data model DBMSs
- Enabling the same end-user language to acquire data from multiple DBMSs
- Making the end-user program significantly less complicated
- Permitting greater database structure changes without automatically requiring end-user program changes

Notwithstanding the existence of a view mechanism, which may prevent the end-user from seeing what is really going on, static relationship DBMS facilities can only report and process data according to the dictates of the already executed database's structure. Dynamic systems allow users or views to express structure relationships and then report and process from the DBMS-discovered data.

A complete DBMS interrogation program includes:

- Its view, which contains either a set of data manipulation language (DML) verbs to perform database structure navigation, or dynamic relationship specifications which *instruct* the DBMS on the relationships to be executed.
- Reporting and update (add, delete, and modify) facilities appropriate for producing a database add, delete, modify, or report.



Access to the database can be through either compiler languages such as C, COBOL, and FORTRAN, or through natural languages. Compiler languages like C, COBOL and FORTRAN normally do not have DBMS verbs built into them. Thus, an interface is required.

The DML interface to either C, COBOL, or FORTRAN is through a specially built interface constructed by the DBMS vendor. The C, COBOL, or FORTRAN acts as the host for the DBMS verbs that the programmer uses. Hence the name, host language interface (HLI).

The ANSI database committee, H2, has developed generic specifications for HLI operations for both NDL and SQL data models. These may be used by DBMS vendors in any way they wish to interface their DBMS to COBOL, FORTRAN, or any other computer language such as query-update, report writers, and other natural languages. All of the HLI methods described below are compatible with the ANSI H2 specification.

A natural language is a specially created language tailored specifically to suit the needs of the DBMS and to provide a way of accessing a database. Natural languages come in many varieties. For example, a procedure oriented language is typically a many-statement program able to accomplish complex data selection and processing. A query-update language program, on the other hand, is a single sentence oriented language that accomplishes simple reports and updates. Finally, a report writer program can be used to produce sophisticated reports with sorting, break totals, and the like.

Natural language facilities vary from menu-driven, scroll and pick interfaces to complete languages. In the first case the natural language program may never be materialized, while in the second case the program may be hundreds of lines of source language.

This chapter presents an overview of the method of interfacing DBMS access languages to databases, and of the various types of DBMS access languages, both host and natural.

## **5.2 DBMSs and Language Orientation**

The static relationship DBMSs for the most part cater to the complex program and data environment. Their main programmer is normally a data processing professional and who uses COBOL to communicate with the database.

The dynamic relationship DBMSs for the most part cater to the simple program and a more simple data structure environment with dynamic relationships. Its main programmer is someone with good analytic skills who does not need to be connected to the data processing department at all. The skills required for a natural language programmer are similar to those required for the language BASIC. In addition to needing only analytical skills, the natural language programmer is often the dynamic database designer and user. Thus the predominate language of the dynamic DBMS environment is to be understood and used by this analytical, non-data processing person.



### 5.2.1 Static Relationship DBMSs and HLIs

Over the years, static relationship DBMSs have developed HLIs to a far greater degree than natural languages. This is for three reasons: interrogation requirements, fundamental inflexibility of the DBMS, and interrogation audience.

First, the interrogation requirements of many HLI reports often demand data from many different tables, and the involvement of many totals, subtotals, and other types of calculations.

Second, the static relationship DBMS is not very well adapted to ad hoc interrogations. Thus, to develop a very comprehensive and flexible natural language for an on-line environment of ad hoc users is to promise both a capability and a level of performance that can not be delivered.

Third, the audience of most static database applications is generally considered to be *end-users* rather than database designers or programmers. To give these end-users powerful programming tools to create their own reports can cause a significant degradation in overall database performance, and can even lead to significant misinterpretation of database semantics.

### 5.2.2 Dynamic Relationship DBMSs and Natural Languages

Over the years, dynamic relationship DBMSs have developed natural languages to a far greater degree than HLIs. This is for three similar reasons: interrogation requirements, fundamental flexibility of the DBMS, and interrogation audience.

First, the interrogation requirements of many natural language reports require data from normally only a few tables, and while there are totals, subtotals, and other types of calculations, reports are simple enough to be configured in the natural language.

Second, the dynamic relationship DBMS is very well adapted to ad hoc interrogations since it has no preconceived report production orientation. Thus, the development and on-going refinement of a comprehensive and flexible natural language for on-line, ad hoc users in no way compromises the performance capabilities of the DBMS.

Third, the audience of most dynamic database applications is generally considered to be the more analytical user, who also might be the database designer and programmer. Since most use of the dynamic relationship DBMS is through the natural language, almost all of the database design, loading, update, reporting and database reorganization tools are also in a natural language format.

### 5.2.3 Types of Interrogation Languages

There are four general classes of interrogation languages. These are:

- Host language interface (to C, COBOL, FORTRAN, etc.),
- Procedure oriented language,
- Report writer, and
- Query-update



Since the last three languages are all vendor proprietary natural languages, any given vendor's languages fit these categories only in a general way. For example, one vendor's language may have both query and report writing capabilities, but is more oriented to query. Another vendor's language may be procedure oriented but have sophisticated report writing capabilities. Because there are no ANSI standards for the natural languages, both a general description and vendor specific examples are given here.

The host language interface (HLI) is characterized by database utilization through ANSI standard compiler languages such as C, COBOL, and FORTRAN. The principal purposes of the HLI include:

- Data processing intensive applications
- Data migration and transfer to, from, and between databases and DBMSs
- Providing maximum control over user to database interaction

The procedure oriented language (POL) is for utilizing database rows and values through a commonly available, but vendor proprietary, language. The capabilities of this language are like COBOL and FORTRAN, but only at one-tenth of the coding. The principal purpose of the POL, similar to HLI, includes:

- Data selection and processing
- Data migration and transformation to, from, and between databases of the same DBMS
- Reasonable control over the user to database interaction

The report writer language is another DBMS vendor proprietary language. This language is usually specially designed to create complex reports. The report writer is often used for prototyping HLI programs. Building report writer programs typically requires only one tenth the resources necessary to build the same program in HLI. Capabilities typically included are:

- Row and column titles, totals, counts, etc.
- Sorting
- Special computation verbs
- Various levels of break totals and titles
- Table look-ups
- Complete control over the user to database interaction as the user cannot update and may only supply argument values

The final language type, query-update, is also vendor proprietary language. Its main role and purpose is to develop simple reports, lists of rows, and to perform various type of simple updates. The SQL language is often mistaken to mean Structured Query Language as if it were a query language, when in fact, it is a standardized interface language that other languages can employ to access data from an ANSI standard database. The typical amount of time to build a query language instance is likely to be about one-hundredth the amount of time required to build the same run-unit in HLI.



It is important to pick the right language for the job. That language should offer the least programmer resistance to producing the final result. That generally means trying a higher level language first (query), then using report writer or POL, and then finally resorting to HLI. Figure 5.1 provides a comparison of these four language types, their relative work efforts required to accomplish a task, the language's level of user control over the user-to-database interaction, and then the ability to port the resulting language from one DBMS to another.

LANGUAGE SELECTION CRITERIA	INTERROGATION LANGUAGE TYPE			
	HLI	POL	RW	QUL
Task Development Effort	High	Medium	Medium	Low
Relative Work Units	100	10	10	1
Level of User Control Over Database Interaction	Low	Medium	Medium	High
Range of Portability From One DBMS to Another of the Same Data Model	Medium to High	Low	Low	Low

HLI = Host Language Interface

POL = Procedure Oriented Language

RW = Report Writer

QUL = Query - Update Language

**Figure 5.1.** Comparative development efforts by language type.

### 5.3 View Facility

A DBMS view facility (known in the network data model as a subschema (subordinate schema)) enables the organization's database administrator to define a user-to-database interface. Within the view interface there are four major components: language-independent clauses, language-dependent clauses, the data interface specification, and the select clause. The data interface component usually involves a subset of the columns from one or more tables.

Each of the four interface components is included their own special sections of an HLI user program. Because of this inclusion, and because the programmer must directly create program language statements that manipulate the variables contained in these four interface components, the program's database processing logic is tightly bound to the database's data model. This is a critical drawback of the view facility.





### 5.3.1 Language-Independent Specifications

The language-independent specifications include the view name, an enumeration of the table names from the schema that belong in the view, and an enumeration of the columns from the schema to be used within the view row, and some statement of how the rows from the different tables are to be interrelated.

### 5.3.2 Language-Dependent Specifications

The language-dependent specifications are styled after the language used for database access. Included are syntax clauses dealing with data types, data conversion rules, data names, and natural languages.

#### 5.3.2.1 Data Types

Each interrogation language may support data types different from those supported by the DBMS. Rules must be created for the proper translation of the database data types to the language data type. For example, data might be stored in the database in packed decimal format or variable length character strings to save space, but FORTRAN, for example, can understand only integer, floating point, and fixed length strings. Thus, the DBMS has to transform the data from packed decimal to integer or floating point, or from a variable length string to a fixed length string.

#### 5.3.2.2 Conversion Rules

A DBMS might be able to store double precision floating-point numbers or numbers in scientific notation. In COBOL, these data types do not exist and so the data has to be converted from its DBMS stored format to the COBOL format. While the rules for conversion are certainly not within the province of the view, statements about the precision of the conversion need to be made.

#### 5.3.2.3 Names

Each language requires different conventions for names. For example, if COBOL is the language, and if the schema names are multiple words without hyphens, then COBOL does not understand them. On the other hand, if FORTRAN is the language, and the names have hyphens, FORTRAN interprets the hyphens as minus signs. Consequently, the rules for natural language names vary from no blanks allowed in names to blanks allowed, to hyphens interpreted as minus signs, and so on. To overcome these anomalies, views have a well-defined set of alias clauses.



#### 5.3.2.4 Natural Language Editing and Default Formats

Natural languages also use view defined default formats for titles, spacing, edit masks, and so on, and default values for blanks, null values, etc. Figure 5.2 illustrates a view with these facilities. In this example, all the columns have default column titles, and some, in the event of no data, have default values whenever the DBMS representation of a NULL value has to be printed.

```
TITLE 'FALL SEMESTER GRADES'/DATE  
  
COLUMN SPACING IS 5  
DEVICE IS TERMINAL  
PAGE IS 132 WIDE BY 55 LONG  
DOUBLE SPACE  
SUPPRESS BLANKS  
PAGE EJECTS ON VALUE CHANGE  
SKIP AFTER RECORD PRINT  
SKIP BEFORE RECORD PRINT  
SKIP AFTER CONTROL BREAK ON <COLUMN>  
SKIP BEFORE CONTROL BREAK ON <COLUMN>
```

**Figure 5.2.** Report formatting.

#### 5.3.3 Data Interface Specifications

The data interface specifications contain the references to the tables and the columns included in the interrogation. In general, these references are provided automatically whenever a DBMS's natural language is employed. In an HLI environment, these table and column references are contained in either Labeled Common (FORTRAN) or Working Storage (COBOL).

#### 5.3.4 WHERE Clause Facilities

A critical component of any database view access is the WHERE clause. End-user programs, regardless of their language, need to select specific rows from within the set that may be made eligible to a view FIND.

There are seven different types of operators that can be contained within a WHERE clause:

- Unary
- Binary
- Ternary
- Arithmetic
- Statistical



- Boolean
- Parentheses

Unary operators are used to determine whether data exists or does not exist (FAILS) for a particular column. The operator is called unary because it involves only the operator. For example:

... WHERE SHIP-DATE EXISTS

... WHERE SHIP-DATE FAILS

In this example, the interrogation may merely want to access all orders that have either been shipped or have not been shipped. The EXISTS or FAILS operators determine whether there is any data at all in the column.

Binary operators are used to determine specific values, or classes of values. The operator is called binary because it involves the operator and a value. These operators include:

- Less than (LT)
- Less than or equal to (LE)
- Equal to (EQ)
- Not equal to (NE)
- Greater than or equal to (GE)
- Greater than (GT)

These operators are specifically used to select a class of rows that meets the specific condition. For example:

... WHERE ORDER-DATE GE 01/01/87

... WHERE ORDER-DATE LE 01/01/81

... WHERE ORDER-DATE EQ 11/15/86

Ternary operators also deal with classes of rows, but specifically define a range of values to be researched. The operator typically use keywords like SPANS. . . THRU. The operator is called ternary as it involves the operator and two values. An example of a ternary operator is:

... WHERE ORDER-DATE SPANS 01/01/81 THRU 01/01/87

Arithmetic operators deal with classes of rows that pass specific types of arithmetic tests. An example is to select all the orders that have a SHIP-DATE greater than 15 days after the ORDER-DATE. Such a calculation requires the use of date arithmetic to subtract the ORDER-DATE from the SHIP-DATE, and the comparison of the resultant number of days to the constant, 15. For example:



... WHERE ((SHIP-DATE - ORDER-DATE) GE 15)

Statistical operators are a special class of arithmetic operators. These include:

- Minimum (min)
- Maximum (max)
- Average (avg)
- Sum (sum)
- Count (count)
- Standard deviation

An example is to select all orders whose TOTAL-ORDER-AMOUNT is greater than the average of all orders. This example is constructed as follows:

... WHERE TOTAL-ORDER-AMOUNT GT AVG TOTAL-ORDER-AMOUNT.

Boolean operators enable multiple WHERE clause conditions to be connected. The operators are:

AND

OR (inclusive)

XOR (exclusive)

NOT

An example of Boolean operations is to find all the orders that have a SHIP-DATE more than 15 days after the ORDER-DATE with a TOTAL-ORDER-AMOUNT greater than the average TOTAL-ORDER-AMOUNT. Such a class of rows is important to find as they relate to significant amounts of non-invoiced products. For example:

... WHERE ((SHIP-DATE - ORDER-DATE) GE 15) AND  
TOTAL-ORDER-AMOUNT GT AVG TOTAL-ORDER-AMOUNT

A variation on the theme is to find the rows related to either of the two conditions. For example:

... WHERE ((SHIP-DATE - ORDER-DATE) GE 15) OR  
TOTAL-ORDER-AMOUNT GT AVG TOTAL-ORDER-AMOUNT.



The Boolean operator, NOT, accepts only those rows that fail the WHERE criteria. An example is to find all orders shipped earlier than five days of an ORDER-DATE and also later than 15 days of the ORDER-DATE. This involves ternary operators, date arithmetic, and the NOT operator. For example:

```
... WHERE NOT((SHIP-DATE - ORDER-DATE) SPANS 5 THRU 15)
```

The final operator is parentheses. These have already been illustrated within the previous examples. These are especially needed when arithmetic operators and Boolean operators are used. Parentheses are also used in combination with boolean operators. An example is to find all the orders in the West Coast sales region that have a SHIP-DATE more than 15 days after the ORDER-DATE or with a TOTAL-ORDER-AMOUNT greater than the average TOTAL-ORDER-AMOUNT. This example can be constructed as follows:

```
... WHERE ((SHIP-DATE - ORDER-DATE) GE 15 OR  
TOTAL-ORDER-AMOUNT GT AVG TOTAL-ORDER-AMOUNT) AND  
SALES-REGION EQ WEST COAST.
```

The two sets of parentheses are needed for the following reasons. The first set is needed to bracket the arithmetic operation that subtracts the dates. The second set is needed to bracket the two conditions connected by the boolean operator OR so that the result of that condition set is ANDed with the last condition.

As can be seen, WHERE conditions can be stylized to the individual need. The need to specify a WHERE clause is most acute in a query environment. While the view might be the same, the WHERE clauses need to be varied from one query statement to the next.

As database applications are developed, a standard set of views can also be developed that all the application programmers use. Depending upon complexity, some of these programs can be written in a compiler language (e.g., COBOL), some in report writer, and others in query-update. If a fully functional WHERE clause cannot be expressed in each and every one of these language types, then the DBA will need to be involved in virtually every HLI, POL, query-update, and report writer program.

To safeguard against inappropriate or inefficient WHERE clauses, DBMS activity audit trails can be used to capture the expressed WHERE clause and the resources consumed. In sophisticated DBMSs, these audit trails are themselves kept in a special database, and they can be browsed by the DBA, using a query language with a fully functional WHERE clause, to find the queries (and thus the users) whose consumed query resources exceed certain boundaries.

If the DBMS has a fully functional WHERE clause, programmers can specify the exact conditions that must be met before view rows are issued by the DBMS to the program's working storage. If the DBMS does not have a fully functional WHERE clause, or if the programmer cannot express the WHERE clause within the program, program development time and complexity are increased. The programmer still has to construct program-based logic to perform the selection operations otherwise performed by the DBMS. The resultant logic still has to be



tested, documented, and then maintained. If such logic requires an extra 100 lines in a COBOL program, then the life cycle cost of the program rises by \$5000 (\$50 per line).

Not having a fully functional WHERE clause is especially damaging to any natural language environment. Programs that would otherwise have been encoded in a report writer, POL, or a query language have to be encoded in COBOL. Simple reports that might have been constructed with 2 hours of programmer analyst time (\$50/per hour) have to be replaced with 500 line COBOL programs that cost \$25,000 each. The reason for this high cost is that such programs have to be designed, coded, tested, documented, and then maintained. Almost all of this activity is avoided with the use of a natural language.

It is also unsatisfactory to have a fully functional WHERE clause that can only be expressed as part of the view. This requires the DBA to be involved in the development of every query, report writer program, POL program, and HLI program. Clearly, such a requirement is wasteful of scarce company resources.

### 5.3.5 Rationale for Views

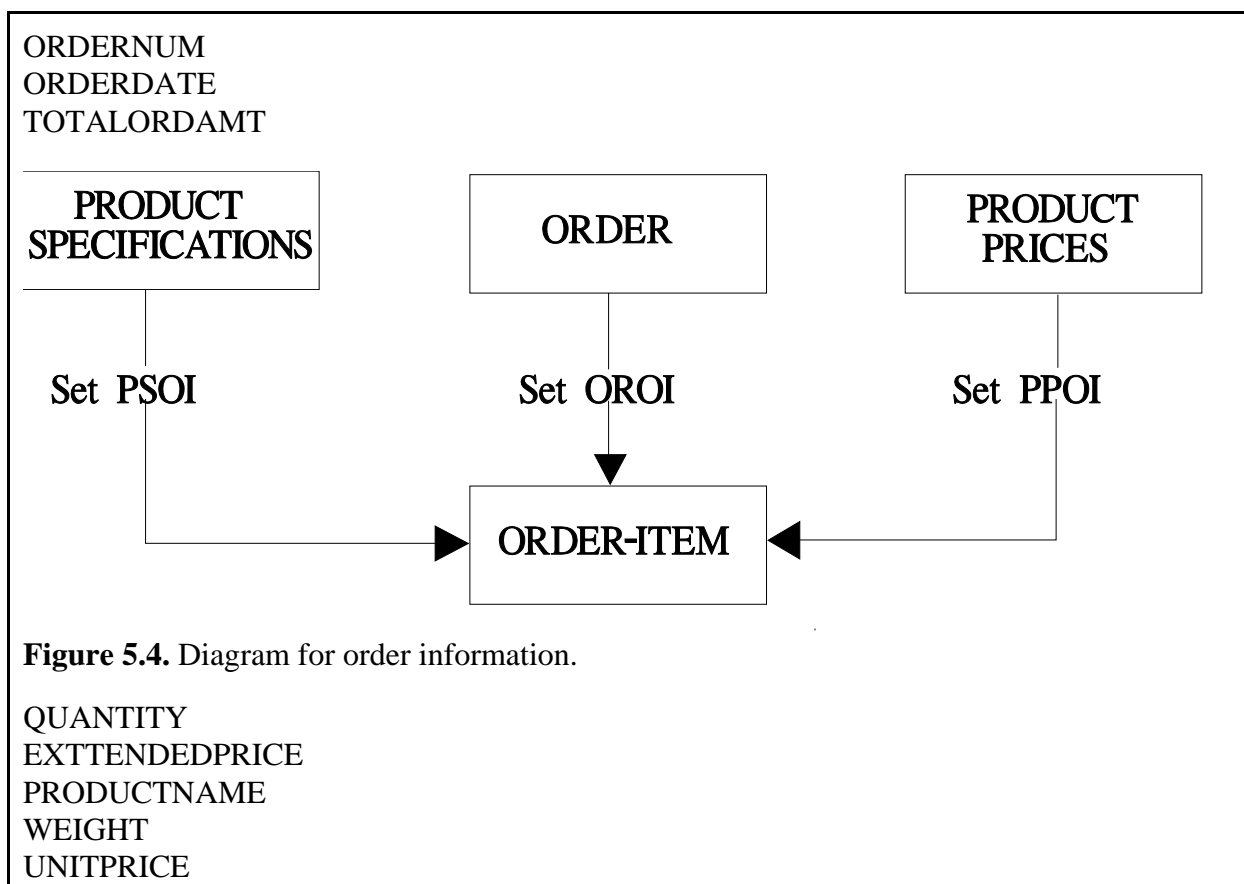
The view obviates the requirement for DBMS users to know the underlying DBMS data model. That means that the database user can think of data as a simple row, as depicted in Figure 5.3, and use verbs such as GET, INSERT, DELETE, or MODIFY. In addition to eliminating the need for programmers to know about the database's data model, the view facility eliminates the program's database processing logic. Thus, the program is not bound to either the DBMS or the database's data model. This is a significant benefit to the view facility.

Figure 5.4 presents the structure of an NDL database. In a traditional view environment, if a report of stored orders involves data from the tables in Figure 5.4, then a data structure interface must be created and made available to the program. Figure 5.5 illustrates the ORDERS schema. In addition to specifying the tables, the sets that must be traversed must also be identified, that is, product price to order item (PPOI), order to order item (OROI), and product specification to order item (PSOI).

When a program is written to access rows from a database structured similarly to that represented in Figure 5.4, the program has to include a view specification similar to the one contained in Figure 5.6. Additionally, in order to produce the order information part of an invoice, the program has to contain logic similar to that contained in Figure 5.7. It should be noted that the illustrations in Figures 5.4 through 5.7 are not intended to be syntactically correct NDL. Rather, they are provided to illustrate a program utilizing NDL without a view facility.

Figure 5.7 clearly shows that the *programmer* has to know about tables, set names, and a navigation oriented data manipulation language.





**Figure 5.4.** Diagram for order information.

**Figure 5.3.** Program view of orders.

In an analogous way, users of an SQL database also have to know about tables (tables), sets (columns from multiple tables with shared values), and a navigation-oriented data manipulation language. Figure 3.28 illustrates the set of tables appropriate to this example. The starred lines indicate the type of relationships that must be present through shared column values. Figure 5.8 provides an SQL version of the DDL for four tables from this relational database structure.



SCHEMA ORDERS

RECORD ORDER

ORDER NUMBER  
ORDER DATE  
TOTAL ORDER AMOUNT

RECORD ORDER ITEM

EXTENDED PRICE  
ITEM QUANTITY

RECORD PRODUCT PRICE

PRODUCT NUMBER  
PRODUCT PRICE  
EFFECTIVE DATE

RECORD PRODUCT SPECIFICATION

PRODUCT NUMBER  
PRODUCT NAME  
PRODUCT WEIGHT

SET PPOI

OWNER PRODUCT PRICE  
MEMBER ORDER ITEM  
INSERTION IS AUTOMATIC  
RETENTION IS OPTIONAL

SET OROI

OWNER IS ORDER  
MEMBER IS ORDER ITEM  
INSERTION IS AUTOMATIC  
RETENTION IS OPTIONAL

SET PSOI

OWNER IS PRODUCT SPECIFICATION  
MEMBER IS ORDER ITEM  
INSERTION IS AUTOMATIC  
RETENTION IS OPTIMAL

**Figure 5.5.** Abbreviated ANSI NDL orders database schema.





## View ORDERLIST OF ORDERS

## RECORD ORDER RENAMED ORDER

ORDER NUMBER RENAMED ORDERNUM  
ORDER DATE RENAMED ORDERDATE  
TOTAL ORDER AMOUNT RENAMED TOTALORDAMT

## RECORD ORDER ITEM RENAMED ORDITEM

ITEM QUANTITY RENAMED QUANTITY  
EXTENDED PRICE RENAMED EXTENDEDPRICE

## RECORD PRODUCT SPECIFICATION RENAMED PRODSPEC

PRODUCT NAME RENAMED PRODUCTNAME  
PRODUCT WEIGHT RENAMED WEIGHT

## RECORD PRODUCT PRICE RENAMED PRODPRICE

PRODUCT PRICE RENAMED UNITPIRCE

SET OROI

SET PSOI

SET PPOI

**Figure 5.6** Order List NDL View Specification



START

View ORDERLIST FROM ORDERS

NEXTORDER

INPUT ORDNUM  
IF ORDNUM EQ "O" GOTO END

NESTITEM

FIND ORDER WHERE ORDERNUM EQ "1234"  
IF ORDER NOT FOUND, DISPLAY "NOT FOUND" GOTO END  
GET ORDER  
PRINT ORDERNUM, ORDERDATE, TOTALORDAMT  
FIND MEMBER OROI  
AT END GOTO NEXTORDER, ELSE  
IF ORDITEM NOT FOUND, DISPLAY "ORDER ITEM NOT FOUND" GOTO END  
GET ORDERITEM  
FIND OWNER PPOI  
GET PRODPRICE  
FIND OWNER PSOI  
GET PRODSPEC  
PRINT PRODUCTNAME, QUANTITY, WEIGHT, UNITPRICE, EXTENDEDPRICE  
GOTO NEXTITEM  
END

**Figure 5.7** NDL Pseudo Code to Obtain Complete Order



## SCHEMA ORDERS

## RECORD ORDER

ORDER NUMBER

ORDER DATE

TOTAL ORDER AMOUNT

## RECORD ORDER ITEM

PRODUCT NUMBER

ORDER NUMBER

EXTENDED PRICE

ITEM QUANTITY

## RECORD PRODUCT PRICE

PRODUCT NUMBER

PRODUCT PRICE

EFFECTIVE DATE

## RECORD PRODUCT SPECIFICATION

PRODUCT NUMBER

PRODUCT NAME

PRODUCT WEIGHT

**Figure 5.8** Abbreviated ANSI/NDL Order Database Schema

The database represented in Figures 5.5 and 5.8 illustrate two major differences between NDL and SQL. First, the NDL version of the database contains 11 columns, and the SQL version of the database contains 13 columns. Second, the NDL version contains three relationship specifications (sets) and the SQL contains none. These two differences, however, as far as this example is concerned, are merely different semantic forms of the same component: relationships. In the NDL version, relationships are expressed through sets, while in the SQL version, the relationships are expressed through shared column values (same names in this example).

The SQL subschema facility is more refined than the NDL subschema. Every SQL view can contain both column references (like the NDL) and a WHERE clause that can select rows from WHERE-clause-referenced tables into the program materialized view table. Because of this WHERE clause capability, the SQL view depicted in Figure 5.9 corresponds to the combined NDL facilities from Figures 5.6 and 5.7. The SQL view contained in Figure 5.9 contains the WHERE clauses necessary to determine the joins (shared column values) among the four SQL tables. Figure 5.10 contains an example of an SQL query that acts directly against the view definition in Figure 5.9. Another major difference is that an SQL view can be nested while the NDL view cannot.



```
CREATE VIEW ORDERLIST AS

SELECT ORDER_DATE
      ORDER_DATE,
      TOTAL ORDER_ AMOUNT,
      QUANTITY,
      EXTENDED PRICE,
      PRODUCT _NAME,
      WEIGHT,
      PRODUCT_ PRICE

FROM ORDER, ORDITEM, PRODPRICE, PRODSPEC
WHERE ORDER.ORDER_ NUMBER = ORDER_ ITEM. ORDER _ NUMBER AND
      ORDER_ ITEM. PRODUCT_ NUMBER=PRICE. PRODUCT NUMBER AND
      ORDER_ ITEM. PRODUCT_ NUMBER=PRODSPEC.PRODUCT_ NUMBER
```

**Figure 5.9** SQL-like View Definition

```
SELECT PRODUCT_ NAME, QUANTITY, WEIGHT, UNIT_ PRICE,
EXTENDED_ PRICE FROM ORDER LIST
```

**Figure 5.10** SQL Pseudo Code to Obtain Complete Order

From the foregoing example, it is clear that although the problem to be solved is the same, when the underlying database design is different, the program itself is also different. The view serves as an important intermediary between the program and the database, obviating the need for the programmer having to know the data model under which the database was constructed. The view contains the data interface as well as the navigation logic for the database producing view rows. To support the NDL, the view references the view represented in Figure 5.6, and the navigation logic portion of Figure 5.7. The data interface for the view is depicted in Figure 5.3. The view generation logic is similar to that contained in Figure 5.11.



```

VIEW ORDER FROM View ORDERLIST

START

    FIND ORDER
    IF ORD NOT FOUND, RETURN "NOT FOUND", GOTO END
    GET ORD

NEXTITEM

    FIND MEMBER OROI
    AT END, RETURN "NO MORE ORDER ITEMS", GOTO END
    IF ORDITEM NOT FOUND,
        DISPLAY "ORDER ITEM NOT FOUND", GOTO END
    GET ORDERITEM
    FIND OWNER PPOI
    GET PRICE
    FIND OWNER PSOI
    GET PRODSPEC
    GOTO NEXITEM

END

```

**Figure 5.11** NDL View Specification ORDER

For the SQL, a view row can be from base tables or can be derived from an existing view. The initial view (Figure 5.9) contains all the joins necessary to materialize a complete order. A view defined on the initial view materializes a subset of the columns in the order view and a subset of the rows (rows), for example, those with a total order amount over \$350.00. An illustration of such a derived SQL view is contained in Figure 5.12. Figure 5.3 depicts the program's data interface for the view.

A host language interface program can now be designed to deal with the view as a simple row using four DML verbs, FIND, OBTAIN, MODIFY, and DELETE. Figure 5.13 illustrates the program logic used for both the NDL and the SQL. The user program containing these four verbs is not legal COBOL or FORTRAN. Thus, the program must pass through a precompiler that maps the view back to the database and translates the appropriate NDL or SQL data manipulation logic to the appropriate legal COBOL or FORTRAN calls. The program then proceeds through a normal FORTRAN or COBOL compiler resulting in an executable program.



```
CREATE VIEW BIGORDERS
SELECT *FROM ORDERLIST WHERE TOTAL_ORDER_AMOUNT GT 350
```

**Figure 5.12** SQL View Specification ORDER

```
OBTAIN ORDER WHERE TOTALORDAMT GT 350.00
```

**Figure 5.13** Pseudo Code for View Based Program

For natural languages, all the translation and binding between the natural language program and the DBMS is automatically handled by the DBMS.

This view mechanism is the typical method by which a DBMS vendor offers services from multiple data model DBMSs to application programs without requiring the application programs to be rewritten.

It is important, though, not to carry a good idea too far. For example, if the view mechanism only allows a restricted set of WHERE logic in the view, then there have to be many more views than necessary. For example, suppose there is a requirement to produce the same report but with the following three different sets of data:

WHERE TOTAL ORDER AMOUNT GT \$350.00

WHERE PRODUCT NUMBER EQUAL 745A2

WHERE ORDER DATE WAS *over two weeks old*.

DBMSs requiring the view to incorporate the complete WHERE clause need three different views. If, on the other hand, the DBMS allows the programmer to base the WHERE clause on any set of columns identified in the view, then the number of views is restricted to one. Eliminating unnecessary views reduces view design, programming, and maintenance by the DBA. It does no good to relieve the programmer from the distributed work of defining views if the only result is a large, centralized pile of unfulfilled view requests on the DBA's desk.

In summary, the view facility provides the following benefits:

- Related tables can be seen through one simple row, called the view row.
- Users do not have to express relationships or navigation.
- End-user languages can use data from multiple data model DBMSs
- The same end-user language can use data from multiple DBMSs.



- End-user programs can be simplified.
- Database structure changes can be accomplished without automatically requiring end-user program changes.

The view facility, once defined, is likely to be available in a view library so that different users can use the views through different languages. This allows an application dealing with customers to have a single set of appropriate views. These views can be used by the host language interface programs for complicated retrievals and updates, and by query languages and report writers for producing different types of reports.

If a DBMS offers a view facility, then the specific details related to navigation, whether static or dynamic, can be hidden from the programmer. The programmer must first know the data needed, however, and must communicate the data needs to a DBA. The DBA, in turn, researches the database structures finding the locations of the various columns, and then constructs the complete view, that is, the set of columns representing the program's needs and the navigation logic required to traverse the database's structures to access the data.

## 5.4 Screen Development

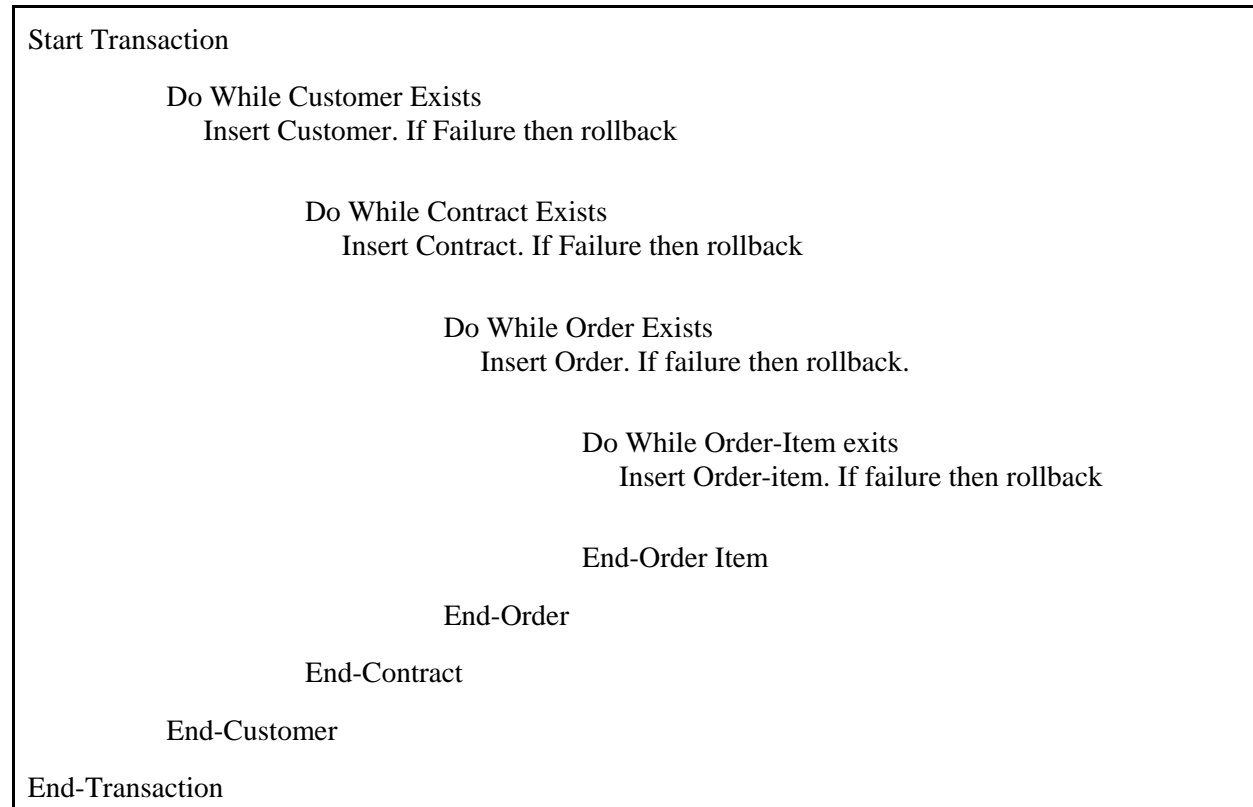
Screens are appropriate for HLI, POL, report writer, and possibly query-update access to the database. Once a screen display format is determined, its programming and database access should be attempted in the highest language possible. That means query-update, then POL, or report writer, and finally HLI. If at each language attempt the screen has to be reprogrammed, the run-unit's development costs rise considerably. In addition, the utility of the screen generator is very questionable.

Most database applications use screens for data entry and update, whether the DBMS that controls the screens resides on a microcomputer, a minicomputer, or a mainframe. Every modern DBMS therefore has facilities for the development, use, and maintenance of screens. It is critical that these screen facilities be available for use by all the appropriate access languages, rather than having to develop screens for host language interface and then a duplicate set for each of the DBMS's natural languages.

Screens are needed for either data display or for data entry. Screens can be developed to provide menu access to types of update transactions, and then, for each update transaction, to redisplay the changed view row, seeking confirmation prior to writing the data back to the database. Since most applications are self-evident only to their creators, HELP screens should be available that can be called upon from any other screen, returning control to the calling screen.

Most complicated applications need to have multiple screens since a given transaction can easily extend beyond one screen. For example, the business event, ADD CUSTOMER, defined in Figure 5.14, certainly contains more than one screen.





**Figure 5.14.** Business Transaction: Add Customer consisting of many run-unit transactions.

In fact, each INSERT <table name> probably contains one or more screens. Thus, there is a natural progression from the CUSTOMER screen, to the CONTRACT screen, to the ORDER screen, then a cycling of the ORDER-ITEM screens until the order is finished. Then there can be a screen to ask if there is another order. If there is, then the ORDER screen appears again. In short, there is not only a need for screen development, but also dialogue development for branching, looping, and the like.

For each screen, a number of characteristics need to be defined. The identifying characteristics include a screen's name, versions, authors, date of creation and update, and restrictions for updating and use. A good screen facility provides the ability to clone a different screen and make trivial changes to make the screen unique. Once a screen is defined, any error messages are flagged so that flawed clauses can be corrected.

As to actual screen definition and maintenance capabilities, the following list identifies capabilities of a sophisticated screen facility:

- Positioning literal fields
- Positioning data fields





- Varying the color of literal fields
- Varying the intensity of literal fields
- Changing the color of data fields
- Varying the intensity of data fields
- Specifying the position of data fields
- Changing the color of borders
- Performing automatic data editing and validation from IRDS stored tables
- Performing automatic table look-up from IRDS stored tables
- Enforcing required fields
- Automatic redisplaying of screens whenever errors are entered
- Automatic highlighting of errors through color or intensity change
- Automatic mapping of screen data fields to table columns
- Automatic invoking of database-defined data editing and validation facilities for screen data fields
- Automatic invoking of database-defined table look-up facilities for screen data fields
- Automatic building of menu screens once menu alternatives are defined
- Automatic building of help screens once help alternatives are defined

Once a screen has been developed, it should be usable in either a data entry or reporting mode. Screens should be usable for both the HLI and POL languages.

A screen becomes a useful component in any information system as there are only two sources for determining any statistics about the quantity of data stored in the database: screen data entry and batch data loading. A screen facility should therefore be able to generate use statistics and store them in a form for later reporting.

Once data is entered through a screen, the screen facility typically performs data editing, validation, and table look-up from IRDS stored tables, as well as enforcing data entry to required fields. This is all accomplished prior to the transmission of the row to the database. When an



error occurs, the screen facility highlights the error and provides a method for explaining the presented error message.

A final capability often present in screen facilities is the generation of statistics written by the facility to the IRDS so that the DBA or application manager can review the number of transactions attempted and the number that succeeded.

A necessary component of any screen driven program is the development of dialogues. This facility is needed to advance screens, transfer to other screens, and the like. The key capabilities that must be available in any screen navigation component are:

- Use PF keys (or mouse controls) to transfer control to next, prior, and top screens.
- Invoke screens from other screens by entering the target screen's name or identity.
- Invoke subroutines encoded in a compiled language, automatically when the screen's data is transmitted, to accomplish complicated processing.
- Present dialogue management messages so that flawed clauses can be modified.
- Store the *compiled* dialogue.

It is clear that screen and dialogue development and management are actually a whole programming environment in themselves. The results of this activity are a critical component of any database project. A sophisticated facility saves hundreds of thousands of dollars of design, development, and maintenance efforts.

## 5.5 Relationship Between Screens and Views

There is a definite relationship between views and screen. A view is a set of view columns presented to the program, and a screen is the format for the presentation of the view columns in the view. If all the view columns can fit on one screen, then the relationship is 1 to 1. If, however, there are more columns in a view than can fit on one screen, then there has to be more screens. Additionally, some of the view columns from an ORDER view might map to the CUSTOMER screen, some to the ORDER HEADER screen, and others still to the ORDER-ITEM screen. The point is that the screens and views must be defined separately, and then related.



## 5.6 System Control Capabilities

The facilities of system control normally available through interrogation language programs include:

- Audit trails
- Message processing
- Backup and recovery
- Concurrent operations
- Multiple database processing
- Security and privacy
- Reorganization

### 5.6.1 Audit Trails

Audit trails, or journals, are logs of DBMS/database activity that organizations wish to track. Depending on the DBMS, audit trails can be kept for particular tables, databases, applications, and entire central version environments. For very sophisticated DBMSs, a DBA might determine that the tracking is to be for updates only, for certain types of reports, for interactive but not batch updates, etc.

Audit trails oriented towards static relationship databases are normally well-developed and centralized to contain a single trail for all updates to all the tables in the entire database. Some static relationship DBMSs also allow audit trails to be started, stopped, kept for times of the day, certain applications, and even for certain access languages but not others.

Audit trails oriented towards dynamic relationship databases are usually not so well developed, and often there is one audit trail for each table. Thus, the only way to have a multiple table audit trail is to have the DBMS' central version operate the audit trail on behalf of all tables. This is further reason to keep updating strictly isolated to a single table when using a dynamic relationship DBMS.

DBMS central version audit trail transactions are kept principally to enable central version rollback and/or recovery in event of a hardware/software failure. Thus, this type of audit trail environment is independent of the static or dynamic orientation of the database. Audit trails are normally automatically available when the run-unit is operating, and are not optional. They are initiated and controlled by a centralized DBA group.

As expected, the sophistication of the audit trail is directly related to the cost of its accomplishment. For example, if the DBA decides that all updates are to be retained in a reprocessable and reportable manner and that along with the update transaction, the source of the



update, the identification of the run-unit, the source language of the run-unit (HLI, POL, etc.), time of day, etc., along with the before and after image of the schema row that was changed, then a lot of extra processing and storage has to occur to accomplish that work.

For example, if there are 1000 on-line, update users to a database and they store one update every three minutes for a six hour day, and the update only affects one schema row, and if the schema row affected is 100 characters long, then the database grows by at least 24,000,000 characters each day ( $1000 \text{ users} * 20 \text{ transactions per hour} * 6 \text{ hours per day} * 100 \text{ characters per schema row} * 2 \text{ copies for before and after image}$ ), just to contain the before and after images of the audit trail. Added to that is the actual audit trail row. If that is only 50 characters long, then the audit trail grows additionally by 6,000,000 characters ( $1000 * 20 * 6 * 50$ ). The total daily audit trail size is then 30 million characters. At the logical transaction rate computed in Chapter 4 of 80 transactions per wall clock second, that takes an additional 4 hours of daily processing ( $((1 \text{ audit trail row} * 1000 \text{ users} * 20 \text{ per minute} * 6 \text{ hours per day}) / (80 \text{ transactions per second} * 3600 \text{ seconds per hour}))$ ).

In a multiple database environment, audit trails must be controlled by the DBMS across all active databases. The DBA must then determine whether there is to be a single combined audit trail, one for each database, and the like. Critical questions like the following must be answered:

- Can audit trail rows stored on a single instance of a journal file be restricted to store updates to one or more databases?
- Can audit trail rows stored on a single instance of a journal file be restricted to store updates to one or more database areas?
- Can audit trail rows stored on a single instance of a journal file be restricted to store updates to one or more defined view table instances?

If answers to these questions are positive then the audit trail resources can be reduced to an acceptable quantity.

### 5.6.2 Message Processing

A good message processing facility within any interrogation language automatically intercepts and processes DBMS or operating system messages, making them available to the user.

### 5.6.3 Backup and Recovery

In either the static or the dynamic database environments, well-designed backup and recovery are essential. A backup is the process of making a complete copy of the database onto another medium. Since backup is a proper activity for the database administrator rather than an application program, it is covered in Chapter 6.



There are three types of recovery: user instigated transaction roll-back, DBMS restart, and full recovery. The only type of recovery that is applicable to the run-unit is user instigated transaction roll-back. The other types of recovery are treated in Chapter 6.

The transaction roll-back activity consists of executing the transaction ROLLBACK. The roll-back operation reverses the effect of one set of transactions. A transaction set is bounded by the two commands START TRANSACTION and END TRANSACTION. For some DBMSs, a COMMIT operation signals an END TRANSACTION and an immediate START TRANSACTION. The command FINISH indicates a commit with an END DBMS command. In HLI languages, and in some POLs, the use of the two commands START TRANSACTION and END TRANSACTION can be explicit. In many query-update languages, these commands are implicit, and are automatically included by the DBMS in every command that performs an update.

Generally the roll-back operation is limited to just one transaction. Figure 5.14 illustrates roll-back levels programmed into an order-entry application. The logic states that if a failure occurs on any of the INSERT operations, then a roll-back is to occur. And in the case of ADD CUSTOMER transaction, the roll-back is at the highest level. That means if the very last INSERT ORDER-ITEM fails, all the prior INSERTs for that entire customer are rolled back.

#### 5.6.4 Concurrent Operations

The very reason for creating the database discipline is to allow multiple users access to the same store of data--in a controlled manner. Even the most primitive DBMS allows concurrent access. The cornerstone of concurrent access is locking. Locking levels are:

- View row
- Schema row
- DBMS row instance
- All row instances of the same type
- All DBMS row instances in a file
- All instances contained in an area
- The entire contents of a database

In general, the levels of locking cited above go from fine to coarse. A view row however can cross multiple schema rows, and a schema row can cross multiple DBMS row instances.

When a database is opened by a run-unit, or an AREA is readied, or a view row is selected, the interrogation language usually provides some mechanism for locking. The locking level is specified by the DBA rather than the programmer. In SYSTEM 2000, for example, a user can access a database through either a DATABASE IS <database name> command or a SHARED DATABASE IS <database name> command. The former mode is exclusive and the latter is shared. If the site's DBA does not want any access other than shared, the DATABASE IS <database name> command can be disabled by the DBA.

Most DBMSs have a method of releasing locks that have been held *too long*. Since this varies both by data model and by DBMS, and is not covered by any ANSI standard, the levels of



locking, their length, methods of release, etc. all have to be determined and understood before they can be properly used in any language run-unit.

In addition to locking at various levels against one database, the locking concept must also deal with multiple databases. An organization might determine that databases are to be defined along application boundaries rather than subject boundaries. Such a decision necessarily requires synchronization of the redundant data. To accomplish synchronization some run-unit updates must execute against multiple databases. That means that locks must be possible on multiple databases at the same time and if an update fails on one database, the updates on the other database must automatically reverse.

The issue of deadly embrace surfaces often in database. If a run-unit needs to lock multiple resources, it is possible that after locking the first resource the DBMS finds the second resource locked by another run-unit. In such a case, the first run-unit is usually programmed to retry a lock on the second resource. During the re-attempt the run unit may hold the lock on the first resource. If there is no finite elapsed time or limit to the number of re-attempt failures that can occur, the first run-unit can be re-attempting forever. Eventually, another run-unit which has locked other resources will attempt to lock a resource that is already locked by the first run-unit. Upon finding it locked, it too will fall into a cycle of re-attempts. If enough of these forever-in-failure run-units are in execution, all real DBMS work stops. To overcome this situation, DBMS vendors have designed and implemented various schemes for deadly embrace resolution. Again, because this varies both by data model and by DBMS, and because deadly embrace is not covered by any ANSI standard, the methods of resolution have to be determined and understood before complicated run-units can be created with any level of confidence. Key questions that must be addressed are:

- Is the DBMS designed such that deadly embrace is automatically prevented and/or resolved?
- Is a deadly embrace resolved by stopping one of the run-units?
- Is a deadly embrace resolved by rolling back all the run-unit's transactions?
- Is a deadly embrace resolved by rolling back only the run-unit's committed transactions?
- Is a deadly embrace resolved by requiring the user to reexecute the run-unit?
- Is a deadly embrace resolved by attempting a run-unit restart a predetermined number of times before permanent abort?
- Is a deadly embrace resolved by attempting a run-unit restart periodically for a predetermined wall-clock time before permanent abort?



Answers and explanations to these questions are critical before multiple-user, multiple database applications can be created. For example, if deadly embrace is resolved by terminating execution of the run-unit that effects deadlock, then these run-units have to be programmed so they are invoked by a higher level program that offers an opportunity to restart. Conversely, if run-units can establish a maximum number of re-attempts, then does this number vary depending on whether the run-units operate in prime or non prime time, or possibly batch, or on-line?

### 5.6.5 Multiple Database Processing

Because a static database is potentially very large, complex, and comprehensive, there is little need for multiple database processing. But when the need exists, the host language often provides the only method to accomplish it. The need for multiple database processing usually arises when someone needs a report that combines, for example, sales and marketing data from multiple divisions that have each implemented the same type of database. Ironically, the relationships that exist between multiple static databases are dynamic. The dashed lines in Figure 3.8 illustrate the fact that almost all hierarchical data model databases need to accomplish multiple database processing. In the dynamic database environment, multiple database processing is its normal mode in either the host language or the POL. Figure 3.28 depicts dynamic relationships, and Figure 4.23 illustrates how they are processed.

Neither the ANSI/NDL or ANSI/SQL standards address multiple database processing in any acceptable manner. This means that each DBMS, ANSI standard or not, has to be examined in this very important area. It must be determined what procedure is required for a run-unit to obtain raw data from multiple databases and combine it into a single integrated report. The technique may range from having a run-unit execute multiple times, one time for each database, to having a run-unit execute against multiple databases in a serial fashion, to having a run-unit being able to have multiple databases opened concurrently and run-unit commands executing against these multiple database, in a coordinated fashion.

### 5.6.6 Security and Privacy

A carefully constructed view facility restricts access to specific tables and to specific column values for specific rows. Beyond that, the DBMS may abort run-units for illegal use of a DBMS operation or the attempted attachment of a table or column. The ANSI/SQL standard provides a reasonably good security facility. Security is not at all specified in NDL because the ANSI H2 committee (different membership at an earlier time) felt that security and privacy specification was the proper purview of an operating system's committee. Equal in importance to the specification of security is knowing how a DBMS reacts to security breaches. Some key questions that must be answered of any DBMS vendor for the proper assessment of a security facility are:

- Is a run-unit immediately terminated when it commits a security violation?
- After a security violation, can a run-unit attempt restart immediately?



- After a security violation, can a run-unit be prohibited from restart for a period of time?
- After a security violation, can a run-unit be prohibited from restart until the run-unit is *manually* revalidated?

### 5.6.7 Reorganization

Normally a language's run-unit is not involved in database reorganization. Some DBMSs, however, provide commands to the language that can cause the creation of an entirely new table including columns, index indicators, etc. This kind of capability is quite useful in the multiple database example described above. An application program can serially operate against multiple databases, creating and populating a new database and associated tables so that a report writer or query-update language can operate against the combined data.

## 5.7 Host Language Interface

Any host language interface (HLI) provided by DBMS vendors consists of the following components:

- HLI-DBMS interaction protocol
- Access language form
- View access
- Row access
- Screen development and utilization
- HLI cursors
- Updating
- DBMS and database attachment
- System control capabilities
- Error control

### 5.7.1 HLI-DBMS Interaction Protocol

The interface between a host language program and a database often requires a specially named data storage area within the program to act as a staging area for passage of data to and from the database. Figure 5.15 illustrates how a COBOL program interacts with IDMS/R. Within a program's data division, a view section identifies the name of the database and within the working storage section there are definitions of the tables that act as the data staging area.

Additionally, each program has a specially configured set of DBMS commands to start and stop the DBMS; to find, select, add, delete, and modify rows; and to traverse row relationships. Each non ANSI standard DBMS creates these interface components differently. If there is an intent to create programs that are DBMS independent, subroutines should be built to





hold these DBMS interfaces. Programs call these interface subroutines to GET or PUT data regardless of the DBMS employed.

Analogous to data interfaces, application programs have an area for the passage of messages and status indicators to the DBMS regarding the database. Figure 5.16 illustrates the various types of variable names that appear in the messages area for one DBMS. Since there are variables for area and set, it can be assumed that the messages deal with a CODASYL type network DBMS. There is an especially large difference among DBMSs with respect to cursors. In static network relationship DBMSs, there are a large number of cursors (sometimes called currency indicators) that identify which database, area, relationship (set), view row, and view row is currently active. Because the network data model is very complex, network cursors are very complex. As the DBMS's data model becomes less complex, that is, from network to hierarchical to independent logical file to relational, the complexity of the cursors lessens.

The HLI program also includes commands specially designed to manipulate the data represented by the DBMS's data model. Both static data models use relationship hypothesis expressions to navigate owners and members and among members. The network data model also supports finding multiple owners for a member and performing recursive set operations, while the hierarchical data model cannot.

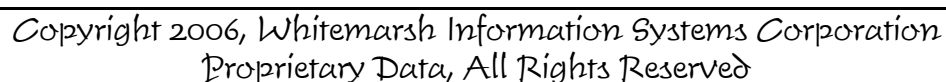


```
ID DIVISION
|
ENVIRONMENT DIVISION
|
DATA DIVISION
|
SCHEMA SECTION
|
DB STAFF WITHIN PERSONNEL
|
FILE SECTION
|
WORKING STORAGE SECTION
|
01 SUBSCHEMA - EMPLOYEE
   05 EMPLOYEE-ID
   05 EMPLOYEE-NAME
   05 ETC
```

**Figure 5.15.** Data interface area. COBOL program, data division. View section.



Some DBMSs have only simple commands such as FIND and GET, where others contain complex commands to represent the combined functionality of more than one command. For example, FETCH combines FIND and GET. Notwithstanding these variations, the following functions are normally supported by a host language interface capability:



Once a program has been precompiled, compiled, and link-edited, an execution version of the program (load module) is created. In some environments this execution version consists of all the routines necessary to complete the program's execution, while in other environments the execution version is able to invoke other load modules dynamically during the program's execution. In the first case, the execution time of the program is faster because no time is spent acquiring memory, I/Os, etc. for the dynamically loaded routines for DBMS, operating system, and telecommunications services. The second case, however, has an advantage because changes in the dynamically loaded modules does not force the program to go back through the precompiler, compiler, and link phases. This difference can be very practical. During the time an application is in development, aspects of a DBMS interface, such as a view, might change. Any such changes must be reflected in the programs using the view. If the programs have to pass back through the precompiler and compiler phases, the time to get affected programs back into production status can be longer. Clearly, the best environment is to have the option of including all components of the application program in the execution module or having them dynamically loadable during execution. When this option is available, quick changes can be made without re-performing all the compile steps, but once an application is finished and stable (changes only once each six months or so), the benefits of faster execution can be realized.

### 5.7.2 Access Language Form

There are three methods popularly used by DBMS vendors to interface compiler level languages (e.g., C, COBOL, FORTRAN, etc.) to database. Two are calls and the other is a specialized language. The CALLS formats are either function specialized or generalized. The function specialized CALL mechanism employs specially named CALLS like CALL GODHRA USING <variable>, <variable>, . . . , <variable>. Figure 5.17a illustrates this specialized CALL type of interface.

The second CALLS incorporates a generalized CALL facility like CALL 'ADABAS' USING <variable, variable, . . . >. In this case, each of the variables carries information to the DBMS, such as what operation is desired, the name of the view row accessed, selection criteria, and so on. The value of this type of mechanism over the previous one is that all CALLS are through a single paragraph in the user's program, albeit not very structured. Even though there is only one place for calling the DBMS there still must be the various instruction sets in the program to set the proper values for the different variables. Figure 5.17b illustrates this technique.

The third technique is through a vendor tailored language. This language is a series of data manipulation language (DML) verbs such as START DBMS, OPEN <database name>, and FIND <table name>. This technique is the one traditionally used in the various CODASYL DBMSs (network), SYSTEM 2000 (hierarchical), and in all ANSI/SQL DBMSs. Clearly, this technique is the most popular. The English-like statements are included in the source program. The combined-set-of-languages-run-unit is then submitted to a vendor provided precompiler. The output of the precompiler is a series of statements that are all C, or all COBOL, or all FORTRAN, etc. Figure 5.17c illustrates the translation through the precompiler of the three English-like lines to the five lines of COBOL.



In all three methods, the ultimate result is the same: data is accessed for a specific purpose. HLI mechanisms usually have automatic error procedure invocation, automatic data type translation, and automatic movement of data from the database to the user work area in the program. The HLI interfaces seem to be better developed in the static relationship DBMSs than dynamic relationship DBMSs. Most DBMSs seem to have either the generalized CALLS or the vendor tailored method of host language interface.

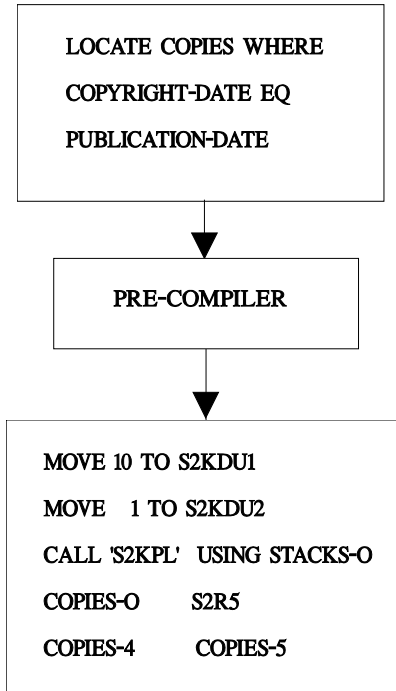
CALL "GODHRA" USING ERR WORKAREA EDIT - 1

**Figure 5.17a** Specialized CALLS interface.

CALL 'ADABAS' USING  
CONTROL - BLOCK  
FORMAT - BUFFER  
RECORD - BUFFER  
SEARCH - BUFFER  
VALUE - BUFFER  
ISN - BUFFER

**Figure 5.17b** Generalized CALLS Interface.





**Figure 5.17c.** Precompiler interface.

### 5.7.3 View Access

View facilities impact the requirements for data interface. With view facilities an application program must contain a set of view cursors, data areas for messages, and a simple set of database access commands.

#### 5.7.3.1 View Operations

The basic view operations are:

- **FIND**, a command that determines the identifiers for a set of view instances. These instances are then available for row-at-a-time processing.
- **GET**, a command that obtains a specific view row from the queue of view instances found.
- **DELETE**, a command that removes a view row from the database. Some DBMSs will only allow database changes (insert, modify, or delete) to be performed on schema table instances (base table instances).



- INSERT, a command that stores a new view row in the database.
- MODIFY, a command that changes existing view column value instances from one value (null, blank, or valued) to another value.
- START TRANSACTION, a command that indicates the start of an identified set of run-unit database change commands having their actions reversed--in unison--by the rollback command, or whose effects are automatically reversed by the DBMS after a DBMS or O/S failure.
- END TRANSACTION, that indicates the end of a set commands that are subject to a rollback.
- ROLLBACK, a command that can be invoked by the run-unit to cause the removal of a set of commands that have been demarcated by a start and end transaction.
- COMMIT, a command that causes the permanent storage of the run-units database changes.
- FINISH, a command that indicates that the uncommitted transactions are to be committed and that is the last command to be executed by the run-unit.

### 5.7.3.2 View WHERE Clauses

WHERE clause facilities impact program design and implementation. If the WHERE clause is not fully functional, then extra logic has to be built into the program to refine a gross set of view instances to just those view instances needed by the program. If the WHERE clause cannot be placed in the program, then the DBA has to create a view for every program. Both these additional requirements expand the analysis, coding, debugging, testing, and documentation time for each program.

The WHERE clause is appropriate for the FIND, DELETE, INSERT and MODIFY commands. A sophisticated WHERE clause contains:

- Unary operators (Exists and Fails)
- Binary operators (GT, GE, EQ, NE, LE, LT)
- Boolean operators (AND, OR, NOT)
- Ternary operators (SPANS, RANGES (i.e., GE <value> AND LE <value>))
- Arithmetic operators (+, -, \*, /)
- Statistical operators (min, max, avg, sum, standard deviation)
- Parentheses (nesting)
- Partial column value searches
- Exact mask searches



- Prefix searches
- Suffix searches
- *Contains* searches

While it is certainly possible to include indexed and non-indexed columns, it is important to know whether the WHERE expression can contain:

- All non-indexed columns from different tables, or must contain at least one indexed column
- Multiple indexed columns from the same table
- Multiple indexed columns from different tables

Key to understanding the performance of a DBMS is knowing its access strategy, and knowing how that access strategy is applied to the different indexed and non-indexed view columns, different WHERE clause operations, and the effects of having different view columns from different schema tables, and finally the difference between static and dynamic relationship DBMS. The minimum set of cases that must be examined to determine if WHERE clauses are optimized are:

- Multiple non-indexed columns from the same schema table, rather than just one non-indexed column.
- Multiple non-indexed columns from different schema tables, instead of just one non-indexed column.
- A single indexed column in place of just one non-indexed column.
- Multiple indexed columns from the same schema table, other than just one indexed column and other non-indexed columns from that schema table.
- Multiple indexed columns from different schema tables, rather than just one indexed column from one schema table, along with non-indexed columns from other schema tables.

Some DBMSs publish exactly how their WHERE clauses are evaluated and processed. This gives the knowledgeable user the ability to construct WHERE clauses that take advantage of the DBMS's processing logic. For example, in SYSTEM 2000, WHERE clause processing logic is from right to left. Thus, when processing a WHERE clause that includes view columns from different schema tables, the WHERE clause processes faster if ordered right to left to contain view columns highest in the hierarchy as well as the most restrictive in value hits. A more sophisticated DBMS will upon submittal of a WHERE clause analyze it to ascertain the most efficient method of processing, that is, indexed, sequential, and serial searches.

Since each DBMS vendor has complete freedom to construct the DBMS's storage structure and access strategy, only a careful examination and very carefully crafted benchmark tests determine the DBMS that is best for a given application.





A static relationship DBMS requires that relationship mechanisms exist between related tables. Thus, a select clause cannot contain view column WHERE criteria from unrelated schema tables. In contrast, a dynamic relationship DBMS select clause can express relationships across different tables within the WHERE clause. Some DBMSs require the joined columns from the different tables be named the same whenever a relationship operation is to be performed. Other DBMSs expressly prohibit columns to be named the same. Some DBMSs required the joined columns from the different schema tables to have exactly the same data types and picture clauses, while other DBMSs only require the data types to be generally the same.

### 5.7.3.3 Run-Unit Sort Clauses

The FIND command typically locates a set of view rows as a consequence of the WHERE clause. Since different run-units often want the view rows in different orders, sophisticated DBMSs provide sorting operations. Traditionally, most static relationship DBMSs provide sorting through a sort clause option on the DDL SET clause (see Chapter 3). If a user wants the view rows in the order maintained by a SET clause, everything is fine. However, if the desired order is to be different from the SET maintained order, the user has to FIND the view rows, retrieve them, place them in a separate O/S controlled file, invoke an O/S sort utility, sort the view rows, and then re-retrieve them into the run-unit. While this procedure is practical for a COBOL run-unit, it is not practical for POL, report writer, or query-update run-units. As a consequence, DBMS vendors include their own sort clause capabilities in these languages. All dynamic and some static relationship DBMS vendors include sort clause capabilities in their HLI. Dynamic vendors always include sorting because maintaining multiple ordered relationships among the schema rows is contrary to the very nature of the dynamic data model (ILF and relational data model DBMSs).

Sophisticated sorting supports:

- Single view column low to high sorting
- Multiple view column low to high sorting
- Single view column high to low sorting
- Multiple view column high to low sorting
- Combinations of view column high to low and low to high sorting

Sorting efficiency is critical. Some DBMSs speed up sorting whenever the sort column is indexed.

### 5.7.4 View Access

If a view facility is not present within the DBMS or if the view facility is not sophisticated enough for the type of processing that is required, then it is necessary to process directly against the schema tables, mapped through a view facility. In this case, the data model of the DBMS cannot be shielded, and the commands that perform schema table operations and relationship



operations must be present. The application program must set aside a working storage area for each schema table (or subset of columns). In addition, the individual program must include a full set of navigation commands to traverse the database's structures. Each of these additional requirements expands the analysis, coding, debugging, testing, and documentation time for each program. The table operations generally common to both static and dynamic relationship DBMSs are:

- Find
- Insert
- Delete
- Modify

Similarly, the operations that perform transaction control operations, also independent of data model, are:

- Start transaction
- End transaction
- Rollback
- Commit
- Finish

Relationship operations are dependent upon the data model of the DBMS. In general, the network and hierarchical data model DBMSs support the following relationship operations:

- Get first
- Get last
- Get next
- Get prior
- Get member
- Get owner
- Connect
- Disconnect

In general, the independent logical file and relational data model DBMSs support the following relationship operations:

- Join
- Outer join
- Division
- Union
- Outer union
- Intersection
- Difference



When there is direct manipulation of the schema rows rather than view rows, the sophistication of the WHERE clause is usually governed by underlying sophistication of the DBMS and whether the DBMS is static or dynamic. In static relationship DBMSs, the presence of DDL based relationship clauses usually permits the WHERE clauses to have columns from different schema tables. In dynamic relationship DBMSs, there are two main subcases: SQL and all others. In SQL, the WHERE clause has a series of hierarchically nested relationship clauses within the WHERE clause. In *all others*, the typical dynamic relationship DBMS's WHERE clause determines a set of primary keys based on the conditions present in only one schema table. These primary key values are retrieved and stored in a program defined array. A programming loop iterates through the array of primary key values and appends any other WHERE clause conditions relevant to each. Figure 5.18 illustrates this technique against the data structure depicted in Figure 5.8.

Sort clauses in the direct schema table access environment are directed only to one schema table at a time. These sort clauses should contain the same capabilities as described above.

### 5.7.5 Screen Development and Utilization

If an organization makes full use of an IRDS then the appropriate screen characteristics about data definitions are available from the IRDS component to define an interrogation screen. A critical characteristic of an IRDS is that once it contains the appropriate set of semantics and rules, the screen facility must then enforce them. A screen facility must not contain contradictory definitions. Further, in the event these centralized semantics change, the specific screen affected by these changes must be automatically disabled from working. Such automatic semantic enforcement and/or automatic disabling of screen executions demarche active IRDS from passive IRDS.

Screens are almost always a critical requirement of an HLI based report or update program. If screens can only be defined with COBOL facilities rather than be copied from a library of screens, then the analysis, coding, debugging, testing, and documentation time for each program is expanded. The facilities of a good screen generator are described in Section 5.5 of this chapter.



```
10 TABLE ORDERS (ORDER_ NUMBER, TOTAL_ORDER_AMOUNT)

15 TABLE ORDITEM (ORDER _NUMBER, LINE_ITEM_NUMBER,
EXTENDED_PRICE)

20 ARRAY ORDNUM (1000)

30 FIND ORDERS WHERE TOTAL _ORDER_AMOUNT GT 3500.00

40 FOR I = 1 TO COUNT. ORDERS

45 GET NEXT ORDERS

50 ORDNUM (I) = ORDER _NUMBERS

60 NEXT I

65 FOR I = 1 TO COUNT.ORDERS

70 FIND ORDITEM WHERE ORDER NUMBER = ORDNUM (I) AND
    EXTENDED _PRICE GT 1000.00

75 .....

80 NEXT I
```

**Figure 5.18** Basic language-like example of schema table access

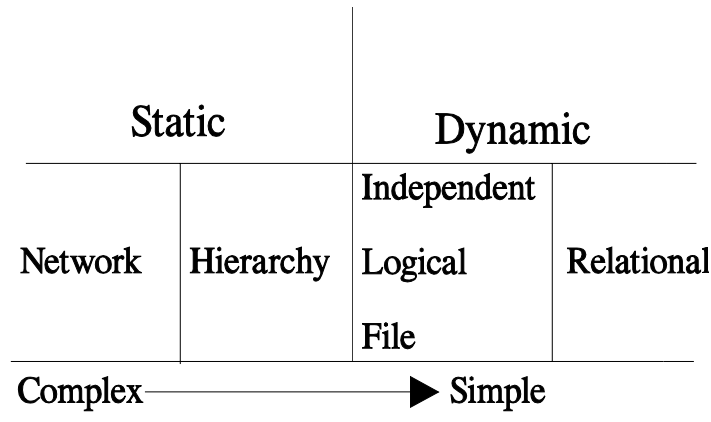
### 5.7.6 HLI Cursors

Cursors are indicators of position. For example, if a WHERE clause has identified 500 view rows and 75 have been retrieved, then the cursor position is 76. In the various data models, the number and types of cursors vary. The more complex the data model, the greater the number of cursors. This is illustrated in Figure 5.19.



**Figure 5.19.** Cursor complexity

Figure 5.20 illustrates the complexity and number of different cursors that exist in



### Cursors (currency)

IDMS/R network. It shows that there is an indicator of the current RECORD NAME, AREA NAME, and SET NAME.

Figure 5.21 illustrates the SYSTEM 2000 cursor model. In SYSTEM 2000, a hierarchy can be 32 levels (0 to 31), and there can be up to 16 (0-15) different open WHERE clauses. Once a WHERE clause is executed, the identifiers of the relationship pointers (SYSTEM 2000 has separated relationship pointers shown in Figure 4.18) are placed into a programmer identified queue that in SYSTEM 2000 is called a stack.

Not only can there be a greater quantity of different cursors, but a DBMS may have multiple cursors of the same type. If there are 100 different run-units executing concurrently against a database then there must be at least 100 different cursors. If each run-unit is dealing with 10 different WHERE clauses then there are 1000 open cursors.

In ANSI/SQL and the ILF data model, there is basically only one cursor type: that which results from a SELECT statement. This is illustrated in Figure 5.22. The SELECT statement is contained within a DECLARE CURSOR <cursor name> statement that identifies the name under which the view rows are to be obtained.

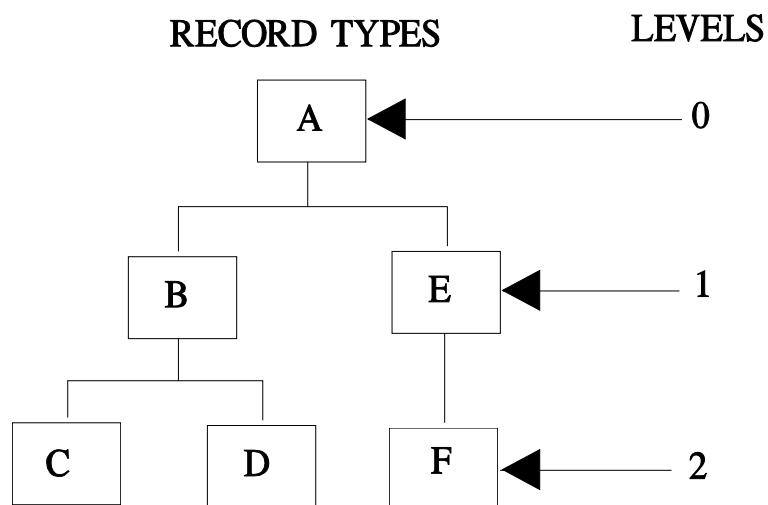


01 VIEW-CTRL	
03 PROGRAM-NAME	PIC X (8) VALUE SPACES
04 ERROR STATUS	PIC X (4) VALUE "1400"
88 DB-STATUS-OK	VALUE "0000"
88 ANY-STATUS	VALUE "0000"-"9999"
88 ANY-ERROR-STATUS	VALUE "0001"-"9999"
88 DB-END-OF-SET	VALUE "0307"
88 DB-REC-NOT-FOUND	VALUE "0326"
03 DBKEY	PIC S9 (8) USAGE COMP SYNC
03 RECORD- NAME	PIC X(16) VALUE SPACES
03 RECORD - DEF	REDEFINES RECORD-NAME
05 SSC-NODN	PIC X (8)
05 SSC-DBN	PIC X (8)
03 AREA-NAME	PIC X (16) VALUE SPACES
03 SET NAME	PIC X (16) VALUE SPACES
03 ERROR-SET	PIC X (16) VALUE SPACES
03 ERROR RECORD	PIC X (16) VALUE SPACES
03 ERROR-AREA	PIC X (16) VALUE SPACES
03 DBMSSCOM-AREA	PIC X (100) VALUE LOW-VALUE
03 DBMSSCOM	REDEFINES IDBMSCOM-AREA PIC X
01 VIEW-SNAME	PICX (8) VALUE "DEMOSO1"
01 VIEW-RENAMES	

**Figure 5.20** IDMS/R Network Cursor Variables

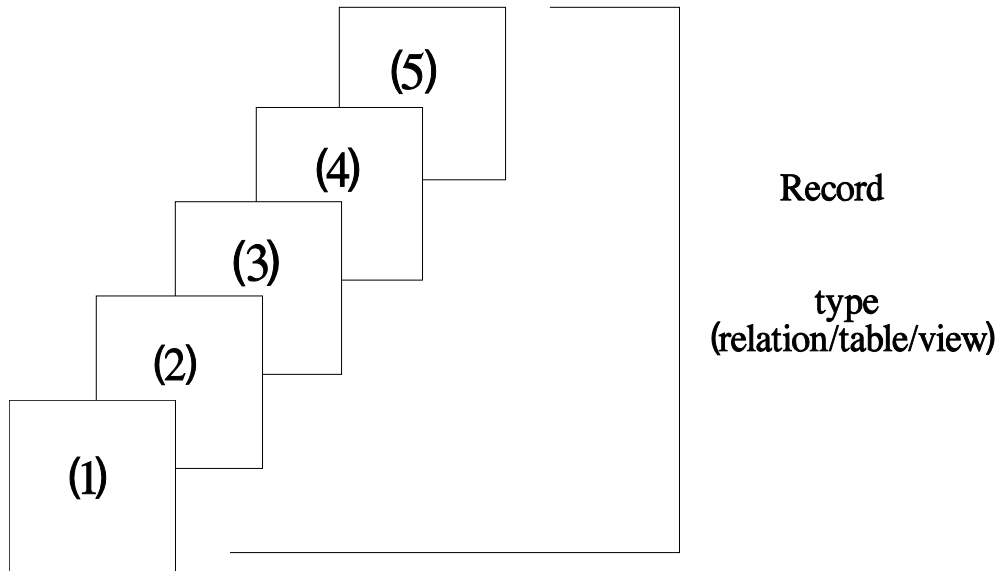


LEVEL	STACK NUMBER			
	0	1	2	3 ..... 15
0	a,1			
1	b,1		e,1	
2	c,1	d,1	f,1	
3				
4				
5				
6 . . . . 31				



**Figure 5.21.** System 2000 hierarchy cursor model.





**Figure 5.22.** ILF/Relational cursor model.

There are more cursors in the NDL data model because its underlying network data model is far richer in explicitly definable relationships than is possible in SQL's underlying relational data model. The cursors that NDL supports are

- Schema table
- schema row
- Set type
- Set position

In the CODASYL version of the network data model the AREA cursor is also present. Figure 5.23 summarizes the types of cursors present in each data model. Since ANSI standards only cover the NDL and the SQL, the cursor models of the various hierarchical and independent logical file data model DBMSs are different in detail, although fundamentally similar. The cursor models of the CODASYL network model DBMSs differ for two reasons: CODASYL was never an enforced standard, and the different CODASYL DBMSs were implemented from different versions of the CODASYL Journals of Development.





CURSOR MANIPULATED ON BEHALF OF	DBMS TYPE			
	STATIC		DYNAMIC	
	Network	Hierarchy	ILF	Relational
Database	YES	YES	YES	YES
Area	YES	NO	NO	NO
Set Type	YES	NO	NO	NO
Set Instance	YES	YES	NO	NO
Table	YES	YES	YES	YES
Row	YES	YES	YES	YES

**Figure 5.23** Cursor Types by Data Model

### 5.7.7 Updating

There are three types of updating: row, column value, and relationships. Row updating consists of adding, inserting, or deleting. If a row is added or deleted from a static relationship DBMS, then not only is the row affected, but so are any relationships that may be defined and any static referential integrity. In the case of a dynamic relationship DBMS, while there are no formally defined relationships, there may be referential integrity constraints defined. Thus, deletion of a row may be disallowed. Additionally, if a row is added, and is subject to an *owner* referential integrity action constraint, then the addition may be disallowed unless the proper *owner* row is already stored.

Column value updating consists of changing a NULL to a blank or other value, changing a value to another value, and changing a value to blank or to NULL. Column value changes can be governed by clauses that prohibit the storage of NULL values, or when a value is removed replacing it with a default value. In ANSI/NDL, if a referential integrity clause is defined by a SET statement that is constrained by a column's value, then the update may be disallowed without first having removed all the rows associated with that value. In ANSI/SQL, if a foreign key column value is changed then the basis of the relationship is changed.

In static relationship DBMSs, relationships are changed by connecting or disconnecting a row from the maintained relationship. In dynamic relationship DBMSs, column value updates can change relationships. In the ANSI/NDL the special verbs that perform relationship updating are CONNECT and DISCONNECT. These verbs are needed when the insertion option on SET membership is MANUAL rather than AUTOMATIC, or when the retention option on set membership is OPTIONAL.

If the HLI interface is flexible then the programmer can direct which view columns are updated by providing specific names rather than letting the DBMS update all the view columns



contained in the view. Some DBMSs disallow updates whenever the view contains view columns from multiple tables. In such a case there is usually one set of views for updates and another set (multiple table views) for retrievals. The critical update questions relate to whether a single run-unit command can add, modify, or delete a single view row that maps to

- A single instance from a single schema table
- Multiple instances from a single schema table
- A single instance from multiple schema tables
- Multiple instances from multiple schema tables

Clearly every DBMS can accomplish the first type of updating. Simply this means that the view maps to one schema table and affects one schema table instance. An example is to add a new schema table instance for PRODUCT-SPECIFICATION, or to select and then modify a PRODUCT-SPECIFICATION schema table instance, or to select and delete a PRODUCT-SPECIFICATION schema table instance. In the delete case, such a deletion can cause a real problem for all the other ORDER-LINE-ITEM schema table instances that reference the same PRODUCT-SPECIFICATION schema table instance. In such a case, the defined referential integrity (static or dynamic) should prohibit the actual deletion of the PRODUCT-SPECIFICATION schema table instance until all other ORDER-LINE-ITEMs using that PRODUCT-SPECIFICATION are deleted. If the referential action is CASCADE DELETE, then the ORDER-LINE-ITEM schema table instances are also be deleted. That is unacceptable. If the ORDER-LINE-ITEM schema rows are related to other schema table instances, such as ORDER, its referential integrity clause prohibits the deletion of the ORDER-LINE-ITEM schema rows.

The second type of updating is more difficult as it requires the DBMS to select and affect multiple schema table instances from the same schema table. For example, there may be a need to increase the prices for all products by 10%. If the DBMS cannot affect multiple schema rows from the same type through a single command, then the programmer will be required to develop looping logic. In the example above, a deletion action that affects multiple schema rows would wreak havoc throughout the database if all the referential integrity clauses and actions are not properly thought out.

The third type of update affects multiple schema tables, but only one instance from each. This type of update can create effects difficult to predict because of the relationships existing among the schema tables.

Beyond obvious relationships, there can be referential actions and data integrity rules that can affect a table and its instances beyond the scope of the view. For example, if a new employee is added to the database, the addition could produce new instances to many different schema tables. The addition can also set off a triggered procedure (data integrity rule) that adds the value of the EMPLOYEE SALARY column to the existing value from the TOTAL EMPLOYEE SALARY column in the DEPARTMENT table. If however, there was a constraint clause that limited the maximum value of TOTAL EMPLOYEE SALARY, (e.g., FROM 1 to 999,999.99), then the message EXCEEDED VALUE RANGE produced by an update to a schema table far outside the scope of the programmer's view is impossible to deal with. The only course of action for the run-unit is to backout the new employee and report that the rollback is due to an



unacceptable update to a schema table beyond the scope of the view. To report a more detailed message violates the implied scope of the view.

Because of the inherent complications of updating through views that embrace multiple schema tables and multiple schema table instances, many DBMS vendors, both static and dynamic, and many DBMS installations prohibit any type of updating except through views that map onto one and only one schema table. While such prohibitions may initially appear onerous to the programmer, they are actually helpful as a programmer is likely to be far more capable of dealing with row violation messages dealing with a few schema tables and only a few schema table instances than having attempted an action that affects many schema tables and instances that produces hundreds of error messages.

The fourth type of update is even more complicated than the third. Until all the problems of the third type of updating are solved, it is well beyond this book's scope to delve into its complications.

### **5.7.8 DBMS and Database Invocation**

The DBMS invocation command is functionally similar in both static and dynamic relationship DBMSs. In a static environment, a database OPEN statement has many options. They deal with logging, locking, shared retrieval and update controls, and exclusive retrieval and update controls. If a view is not used in a dynamic environment, the database OPEN statement is usually restricted to a single table, and when it is opened for update, it may automatically be locked to all other use.

Most organizations have multiple databases, each serving different subject areas. In addition to vertical segmentation of an organization's data, there may also be a horizontal segmentation, that is, a database for operational data in one subject area, and another database for MIS level data in the same subject area. Reasons to have multiple databases include segmentation processing for more cost effective backup and recovery and for distributed processing.

In response to the need for consolidated reporting from multiple databases, the HLI facilities must allow for multiple database OPEN commands, select, retrieval, and update commands.



### 5.7.9 System Control Capabilities

Specific details about system control capabilities available through interrogation facilities are presented in Section 5.7 of this chapter. The facilities normally available through an HLI run-unit include the following:

- Audit trails
- Message processing
- Backup and recovery
- Concurrent operations
- Multiple database processing
- Security and privacy
- Reorganization

### 5.7.10 Static and Dynamic Differences

The principal difference in the HLI between a static and dynamic relationship DBMS environment is the method by which database structure navigation is accomplished. For example, in the static environment there are specific navigation commands to traverse an owner to a member. In the dynamic environment the programmer is often required to acquire an owner row through a DML call statement, extract the connecting data value, and then use the value as a secondary key selection criterion in another DML call statement to find the members of the relationship. Figure 5.24 summarizes these differences.

DBMS TYPE		
CAPABILITY	STATIC	DYNAMIC
Sorting	Low	High
Selection	Medium to Low	High
Navigation	High	Low
Retrieval	Ok	Ok
Updating	Ok	Ok
System Control	Varied to Low	Varied to High

**Figure 5.24** Static versus Dynamic HLI Differences



## 5.8 Natural Languages

Natural languages are languages that provide access to the database for either reporting or updating. Common to all these languages is that they are proprietary to the DBMS vendor. Thus, the source code of one vendor's language is normally not able to be ported to another DBMS vendor's product. Some vendors of these languages, in an attempt to increase market share, are developing interfaces between these languages and other DBMSs. For example, Information Builder, Inc.'s FOCUS operates against many different DBMSs, hardware environments, and operating systems.

### 5.8.1 Natural Language Types

DBMSs generally employ three different types of natural languages. The first is a procedure-oriented language (POL), which is often used by ILF data model DBMSs as an alternative to HLI. The second natural language is report writer, which seems to be implemented regardless of the static or dynamic nature of the DBMS. The third language is query-update. In the static relationship DBMSs the query-update language serves mainly as a quick report writer against a well-structured database. In a dynamic relationship DBMS, the query-update language serves mainly as a heuristic research and reporting mechanism.

A very important distinction in the types of natural languages is whether each query command accesses the database directly or whether it accesses a subset of the database. Some DBMSs' query facilities cause the initially selected data to be extracted and placed in a work file that is used for the subsequent queries during that session. Others have each query statement in the query session execute directly against the database. Each alternative has its advantages and disadvantages. For the subset execution type, the entire subset of data is consistent and unchanging during the entire query session, but reports may not reflect the most current data. For the direct execution type, the advantages and disadvantages are reversed. For example, if there is a summary column that represents the number of employees in a department, NBR-EMPLOYEES, the column may not be the same as the count of employees in the chain if an ADD or DELETE executes completely between the start and the finish times of the query. An advantage to direct execution is that if the column is AMOUNT-UNENCUMBERED-DOLLARS, then the most current value can be retrieved directly so that funds are not over committed.

One type of natural language is not superior to the other. Instead, what must be decided is which type is more appropriate for the application.



## 5.8.2 Natural Language Execution Alternatives

There are often four different types of natural language run-unit execution strategies. These are

- Host language interface
- Interpretive
- Direct
- Compiler

An HLI execution natural language is actually a large sophisticated application of the DBMS written through an interface to a host language. Thus, the natural language command streams are actually strings of data that the HLI program reads, interprets, and then formulates into HLI calls to the database. An advantage of this form is that the natural language can be created merely by defining the mapping between its syntax units and the functions of the HLI. However, the translation of natural language to its actual executing functions cannot involve functionality not already present in the DBMS's HLI. This type of language is typically transaction oriented. That is, submit a unit of work, have it translated and executed, and then have a complete answer passed back.

An interpretative language form is one in which the DBMS determines what must be accomplished only after it has *seen* the command. If a loop is involved, the DBMS must interpret each command for every iteration of the loop, resulting in longer interpretation times, continuous reinterpretations for each cycle of the loop. Another disadvantage is that a natural language run-unit is often desired to be transaction oriented and the interpretive mode of run-unit construction and processing rarely supports transactions. The advantage of the interpretive language form is that its user is often able to interrupt and provide immediate alternatives to a failed syntax unit, and to interrogate the current values of variables, making corrections and simple debugging.

A direct mode execution natural language reads the entire command stream, determines what must be accomplished, and then performs all the needed operations by CALLing and then executing specifically created DBMS programs. An advantage of this mode is that the translation and execution are faster. Another advantage is that this language form is often able to have additional functions that are not available from HLIs, for example, automatic formatting.

The final type of natural language execution is a compiler variety. This type reads the source language and then creates a directly executing module that executes every time it is invoked. An advantage of this form is that once created, the resultant run-unit often executes much faster. That is always a strong benefit in production applications. Its one big disadvantage is the length of time initially taken to see run-unit execution results because the run-unit has to go through several levels of translation and intermediate product creation and storage.

Independent of all the modes of execution, the run-unit's languages must be able to direct its output to print files or disk files through commands within the natural language program. Any sophisticated natural language allows the invocation of user-written programs such as COBOL or FORTRAN. Finally, the natural language permits command redefinition for the establishment of a language designed for a specific audience.



### 5.8.3 Run-Unit Development

Since a natural language run-unit is actually an instance of a set of programming language commands, the development environment is typically interactive. In the interactive mode, syntax errors are flagged immediately after a line is entered. Once an error is presented, it should be easily correctable, allowing run-unit development to continue.

Run-units are also able to be created through a text editor. Once the syntax stream is submitted, the syntax errors surface. Because some POL run-units can be several hundred statements long, some POLs allow syntax only processing. Sophisticated POLs attempt to find all the errors they can during any syntax check. This is sometimes difficult. For example, if a literal is supplied but the closing quote mark is not included, the POL might assume that the next line is a continuation of the literal rather than a new command.

In large database environments, there is likely hundreds, maybe thousands, of natural language run-unit instances. Sometimes a database's schema definition changes affect the view used by many natural language run-units. When this happens a fully functional IRDS produces a report identifying the names of the affected POL programs and also automatically invalidates the POL run-units' views.

A critical aspect of a POL is the ability to restrict computer resources required for execution. Resources includes the quantity of view rows selected, the CPU seconds, I/Os, etc.

The next three sections (5.10, 5.11, and 5.12) describe each of the three major types of natural languages: procedure oriented, report writer, and query.

## 5.9 Procedure-oriented Language

While the procedure-oriented language (POL) was originated during the late 1960s as the primary natural language of dynamic relationship DBMSs, and in particular the ILF data model DBMSs, this language type is present today in almost every modern DBMS. Today, however, the POL languages are known by their catchy name: fourth generation languages (4GLs). From a historical perspective, it is important to note that these languages were implemented by DBMS vendors (e.g., Infodata (Inquire), Computer Corporation of America (Model 204), and Mathematica (RAMIS)) in the middle sixties, which is about the same time COBOL (a third generation language) was becoming popular. Thus, to call them *fourth* generation languages is somewhat inaccurate.

Because POL programs are not portable from one DBMS vendor to another, it becomes risky to create whole systems in POL if there is even an intention of changing DBMSs. This risk can be greatly alleviated if an ANSI standard language existed for POLs.

Most POL vendors have recognized the value of their language designs and have invented interfaces between the language and other DBMSs. A de facto standard fourth generation language is FOCUS from Information Builders, Inc., in New York City. FOCUS is both a DBMS and a language that can access data stored by almost every major DBMS, for example, IDMS/R, DB-2, and ADABAS. The most valuable of these extensions is the one to DB-2. That is because FOCUS is then able to interface with any SQL based DBMS. Since



ANSI/SQL is the standard language of interface, FOCUS can interact with any DBMS that can *read* SQL. FOCUS as a DBMS operates on IBM mainframes, DEC's, Wangs, IBM/PCs (XTs, ATs and close compatibles).

With the emergence of the Open Database Connectivity (ODBC), and Java Database Connectivity (JDBC), there is now standard access to DBMS engines through this facility. Thus, it is now safe for an independent POL vendor to create an environment as it is not locked to a particular DBMS. A very high quality POL is Clarion for Windows ([www.Softvelocity.com](http://www.Softvelocity.com)). This environment enables the creation of whole application systems on Windows based PCs with access to DBMSs through ODBC or JDBC.

In general, every POL contains clauses for row selection, sorting, terminal prompting, formatting, and logic branching. Originally, if a DBMS had a POL type of language, its HLI was seldom used, resulting in underdeveloped HLIs. When static relationship DBMSs have POLs, the approach to data access and navigation is similar to that illustrated in Figure 5.25. When dynamic relationship DBMSs have POLs, their approach to data access and navigation is similar to that illustrated in Figure 5.26. The minimum capabilities of POL, as illustrated through examples in FOCUS, that should be included in a POL, static or dynamic are:

- Basic capabilities
- View access
- Table access
- Screen development and utilization
- Dialogue management
- Updating
- Reporting
- Batch job execution
- System control
- Error control

### 5.9.1 Basic Capabilities

A POL is a procedure oriented language. That means that whole programming procedures can be encoded. Such support requires capabilities for logic branching, screen generation, dialogue management, and reading and updating of data from non-database files. POLs permit command echoes, to keep the user aware of a program's progress. Sophisticated POLs allow for both column value and column name arguments. POLs permit graphic output, as well as tabular output. Supported are bar histograms, connected point plots, and scatter diagrams. Definable are labels, scales, headers and footers, and point annotation. Finally, natural language programs can be created, stored, and edited by their users.





**Problem:** print salespersons who sell products manufactured where they were born

**Solution:**

```
10 SELECT SALESPERSON, ON ERROR GOTO 100, AT END GOTO 110
20 GET MEMBER CONTRACT, ON ERROR GOTO 100, AT END GOTO 10
30 GET MEMBER ORDER, ON ERROR 100, AT END GOTO 20
40 GET MEMBER ORDER -ITEMS, ON ERROR GOTO 100, AT END GOTO 30
50 GET OWNER PRODUCT, ON ERROR GOTO 100
60 GET OWNER PRODUCT, ON ERROR GOTO 100
70 IF PRODUCT-MFG- CITY EQ SALESPERSON -BIRTH- CITY AND
    PRODUCT-MFG-STATE EQ SALESPERSON - BIRTH- STATE
    THEN PRINT SALESPERSON- NAME, PRODUCT-NAME,
    SALESPERSON BIRTH-CITY, SALESPERSON-BIRTH-STATE
    ELSE GOTO 40
100 PRINT "ERROR"
110 END
```

**Figure 5.25** Static POL Example



**Problem:** print salespersons who sell products manufactured where they were born

**Solution:**

CONNECT PRODUCT TO ORDER-ITEM

VIA PRODUCT-NBR EQ ORDER-ITEM-PRODUCT-NBR FIND MATCH AND KEEP  
PRODUCT-NAME, ORDER-ID, PRODUCT-MFG-CITY, PRODUCT-MFG-STATE IN  
LIST-1

CONNECT SALESPERSON TO CONTRACT VIA SALESPERSON-NBR EQ  
CONTRACT-SALESPERSON-ID

FIND MATCH AND KEEP SALESPERSON-NAME, CONTRACT-ID, SALESPERSON-  
BIRTH-CITY, SALESPERSON-BIRTH-STATE IN LIST-2

CONNECT LIST-2 TO ORDER VIA CONTRACT-ID OF LIST-2 EQ ORDER-  
CONTRACT-ID

FIND MATCH AND KEEP ORDER-ID, SALES-PERSON-NAME SALESPERSON-  
BIRTH-CITY, SALESPERSON-BIRTH-STATE IN LIST-3

CONNECT LIST-1 TO LIST-3 VIA ORDER-ID OF LIST-1 EQ ORDER-ID OF LIST-3  
AND SALESPERSON-BIRTH-CITY EQ PRODUCT-MFG CITY AND SALESPERSON-  
BIRTH-STATE EQ PRODUCT-MFG-STATE

FIND MATCH AND PRINT SALESPERSON-NAME, PRODUCT-NAME,  
SALESPERSON-BIRTH-CITY, SALESPERSON-BIRTH-STATE

**Figure 5.26** Dynamic POL Example



## **5.9.2 View Access**

View facilities have a definite impact on POL programs even though POL programs typically assume the responsibility for allocating their own working storage space, mapping to DBMS field names, data typing, and the like. If the DBMS does not offer a view facility, the individual POL program must include a full set of navigation commands to traverse the database's structures.

### **5.9.2.1 Run-Unit WHERE Clauses**

WHERE clause facilities impact the utility of a POL. If the WHERE clause is not fully functional, then extra logic must be built into the program for these winnowing operations, and if the POL does not have sufficient selection logic capabilities, the POL cannot be used. That means writing the program in a third generation language, with the consequence of transforming three days of analysis and programming into a one to three month effort. Obviously, a fully functional WHERE clause is a critical component of any POL. The WHERE clause capabilities of the HLI should be equally present in the POL. WHERE clauses are more fully explained in Section 5.8.3.2

### **5.9.2.2 Run-Unit Sort Clauses**

Since the POL is often used as the only programming language of end users, sorting capabilities like those described in the HLI section on sorting (see Section 5.8.3.3) are a very important characteristic of any sophisticated POL, regardless of whether the DBMS's orientation is static, dynamic, or both.

## **5.9.3 Table Access**

If the DBMS does not contain a view facility then there must be navigation facilities present in the POL language. The requirements of such facilities are described in the corresponding HLI sections (see Section 5.8.4) and must be present whether the DBMS is static, dynamic, or both.

## **5.9.4 Screen Development and Utilization**

Screens are almost always a critical requirement of a POL based report or update program. If screens cannot be defined then the POL is not a really useful language. As stated in the earlier section on screens, there is not always a 1:1 mapping between screens and views. Some applications will require multiple screens to process all the data within one view.

Screens should not be tied to just one language type, for example, HLI or POL. The characteristics of a quality screen development facility are provided in Sections 5.5 and 5.8.5.



### 5.9.5 Updating

Updating capabilities of a POL must support single-value changes, multiple-value changes, view row additions and deletions, etc. The updating capabilities in a POL are the same as those described in the HLI section (see Section 5.8.7). A significant difference is that some POLs are designed to accomplish set-at-a-time processing without having to declare and manipulate cursors, while other POLs, e.g., FOCUS' MODIFY language are row-at-a-time processing language, but still without cursors.

### 5.9.6 Reporting

Because the POL is a natural, vendor proprietary language, its capabilities for reporting vary widely, and are all nonstandard. This book illustrates the most widely accepted POL, FOCUS, as a model for report writing capabilities. The examples provided are taken from the PC/FOCUS Users Manual, Release 5.5. Since Information Builders FOCUS procedures can translate into ANSI/SQL compliant access clauses, most every organization can use FOCUS as their de facto standard for POL and remain ANSI/SQL compliant with respect to the interface between FOCUS and any SQL DBMS engine that is used by FOCUS. It is important to emphasize that:

*No report writer facility or any POL of any SQL compliant DBMS is ANSI standard. That is because the ANSI H2 committee only standardizes those aspects of SQL that define, protect, and modify the characteristics of tables and views, and that select and update view instances.*

Any SQL compliant DBMS vendor claiming that their POL is ANSI/SQL compliant:

- Misunderstands the process of standardization
- Does not understand the content of the ANSI standards that have been issued to date
- Presumes the reader is easily fooled

The only POL component that can be compliant is the select clauses.

In general, high quality POLs permit column and row totals, sums, counts, sorting by high to low and reverse, headers and sub-headers, footers and sub-footers, underlining, column headers, line folding, column spacing, control breaks, page breaks, and centering. The output from view column values allows for editing, such as zero suppression, floating dollar signs, and variable date formats. The output allows for multiple column value concatenation and encoded value table look-up.

When operations are performed on multiple dynamic databases, the relational operations of matching and merging should be allowed, as well as navigation between databases.



Reporting capabilities fall into eight categories:

- User prompting
- Statistical operators
- Titles and report headers
- Control break formatting
- Page formatting
- Sorting
- Graphical output
- Derived data

### 5.9.6.1 User Prompting

A critical component of any POL program is user supplied values to program defined arguments that allow prompted, user supplied argument values to be passed to the report clauses. In the FOCUS example contained in Figure 5.27, the user supplied arguments are indicated by the & character preceding the variable name, for example, &WHICH CITY. The - character which precedes some of the commands identifies those commands as part of the FOCUS dialogue manager facility. Briefly, the program's execution is as follows:

- Execution is started at the command just after the label TOP.
- A prompt is provided the user to solicit the value for the city or the value *DONE*.
- If the value *DONE* is provided, control is transferred to the label QUIT, otherwise the next command is executed.
- The TABLE command identifies the file name from which the data is to be acquired. A file in FOCUS's terminology can be synonymous with an SQL view.
- The first operator is WRITE, which with the *SUM* operator indicates that the values of UNIT-SOLD are to be totaled for each unique value of PROD-CODE.
- The WHERE clause is *IF CITY IS &WHICH CITY* and selects only those view rows that pass the WHERE clause select operation.
- The END statement is the last statement in the report request.
- The -RUN command starts the execution of the report request. Once the view rows are selected and the report is printed, control is transferred back to the TOP label.

FOCUS allows any syntactic unit's value to be provided through a variable. Thus, the relational operator *IS* can be supplied with an argument &OPERATOR and become *IS* or *IS NOT*.



```
- TOP

- PROMPT &WHICH CITY. ENTER NAME OF CITY OR DONE.

- IF &WHICH CITY EQ DONE GOTO QUIT;
  TABLE FILE SALES
  WRITE SUM.UNIT_SOLD
  BY PROD_CODE
  IF CITY IS &WHICH CITY
  END

- RUN

- GOTO TOP

- QUIT
```

**Figure 5.27** Example of User-supplied Values from FOCUS

### 5.9.6.2 Statistical Operators

Sophisticated POL provided statistical operators include:

- Count
- Sum
- Distinct
- Average
- Median
- Mode
- Minimum
- Maximum
- Percent
- Standard deviation

Some of these operations are already defined in the ANSI/SQL standard and apply to the data that has been selected for presentation through the view. For example, if an employee has had several salaries throughout the years, the AVERAGE SALARY for employees who are managers is presented through the following:



```
SELECT AVG(SALARY) FROM EMPLOYEE WHERE JOB = MANAGER
```

As another example, the first names of all employees named SMITH are acquired through the use of the query:

```
SELECT DISTINCT(FIRST_NAME) FROM EMPLOYEE WHERE LAST_NAME =  
SMITH
```

But what of the statistical operations not provided by SQL, for example, MODE, MEDIAN, and STANDARD DEVIATION? The FOCUS strategy is to use the statistical operations included in the ANSI/SQL standard as well as those not included. For those included in ANSI/SQL, the FOCUS-to-SQL translation program passes the operation through to the SQL-compliant DBMS. For the statistical operations not performed by the SQL-compliant DBMS, the FOCUS translator executes a select clause that returns the view instances to the FOCUS interface module, which then performs the statistical operation. For example, an operation that computes the percent of employees that each group represents within the data processing department can be expressed as follows:

```
WRITE PCT.CNT.EMP_ID BY GROUP IF DEPARTMENT IS DATA PROCESSING
```

The PCT.CNT.EMP\_ID function initially causes a count of employees by group within the data processing department, and then the PCT function recasts the counts into percentages.

### 5.9.6.3 Titles and Subtitles

Every POL must support sophisticated titling to make the output suitable for managerial use. The minimum set includes:

- Column spacing
- Column totals
- Computer date access for headers or footers
- Computer time access for headers or footers
- Floating dollar signs
- IRDS-provided column headers
- Left and right data justification
- Line folding
- Multiple line column headers
- Multiple line footers
- Multiple line headers
- Multiple view column value concatenation
- Row titles
- Row totals
- Single line column headers



- Single line footers
- Title centering
- Underlining
- Variable date formats
- Zero suppression

#### 5.9.6.4 Control Break Formatting

A control break provides the opportunity to perform certain actions whenever a particular event occurs. Typically the event is a change in value of one of the view columns presented to the report writing feature. If the control break view column is the row's primary key, then when the primary key value changes (every row change), the action occurs. Control breaks can be based on view columns other than the view instance primary key. For example, the EMPLOYEE view rows can be sorted by three different view columns, such as DEPARTMENT, then GROUP, and then JOB TITLE. In such a case, the action occurs as if there are three nested loops: JOB TITLES within GROUP within DEPARTMENT.

The action taken can occur just after the value changes but before the new view instances are presented. In such a case, the action can be to present sums, counts, and the like. If the action is to occur before the new set of view instances is presented, sum and count variables can be reset, page ejects can occur, and new titles can appear.

Regardless of the type of action, it is important that the following minimum set of actions be available:

- Break titles
- Special formatting
- Break totals
- Page totals
- Final column totals

#### 5.9.6.5 Page Formatting

Sophisticated page formatting prevents output reports from being placed first into output files and then loaded into some type of word processor or desk top publishing facility. Page formatting controls

- Page length
- Page width
- Logical pages within physical pages
- Page numbering
- Page titles
- Page footers





### 5.9.6.6 Sorting

While the database's rows may already be sorted (logically, of course) through clauses in the schema definition language, it is complete folly to assume that the schema's specification for row sorting will be sufficient for all users. Consequently, upon selection, the view rows must be able to be sorted by one or more view columns in either a low to high or a high to low fashion. Without sorting the whole concept of control breaks is lost.

### 5.9.6.7 Graphical Output

Clearly some reports are best presented graphically rather than through columns of numbers. The most popular types of graphical outputs produced are:

- Pie charts
- Bar histograms
- Connected point plots
- Graphs, and
- Scatter plots

Supporting these graphical outputs must be facilities for headers, footers, column headers, graph labels, scales, and point annotation. Figure 5.28 illustrates the command language necessary to produce graphical output FOCUS on a CAR database.

The data definition language for the CAR database is contained in Figure 5.29, and a logical Bachman diagram illustrating its structure is in Figure 5.30. The pie chart produces a number of *slices* equal to the number of CAR companies, and the size of each slice is equal to the average retail cost. The actual pie chart obtained through a screen capture utility appears in Figure 5.31.

```
GRAPH FILE CAR  
  
ON GRAPH SET PIE ON  
  
SUM AVE. RETAIL_COST  
  
ACROSS CAR  
  
ON GRAPH SET HIST OFF  
  
END
```

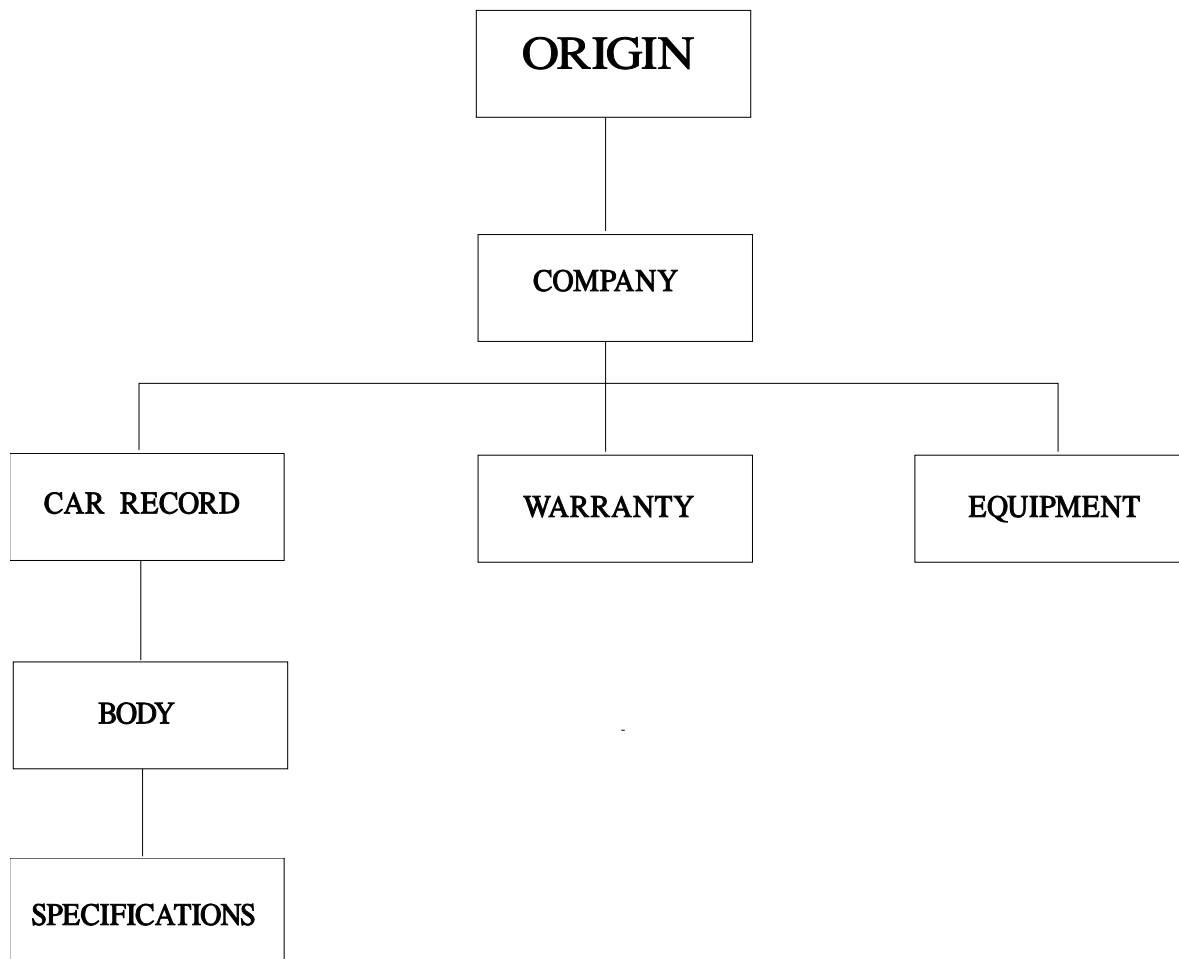
**Figure 5.28** FOCUS Graphics Command Language for a CAR Pie Chart



```
FILENAME=CAR, SUFFIX=FOC
SEGNAME=ORIGIN, SEGTYPE=S1
  FIELDNAME=COUNTRY, COUNTRY, A10, $
SEGNAME=COMP, SEGTYPE=S1, PARENT=ORIGIN
  FIELDNAME=CAR, CARS, A16, $
SEGNAME=CARREC, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=MODEL, MODEL, A24, $
SEGNAME=BODY, SEGTYPE=S1, PARENT=CARREC
  FIELDNAME=BODYTYPE, TYPE, A12, $
  FIELDNAME=SEATS, SEATS, I3, $
  FIELDNAME=DEALER_COST, DCOST, D7, $
  FIELDNAME=RETAIL_COST, RCOST, D7, $
  FIELDNAME=SALES, UNITS, I6, $
SEGNAME=SPECS, SEGTYPE=U, PARENT=BODY
  FIELDNAME=LENGTH, LEN, D5, $
  FIELDNAME=WIDTH, WIDTH, D5, $
  FIELDNAME=HEIGHT, HEIGHT, D5, $
  FIELDNAME=WEIGHT, WEIGHT, D6, $
  FIELDNAME=WHEELBASE, BASE, D6.1, $
  FIELDNAME=FUEL_CAP, FUEL, D6.1, $
  FIELDNAME=BHP, POWER, D6, $
  FIELDNAME=RPM, RPM, I5, $
  FIELDNAME=MPG, MILES, D6, $
  FIELDNAME=ACCEL, SECONDS, D6, $
SEGNAME=WARRANT, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=WARRANTY, WARR, A40, $
SEGNAME=EQUIP, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=STANDARD, EQUIP, A40, $
```

**Figure 5.29** CAR Database Master-File Description (DDL)

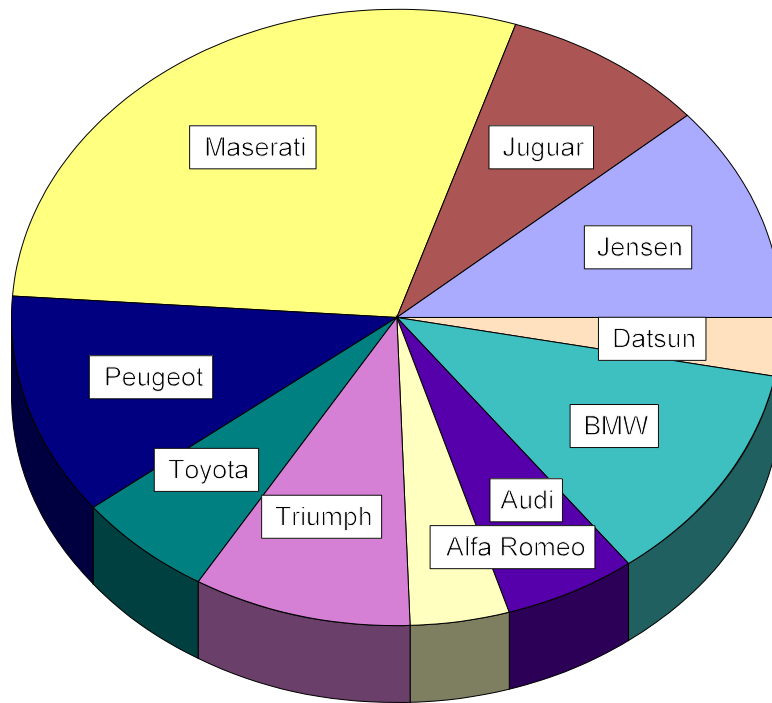




**Figure 5.30.** Car database logical data model diagram.



Preference of Cars in Area



**Figure 5.31.** Pie chart graphic for subset of data from the cars database.



### 5.9.6.8 Derived Data

Derived data is critical. Derived data takes on many different forms. Among these are:

- The use of table look-ups, for example, MD for Maryland
- The use of a DBMS provided sequence number, for example, as a LINE-NUMBER in a financial report
- The computation of new values from view column values in the view row, for example, deriving REMAINING-CAPITAL-BALANCE by subtracting CAPITAL-PAYMENTS-TO-DATE from ORIGINAL-AMOUNT
- The computation of the new value from a view column and a constant, for example, deriving AGE by subtracting BIRTH-DATE from the system constant CURRENT-DATE
- The computation of a summary value from values in multiple view rows, for example, deriving CAPITAL-PAYMENTS-TO-DATE by summing all CAPITAL-PAYMENTS

In addition to generating these new values, it is important to be able to place them where appropriate in a report, for example, as an output column with its own title and control breaks. Another critical issue in the use of derived data is sorting, for example, sorting LOAN view rows by REMAINING-CAPITAL-BALANCE. In FOCUS, the POL (the TABLE and MODIFY languages) must have the REMAINING-CAPITAL-BALANCE data field defined and computed on the selected view rows prior to using it as a sort field.

### 5.9.7 Batch Job Execution

Once a POL run unit is created and tested, it should be executable in both interactive and batch modes. Run-unit considerations include:

- Argument values
- Output report direction
- Output report formats
- Reuse of run-unit outputs
- Use of non-DBMS files

Argument values are critical to the development of general purpose POL run-units. In the interactive mode, these argument values are requested from the terminal user. For example,



SELECT CARS WHERE MAKE EQ &MAKE *ENTER THE MAKE OF THE  
AUTOMOBILE*

causes the quoted literal to appear on the screen so the user knows what to enter. In a batch environment, such data has to be provided by a specially created file, and the identity of the file has to be known to the POL run-unit. Such identity is often provided by a DATA statement that appears either at the start or the end of the procedure. For example:

DATA FILE IS <mydata>

Argument values are not just restricted to variable values. Operators, such as GE, LT, SPANS, etc., can be placed into arguments to make the POL run-unit more general purpose, for example:

SELECT CARS WHERE MAKE &OPER *PLEASE ENTER THE OPERATOR* &YEAR  
*AND ALSO PLEASE ENTER THE YEAR OF THE AUTOMOBILE*

Output destinations are another consideration in the effective use of POL run-units. In the interactive mode, the POL's output destination usually defaults to the input terminal. In the batch mode, the default might be the system printer, which is usually known to the system through file name conventions. If destinations other than defaults are desired, the POL should have the ability to specify names and devices. For example, FOCUS provides the ability to specify the destination of an output report through a SAVE command placed at the end of the procedure. For example,

SAVE on <filename> . . .

A number of languages also provide control over saved data formats. In FOCUS, if the command REPORT IS OFFLINE is provided at the start of the procedure, then the output is formatted to fit on a 66 line page rather than a 26 line screen. Additionally if the SAVE statement has a *FORMAT IS WP*, then the data is saved so that it can be used by a word processor. Other formats supported by FOCUS include the DIF format with a choice of value delimiters. Data in this format can be fed into other packages that can import delimited data. Harvard Project Manager, for example, is a PC based project management package that supports this type of data importing. A database project's manager might for example, use PC/FOCUS for the staff's work product and time reporting, and then at the end of each week use a FOCUS procedure to generate an ASCII delimited file for input. Harvard can produce CPM and Gantt charts for the project.

Another batch processing consideration is that the POL can output data from within one run-unit, that can be immediately used during the execution of another run-unit against the same or different database.

A critical facility in any POL is the ability to manipulate rows from non-database files and invoke automatic IRDS editing and validation. Accepted row can be written to a database, to reports, back to the file structure, or to other non-database file structures. With this facility, the POL supports nonDBMS files, as well as DBMS databases.



## 5.9.8 System Control

The system control capabilities normally available through a POL include:

- Audit trails for all updates
- DBMS message processing
- Backup and recovery
- Concurrent operations
- Multiple database processing
- Security and privacy
- Reorganization

The capabilities of each of these facilities are generally as described in previous sections on system control (see Section 5.7) and the HLI subsection addressing system control subsection (see Section 5.8.9) except as provided hereafter.

### 5.9.8.1 Message Processing

POLs usually have a fully functional message facility that is available through a program function key, for example, PF1.

Messages that are run-unit specific and intended for user response can be encoded right into the run-unit's language. Figure 5.32 illustrates a part of a POL program from a sports application that displays a message when there is an error situation. The run-unit has read the FAMILY-ID from an external file, and attempts to retrieve that family's row from the database. If the family's identifier is not present, then the error message *Illegal Family Id* is written to an error file. These messages are reviewed after the batch of updates have executed and the erroneous data is corrected and resubmitted.

```
CASE FAMILY
```

```
    MATCH FAMILYID
```

```
    ON NOMATCH TYPE ON ERRFILE "ILLEGAL FAMILY ID"
```

```
    ON NOMATCH GOTO BIGERROR
```

```
    ON MATCH GOTO PLAYER
```

```
ENDCASE
```

**Figure 5.32** POL Run-unit Application Message



### **5.9.8.2 Backup and Recovery**

Most often, backup and recovery is a function performed by the DBA. Some aspects of backup and recovery may be allowed to certain HLI users, but seldom is the control over backup and recovery allowed to POL users. If backup and recovery features are available through POL, they are usually automatic. If during the execution of a POL run-unit the database fails, the DBMS usually begins an automatic recovery cycle that restores the database to the last committed transaction of each active run-unit.

In DBMSs that have sophisticated view facilities, transaction rollback and some aspects of backup and recovery may be encoded in the view facility logic to control the interaction between the run-unit and the DBMS.

### **5.9.9 Reorganization**

Common to many POLs is some database reorganization capabilities. Typically available is the ability to create an entirely new table including columns, index indicators, etc. Once created, the POL run-unit causes the creation of an O/S file for selected rows from the database. This store persists beyond the end of the run-unit's execution.

This kind of capability is especially useful for DBMS users that view POL as their only programming language. With the ANSI/SQL manipulation facilities a user can define, delete, and modify the attributes of any table. Such capabilities are subject to security controls that may have been imposed by a DBA and thus may not be available to all users.

### **5.9.10 POL Summary**

A brief review of the capabilities contained in a POL quickly leads to the conclusion POLs are complete programming languages that often eliminate the need for other types of database access languages. For many years, users of FOCUS, Model 204, Nomad, and RAMIS DBMSs never used any other types of languages. These users designed, programmed, loaded, reported, and updated complete database applications with POLs.

### **5.10 Report Writer**

A report writer is usually the first language developed after the HLI by static relationship DBMS vendors. The majority of static relationship DBMS vendors have only lately developed POLs (4th generation languages). Except for the fact that report writers do not have database updating commands, the capabilities of POLs and report writers are very similar. Thus, references are made in this section back to the POL section (Section 5.9) for detailed explanations of most of the report writer's facilities.

In addition to the similarity between POLs and report writers, there is also a similarity between report writers of both static and dynamic relationship DBMSs, with the exception of





row selection and navigation strategies. Finally, report writers are often used as prototyping mechanisms for HLI programs.

Clearly the most popular report writer is Crystal Reports. This report writer operates through ODBC and thus it is generally independent of any particular DBMS.

The typical capabilities found in report writers include report titles, column titles, row titles, multiple levels of breaks, the ability to skip lines and pages, page headers and footers, output data editing such as floating dollar signs, selection clauses, page counting, logical and physical dimensions for report pages, left and right justification for alphanumeric data, setting and resetting of program variables and multiple report copies.

Sometimes, report writing capabilities do not exist as a completely separate language, but as an extension of one of the other natural languages. The execution alternatives also vary from interpretive, to direct, to compiler oriented.

Because report writer languages are not portable from one DBMS vendor to another, it becomes risky to create large quantities of reports intended to be large critical components of systems. Use of a vendor proprietary report writer is justified if the report formats change frequently, or if they are short lived. These risks would be eliminated if there was an ANSI standard language for report writers. There is, however, no committee working on such an effort.

The fundamental difference between static and dynamic report writers centers not on capabilities, such as footers, headers, and break totals, but on the fundamental nature of the DBMS. In a static relationship DBMS, the report writer output must mirror the basic design of the database through its already defined relationships among the tables. In a dynamic relationship DBMS, report writer output can mirror any relationship that can be expressed between two union compatible columns from different tables. Figure 5.33 contains an example of a report writer language program. It does not show the view used, and assumes that all database variables are automatically defined. The only variables defined are those *local* to the report writer program.

The following topics are applicable to report writers:

- Basic Capabilities
- View Access
- Table Access
- Reporting
- System Control
- Error Control



**COMPOSE:**

```
FOR REPORT SALES,  
PHYSICAL PAGE IS 55 BY 55:  
DECLARE SUM1 = RCOUNT OF ORDER-ID:  
DECLARE SUM2 = RCOUNT OF CONTRACT - ID:  
SELECT RECORD IF ORDER OCCURS:  
ORDER BY NAME OF CUSTOMER-ID, CONTRACT-ID, ORDER-ID:  
FOR CUSTOMER, SKIP TO NEW PAGE, SKIP 2 LINES,  
AT END, PRINT (15) $TOTAL NUMBER OF CONTRACTS$, R (45) SUM2:  
FOR CONTRACT, SKIP 2 LINES, COMPUTE SUM2,  
    PRINT (15) $CONTRACT-ID$, (30)$DATE-SIGNED$, (45) $MAXIMUM-  
COST$,  
AT END, SKIP 1 LINE, PRINT (15)$TOTAL ORDERS$, R(28) SUM 1,  
    (40) $TOTAL VALUE$, R (55) TOTAL-ORDER-COST.  
FOR ORDER, AT END, COMPUTE SUM1  
END REPORT:
```

GENERATE ACTIVITY WHERE CUSTOMER-ID EXISTS:

**Figure 5.33** SYSTEM 2000 Example of Report Writer Program



### 5.10.1 Basic Capabilities

The basic capabilities of the report writer are similar to that of the POL except the report writer does not allow updating to the database. To accomplish sophisticated reports the report writer must support:

- Logic branching
- Looping
- Reading non-database files
- On-line reporting
- Off-line reporting

### 5.10.2 View Access

WHERE clause facilities have a definite impact on the requirements of any report writer program for the same reasons as with the POL. If there is not a sophisticated WHERE clause, the report will likely have to be written in COBOL. The previous sections on view facilities and WHERE clauses are discussed in Sections 5.8.9 and 5.8.10.

### 5.10.3 Reporting

Screens are seldom a critical requirement of a report writer as reports are generally destined for printing on a computer's high speed printers. If, however, a report is to be on-line, then the screen to which the printing is formatted might be defined by the DBMS's screen facility. It follows then that a necessary component of any screen driven report writer feature is controls for dialogue management. These are usually restricted to page up and down, and possibly side to side. The capabilities that must be present in screens and dialogue management are presented in earlier sections of this chapter.

The report formatting capability of a report writer falls into eight categories:

- User Prompting
- Statistical Operators
- Titles and Reporting
- Control Break Formatting
- Page Formatting
- Sorting
- Graphical Output
- Derived Data



All these capabilities are described in the corresponding section of the POL language (see Section 5.9.6, and its subsections).

#### **5.10.4 System Control**

The capabilities normally available through interrogation language programs include:

- DBMS message processing
- Multiple database processing
- Security and privacy

Since report writers are not able to update databases, most of the system control capabilities are not even of concern. Thus there is no need to be concerned about:

- Audit trails for all updates
- Backup and recovery
- Concurrent operations
- Reorganization

The capabilities of each of the relevant system control facilities are generally as described in the earlier section on system control in this chapter.

##### **5.10.4.1 Message Processing**

Message processing in a report writing environment is primarily concerned with run-unit development. The only other place messages appear is during report execution when some aspect of the data cannot be found, for example, selecting a row on the basis of a primary key when the row is not present in the database. The messages are also generally not interactive. Rather, the report is executed, and then all possible messages are made available.

##### **5.10.4.2 Multiple Database Processing**

Multiple database processing is important in report writing because every organization is likely to have multiple DBMSs and also multiple databases. Sophisticated report writers can access multiple databases from the same DBMS. Given the wide scale acceptance of ANSI standard SQL, there are now report writers that can access both multiple databases from the same DBMS vendor and also databases from multiple DBMS vendors.



### 5.10.4.3 Security and Privacy

The only relevant aspects of security and privacy are those that affect views and view column instances values. If the security is not granted, then while a view may be permitted to be referenced in a report writer run-unit, it may be disallowed when first accessed. In addition to allowing/disallowing access to views, the security may also be used to further permit screening of view instances on the basis of view column values. For example, the security facility may prohibit access to all EMPLOYEE view instances that are managers or that have salary above certain levels.

### 5.10.5 Report Writer Summary

Report writers are a critical component of any database environment. Typically, they exist either alone or as extensions to either POLs or query-update languages. Most static relationship DBMS vendors have stand alone report writers. Most dynamic relationship DBMS vendors have report writers as extensions to either POL or the query-update languages. Today, sophisticated report writers are being created by both DBMS vendors and by independent software vendors. These report writers can access data from multiple databases and from different DBMS vendors' databases. As this trend continues, and if one vendor's report writer becomes overwhelmingly popular, there may be a de facto standard for report writers just as FOCUS is the de facto standard for 4GLs (POL).

## 5.11 Query-update Languages

A query-update language interrogation program is typically constructed as a single sentence, while the POL and the report writer language interrogation may typically contain from 50 to 150 lines. The query normally consists of three types of clauses: output, sorting, and data selection. The output clause usually contains titles and a list of the columns to be printed. The sort clause specifies the fields for sorting, the sort order, and whether the sorting is to be high to low or reverse. The selection clauses specify the criteria for row selection, which may include Boolean operations, relational operators, and the like. A typical query-update sentence might be:

```
PRINT <column-name> [, <column-name>,. . . ]  
  
[ SORTED BY <column-names>,. . . ]  
  
[ WHERE <column-name> <relational-operator> <value>  
  
[ <Boolean operator>  
  
<column-name> <relational-operator> <value>. . . ] ]
```



Query-update languages often contain less sophisticated formatting capabilities than report writers. Query-update languages also offer processing logic less complex than POLs. Query-update languages do not compete with POLs or report writer languages. Rather, they offer the ability to do something simple in a simple way. As a consequence, query-update languages are very popular with the ad hoc interrogator who obtains a simple listing of data from one or more tables or who performs a single update.

The typical capabilities present in a query-update language include tabular reporting with titles, sorting by one or more columns, arithmetic functions such as MIN, MAX, and AVG, results of arithmetic formulas, and selection clauses that include arithmetic, relational, and Boolean operations.

Again, the fundamental difference between static and dynamic query-update languages centers not on capabilities, but on the fundamental static or dynamic nature of the DBMS. In a static relationship DBMS, query-update languages produce reports that conform to the structure dictates of the database. In a dynamic relationship DBMS, query-update languages can produce reports from different tables that can be related through union compatible columns. Figure 5.34 contains three examples of SYSTEM 2000 query-update programs.

WHERE clauses are the most critical component of a query-update language as there are no other facilities available for row selection or relationship navigation that are available in an HLI or a POL.

Query-update languages are understood through:

- Basic capabilities
- View access
- Table access
- Updating
- Reporting
- System control
- Error control

```
LIST/TITLE  L(5) CUSTOMER-ID,  L(15) CUSTOMER-NAME,
            L(40) CONTRACT-ID, R (50) TOTAL-ORDER-COST,
            ORDERED BY CUSTOMER-ID, CONTRACT-ID
            WHERE CONTRACT-ID EXISTS:

ASSIGN SALES-PERSON-ID EQ 123-45-6789
            WHERE CONTRACT-ID EQ 76498:

REMOVE CONTRACT FROM WHERE CONTRACT-ID EQ 76498:
```

**Figure 5.34** SYSTEM 2000 Example of Query-Update Language Program



### 5.11.1 Basic Capabilities

Because of the very nature of query-update languages, that is, single sentence orientations, their basic capabilities are typically restricted to being able to:

- Read non-database files
- Read and update view rows
- Report and update on line
- Generate off-line reports

### 5.11.2 View Access

Since the view access capabilities needed for query-update languages are the same as those for all natural languages, the sections earlier in this chapter address these.

### 5.11.3 Updating

Updating is a very important component of the query-update language. It is critical that the updating facilities be more sophisticated than those in either the HLI or the POL because the query-update language run-unit's orientation is single sentence oriented, and does not contain the language facilities for looping. To be sophisticated, a query-update facility must be able to

- Allow for prompted user-supplied arguments in the update clauses.
- Add, modify, or delete a view instance(s) that maps to a single instance from a table instance.
- Add, modify, or delete a view instance(s) that maps to a multiple instance from a table instance.

Very sophisticated query-update languages can also:

- Add, modify, or delete a view instance that maps to a single instance from multiple tables.
- Add, modify, or delete a view instance that maps to multiple instances from multiple table instances.

Without the first three update capabilities, the query-update language can be used only for simple reports, and for single table instance adds, deletes, or modifies.



#### 5.11.4 Reporting

The report formatting capability in a query update language falls into eight categories:

- User prompting
- Statistical operators
- Titles and report headers
- Control break formatting
- Page formatting
- Sorting
- Graphical output
- Derived data

These capabilities are drawn from the same set as described in the section on POL. The only differences are usually related to the degree of sophistication allowed. As the query-update language adds sophistication, the *single sentence* language construct becomes stretched until the language becomes either a POL or a report writer. Neither IBI's FOCUS nor Computer Corporation's Model 204 contains separate and distinct query-update languages. Rather, they each contain a POL that can be used in a single sentence fashion to accomplish simple reports.

#### 5.11.5 System Control

The system control capabilities generally supported by the query-update language are the following:

- Audit trails for all updates
- DBMS message processing
- Security and privacy
- Reorganization

The capabilities of each of these facilities are generally as described in the Sections 5.8.9 and 5.9.8.

### 5.12 Choosing the Right Interrogation Language

Generally, the choice of an interrogation language should follow this sequence:

- Attempt to develop the report/query with the least amount of programming effort. That means using either the query-update language, report writer, or POL. What is being developed is a prototype of the task rather than a production report.





- Demonstrate the result to the person who requested the report/query to determine whether it satisfies the needs. If not, change it until it does.
- When the report or query is acceptable, determine the frequency of operations and estimate the amount of data to be processed, the expected amount of processing time, and the amount of required system control support. Then choose a final programming language for the interrogation that satisfies the combined requirements--assuming that satisfying them is possible.
- Leave the prototype report or query in place for use by the requestor--if its performance is tolerable--until its replacement can be created.

The following benefits are derived from this process:

- The prototype is created with minimal human resources.
- The design is refined with minimal resources.
- The final product is created only after the design has been validated.

### 5.13 Interrogation Summary

No DBMS can report data it cannot model. Thus to compare the overall reporting capabilities of a static relationship DBMS interrogation language to those contained in a dynamic relationship DBMS to determine which DBMS is better is like comparing the turning radius of a bus to that of a sports car in order to determine which vehicle is better. The bus has one purpose and the sports car has another. Neither the turning radius of the vehicle nor the capabilities of a DBMS's interrogation language alone is sufficient to determine which vehicle or DBMS is better.

The languages available in a particular DBMS can only mirror those functions the DBMS performs in an acceptable manner. No vendor is going to issue a language that will constantly *break the DBMS's back*. Thus, the fundamental difference between static and dynamic interrogation languages centers not on capabilities, but on fundamental DBMS design. In a static relationship DBMS, these languages must mirror the basic design of the database. In a dynamic relationship DBMS, these languages define relationships that allow any combination of tables to be used in the interrogation.

A well-engineered DBMS contains multiple database interface languages. Although there is a growing tendency to rely on natural languages, the compiler interface languages, such as COBOL and FORTRAN, are important for data loading and data update, for more complex multiple database processing, and for building DBMS-independent applications.

The point of having multiple languages is to be able to choose the most appropriate language for the job. If a DBMS has only an HLI or a POL, then its vendor cannot be too interested in providing a helpful environment to its users.

A final aspect of interrogation is interfaces between DBMSs. Because of ANSI/SQL it is now cost effective to access a high-speed production database, say run under IDMS/R, extract data and load the extract into a Model 204 database for very high speed ad hoc queries.



Figure 5.35 compares and contrasts the different types of interrogation languages available from static and dynamic relationship DBMSs. Figure 5.1 compares and contrasts the relative work efforts for developing the same result through each language type.

LANGUAGE	STATIC	DYNAMIC
Most Language	Well developed, good function and facilities	Poor to acceptable development
Procedure Oriented Language	None at all	Well developed, good function and facilities
Query Update Languages	Under developed usually no update Constrained by database structure	If developed then done well
Report Writers	Used only as a short cut to HLI Still constrained by database structure	If developed, Usually an Extension of POL or QUL

**Figure 5.35** Interrogation Static and Dynamic Relationship DBMS Comparison



## 6

# SYSTEM CONTROL

### 6.1 System Control Components

System control identifies facilities provide for the protection of the database, the smooth operation of applications that use the database and the DBMS, and the effective use of the DBMS.

A fully functional DBMS's system control facilities include the following:

- Audit trails
- Message processing
- Backup and recovery
- Reorganization
- Concurrent operations
- Multiple database processing
- Security and privacy
- DBMS installation and maintenance
- Application optimization

The DBMS's audit trails capture update transactions according to criteria such as user, table, date, time, and program. These transactions are selectively reportable, and available during backup and recovery operations.

Message processing facilities provide sophisticated on-line help for investigating the cause of a user, database, or DBMS error.

Backup and recovery facilities permit unaffected users to continue work while backups are taken of production databases and to undergo minimum interruptions while the DBMS recovers a database from a DBMS, user, or computing environment induced error.

Reorganization involves both logical changes to the database's column and relationship types, and physical reorganization of the database's storage structure components.

Concurrent operations let a single run-time copy of the DBMS support multiple concurrent update and retrieval transactions from multiple interrogation language run-units to the same or different databases without any compromise to database integrity, and with automatic detection and resolution of execution deadlock.

Multiple database DBMS allows more than one database to operate independently under a single central version. This allows logical database changes to be isolated and applied without affecting any of the other databases that may be running under that central version. Furthermore, if one database crashes, then only that database is affected, and while it is being recovered, the others remain operating.

Security and privacy support the definition of a comprehensive set of users, profiles, and passwords to prevent and report on illicit data access and database operations. The DBMS security provides protection at the database, table, and column level for at least update and retrieval access, and ideally also for select clauses.



DBMS installation and maintenance facilities, if sophisticated, enable the generation of special run-time versions that favor certain types of update or retrieval processing.

Finally, application optimization is the process of using DBMS generated statistics of its own database application access efficiencies to guide physical and logical reorganization, or to create special DBMS versions. Sophisticated DBMSs provide performance assessment aids that enable the physical database designers to tune the performance towards critical applications.

The principal difference between static and dynamic system control relates mainly to the domain of each system control facility. In a dynamic relationship DBMS, the domain is normally a table because a table is usually stored on one O/S file. In a static relationship DBMS, the domain is usually the database, as it is a large collection of interrelated tables stored in a complex way across a series of O/S files. What this usually implies for a dynamic relationship DBMS is that audit trails, reorganization, and concurrent operations are isolated in their effect to the table. For a static relationship DBMS, these same system control facilities affect the large, complex database as a whole. This means that with a dynamic relationship DBMS, normally just a table is locked during reorganization. With a static relationship DBMS, normally the whole database is locked during its reorganization. In short, the difference relates to both scope and size. Figure 6.1 provides an overview of the differences between system control facilities in static and dynamic relationship DBMSs.

System control facilities range from a series of DBMS vendor supplied software capabilities, manual processes, activities, and lessons learned through classes and experience. Collectively, these facilities have to be enhanced with well- engineered and established procedures that are centrally controlled within a database administration department/group. These activities have to be centrally administered and controlled because sometimes their invocation can occur only after the entire database environment is shut down. This is especially the case in a corporate database that is under the control of a static relationship DBMS.

While the activities within system control are not normally a monolithic set of utilities accessible by a large menu, many of these functions do exist in menu driven mechanisms that are easily used by the database administrator. This ease of use is both good and bad, especially in a dynamic environment where the database administrator is the same person who designs databases, loads data, updates data, and is *chief cook and bottle washer*. It is good in that this person really controls all the DBMS facilities. It is bad because all these facilities are so easily--and irrevocably--controlled.



SYSTEM CONTROL FACILITY	DBMS TYPE	
	STATIC	DYNAMIC
Audit Trails	Poor to Good	Poor to Good
Message Processing	Poor to Good	Acceptable
Backup and Recovery	Good	Good
Reorganization	Acceptable	Acceptable to Good
Security and Privacy	Poor to Acceptable	Poor to Acceptable
Multi-Database Processing	Poor to Good	Good
Concurrent Operations	Acceptable to Good	Good
Application Optimization	Acceptable to Good	Poor to Acceptable
Installation and Maintenance	Poor to Acceptable	Acceptable to Good

**Figure 6.1** Static versus Dynamic System Control Comparison

Before proceeding, the following must be stated about what a database and DBMS environment is like without these nine critically important facilities.

- Not having audit trails means that there is no accountability for database updates, reports, and other types of use.
- Not having sophisticated message processing will not stop errors. Rather users will become frustrated, and stop using the system.
- Not having backup and recovery results in total database loss as important update transactions are lost and critical databases are not recoverable after crashes.
- Not having reorganization ultimately causes the database to become antiquated and *freeze-up*. This is due to structural inefficiencies or because the database does not contain the right columns, relationships, and tables.
- Not having security and privacy enables valuable corporate data to be stolen.
- Not having multiple database processing causes a redundancy of facts, which in turn causes errors through time, and an eventual breakdown in the usefulness of database projects.



- Not having concurrent operations capabilities and guidelines will--at some time--prevent the generation of an extremely important report because of some trivial, but database locking, operation.
- Not having database application optimization causes the expenditures for hardware to increase as a consequence of unimproved database designs, inefficient physical structures, and inappropriate DBMS configurations.
- Not having good installation and maintenance procedures eventually causes special software libraries to be lost, different versions of modules to be linked together, and databases to be lost due to corruptions caused by mis-connected DBMS modules.

Although time consuming and seemingly nonproductive, the only thing more expensive than good system control is not having system control.

## 6.2 Audit Trails

An audit trail is a time-ordered and user-sequenced record of operations that have been executed against the database. Without an effective and complete audit trail, there may be no way to determine the source of a destructive update.

Audit trails are used to determine the source of an update, rather than fixing a damaged database. Since the audit trail capability can operate effectively from different run units and in different modes, such as batch, interactive, and exclusive or shared use of one or more databases, the audit trail file(s) represent a complete chronology of the activities against the database.

A sophisticated audit trail contains enough information to identify correctly the source of the update. Typically, each audit trail transaction contains:

- Time and date stamps
- Database identification
- User-identification
- The update transaction itself

Under the assumption that the audit trail tables themselves are stored in a database, the various DBMS languages isolate a single user's transactions from within a multi-user environment, or report the activities by transaction type, database, table, time of day, and the like. Figure 6.2 illustrates the typical table format for an audit trail file. As seen from this table, all the information should be present to completely reconstruct the environment of the update.

With sophisticated audit trails, transactions can be from

- All the databases operating under a central version
- Certain dates or time periods
- Specifically identified databases



- Specifically identified areas within a database
- Specifically identified schema tables within a database
- Specifically identified views within a database
- Specifically identified interrogation language types, e.g., query-update, or procedure oriented languages
- A named application program
- A class of application programs

Date =   yyyymmddd   (julian format)
Time =   hh:  mm:  ss:  100
Submitter Id
Database Id
DBMS Version Id
Database Data Cycle Id
Actual Command
Table Name
Before Image
After Image
etc.

**Figure 6.2** Transaction Content

Audit trails are stored either on tape or disk. Each has advantages and disadvantages. Tape is safer in that it almost never crashes. However, a tape drive is needed for each central version of the DBMS. Another disadvantage is that when the tape runs out, another must be started and/or mounted. During that time, all updates under the control of that central version have to be suspended, locking out the users.

Disk is less safe because it is subject to head crashes. Over the last several years, however, disks have become almost as safe as tape. Thus, when audit trails are properly established, there is no difference.

Under either tape or disk, the writes must be direct, not buffered. This means that only one transaction is lost in the event of a DBMS, operating system, or hardware crash.

A sophisticated audit trail is used to follow the data value changes for the purpose of isolating either an incorrect data value that is added to the database, or to identify the source of a valid but inappropriate value. Given that the audit trail rows are in a DBMS central version database, then the DBMS typically supports standard reports of audit trail rows on the basis of



- User-Id
- Date
- Database
- Table
- Run-Unit-Id
- Language Type
- Error Message Type

Natural languages must also support ad hoc reports from audit trail logs.

Before learning about any particular DBMS's audit trail capabilities and declaring them to be either good or bad, a careful assessment of exact needs for database integrity, the type of update auditing, various legal considerations, etc., has to be made. Armed with these requirements, a careful examination of DBMS capabilities will be productive.

There is no real difference in audit trails between static or dynamic relationship DBMS. Audit trails within a DBMS are either sophisticated or they are not. Static relationship, DBMSs however, have been around for a longer time, and are likely to have a better developed approach to audit trails. Other than for evolutionary differences between static and dynamic relationship DBMSs, there are no real differences. Figure 6.3 tabulates these natural differences.

AUDIT TRAIL ASPECT	DBMS TYPE	
	STATIC	DYNAMIC
Database Definition	Many interconnected tables	Single table per database
Database Sources	Well developed	Acceptable
Transaction Content	Poor	Poor
Transaction Storage Media	Ok	Ok
Reporting	Ok	Ok

**Figure 6.3** Static versus Dynamic Audit Trail Differences





### 6.3 Message Processing

In a database application there are many sources of messages, from the DBMS, the O/S, various systems software, and the application. The sources of these messages must be determined, and there should be a central scheme for their definition, maintenance, and issuance. A well-organized DBMS has all its messages categorized and summarized in an easy to read and understand language. Five typical types of messages are:

- Syntax errors of various DBMS languages
- Application generated errors such as incorrect column names
- Data generated errors such as incorrect employee Ids
- Database integrity errors that indicate that a database storage structure component is missing or damaged
- DBMS integrity errors such as missing DBMS subroutines, bad code, etc.

Figures 6.4 thorough 6.8 illustrate typical message types and a probable cause for each. These messages are intended to invoke some action. Typically, these actions require

- Nothing when the message is informational
- Reentry of the data or command when the DBMS doesn't understand
- User termination from the run-unit or DBMS when a password is not acceptably provided
- Detaching all users from the database when damage has occurred to a storage structure component
- Termination of the operation of the DBMS due to a missing subroutine or a logic error



**Example:**

MODIFLY EMPLOYEE-NAME WHERE

<<syntax error>>

**Source:**

DDL language

interrogation languages

system control languages

**Figure 6.4** Typical Message Types: Type = 1: DBMS Syntax Errors

**Example:**

PRINT EMPLOYEE-MAME WHERE.....

<<no valid field employee-mame>>

**Sources:**

DDL languages

Duplicate field name

Duplicate schema/subschema

Interrogation languages

Illogical request

Untrue selection clause

System Control language

Illegal password

**Figure 6.5** Typical Message Types (cont.): Type = 2 Application Generated



**Example:**

MODIFY EMPLOYEE-NAME EQ 'TED CODD' WHERE  
EMPLOYEE-ID EQ 485927143

<<illegal employee-name value>>

**Sources:**

Data loading

Records out of sequence

Illegal data value

Interrogation

Insert records/columns

Modify records/columns

**Figure 6.6** Typical Message Types (cont.) : Type = 3 Application Generated

**Example:**

Open sales (mike)

<<sales database damaged>>

**Sources:**

Damaged pointers

Unreadable dictionary,

index,

relationships, or

data

Missing storage structure components

**Figure 6.7** Typical Message Types (cont.): Type = 4: Database Integrity



**Example:**

START DBMS

<<unrecoverable error 123>>>

**Sources:**

Missing subroutines

Time bomb expired

Trapped logic error

**Figure 6.8** Typical Message Types (cont.): Type = 5: DBMS Integrity

Figures 6.9 through 6.13 illustrate a message from each severity type and an example cause of the message.

There is no real difference in the types, kind, and result of messages between static and dynamic relationship DBMSs, except that if a single dynamic database is damaged, others will most likely continue to process, and thus only a few users are affected. While in a static environment, the termination of a single database stops the activities of large groups of people, such as a nationwide network of order entry clerks.

Figure 6.14 presents these message types and severity levels in a matrix. Each cell indicates the probable effects of the message type and severity on a transaction, depending on whether the executing DBMS is operating in single user mode (local version) or multiple user mode (central version).

It is important that sophisticated DBMSs contain an on-line message database that is available during any one of the on-line sessions. The table description of such a messages database is illustrated in Figure 6.15.

**Example:**

DELETE EMPLOYEE WHERE SSN EQ '184-86-7787'

<<employee deleted>>

**Figure 6.9** Severity Levels: Severity = 1: Information Only



**Example:**

PRINT <EMPLOYEE RECORD> IF SSN EQ '34-134-8618'

<<no records found>>

**Figure 6.10** Severity Levels (cont.): Severity = 2: Correctable

**Example:**

MODIFY SALARY = 100000 IF EMPLOYEE - NAME EQ 'ME'

<<update authority not allowed>>

<<run - unit terminated>>

**Sources:**

Request for inactive database

Request for an already exclusively controlled function

Security violation

**Figure 6.11** Severity Levels (cont.): Severity = 3: Run Unit Fatal



**Example:**

A read to index that has bad track  
<<database damage>>  
<<run-unit terminated>>  
<<no database access allowed>>

**Sources:**

I/O errors  
Memory failure  
DBMS bug

**Figure 6.12** Severity Levels (cont.): Severity = 4: Database Fatal

**Example:**

An O/S read to a DBMS subroutine that is not present  
<<session terminated>>  
<<O/S prompt for next command>>

**Sources:**

Malformed load of DBMS software  
Hardware disk failure  
O/S loader error

**Figure 6.13** Severity Levels (cont.): Severity = 5: DBMS Fatal



MESSAGE TYPE AND SOURCE		MESSAGE SEVERITY				
		1 INFO	2 CORRECTABLE	3 RU FATAL	4 DB FATAL	5 DBMS FATAL
1	Syntax Error	N/A	Local CV	Local	N/A	N/A
2	Application	N/A	Local CV	Local N/A	N/A	N/A
3	Data	N/A	Local CV	Local	N/A	N/A
4	DB Integrity	N/A	N/A	N/A	Local CV	N/A
5	DBMS integrity	N/A	N/A	Local N/A	Local CV	Local CV

Legend:

local = local mode (single user)

CV = multiple users from batch or on-line

N/A = not applicable

**Figure 6.14** Cross Reference Between Message Types and Severity Levels

<b>Access keys</b>  member number  short name  long name
--

**Figure 6.15** Messages Database Table Format



Finally, it is important that certain classes and types of messages be logged to the audit trail. If this is accomplished, then user training can occur as a consequence of reviewing the message logs. The message classes that should be logged are those above class 2, and the message severities that should be logged are those above severity level 3.

Figure 6.16 illustrates the format of the message audit trail table. This table should be a fundamental component of data dictionary system's database, and the rows in this database are used to assist in understanding problem areas in a database application design. If there is a large number of data entry editing and validation violations, or a large number of query language syntax error entries, then there is a need for remedial training.

message number
short name
long name
message explanation
suggested remedies

**Figure 6.16** Audit Trail Columns For a Help Transaction

## 6.4 Backup and Recovery

Backup and recovery embraces two different concepts. Backup is the ability of the DBMS to make a copy of one or more databases that are under the control of an executing DBMS. Recovery is the ability of a DBMS to recover from a failure that may be user, DBMS, O/S, or hardware caused. Since some types of recovery involve database backups, both topics are often presented together.

### 6.4.1 Database Backup

Prerequisite to any safe backup and recovery facility is the ability to backup the database. That means the DBMS, through a command, copies all the files associated with all the storage structure components of a database from an on-line media to an off-line media. The backup facility should also have a similar facility to bring back the database copy from off-line to on-line, under the same database name, or under a different name. The ability to change a database name is important because it enables a duplicate database for activities, such as database comparisons and training.

The practicality of making a database backup is dependent upon the sophistication of the DBMS's storage structure, and the ability to operate multiple databases within a central version.





If there are many different databases within a central version, then backing up any one database is practical. If, however, the DBMS only allows a single database within a central version, then it is quite impractical to backup a multiple billion character database.

In a simple database and DBMS environment, backup is simple. On a micro-computer there might be a relational DBMS and under its control a single database of only 20,000 rows from all types, totaling only about 2 million characters. To backup that database might require only a single command and a few minutes. However, in a large mainframe environment when the database application involves hundreds of users, a hundred or so tables, and millions of rows, the problem of backups becomes more complicated and time consuming. Such a database might be 4 billion characters in size and take four hours to backup

In large database applications, backup needs to be a great deal more sophisticated than just locking up the database and sending all the employees home while database backup is underway. It should be possible to backup a database while only retrievals are occurring. It should even be possible to backup a portion of the database while update activity is occurring on other portions. Incremental backups should also be possible, that is, only backing up the tables and/or O/S files that have been updated since the last backup. This type of backup is popular on large mainframe environments and even on PCs. Database is inherently more complicated, however, because there are storage structure components for the schema and subschema/view, the indexes, relationships (static only), and the rows. Because of these complexities, the following questions need to be asked of any DBMS vendor before arriving at a complete understanding of the DBMS' backup facility:

- Can a database schema table be updated while a backup of that same schema table is in progress?
- Is there a DBMS facility to prevent updates while a backup is in progress?
- Can an O/S file be updated while a backup of that same O/S file is in progress?
- Can a database be updated while a backup of that same database is in progress?
- Does the DBMS support incremental backup of only those schema tables that were updated since the last backup?
- Does the DBMS have the ability to report information on the incremental backups?
- Does the DBMS contain parameters to maintain the number of backups that are to be stored?



- Does the central version need to be off-line when a backup of one or more databases is in progress?
- Does the DBMS include a backup copy utility?
- Does the DBMS have the ability to report information on the backup copies?

### 6.4.2 Database Recovery

Database errors occur mainly during update. An update may not be completed successfully because there has been a computer hardware failure, or because of a problem in the system software, the DBMS, or the application. In addition to these types of failures, there may have been updates that are mistakes (for example, a whole group of customers may have been assigned to the wrong salesperson's territory). Under any of these situations the database may have to be moved *back* in time to a state in which the errors did not occur. Recovery capabilities are usually a combination of manual and DBMS-based procedures performed by the database administrator. The transactions that the DBMS recovery processor uses are often stored on a disk or tape file called the journal.

Critical to recovery are the capabilities provided by the DBMS for on-line and batch users, and the role a central database administration function may play during certain database damage situations. For example, the DBMS should have the capability to invoke transaction rollback to purge an unwanted update while a run-unit is in operation. This capability should be available from either the host or natural language environment.

There are five distinct DBMS alternatives that are typically available to bring a database back to a consistent state:

- Rerun the job
- Roll-forward from a checkpoint
- Rollback the database from a current state to a prior checkpoint
- A combination of rollback and roll-forward
- Transaction rollback instigated by the run-unit

There are two expense components in any database recovery. The first is the resources consumed by the DBMS in anticipation of a database crash. The second is the resources consumed during the actual recovery process. In general, these two factors are inversely related to each other. That is, the more resources expended during the update operations in preparation for database recovery, the quicker and less expensive the recovery. And, the fewer resources expended during update operations in preparation for database recovery, the longer and more expensive the recovery. Figure 6.17 identifies six classes of failures, and the types of recovery that are possible under three scenarios: immediate, rapid, and slow.



CAUSE OF FAILURE	RECOVERY METHOD		
	IMMEDIATE	RAPID	SLOW
<b>Run Unit Exception</b>	Not impossible except for transient failures (full redundancy)	Run-unit rollback	Run-unit roll-back or roll-forward and restart
<b>O/S or DBMS Failure</b>	Not possible except for transient failures (full redundancy)	System roll-back; restart from checkpoints	System roll-back or roll-forward and restart or rerun
<b>Failure to accept CPU instruction</b>	Hardware Redundancy	SAME AS ABOVE	SAME AS ABOVE
<b>Database Unreadable</b>	Dual files	Dual Files	SAME AS ABOVE
<b>Corruption detected by DBA</b>	IMPOSSIBLE		Log analysis: DBA takes manual action
<b>Corruption detected by user</b>			Analysis of run log and results correction of applications . Rurun

**Figure 6.17.** Recovery methods vs causes of failures.

Choosing the most cost effective method of recovery involves choosing the level of protection that is both the maximum affordable and the minimum necessary. Basically that means balancing the cost of database unavailability against the cost of database recovery. For example, if the application is an on-line order entry system, then for every minute the order-taking database is not available there is going to be the unrecoverable cost of lost orders. The real cost of recovery is the actual cost of recovery minus the value of the lost orders. In contrast, if a batch job is established to update the database, and the batch job has a whole weekend to run, then it does not really matter if the job is finished Friday night, Saturday afternoon, or Sunday morning. In this case, almost any expense incurred for rapid recovery is wasted. For either case, the cost of recovery is essentially insurance premiums paid against a certain loss. Only the organization can determine the appropriate premium cost.



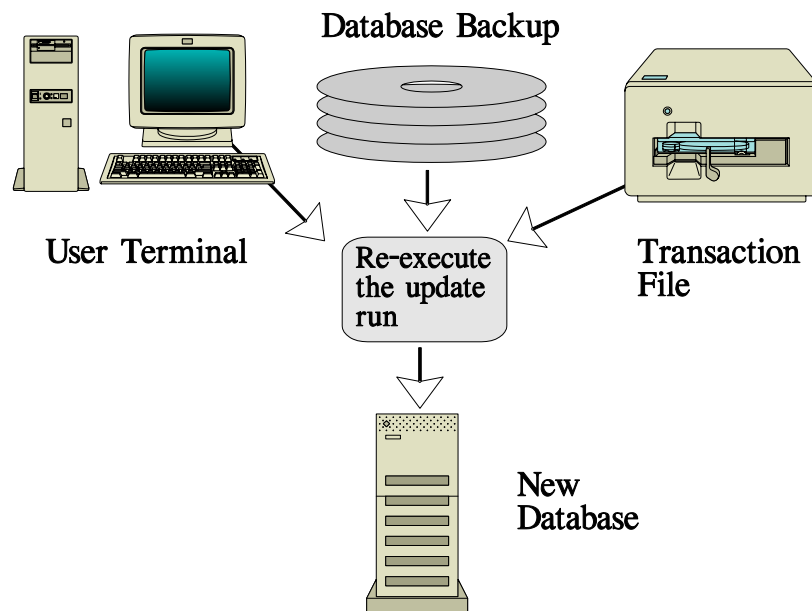
### 6.4.2.1 Re-run the Job

The re-run the job method of recovery has the least preparation cost and is thus the longest and most expensive to execute. For example, if a database backup is taken just prior to the execution of a job, and if the job crashes 99% from the beginning, then all the work is scrapped, the backup restored, and the database job rerun. The cost to prepare for this recovery is only the cost of the backup. The cost to execute the recovery is the cost of rerunning all the update transactions that are present in the job.

This recovery method is practical when the probability of a hardware crash is pretty low, and when there is a very large volume of updates against a good portion of the database. This typically occurs in overnight batch update runs. Figure 6.18 illustrates the basic process of rerunning a database job as a method of recovering after a crash.

### 6.4.2.2 Roll-forward

The roll-forward method of recovery comes in two variations. They are both similar in that they initially require a backup of the database, and write the update transactions to a journal file during the update run. In the event of a crash, the backup is restored, and the journal file containing a log of the update transactions is run against the database until the last successful



**Figure 6.18.** Single User Recovery: Rerun the job.

update as been reapplied.

In the first variation, the journal file contains a form of the database update command. Something like: UPDATE FILE 5, PAGE 10, WORD 47 TO VALUE 88. The journal file contains a whole series of these Spartan commands, and reapplies them very rapidly. The space



required for each command is quite small, so the journal file may contain a large number. Each file and each DBMS row has to be located and brought into memory, the value changed, and the DBMS row written back to disk.

In the second variation, the journal file contains the actual image of the database DBMS row that is written to disk. The journal file contains a whole series of these DBMS rows, and reapplies them even more rapidly than the first variation. This is because the only requirement of the recovery is to write these DBMS rows directly on top of the corresponding DBMS rows of the database.

Both roll-forward variations cost more than the re-run-the-job method of recovery, but less than the rollback method (next section). Both roll-forward variations take less time for recovery than the re-run the job method, but more than the rollback method.

The roll-forward recovery method is most appropriately used during single or multiple update runs against the same database, providing

- Those jobs have coordinated restart in the event of a crash while they were executing.
- The amount of time to recover to the point of job restart is not very critical.

Creating such programs requires careful design and programming, and then careful testing before they are put into place. Figure 6.19 illustrates the basic process of accomplishing database recovery through roll-forward.

### 6.4.2.3 Rollback

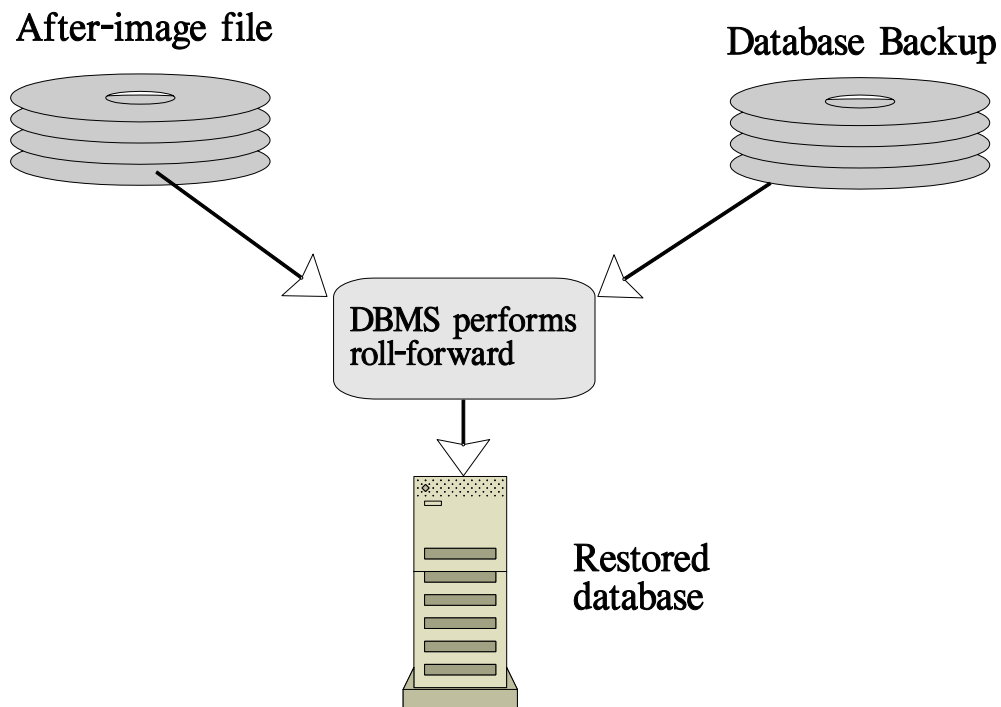
The rollback recovery method departs in philosophy from re-run the job and roll-forward in that it does not require a database backup, only the current database. As the updates occur, the before image of the database DBMS row (instead of the after image DBMS row) is written to the journal file. This enables recovery from the point of the crash back into time to a position in the database normally known as a checkpoint. A checkpoint is a marked point, or an instant when no updates are occurring to the database. The DBMS marks that instant as a point of database consistency, and uses that point of consistency as the *ending* place for a rollback. Figure 6.20 illustrates the basic process of accomplishing database recovery through rollback.

All update jobs running at the time of the crash also have to be rolled back to that checkpoint, and then be restarted. If a user has entered ten or twenty update transactions between the last checkpoint and the time of the database crash, then all those updates have to be reentered. For batch users, such program restarts, if properly programmed, cause all journal files to be reset and then reprocessed. Thus, careful design and programming are necessary.

A real problem arises whenever a job has started and finished between the last checkpoint and the time of the crash. A mechanism has to be created to identify those completed jobs from a job log, and to automatically re-execute those jobs.

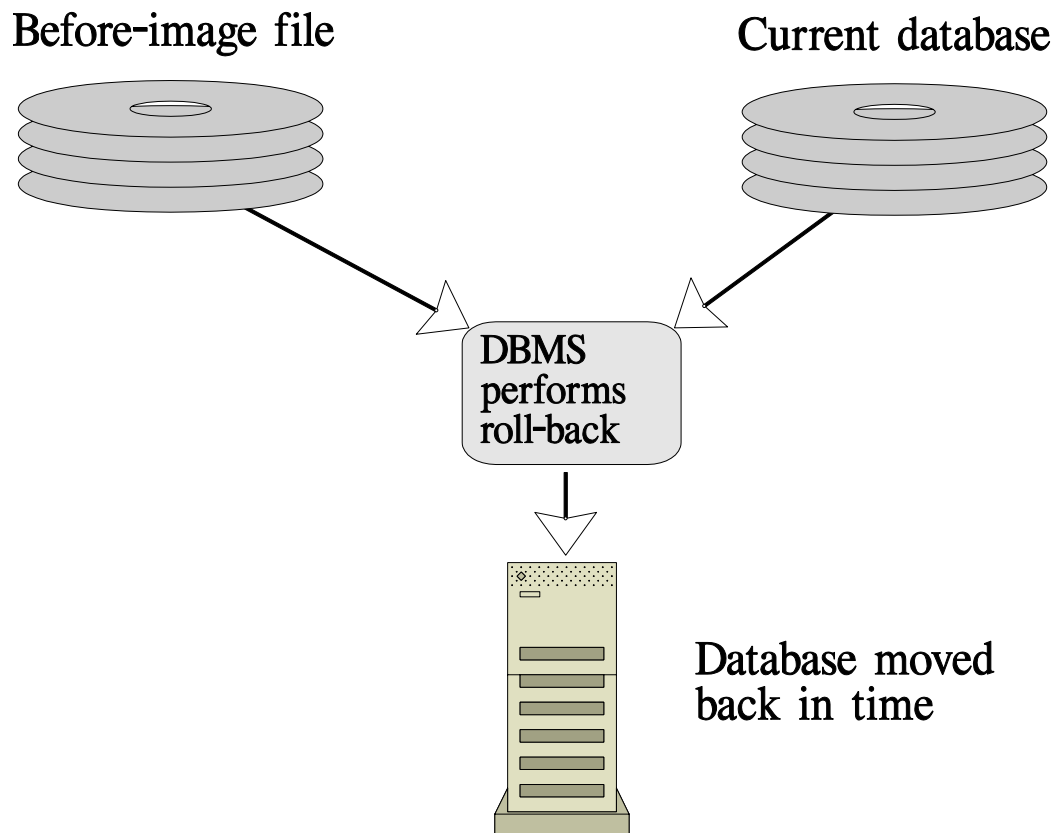


The cost of preparing for rollback recovery is similar to the amount spent on roll-forward, and depending on how far it is to the previous checkpoint, the recovery can be faster or slower.



**Figure 6.19.** Single-user recovery: Roll-forward (after image recovery)





**Figure 6.20.** Single user recovery: Rollback or before image form.



#### 6.4.2.4 Rollback with Roll-forward

The rollback with roll-forward recovery mechanism starts recovery at the point of the crash and moves backwards in time to the most recent checkpoint. The roll-forward method then recovers from the last checkpoint to the last successful transaction so that all jobs running at the time of the crash are recovered. The cost of implementing this capability is the most expensive because the journal file log must contain both the before images and the after images of database DBMS rows that have changed.

The most appropriate use of this recovery method is on-line applications that cannot tolerate long recovery periods. Even though the total cost of recovery is high, the cost of preparation plus the cost of recovery, the cost is spread over each transaction.

This method also has the advantage that transactions which occur earlier than the last checkpoint are moved off-line because they are not needed for recovery. This is in contrast to roll-forward recovery which requires all the after images from the point of the last database backup to the point of the crash.

Most modern on-line database update environments that use multiple databases under a central version with updates from different sources and languages make use of the combination rollback and roll-forward methods of recovery. During the time of such recovery, the database is usually locked out. Figure 6.21 illustrates this lockout and types of recovery.

Another very important reason for having multiple database operations within a single DBMS central version is recovery. If one of the databases within a central version crashes, then that database undergoes recovery without impacting any of the other databases that are still operating under the control of that central version. If there is only one database within a central version, then when it crashes, everyone operating within its domain also crashes.

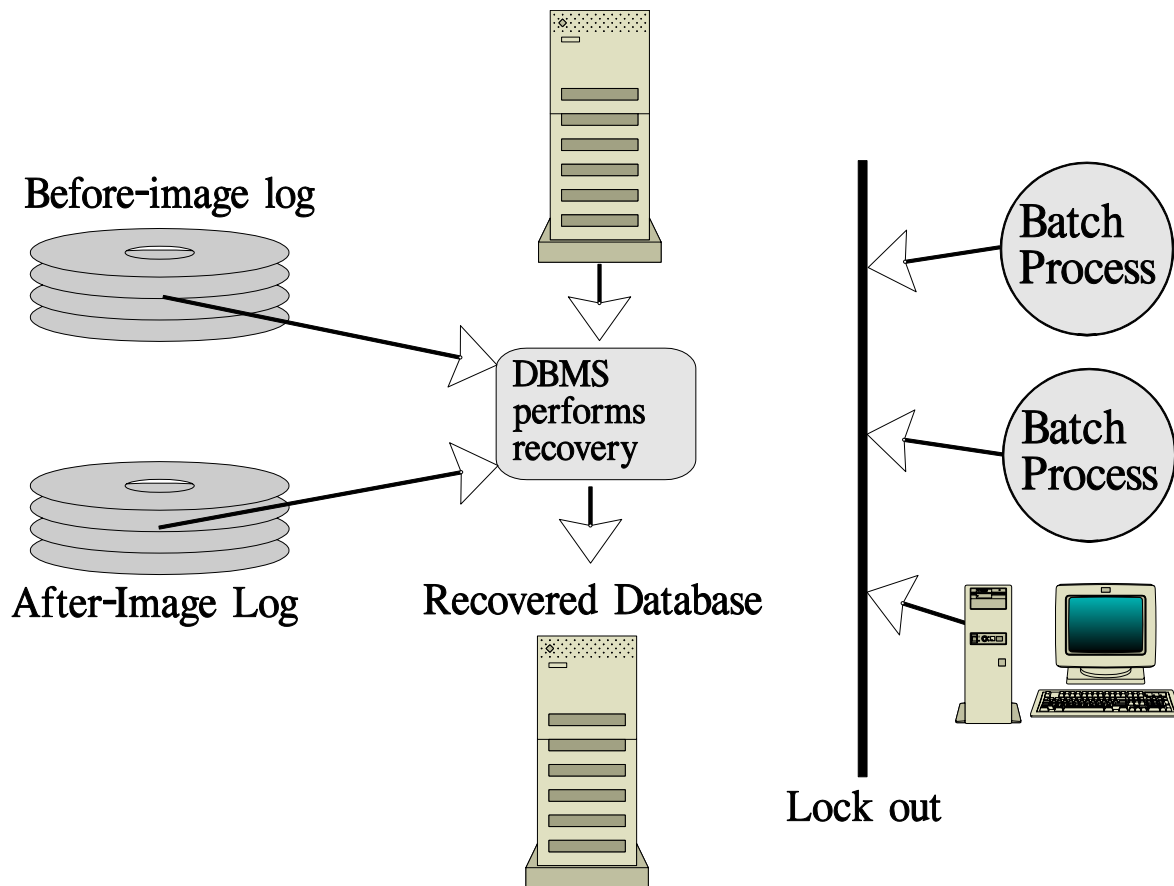
#### 6.4.2.5 Transaction Rollback

If during an update, the user determines that the transaction is really not valid, a facility called transaction rollback lets the user reverse the transaction. It simply cancels the effect of the transaction on the database. There are two varieties of transactions: explicit and implicit. Figure 6.22 illustrates both these transaction types. When the transaction is explicit, then the run-unit issues a <START FRAME> command to the DBMS. The run-unit then issues one or more updates, and if acceptable, the run-unit issues a <COMMIT> transaction. If, however, the run-unit issues a <ROLLBACK> transaction, the entire set of DBMS transactions is rolled back.

If the run unit is interrupted by a software or hardware failure, then upon recovery, the DBMS backs out all uncompleted transactions. The three versions of rollback are summarized in Figure 6.23.

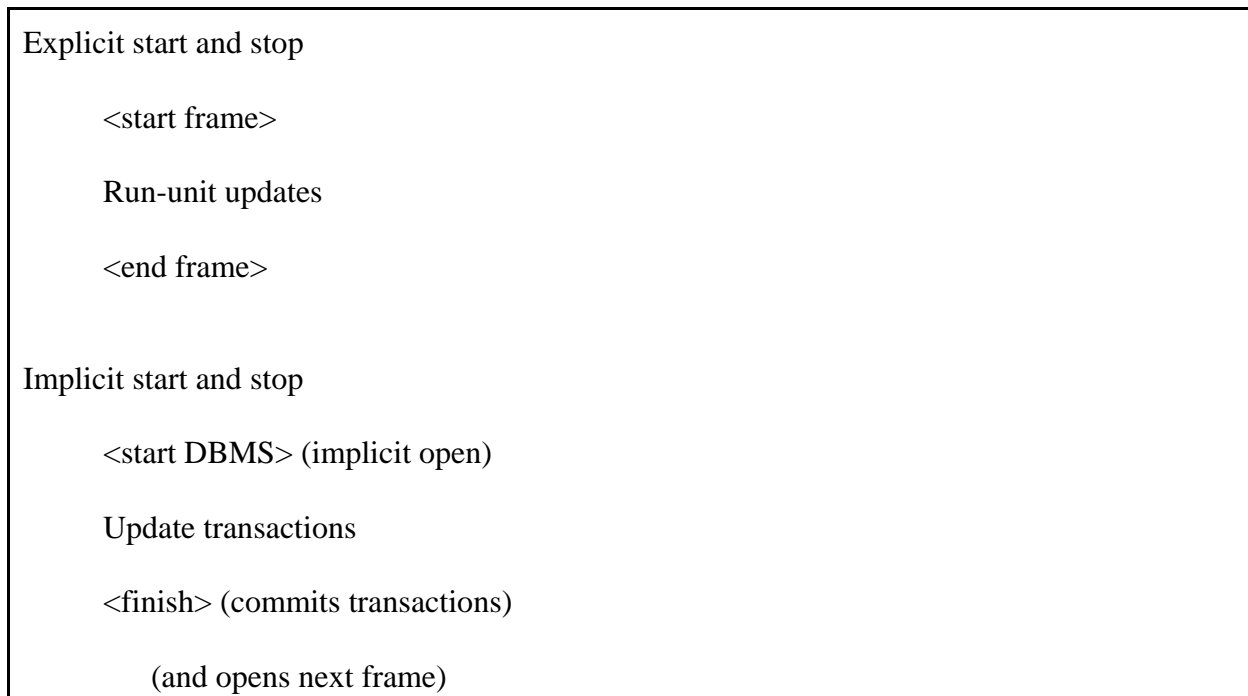




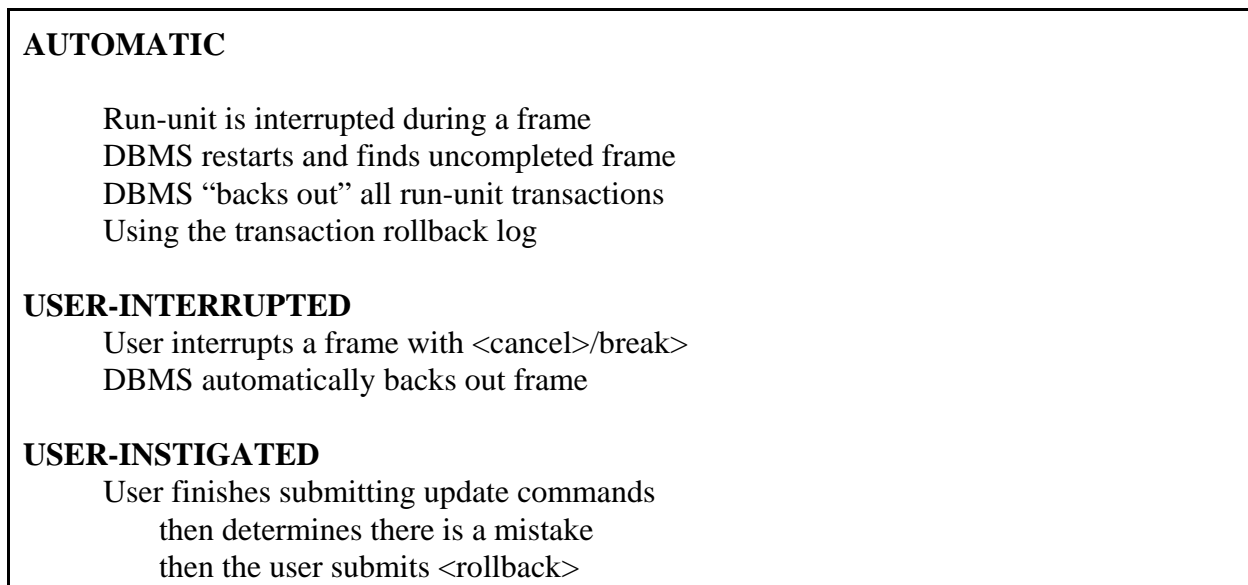


**Figure 6.21.** Multi-user recovery.





**Figure 6.22** Transaction Rollback Alternatives

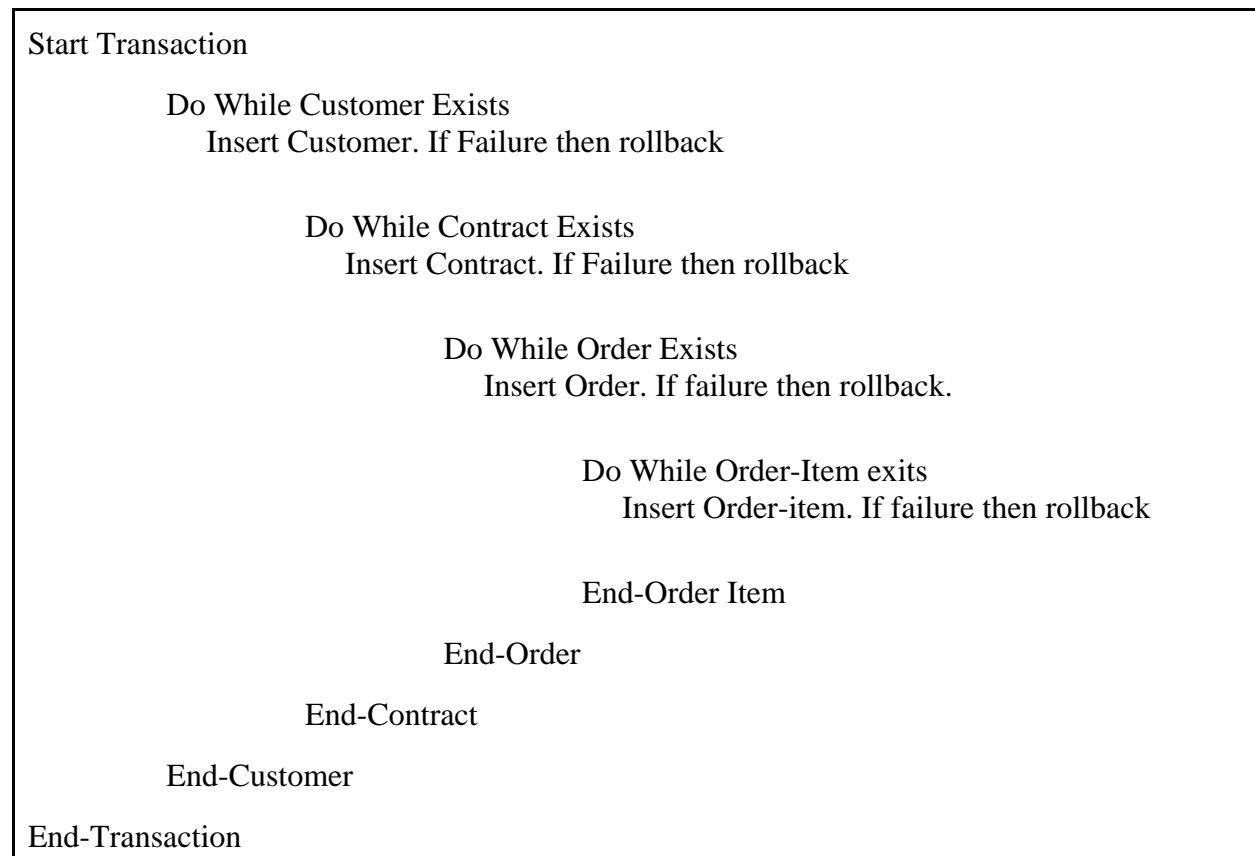


**Figure 6.23** Transaction Rollback Alternatives (cont.)



The reason transaction frames are needed is that a business transaction seldom maps to a run-unit transaction. The pseudo code illustrated in Figure 6.24 illustrates this fact. From the user's perspective, the transaction is ADD CUSTOMER. From the run-unit's perspective, the steps within the transaction are

- Insert the customer row
- Insert all the available contracts for the customer
- Insert for each contract each of the orders
- Insert for each order each of the order-line-items



**Figure 6.24.** Business Transaction: Add Customer consisting of many run-unit transactions.



The rule established for the user transaction is to add all or add nothing. The pseudo code reflects this with the inclusion of the explicit START and END TRANSACTION statements, and with the inclusion of the ROLLBACK statement at every subcommand within the user transaction.

From the DBMS's perspective, there are at least as many DBMS transactions as there are run-unit transactions. There are additional DBMS transactions for each index and/or relationship update, and for each recording of a before and/or after image update.

Since each type of business transaction is likely to contain a different number of run-unit and DBMS transactions, the business transaction must be carefully crafted to ensure that it is rolled back in its entirety rather than just some of its contained run-unit transactions.

If a transaction rollback facility exists, then what happens to other transactions that might have used its results for their updates? Are they rolled back? What of the user who has used the updated data for a report, and after the report is finished, a whole day's transactions are rolled back? These types of questions have troubled DBMS designers to no end.

The solutions seem to be partially user designed and implemented, and partially DBMS implemented. From the user point of view, each update requests exclusive control over the row or table (all rows) or database (the entire storage structure) that is being updated, thus ensuring that no other person interferes with the update. And when the person or program that submits the update is satisfied, the update's effects are committed to the database and then all the resources that were locked are released.

From the DBMS point of view, it imposes single threaded operation for all updates to the same database, and does not allow any other updates to occur until the submitter of the update has committed it.

#### 6.4.2.6 Backup and Recovery Responsibility

Figure 6.25 identifies the different processing states through which an online transaction proceeds. As is seen, most states are the responsibility of other processors. Distributed processing complicates this chain even further. Distributed processing introduces additional processor state interactions between states 4 and 5, and between states 7 and 8. The complete set of these interactions is being defined by the ANSI H2.1 subcommittee for remote data access protocol (RDAP). The H2.1 committee interfaces with the ISO committee for RDAP which is also a subgroup of WG3.

Because of this complex and shared responsibility, it is critical to know which processor is in charge of a transaction in the event of a system failure to determine which transactions have to be re-entered. While that knowledge is clearly available after considerable research has been performed, it is more practical to research the data contained in the database after the environment has been recovered to determine whether transactions arrived there, and if not, to reprocess them.



STATE	ACTION	RESPONSIBLE
1.	User to TP monitor	User
2.	TP monitor response to user	TP
3.	Activation of TP processing module	TP
4.	TP processing module to user program	TP
5.	User program request to DBMS	O/S
6.	DBMS to database	DBMS
7.	DBMS response to user program	O/S
8.	User program to TP processing module	TP
9.	TP processing module response to user	TP

**Figure 6.25** Processor States Responsibilities in On-Line Transactions

### 6.4.3 Checkpoint and Restart

A checkpoint transaction is a special transaction that is placed on the journal file for performing database recoveries. The checkpoint transaction indicates that once a database is restored to that transaction, the database is consistent. A checkpoint transaction may be instigated by the DBA, or automatically once an hour. When the checkpoint transaction starts, it usually requests an update lock on the entire database and does not allow new updates to commence. When the exclusive lock is obtained, no other updates are in progress. At that moment, all buffers are cleared to disk and the checkpoint transaction is written to the journal, signaling that the database state is consistent. The checkpoint transaction generally requires specialized information to be sent to the journals so that the entire database environment is acceptably restored as of that checkpoint.

Checkpoints are needed for recovery methods that involve roll-forward. The opportunities for establishing checkpoints include

- The start of the first update job when it is bound to the environment
- The end of the last update job, when binding is terminated
- A DBMS data manipulation language commit point that may be implicitly or explicitly programmed into a run-unit
- A data manipulation language abort activity (rollback) that may have occurred within a job



- A data manipulation language *ready area* command (CODASYL)
- A journal file initialization
- During DBMS restart or recovery
- Job rollbacks that return the database to a consistent state (i.e., before the <start of job>)

#### 6.4.4 Database Lockout

Database lockout is the DBMS process that precludes the execution of conflicting operations. In general, recovery instigates lockout by preventing the start of any new update run units. Eventually all the currently running update jobs come to an end, and the database is then locked from all updates. In some cases, retrieval jobs are also locked out from execution. For example, if the reason for the lockout is to add a new column to a table then the DBMS might require that table and all its instances be locked out from all update or retrieval activity.

Static relationship DBMSs typically lock a larger portion of a database's storage structure than does a dynamic relationship DBMS. This is because the static relationship DBMS database is so much more complex. A static relationship database locks at least one O/S file, and if the file contains rows from many different tables, then all those different tables are also locked.

If the static relationship DBMS allows the instances from a single table to be spread across multiple O/S files then the lockout necessarily spans multiple O/S files.

In addition to locking the rows, any index structures associated with the table's columns are also locked out, if the indexes are built in a sophisticated way. For example, the index types described in Figure 6.10 that involve either DBKEYs or bit maps also have to be locked during any reorganization of rows as those addresses (a DBKEY or a bit map address) are also affected. The index organizations that only involve the use of primary keys in the multiple occurrence lists are untouched by any such reorganization.

A dynamic relationship DBMS, in contrast, typically only locks a single table and its associated instances as they alone may be contained in a single O/S file. Indexes associated with that single table are or are not locked depending upon the level of sophistication of the index designs.

Static relationship DBMSs have traditionally locked more of the storage structure than dynamic relationship DBMSs because their databases are used--most often--for corporate wide, large, complex databases. In recent years, these same use characteristics (corporate wide, large, and complex) are being imposed on dynamic relationship DBMSs. To acceptably fulfill these demands, the dynamic relationship DBMS vendors are making their database storage structures more complex. In short, they are starting down the same road traversed long ago by the static relationship DBMS vendors. At the end of that road will be broader locking domains and increased database reorganization times. That is the natural consequence of deploying sophisticated storage structures to achieve greater application performances.



A critical issue is the coordination of the recovery of a multiple database environment. Such an environment exists within the operation of a single DBMS instance or across instances of the same or different DBMS vendors in the case of a distributed environment. If the database is more simply structured in such environments, that is, if there is only one table to the O/S file, a minimum of indexes, and the indexes are all simply structured, then the required locking during any reorganization is minimized. However, those characteristics are typically the inverse of the characteristics of a fast running, corporate wide, complex database environment.

#### **6.4.5 Differences between Static and Dynamic Relationship DBMSs**

Generally, a database implemented through a static relationship DBMS is more complex than a dynamic database. It is likely to have more tables (60 to 90 versus 10 to 15). It is likely to have larger quantities of data (billions versus millions of characters) and have many more programs (thousands versus hundreds) and more concurrent users (thousands versus hundreds) doing both updates and retrievals. Because of the multitude of complexities, the static relationship DBMS is likely to have more sophisticated backup and recovery that costs more per each update, and also takes longer to recover since the volume of transactions is greater. The recovery, however, is likely to be more reliable as recovery applies to the entire database of complex tables.

In a dynamic environment, the DBMS is likely to restrict recovery to individual tables, so it is important to design both the tables and the updates to be as independent as possible. That is, one update, one table; or two updates, two different tables; or two updates, two rows of the same type. Figure 6.26 tabulates these important differences.

#### **6.4.6 Disabling the Journal File**

For certain databases operating within an executing central version of the DBMS, there may be a need to disable the journal file. This capability, if carefully used, leads to efficient processing of large volumes of transactions for which backup and recovery are more expensive than rerunning the batch job. If this capability exists, but only at the central version level, then the capability is not acceptable. It must exist at a level below that of the central version. That is, at the level of one or more databases within a central version, and preferably, at one or more separable subunits within a database, for example a CODASYL area or a table that is restricted to just one O/S file. In either case, if indexes are present, there must be a utility to regenerate them after the updates are completed.

To make this capability safe, database locks must be installed on the portion of the database that is not be recovered through normal recovery mechanisms. This exclusive area of the database can be updated through the batch run, and when the processes are finished, the unit is released for general use.



ISSUE	STATIC	DYNAMIC
Inherent DB Structure	Many Tables	Few Tables
Number of Users Per DB	Many	Few
Probability of Damage	Higher	Lower
Time For Recovery	Longer	Shorter
Recover Completeness	Greater	Lesser

**Figure 6.26** Recovery: Static versus Dynamic DBMS.

### 6.4.7 Backup and Recovery Summary

This very critical subject is too little understood by most database application groups. There is too little understanding of how to achieve serialization of transactions without requiring that all database update transactions occur in a serial mode. Serial mode requires single user update, and that is something user communities will not tolerate.

As stated above, the five types of DBMS recovery are

- Rerun the job
- Roll-forward from a checkpoint
- Rollback the database from a current state to a prior checkpoint
- A combination of rollback and roll-forward, and finally
- Transaction rollback instigated by the run-unit

Careful attention should be paid to evaluating the backup and recovery capabilities of the DBMS because recovery subsequent to a hard crash could take hours. For example, if there was only one database operating under a central version, and if the log files were collecting update images all week, then in the event of a crash on Thursday, the time to recovery could be hours. If, however, the DBMS could handle multiple databases under the central version, each with its own log files, then recovery only has to be performed on the ones that were *open for update* at the time of the crash.

In short, the least desirable option is that all users are locked out of a central version while the database is being backed up and/or recovered. More desirable is that users of individual databases within the central version--not undergoing backup--are free to perform updates and retrievals, and the retrieval users of the database being backed up are also free to operate. The most desirable option is the option just stated, with the addition that users who are not addressing the subunit of the central version database being backed up are free to perform retrievals and updates against other subunits of the database.

Sophisticated DBMSs enable the recovery mechanisms to operate independently against different databases within a central version. That way, if a specific database under a central





version is damaged, others remain operational while the affected database is undergoing recovery.

As an extra measure of security, the DBMS should have the ability to *double* write the journal file. This greatly lessens the possibility of permanent damage due to bad writes to the journal file.

## 6.5 Reorganization

There are two types of reorganization: logical and physical. In general, logical reorganization refers to processes the DBMS makes available to change the database's logical structure, that is, to add/delete a table, to add/delete/modify a column within an existing table, or to add or delete relationship type between or among existing tables.

Physical reorganization, often automatically invoked through logical reorganization changes, also includes adding or deleting an index, re-optimizing the order of rows, and combining or separating rows from different types into one or more O/S files.

The greatest need for logical database reorganization is to accommodate new database application requirements. The largest quantity of changes happen during the first several years of a database application's life cycle. If the database application is implemented with a static relationship DBMS then logical database changes always consume more resources to incorporate than do changes to a dynamic relationship DBMS. This is of course due to the very different nature of a dynamic and static relationship. Additionally, these differences are often due to the fact that a static database storage structure is often very complex, has many tables represented in the same O/S file, etc. In contrast, the dynamic database storage structure is more simple, typically one table in one O/S file. But, as is stated in the previous section of this chapter, dynamic relationship DBMSs, in response to demands for greater performance, are allowing implementation of more complex storage structures, leading to the same reorganization impact as static relationship DBMSs.

Because complex storage structures are becoming a fact of life, regardless of whether the DBMS is static or dynamic, strategies have to be developed for database storage structure implementations to avoid the inevitable set of logical (and automatic physical) database reorganizations.

Two strategies are used. First, include generalized or restrictive column type clauses for columns that are not yet completely understood. For example, if EMPLOYEE SALARY is a column, then a picture of 99.99 can only allow up to a \$99.99 salary, with the presumption that the salary is hourly. If an employee is hired and the salary is \$75,000 per year, then that salary value is not be allowed until it is converted to a hourly rate. If the following two columns are included instead, the problem never arises:

- EMPLOYEE SALARY PIC 9,999,999.99
- EMPLOYEE SALARY PERIOD CHAR 2



In this case, the salary ranges from a very small number to a very large number, but its real meaning is defined by the second column's values:

- YR--yearly
- MO--monthly
- WK--weekly
- HR--hourly

It is unlikely that such a pair of columns would be subject to logical reorganization for a long time. A problem with this example, however, is that two column values are needed to fully understand the complete semantics of the value set.

The second strategy is to isolate the components of the database that are not yet well defined. These can be isolated into their own storage structure components so that when they are reorganized, there is either no impact on stable components or the input is minimized. For example, if the number of columns in the EMPLOYEE BIOGRAPHIC table is subject to a great deal of change, but there is absolute certainty about the EMPLOYEE PAYROLL table, the EMPLOYEE BENEFITS table, and the other tables related to an employee, then the stable tables and all their rows should be stored in a single O/S file while the rows of the EMPLOYEE BIOGRAPHIC table should be stored in its own O/S file. When the inevitable logical reorganization occurs, only the storage structure component representing the EMPLOYEE BIOGRAPHIC table needs to be reorganized. The applicability of this technique varies widely among all the static relationship DBMSs, and should always be present in the dynamic relationship DBMSs. The inverse ability to design complex storage structures to enhance performance varies widely in dynamic relationship DBMSs.

These two techniques support DBMS's first principal: flexibility and performance are inversely related.

### 6.5.1 Logical Database Reorganization

As database applications evolve, there is always a need to add new database component types (tables, columns or relationship) to the database, to modify existing types, and to delete unnecessary types. The process of accomplishing these changes is called logical reorganization.

Logical reorganization includes the addition, deletion, or modification of

- Columns and subclauses
- Tables and subclauses
- Relationship types and subclauses

Specifically, for columns, the changes are to:

- Add/delete columns
- Modify column characteristics
  - \* Change its defined name



- \* Change its defined length
- \* Change its defined data type
- \* Change its editing and validation clauses

When column clauses are changed so already stored data is no longer acceptable, it is very important to know what the DBMS's reaction will be before such changes are incorporated. One DBMS automatically invokes detailed checking at the column instances level, and if any of the values are not in conformance, the appropriate set of messages and offending values are printed along with the message at the end indicating rejection of the new set of rules. Changes for tables include:

- Adding new or dropping existing tables
  - Modifying table clauses
- 
- \* Physical access clauses
  - \* Editing and validation clauses
  - \* Stored procedure references

Changes for relationships between tables vary greatly between static and dynamic relationship DBMSs. For ANSI/NDL, the process is to add, delete, or modify set clauses or subordinate clauses. For ANSI/SQL, the process is one of column type addition, deletion, or modification. A very powerful subordinate clause within static relationship specification is the sort clause. When the sort clause component of the static relationship is changed, the row references participating in all the relationship instances are automatically resorted. This resorting is done either all at once and immediately, or as the relationship instances are accessed.

Another critical issue is the effect of changing the definition of referential integrity. A quality DBMS examines--at the end of each referential integrity maintenance action--all the referential integrity definitions, rejecting those that could result in ambiguous INSERT, DELETE, or MODIFY situations. It is critical to know what the DBMS's reaction is going to be if the business rules are *tightened down* before a database reorganization begins.

Some DBMSs only accomplish the logical reorganization changes as the rows are touched (added or modified). For example, if a new column is added to a table, some DBMSs only add the space and the column value required as the row is accessed. In the case of column deletes, some DBMSs merely mark the column as being deleted in the dictionary, and then return an appropriate message whenever the deleted column's value is requested.



While DBMS procedures vary widely in accomplishing logical database reorganization, the general process is:

- Access a special DBMS software module
- Input changes either interactively or batch
- Perform syntax checks
- If no errors are found, make changes, which in turn often commence some type of physical database reorganization

### 6.5.2 Physical Database Reorganization

Physical reorganization refers to the DBMS process to bring the physical order of the database back into close relationship with its implied logical order. Generally, when a database is initially loaded, its physical order closely matches its logical order. As rows are deleted or added, the DBMS automatically rearranges the physical order of the database. Physical database reorganization is often invoked whenever logical database reorganization occurs.

Physical database reorganization involves the restructuring of one or more of the database's storage structure components:

- Dictionary
- Indexes
- Relationships (static only)
- Rows

Restructuring the dictionary is logical database reorganization, and is treated in an earlier section. Index reorganization might be to

- Reallocate the number of index levels to improve performance
- Change blocking factors
- Allocate padding to index DBMS rows for extra values without reorganization
- Resequence index DBMS rows to achieve optimum logical order

Relationship reorganization, applicable only for static relationship DBMSs, enables the resorting of the rows into an ordered set of relationship references for the primary relationship. In CODASYL, the primary relationship is called *stored via*. Data reorganization is employed to

- Reclaim deleted row space
- Expand space in DBMS rows for precise loading
- Change blocking factors
- Sort rows for specific processing improvements



Physical database reorganization is needed because the physical organization of the database becomes inefficient with respect to updates and interrogations. This means that programs are operating more slowly than when the database is *new*. Normally this is because the storage structure has become fractured. Physical database reorganization is the process of recapturing the efficiency that is in the database when it is *new*.

Physical database reorganization capabilities should include the ability to add or delete indexes, and reload rows to optimize data placement and increase index performance.

### 6.5.3 Impact of Reorganization

A very significant impact on logical database reorganization is whether the DBMS has a central version that handles multiple databases. If there is only one database within a central version, the effect of logical database changes is likely to be a complete user lockout. If the DBMS central version handles multiple databases, then only the users associated with the database being reorganized are affected.

A very valuable benefit of views, as described in Chapter 5, Interrogation, is that many logical database changes are accomplished without affecting operational programs. As stated in Chapter 5, the view becomes an intermediary between the program and the database's underlying schema structure. Changes to the database structure are shielded from affecting the program if they do not affect the view or only affect the view's navigation logic. In a static relationship DBMS, the view's navigation logic is accomplished with commands such as GET NEXT, GET MEMBER, GET OWNER. In an ANSI/SQL compliant DBMS, the view navigation logic is completely contained within the CREATE VIEW statement's SELECT clause. If, for example, a column is added to a table, then the programs using a particular view that do not need that newly defined column do not have to be modified in any way. Nor does that view definition have to change. If however, a new table is added in between two existing tables, then the view's logic might have to change. Such changes should not affect the existing program, unless views are bound into load modules with the compiled programs. In that case all the programs using the affected views have to be recompiled.

### 6.5.4 Reorganization Locking

When reorganization occurs, locking occurs. The level of locking is typically dependent on whether the DBMS's storage structure is simple or complex. As the storage structure progresses from simple to complex, more and more of the storage structure is probably locked. With simple storage structures only the table, its instances, and associated indexes are locked. With complex storage structures, the entire database is locked from all users. With complex storage structures, database reorganization efforts must be carefully planned and timed to occur during the least important periods.



### 6.5.5 Static and Dynamic Differences

The effect of adding or modifying column types, tables, and relationships differs from DBMS to DBMS. As the storage structure becomes more sophisticated, the complexity of making logical database changes increases.

The difference between static and dynamic relationship DBMSs is striking. Figure 6.27 contrasts the types of logical reorganization changes with the static or dynamic nature of the database being reorganized. Logical reorganization in a dynamic relationship DBMS is easier and is accomplished more rapidly because the storage structure domain of the dynamic table is very restricted. Some dynamic relationship DBMSs even allow changes to the individual table without physical reorganization. The static relationship DBMS requires physical reorganization for all but the simplest logical database changes.

As might be expected, the domain of the affected tables in a static relationship DBMS is much larger than in a dynamic relationship DBMS. Consequently, physical database reorganization in a static relationship DBMS is more extensive and consumes greater resources.

### 6.5.6 Reorganization Summary

Database reorganization is critical to the life of a database. Through its logical aspect, the database continues to be relevant to the needs of its users, and through its physical component, the DBMS can process user requests in an efficient manner.

As database structures become more complex, that is, as there are more and more tables, relationships, and rows, sophisticated DBMSs provide utilities to monitor a database's performance so that physical reorganization is planned cost-effectively. It is cost-effective when the reorganization cost is less than the cost of running with degraded performance.

Change	DBMS Type	
	Static	Dynamic
Column	Reload at least an area	Reload at least a table
Table	Reload at least an area	Reload at least a table
Relationship	Add, delete, or modify relationship by writing a program to record the relationship mechanism into rows	Change a column's value

**Figure 6.27** Static versus Dynamic DBMS Logical Reorganization Comparisons



## 6.6 Concurrent Operations

Concurrent operations are the DBMS facilities that allow the DBMS to service multiple transactions--concurrently. There is a hierarchy of transaction types to consider: business, run-unit, and DBMS. A business transaction, the highest level, typically causes the execution of multiple run-unit transactions. A business transaction might be to ADD CUSTOMER. This business transaction is depicted in Figure 6.24. The run-unit effecting this business transaction, because of the looping implied in the business transaction, executes many run-unit transactions within one business transaction. For example, if there are 10 order-line-items for each of 5 orders for each of two contracts for only one customer, then there are 100 run-unit transactions. Each run-unit transaction typically consists of multiple DBMS transactions. For example, the run-unit transaction

INSERT ORDER, IF FAILURE THEN ROLLBACK

might involve (based on various table clauses) at least the following four DBMS transactions:

- Inserting the order row only if all the appropriate columns are valued and all the columns pass editing and validation.
- Inserting the order row only if there are no other orders with the same order primary key.
- Updating all the affected indexes associated with the ORDER table, for example, the index SALESMAN-ID-OF-RECORD.
- Updating all the appropriate relationships in which the ORDER row participates.

If these same DBMS transactions are required to occur for each run-unit transaction then there are about 400 DBMS transactions for the 100 INSERT ORDER business transactions. Automatic DBMS rollback is at the run-unit transaction level. That means that if any of the relationships in DBMS transaction 4 were not able to be updated properly, then the entire run-unit transaction is rolled back. And then, according to the logic depicted in Figure 6.24, all the run-unit transactions are rolled back.

A key goal of DBMS is to keep the execution time of a run-unit transaction to a minimum. Prior chapters have addressed this from the point of view of index strategies, types of relationships, or the placement of logically related rows from the same or different types. There are also optimizations that are achieved by making sure that storage structure components are placed on different computer hardware I/O channels and in different O/S files. This section assumes that all that has been accomplished, and that the run-unit transaction execution time is as short as possible.

Concurrent operations are commonly viewed at the run-unit transaction level (INSERT ORDER, IF FAILURE THEN ROLLBACK). Since a run-unit transaction is broken into multiple DBMS transactions, a DBMS must be designed to handle that lower level of concurrency. As



shown below in Section 6.6.3, a DBMS transaction can also be divided into subordinate transactions. Sophisticated DBMSs handle concurrency at all three levels.

A key component of concurrent operations is serializability. Serializability is the DBMS characteristic that enables a set of run-unit transactions (a business transaction) operating in a multiple user environment to produce the same results each time they are run in the same sequence.

When run-unit transactions execute against different tables resident on different O/S files, then there are few problems associated with concurrent operations. However, when run-unit transactions affect the same row, or when run-unit transactions submitted by different users affect overlapping rows, or when run-unit transactions each require/request complete locks of the database, then the DBMS needs to be very sophisticated to prevent unwanted effects or very, very long response times.

The term *concurrent operations* embraces two very different areas of conflict:

- Those between mixed data interrogation or update transactions to the same or different databases operating under the same executing copy of the DBMS
- Those between system control transactions, such as logical and physical reorganization, security and privacy, etc.

Generally, the second type of conflict is well documented and the effects on users are well known through messages from the database administration group:

*Attention users, the database will be down between 0800 and 1200 for reorganization.*

The remainder of this section deals with transactional conflicts among interrogations and updates against the same or different databases operating under the same executing copy of the DBMS.

There are three different concepts that exist in pairs and in various combinations contrasting the different types of DBMSs in the marketplace. The pairs are

- Single or multiple user DBMS
- Single or multiple database DBMS
- Single or multiple threaded DBMS

### 6.6.1 Single or Multiple User DBMS

A single user DBMS is one that only allows access by a single user to an executing copy of the DBMS. The database identified for use is then typically locked until the user session terminates. If another user attempts to attach the database, a *database in use* message is displayed, and no access is allowed.

A multiple user DBMS is one that processes multiple run-unit transactions from different run units concurrently. Such a DBMS configuration is employed in distributed processing, or multiple user micro, mini, or mainframe environments. Unless otherwise indicated, the run-unit transactions all process serially through the DBMS. Thus, the next run-unit transaction has to





wait for the completion of the current run-unit transaction. Also, unless otherwise indicated, transactions from each run-unit only access a single database within a given user session.

### 6.6.2 Single or Multiple Database

A single database DBMS is one that permits one or more users access to only one database under the DBMS's control. If there are multiple databases concurrently being accessed, there must be one DBMS for each database.

This means that each user only accesses one database. Multiple users access that one database, but a single user cannot access multiple databases.

A multiple database DBMS is one that permits run-unit transaction access--from one user--to different databases that are under the control of the single instance of the DBMS. The DBMS processes to completion only one run-unit transaction at a time regardless of which database is being accessed. This configuration is typical on single user PCs.

### 6.6.3 Single and Multiple Threaded DBMS

A single-threaded DBMS only executes one run-unit transaction at a time and cannot service another run-unit transaction until processing finishes for the first run-unit transaction. A multi-threaded DBMS concurrently executes multiple run-unit transactions from multiple run-units against a single database.

Executing multiple run-unit transactions at the same time requires a computing environment that either has multiple CPUs to process more than one run-unit transaction at the same time, or multiple I/O device channels so that multiple channel control programs (I/O service routines) execute at the same time.

Key to executing multiple run-unit transactions at the same time is being able to decompose the DBMS transactions comprising a run-unit transaction into subtransactions. For example,

Insert the order row only if there are no other orders with the same order primary key.

can be divided into the following DBMS subordinate transactions:

- Employ the CPU to formulate the appropriate command to attempt to find an order row that is already stored with the same primary key value
- Employ the I/O to determine that none is found
- Employ the CPU to formulate the store operation to actually place the row on disk
- Employ the CPU to formulate a *success* message to be passed back to the run-unit



If the environment only has one CPU, but multiple channels, then an I/O request is issued to the channel, which in turn searches its mass storage devices for the data until it is found. When the data is found it is passed back to the computer's main memory for use by the CPU. While the channel is busy finding the data, another channel control program can be executing in another channel to find the data required by another DBMS subtransaction. Keeping all these DBMS subtransactions properly organized requires a very sophisticated DBMS design as well as a sophisticated computing environment.

#### 6.6.4 Commonly Existing DBMS Combinations

The combinations that commonly exist are, in increasing order of sophistication:

- Single user, single database DBMS
- Multiple user, single database, single thread DBMS
- Multiple user, multiple database, single thread DBMS
- Multiple user, single database, multiple thread DBMS
- Multiple user, multiple database, multiple thread DBMS

These are depicted in Figure 6.28. The platforms that accommodate these different levels of sophistication appear in Figure 6.29.

Increasing Level of Sophistication	User		Database		Threads	
	Single	Multiple	Single	Multiple	Single	Multiple
1	YES		YES		N/A	N/A
2		YES	YES		YES	
3		YES		YES	YES	
4		YES	YES			YES
5		YES		YES		YES

**Figure 6.28** Sophistication Levels for Concurrent Operations Configurations



Typical Computer Platforms	USER		DATABASE		THREADS	
	Single	Multiple	Single	Multiple	Single	Multiple
PC's	YES		YES			
PC Networks		YES	YES		YES	
Mini Networks		YES	YES		YES	
Mainframes		YES	YES		YES	
Mini computers and Mainframes		YES		YES	YES	
		YES		YES		YES

**Figure 6.29** Typical Platforms for Concurrent Operations Configurations

DBMSs that are typically single user are also likely to be single database. These most often operate on PCs or time sharing services. DBMSs that are multiple user either are single or multiple database. Multiple database operations are clearly a level of sophistication over single database. Finally, DBMSs that are multiple user either are single or multiple threaded. Multiple threaded operations are significantly more sophisticated.

#### 6.6.4.1 Single User, Single Database DBMS

The single user, single database is most common on single user PCs. However long waits are possible as the user submits one run-unit transaction and must wait until it is finished before starting another. In fast PCs, some DBMSs allow printing of one report concurrently with other activities. To achieve that the DBMS usually *electronically* prints the report to a spool file, and then *steals* cycles from the foreground job to send print records from the spool file to the printer. Simply put, there is a one-to-one-to-one relationship between the user, the database, and the DBMS.

Because of the speed of the Intel Pentium chip, multiple tasking operating systems are now viable on PCs. Even in the single user mode a multiple tasking operating system can start multiple jobs, each operating under different copies of the DBMS that run concurrently. Under this mode there is no concurrency control. Therefore the only safe run-units are reports against the same or different databases, or update run-units against different databases.

#### 6.6.4.2 Multiple User, Single Database, Single Thread DBMS

A multiple user, single database, single thread DBMS configuration generally exists on PC networks, distributed networks, minicomputers or mainframes. In this environment, multiple



users access only one database, as only one database is controlled by the executing DBMS. As a second user's run-unit transaction reaches the DBMS, it waits in a queue until the run-unit transaction currently executing is finished. Long waits are possible as the currently executing run-unit transaction might be performing updates to many rows.

#### **6.6.4.3 Multiple User, Multiple Database, Single Thread DBMS**

The multiple user, multiple database, single thread DBMS environment enables the DBMS transactions from the different users to access different databases all under the control of a single executing copy of a DBMS. Having multiple databases means that there are multiple sets of dictionaries, indexes, relationships, and data. Thus, under the control of one executing copy of the DBMS, the following transactions can occur concurrently to the different databases known to the single executing copy of the DBMS:

- One database is varied off-line through a DBMS backup transaction and then a O/S command is employed to delete the files from mass storage.
- A second database is accessed for retrievals, updates, etc.
- The third database is undergoing logical or physical reorganization.
- A fourth database is completely locked for special updating.
- Definition of a fifth database is occurring through DDL commands.

If the DBMS is single threaded, then all these run-unit transactions to the different databases queue up--as received--and are executed one at a time. A run-unit transaction from a user to the first database is executing at the time a run-unit transaction to the fifth database arrives. When the first database run-unit transaction finishes the run-unit transaction affecting the fifth database starts. During the execution of the run-unit transaction affecting the fifth database, a run-unit transaction for the third database might arrive and await in the queue. Then a run-unit transaction for the fourth database might arrive and also wait in the queue. After the run-unit transaction to the fifth database finishes, the first run-unit transaction in the queue starts. After it finishes, the next run-unit transaction in the queue starts. If all the run-unit transactions are short, the computing environment is sophisticated or fast enough, and the number of run-unit transactions in the queue is reasonable, then the waits are tolerable as the sophistication and speed of the computing environment overcomes the number of run-unit transactions in the queue. If the computing environment is neither sophisticated nor fast enough, then the queue becomes unacceptably long. Or, if the number of users suddenly increases, then the computing environment cannot keep up with the arrival rate of run-unit transactions and the queue lengthens.



#### **6.6.4.4 Multiple User, Single Database, Multiple Thread DBMS**

A multiple user, single database, multiple thread DBMS configuration is like the one described in Section 6.6.4.2 except that the DBMS handles the simultaneous execution of multiple run-unit transactions. Still, all the run-unit transactions must occur against only one database. Such a configuration requires a sophisticated computing environment found on either mainframes or multiple CPU database machines.

Since the typical DBMS transaction performs a minimum amount of CPU activity in contrast to the number of disk accesses (I/O requests), multiple DBMS transactions are serviced if the computing environment concurrently executes multiple I/O requests. Being able to have multiple and concurrently executing CPU processes is less important as database is typically I/O intensive, not CPU intensive. In the case of ANDing and ORing index lists, multiple CPUs enhance performance.

#### **6.6.4.5 Multiple User, Multiple Database, Multiple Thread DBMS**

The most sophisticated DBMS capability is a multiple user, multiple database, multiple threaded DBMS. In this environment the DBMS installation process often requires the definition of the number of threads (queues) and the maximum number of transactions allowed in each queue. Another installation parameter might allocate a service priority among the queues. There might also be a servicing algorithm for the queues that is affected by installation parameters. During the day time, query-update language transactions might receive a higher priority than batch requests. At night the reverse might be true. Thus, classes of run-unit transactions would be automatically allocated to certain queues. In this example, the COBOL batch jobs would take longer to execute during the day as they would be assigned to lower priority queues. Since query-update language run-unit transactions would be allocated to higher priority queues they would be serviced quicker.

In general, the dynamic relationship DBMS performs the multi-threaded, multiple database set of operations with less DBMS vendor software than is required for a static relationship DBMS. This is because in the dynamic database, the tables are typically physically independent, and other aspects of their storage structure are physically and logically decentralized. As dynamic relationship DBMSs respond to demands for increased performance, their storage structures will become more sophisticated.

### **6.6.5 Complex Storage Structure Effects on Concurrent Operations**

In general, the more complex the storage structure, the greater the likelihood that a lock imposed for one user's access will affect another user. In the Figure 6.30 example of two DBMS rows, rows from DEPARTMENT and EMPLOYEE are stored on the same DBMS row instance.



PHYSICAL ROW 1		
ACCOUNTING	ABLE	BAKER
GRODMAN	KERR	PERKINSON
ADMINISTRATION	EDMONDS	CRITENDEED

PHYSICAL ROW 2		
FRANCIS	GIMBLE	HARRISON
JONES	ENGINEERING	APPLETON
QUINCY	RICHARDSON	ZIEGFELD

**Figure 6.30** Complex Storage Structure Effects on Concurrent Operation

If a run-unit transaction performed a GET WITH LOCK obtained the ACCOUNTING department and if the locking strategy locked the entire DBMS row, then when another run-unit transaction attempts a GET WITH LOCK to obtain an EMPLOYEE row for KERR, it will receive the message: DBMS row LOCKED.

### 6.6.6 System Control Operation Conflicts

It is not safe to run certain operations concurrently with others. For example, logical or physical database reorganization transactions along with data updates. A Database backup can execute concurrently with updates if implemented in a sophisticated manner. For example, if a backup is started at midnight and continues through 9:00 A.M., and the on-line update queues are opened at 8:00 A.M., then the backup and after image journal records created after 8:00 A. M. are all that is needed to restore a crashed database environment at 10:00 A.M.

The effect of system control locks is greatly minimized through the use of a multiple database DBMS as described above in Section 6.4.1, Database Backup.



### 6.6.7 Concurrent Operations Locks

Certain DBMS operations impose locks explicitly or implicitly. An explicit lock is achieved when a lock command is issued. An implicit lock is one that is imposed as a consequence of the intensity of the operation. Locks often occur during:

- Massive data loads
- User-instigated DBMS locks
- Audit trail instigation, suspension, or modification
- As a consequence of messages with a severity 4 (database fatal), and severity 5 (DBMS fatal)
- During backups (sometimes) and recoveries (always)
- At the onset of logical and physical reorganization
- During security instigation and maintenance

It is critical to the acceptable operation of the DBMS that each of these locks be understood in terms of the resources consumed and in the elapsed time required. Armed with this knowledge, schedules can be created to minimize these undesired but necessary disruptions.

### 6.6.8 Deadly Embrace

Deadly embrace is a state in which two or more commands mutually await the other's locked resource. Deadly embrace disables only a poorly designed DBMS. A well-designed DBMS resolves the occurrence of execution deadlock by first terminating one of the conflicting operations, and then attempting to restart the terminated run-unit. Typically, deadly embrace is resolved in two steps:

- The DBMS stops one of the run-units.
- The DBMS rolls back all the stopped run-unit's uncommitted transactions.

Once the deadly embrace is resolved, the environment is recovered by either of the following:

- The DBMS requires the user to restart the terminated run-unit.
- The DBMS automatically attempts a run-unit restart for a predetermined number of times or periodically for a predetermined wall-clock time before permanent abort.



### 6.6.9 Static and Dynamic Differences

The main difference between static and dynamic relationship DBMSs, in the area of concurrent operations, again reflects on the basic nature of the DBMSs and their databases. In a dynamic environment, any operation on any table can co-exist with any other operation on any other table so each table is completely independent. In a static environment, the storage structure is complex, and multiple users in fact are often using the same physical file even though they are dealing with separate rows. Because of this complexity, queues often result to prevent destructive competitive operations.

### 6.6.10 Summary

In concurrent operations, it is critically important to know where (storage structure) and when (user commands) conflicting operations can occur. Each must be verified by setting up tests to stop/slow processing via nonconcurrent operations. Finally, once these slowdowns are known, and when they are projected to occur due to required database operations, such as reorganizations, security and privacy installations, procedures must be established to notify users during scheduled and emergency lockouts.

## 6.7 Multiple Database Processing

Multiple database processing is the ability of a single executing copy of the DBMS to know of and to support concurrent activities against multiple databases. This facility is not present in all DBMS products. With respect to two very popular, mainframe DBMSs, however, the vendor's typical response to inquiries concerning the existence of multiple database processing is *yes*. In fact, however, a single executing copy of these DBMSs knows of only one database at a time and thus supports only multiple user activities against one database. To know of, or to support, activities against another database requires stopping the DBMS's execution, changing the O/S file names that represent the name of the database in the DBMS's job control language, and restarting the DBMS.

Other DBMSs know of multiple databases through user database attachment commands. One user could be attached to 10 databases just as easily as 10 users could be attached to one database.

Multiple database processing implies exactly what the statement says, that the DBMS has the ability to process multiple, independently defined and loaded databases--concurrently.

To know whether a DBMS handles more than one database under a single copy of the DBMS requires a clear definition of database, how to recognize when only one database *can* exist within a single copy of the DBMS, and how to recognize when multiple databases *can* exist.

A database is that set of interrelated data that contains information about its own organization (dictionary) and rows (data). A database may include mechanisms for fast access (indexes). Finally, some DBMSs store pre-executed interconnections (relationships) between





rows of the same or different types. A database cannot exist without a dictionary and data. Indexes and relationships, although critical, are optional. To have one database is to have one set of these components. To have multiple databases is to have multiple sets of these components.

A database is independent of other databases operating under a DBMS central version if it can be loaded, unloaded, updated, backed up, and restored independently from any other database also operating under the control of the same executing DBMS.

To have multiple databases within the control of a single executing copy of a DBMS requires, *a priori*, the ability to execute multiple schema statements. That is, the ability to state at least as many times as there are databases operating within the control of that executing copy of the DBMS:

SCHEMA NAME IS <database name>

Whether the DBMS allows concurrent access to multiple databases from within the various interrogation languages by a single user is another matter and is discussed in Section 6.7.6

To test the existence of multiple databases:

- Start a DBMS's execution, then
- Submit a DBMS command to attach to a database and then,
- While attached, execute a command that copies that database instance to off-line storage.
- Then submit another command to attach to another database.
- While attached to that other database, another user, not operating under the control of the DBMS, uses an O/S delete file command to remove the on-line instances of the O/S files representing the database that moved to off-line storage. If that O/S delete file action causes a severity 3 or worse message, then there is only one database.

Another key indicator that a DBMS permits multiple database processing is that a database can be moved from one instance of a DBMS central version to another instance of a DBMS central version without going through the process of row unload and load.

Some DBMSs can only operate one database under the control of a single executing copy of the DBMS, while other DBMSs can operate multiple databases.

### 6.7.1 Types of Multiple Databases

A critical database design activity is the determination of its boundaries, that is, is the database restricted to well-defined subject areas such as human resources, customers, sales and marketing, etc., or is it restricted by fiscal year, calendar year, or by corporate divisions. Whatever boundaries are chosen, they are sure to exclude someone's data.



Two popular methods of setting the boundaries are by corporate level, and by *damage control*. The common corporate levels are executive or MIS, middle management control or administration, and operations. The *damage control* principle splits the data from one very large database into multiple databases, all having exactly the same schema. Thus, if a corporation had multiple divisions, each with their own sales and marketing force, there could be either one on-line database that is all operational or all inoperable, or there might be multiple databases, one for each division so that if one division's database is not operating, the other databases remain operating.

The five aspects of multiple database processing that need to be examined are:

- The DBMS's ability to operate multiple databases within a single central version
- The impact on physical database structures
- The ability of run-units in various DBMS languages to access data from two or more databases
- The impacts caused by a DBMS's system control facilities
- The differences imposed by the DBMS's static or dynamic character

A solution to not being able to handle multiple databases is to have multiple central versions of the DBMS. Central versions themselves consume considerable overhead with respect to computer and personnel resources. Since most database applications are interactive, that means attaching the central version to a teleprocessing monitor. The fewer there are of these connections, the easier the environment is to manage.

Each class of applications could be placed under its own central version. For example, to answer the needs of an on-line order-entry system there could be four central versions. Then there could be other central versions for the other different types of applications, that is, several for testing, some for manufacturing, personnel, financial applications, and the like.

In addition to the considerable overhead of handling multiple central versions, there is also the problem of consolidated reporting. Further, the more central versions there are, the greater the tendency to drift towards application databases.

Because of all the overhead there simply is a need for multiple databases under a central version. There is also a need for programs to interact with different databases under the central version, and between databases in different central versions.

## 6.7.2 Rationale for Multiple Databases

Not all applications within an organization require integrated data. For example, for a manufacturing organization, the need to integrate personnel data with manufacturing quality control is limited to probably just a few reports per year, or none at all. Consequently, the manufacturing and personnel applications could exist in their own databases. If a DBMS does



not allow multiple databases, then these applications have to be contained in the same database. In that situation, the backup of one is the backup of the other, and the crash and recovery of one is the crash and the recovery of the other.

If there are multiple, different logical databases, then the database's definition can be developed and revised without affecting other databases operating under the central version. In a multiple database environment, a database development group does not have to wait until a specific time to perform logical database changes to one database that do not affect other databases.

The need for multiple databases extends well beyond separating different applications. For example, if there is a nationwide order entry system, the order entry hours could well be from 8:00 A.M. AST (Atlantic Standard time (1 hour beyond Eastern)) through 8:00 P.M. Hawaii time (three hours beyond Pacific). That includes Eastern Canada through Hawaii. The total order entry time is then 19 hours. That leaves only 5 hours for consolidated batch processing, reporting, and the like. If instead of a single database, there are three databases, each one representing 8 hours or less, there can be a database for the orders for the Atlantic time through Central time zones, another for the Mountain and Pacific time zones, and one more for the remaining time zones. As the on-line aspects of one database closes, it can begin its batch processing, including creating files for end-of-day sales reports. When the last database is *closed*, its batch processing is accomplished and all the batch transaction files previously created can be used for the final end-of-day sales reports. Each database has non-overlapped time for detailed data editing and validating, summary statistics generation, backup, and the like.

What then of consolidated processing? There are several alternatives. The first is to write a file of data that is loaded into a central database. As the last database completes its processing, this consolidated database receives its last batch of updates. It could then be interrogated and the consolidated reports produced.

If a run-unit accesses data from multiple databases, then an HLI run-unit could produce the consolidated report. If the DBMS requires a one-to-one relationship between the view and the run-unit, these views and database accesses could be accomplished through properly defined subroutines.

The third alternative, assuming that some of these order entry databases are different central versions, is to have the consolidated reporting run-unit access these different databases through their different central versions, and then perform the consolidated report.

The need for multiple databases also arises from having multiple application databases that really belong to one subject area database. For example, there might be a database for all the personnel information, another for managing projects, and a third for personnel training and development. These three probably should all be a part of a single human resources management system, but for political, economic, or traditional reasons they are not. Naturally a problem arises whenever there is a need to develop a complete profile of an employee. One way to solve this problem is to ask for a complete print-out from each database for the employee, and then manually compile the report. While this might be faster for a few employees, it's not for hundreds or thousands. What is needed is a reasonably efficient mechanism to interrelate the data from many databases into single reports. Hence the need for multiple database processing.

If this is the only need, then a case can be made to consolidate the database into one. But as soon as this is done, someone is sure to want consolidated reporting on the financial data that



is kept in decentralized systems, or if the financial data is centralized, to do longitudinal reporting for specific data over several years. The point is that no organization is ever without the need for multiple database processing.

### 6.7.3 Critical Issues

When accomplishing multiple database processing, the critical issues are:

- Schema commonality
- DBMS versions
- Required update coordination

Under some DBMSs, multiple divisions of a corporation employ different physical databases from the same logical design to accomplish the same application. Whenever these different databases need to be combined from these multiple databases, the schemas from each must be exactly the same with respect to the commonly accessed parts. To enforce schema commonality, a single group within the corporation must develop the schema. This group could be *real*, or be a joint committee from the decentralized divisions with the committee's chair elected or from headquarters.

Great care must be exercised in examining corresponding DDL. It must be the same for at least the following clauses:

- Table
- Column
- Set (if ANSI/NDL)
- Edit and validation tables
- Table lockups
- Assertions and triggers

The second area of concern is DBMS versions. If, in a distributed environment, one division is running a database under the most recent version of the DBMS and another is running a version that is a year old, then a run-unit that has to access the two databases might not work because the storage structures of the different databases might be different. Coordination between the different sites with respect to DBMS releases and the application of interim updates by the DBMS vendor should eliminate this type of problem.

In a centralized environment there is never a concern over different versions of the DBMS for current data. Rather, the concern is over the different DBMS versions that are used to save databases for previous years. For example, there might be a set of corporate statistics for data in 1984 that were backed up with the 1984 version of the DBMS, and that data, due to the storage structure changes in the DBMS versions since 1984, cannot be restored with the 1991 version of the DBMS for use with data from 1990. To remedy such a situation, these backed up databases have to be periodically restored and re-backed up every time a DBMS release is made that affects the storage structure.



The third area of concern, update coordination, is especially important. If the multiple databases are multi-level, that is operational control, middle management, and strategic or MIS, there must be great care in the planning of update cycles. Batches of updates created for the lowest level database, then summarized for the next and top levels, must all be applied at the same time. There could be daily updates for the lowest level, weekly for the middle level, and only monthly for the highest level. Weekly summaries will always appear smooth as there are only five days in each week, except for holiday weeks. However, to create monthly summaries with only whole weeks in each month, some months have to be defined as four weeks while others are five weeks. The five week months are 25 work days, while the four week months are 20 work days. Quarters are four 13 week periods, not necessarily three calendar months. If the thirteenth week of the fourth quarter is in January, then that is when the year ends rather than on December 31. In computing corporate income and expenses on a monthly, quarterly, and annual basis, the figures will always look erratic unless they are smoothed to be truly monthly, quarterly, or annual.

When updates occur to the highest level database and fail, there must be well established policies about how to deal with the updates at the lower level.

Even more critical is the situation where information from a new employee is added to several different databases at the same time. A new employee might be added to the human resources tables in the main corporate database for personnel, benefits, and payroll, but the appropriate updates to a decentralized division human resources database might fail. Before the new employee update is considered complete, both databases must be completely updated. In short, if the update fails on one database, there better be well established policies about dealing with the updates to the other databases to avoid partial updates. The reverse is also true if an employee leaves a division, or is transferred to another division, then the appropriate transactions must be generated to update all affected databases.

#### **6.7.4 Logical Database Impact**

The majority of the issues contained in the previous section address the logical database impacts on multiple databases. If there are relationships between rows from tables in the different databases, then these relationships have to be dynamically based rather than statically based. That is because static relationship instances are usually based on DBMS row addresses that could change through various adds, deletes, or modifications. While the DBMS is expected to keep all those addresses straight within one database, it cannot do that activity across multiple databases, especially if one database is varied off-line and its disk space reused.

#### **6.7.5 Physical Database Impact**

Multiple database impact on the physical database affects mainly the operational mode of the DBMS.



### 6.7.5.1 Storage Structure

The storage structure impact from multiple databases is restricted to multiple sets of the storage structure components. If there are five databases, there are five sets of dictionaries, indexes, relationships, and data. Some operating systems impose an upper limit of O/S files known to a run-unit, which is the executing copy of the DBMS. In such cases, if the upper limit is 255, and each storage structure component is a single file, then there could be about 60 databases known to the DBMS. That number is actually too large because the DBMS often requires scratch and sort files, etc. Subtracting 50 files for that purpose, the upper limit of databases is about 40.

A real problem arises, however, if the storage structure requires a separate file for each column's index and each table. If the average corporate database is 30 tables, and there are two indexes per table, and there are 20 files set aside for the dictionary and scratch files, the number of files required for just one database is 110. That leaves room for just two databases. In short, the impact of the storage structure design is significant.

Generally, DBMSs with sophisticated storage structures operate multiple databases. Multiple database processing must be designed into the DBMS product and its database storage structures right from the start.

### 6.7.5.2 Access Strategy

The multiple database impact on access strategies relates to transaction separation and buffer management. Keeping transactions from multiple databases separate is merely an extension to the problem of keeping user transactions separate in a multiple threaded environment. As the number of databases, users, and threads increases, the sophistication of memory resident buffers increases. The buffer management facility must *learn* from the transactions being processed. That is, it must keep in memory the most often and most recently used DBMS rows from the various storage structure components of the various databases. If the buffer management facility is not properly tuned, performance suffers. It is certainly possible that one buffer configuration is highly tuned towards one set of databases, but is improperly tuned for another set of databases.

### 6.7.5.3 Data Loading

Multiple database processing affects data loading only because the name of the database has to be delivered by the run-unit to the DBMS. This enables the DBMS to know which database is the target of the run-unit's activities. If multiple databases are to be loaded then the multiple databases must be opened.



#### 6.7.5.4 Data Update

Updates to a single database are likely to be faster in a multiple database environment as the databases are smaller. Thus, fewer rows are processed, smaller indexes are traversed, and smaller chains of relationships are scanned to update the table's instance.

If an order entry application is partitioned by region, resulting in ten regional databases, then to find the order for a particular product and customer is faster due to the smaller indexes and the smaller multiple occurrence lists that need to be ANDed (list processing indexes presumed). If the databases are statically oriented, summarizing the product orders to decrement inventories and to alert the regional shipping warehouses is faster because the length of the chains is smaller.

#### 6.7.6 Interrogation Impact

Some application programs need to access data from two or more databases, simply because there is never one database schema that addresses all the data required by all the programs. Someone's data is always split across multiple databases.

Database access is either through a host language interface or through a natural language. Assuming the access is through views, then to access multiple databases requires either that the view definition language allow a view to be defined across multiple tables from different databases, or that the run-unit allow multiple views from different databases.

The typical scenario for HLI multiple database processing is to read data from a non-DBMS file or input screen, and then STORE that data into one or more databases. When data passes from one database to another, the sequence is to GET data from one database and STORE the data in another. Such a scenario requires the manipulation of multiple sets of cursors, one set for each database.

In the natural language environment, there are several different existing strategies. One strategy, when data is required from two different databases that have the same schema definition, is to have a database attachment command that allows the inclusion of multiple database names. This strategy is most suited to the query-update natural language.

Another strategy is to develop a command language sequence that gets data from one database, writes that data to a dynamically created database, gets data from another database and writes that data to the newly created database. When all the gets and writes are finished, the program creates consolidated reports from the newly created database. This strategy is suitable to the POL and the report writer languages as these programming languages are capable of branching, looping, etc.



### **6.7.7 System Control Impact**

The impact from multiple database processing primarily affects:

- Audit trails
- Backup and recovery operation, including transaction rollback
- Concurrent operations
- Security and privacy

#### **6.7.7.1 Audit Trails**

The multiple database impact on audit trails centers on obtaining a complete report of the activities that may have occurred on a particular business application. A financial application may, for example, receive transactions from various sources and store them in three journals, one each for the general ledger, receivables, and payables. Once the data is considered acceptable, summaries may then be derived from the payables and receivables journals and stored in the general journal. Summary data may then be retrieved from all three journals and stored in mid-level databases for business units or product lines that support weekly and monthly reports, and finally in high-level databases that support corporate wide reports, trends analyses, and the like.

If the DBMS stores transactions to each of these journals in different databases, then a consolidated audit trail report of financial activity is difficult. To create the audit trail, each journal based program would have to write to a consolidated financial transactions journal as well as to the individual journal databases.

Difficulty in this area does not, however, eliminate the need for multiple database processing. Rather, it increases the challenge to accomplish it in an acceptable manner.

#### **6.7.7.2 Backup and Recovery**

Most backup and recovery operations are restricted to all the databases operating within a single central version. Some DBMSs record all the transactions from all the different databases onto a single centralized journal, while others record the journal transactions onto separate journals, one for each database.

The critical issue is transaction rollback and recovery after a database crashes. If one database crashes and the only transactions backed out are those recorded against the damaged database, then what happens to the recorded transactions for the database(s) that did not crash? If user transactions span databases there is a failure in overall database consistency. In these cases, very careful planning must take place to ensure that the databases remain in synchronization.

#### **6.7.7.3 Concurrent Operations**





Multiple database processing affects concurrent operations only as it relates to backup and recovery, and audit trails. To ensure that a complete set of transactions is accepted by all the databases to which they were posted, there has to be a request for exclusive control over all the involved databases, and for the exclusive control to be maintained until all the multiple database transactions are accepted and a checkpoint transaction is posted.

#### **6.7.7.4 Security and Privacy**

Security and privacy affects the operations performed, the columns, tables, and relationships processed, and possibly the actual rows viewed. Defining security and privacy on behalf of users is no easy task. DBMSs usually take one of two approaches: single database or user profile. In the first approach, the security is defined through the establishment of passwords and authorities allowed for those passwords from the viewpoint of a particular database. Thus, for a user to perform certain activities, the user must possess the password. Anyone possessing the password has the same access. While this approach is suitable for single database environments, it is unsuitable for multiple database environments.

The second approach starts from the viewpoint of the user, not the database. From the user viewpoint, certain processes and DBMS views are allowed into the user's profile. Whatever the user has permission to use is available, otherwise it is not. Thus, if a particular natural language allows access to multiple views (one for each database) or the run-unit accesses multiple databases, then the user can perform whatever the language or run-unit accomplishes.

#### **6.7.8 Static and Dynamic Differences**

Most dynamic relationship DBMSs have very well developed multiple database processing facilities because their databases are often designed with physically separate, simple storage structures. The dynamic relationship DBMSs allow multiple database processing in a variety of interrogation languages. Some DBMSs allow the user to open a list of tables, and if they are all the same structure, the DBMS treats them as a single database.

In a static database, however, the database's storage structure is normally more complex, so these DBMSs usually provide multiple database processing only through the host language interface, or through POL and report writers.

The ANSI NDL, an evolution of the CODASYL model, does not prohibit multiple database processing. The current ANSI/NDL defines modules that are used by various programming languages. Even though a given module is restricted to a single <schema-subschema> pair, there is no restriction on the number of different modules that a program can call.



### 6.7.9 Multiple Database Summary

Multiple database capabilities are needed--always. That is because multiple database processing is the method of integrating:

- Database editions that are either time or volume based
- Distributed databases that are resident at different sites/computers
- Different applications such as personnel, engineering, etc.
- Multiple database levels such as operations, middle management, or high-level MIS applications

A minimally acceptable DBMS, whether it is static or dynamic, and regardless of data model, will offer capabilities to accomplish multiple database processing in its HLI. The more sophisticated DBMSs will additionally offer access through the POL and report writer natural languages.

## 6.8 Security and Privacy

Security and privacy facilities are provided by the DBMS to define boundaries of allowable access, enforce these boundaries by rejecting attempts to penetrate them, and then report on the attempts to pierce these protections.

### 6.8.1 Rationale

Prior to the existence of database and DBMS, only the computer programs specially designed to access data files could make sense out of the data. In essence, security existed because the data structure's definition was fundamentally a part of the computer programs and not the data files. The data files and the programs were normally kept separated, providing additional security. The data files were on tape in a tape library, and the programs were kept in card trays at the programmer's work station. Since the separation of the data's definition from the data was a drawback to the generalized use of data by many different languages, technology came to the rescue and made the data's definition part of the data, giving rise to databases. Now, that solution is the problem, because anyone who has access to a generalized language can access a database since the database contains both the data's definition and the data.

In short, database use necessitates sophisticated security and privacy, since well organized collections of corporate data are readily available through easy-to-use, rapid-access, natural languages. To protect the database, security and privacy facilities must be provided.



## 6.8.2 Areas of Concern

There are more than just rows that need to be secured. In fact, the entire database environment including the logical database, physical database, interrogation, and system control needs protection.

### 6.8.2.1 Logical Database

The logical database, from the DBMS point of view consists of table clauses, column clauses, and relationship clauses. Establishing a database should require prior authorization. With one DBMS, establishing a database's structure was so easy that within a few month's of the DBMS's installation there were hundreds of databases established, loaded, and being used. Of these hundreds of databases, some even had something to do with the mission of the organization. The databases proliferated to such an extent that the command allowing establishment of a new database had to be removed from the DBMS to bring the situation back under control.

Once a database is established, its definition needs to be protected from unauthorized changes, a topic addressed in Section 6.8.3.2.1.

### 6.8.2.2 Physical Database

Within the area of physical database, there are a number components that need to be protected. These are storage structure, data loading, data update, and backup.

#### 6.8.2.2.1 Storage Structure

The storage structure of any database must contain a dictionary and data. It may also contain indexes and relationships. Dynamic relationship DBMSs do not have explicitly declared relationships. Since these components are really all O/S files, they need to be protected from unauthorized reading, copying, updating, and deleting.

The dictionary component of a DBMS's database contains a great deal of important information. While an uninitiated person is unlikely to decode its contents, someone trying to steal a database is unlikely to be uninitiated. Contained in a single database's schema are:

- Schema clause and subclauses
- Table clauses and subclauses
- Column clauses and subclauses
- Relationship clauses and subclauses



If the DBMS supports multiple databases, and if all the dictionaries for all the databases are kept in one dictionary, then that same information is available for all the databases. If that information is decoded and changed, then all these clauses and subclauses can be changed from what is intended. For example, the edit and validation rules could be changed to permit NULL values in an audit trail column USER-ID-OF-PERSON-MAKING-UPDATE.

As another example, the rules for row memberships could change. A payment could be made to a person without first checking the referential integrity rule that the payment be made only when there is a valid invoice.

Indexes can provide a great deal of classified information. For example, an index for SSN could provide all the social-security-numbers. An index for SALARY could provide all the unique salary values and the multiple occurrence lists of the primary keys (possibly the SSNs) of all persons earning specific salary amounts. The SSNs could also lead directly to the rows.

Relationships exist independently from the rows in the static-segregated form. In that form, a count of relationships implies a count of rows, and a count of the relationships within a relationship gives the quantity of rows within that relationship. In this form, or in the form where the relationships are embedded in the rows, changes could be made directly to those relationship pointers. For example, suppose all checks must be issued against approved invoices from an owning department. If the relationship instance between the invoice and its owning department is changed such that there is no owning department for the invoice, the checks attached to that invoice would be *lost*. When a report is run that shows paid invoices for the department these lost invoices would never appear.

The final storage structure component, data, consists of the rows. A utility could be employed to delete specific rows, for example, the record of a written check in a payables journal, or the audit trail transaction identifying the person making a specific update.

How can someone find out how to decode such information from the storage structure components? Simple. All DBMS vendors internally publish detailed information about the exact structure of each storage structure file. These materials are needed for normal vendor programming and testing of the DBMS. They are also needed whenever a database crashes to such an extent that the only way to repair the damage is with an O/S utility that absolutely changes specific bytes. While these manuals are normally restricted to the DBMS vendor's employees, some client organizations have managed to acquire these books, leaving open the possibility of unauthorized duplication within the client's shop.

Because of all the information that can be gleaned from the files in a DBMS's storage structure, the site's operating system needs must prevent access to these files by any system software other than the DBMS.

#### **6.8.2.2.2 Data Loading**

DBMS vendor provided data loading utilities are often designed to bypass the carefully crafted editing and validation facilities that service the needs of on-line updating. Because of these bypasses, these utilities need to be protected from unauthorized use.



#### **6.8.2.2.3 Data Update**

Update security ensures that only authorized changes are made. Thus, there is a need to prevent

- Modification to particular columns and to particular values
- Adding new column values to existing rows
- Inserting new rows
- Amending existing relationships between rows

Changes to columns and rows are usually made either through views or subschemas. Thus, specifically designed views and the run-units using those views must be protected from unauthorized access.

#### **6.8.2.2.4 Backup**

Backups are copies of an entire database. It is almost useless to have highly sophisticated security on database access, the O/S utilities, etc., if someone makes a backup of a database and then hand carries the tape(s) out of the data center. Restrictions must be established on who makes backups and takes tapes out of a data center.

#### **6.8.2.3 Interrogation**

Interrogation, regardless of the language, consists of two components: the view/subschema, and the executing run-unit.

The view definition process must be protected because views are the primary method of accessing databases. Strict procedures need to be in place for verifying that the person requesting a view has the proper authority to perform whatever actions the view allows. A view has a data interface, operators and navigation logic, and select clauses. Each of these three components needs to be carefully examined.

In addition to view definition, view maintenance procedures need to be carefully examined to ensure that views are modified or deleted only by those with authority. Significant havoc could occur if someone deleted all the active views. Even though it may only take about one hour to establish one view, it would then take about 25 staff weeks to restore 1000 deleted views.

Run-units are programs written in one of the DBMS's languages that use a specific view to access a database. Some DBMSs require that run-units be registered with the DBMS so that only registered run-units are allowed to execute. When run-units are registered, their view is identified. When this is done, any change to the view or the run-unit requires a re-registration process.

The data uses to be protected are reading, updating, or selections. Data reads are for output directly or indirectly through statistical operations on the column, such as a count of



SSNs. Updates to data include changing values to NULLs, NULLs to values, or values to different values. For a static relationship DBMS, changes also include relationship operators such as CONNECT and DISCONNECT. For dynamic relationship DBMSs, relationship changes are changes to columns involved in referential integrity clauses, and relationship operators such as JOIN and UNION.

View instance selection is an important component of any security scheme. Selection is either for updating or retrieval. For example, there might be a request to print the list of qualified candidates for job. To select candidates on the basis of a degree in electrical engineering and years experience of at least 10, the print clause might be:

PRINT EMP.FIRST\_NAME, EMP.LAST\_NAME, EMP.SALARY,

and the select clause might be:

WHERE BACHELOR\_DEGREE EQ BSEE AND START\_DATE LE 01/01/81

If the select clause is not protected from unauthorized modification, then someone in personnel might creatively add sufficient conditions to reduce the number of candidates. For example:

AND MASTER\_DEGREE EQ MSEE

#### 6.8.2.4 System Control

System control itself must be carefully protected. The areas sensitive to misuse include

- Audit trails
- Message processing
- Backup and recovery
- Reorganization
- Concurrent operations
- Security and privacy
- RDBMS installation and maintenance

Audit trail facilities sometimes allow the audit trail to be turned off. If for a large batch update to 50% of the rows, the normal audit trail mode is to capture both a before and after image, there are at least three times as much updating occurring. To reduce the activity's resource consumption considerably, the following sequence is undertaken:

- Backup the database
- Turn off the audit trail facility
- Run the large update job
- Backup the database
- Turn the audit trail facility back on



If such a scenario occurs, then the ability to turn the audit trail on and off must be protected, otherwise the following might occur:

- Turn off the audit trail facility
- Wire a big funds transfer
- Turn the audit trail facility back on

This, along with some of the other security areas identified above, might cause an untraceable transaction.

Security concerns in the area of message processing are also important to consider. All sophisticated DBMSs send security violation messages to a journal file, the computer center's operator console, or the DBA's terminal. If the data file in which these messages are stored can be penetrated, and the message:

SECURITY BREACH ATTEMPTED BY <user-ID>, <date>, <time>

is changed to:

OPERATION SUCCESSFUL <user-ID>, <date>, <time>

then the attempts at security breaches might go un-logged and/or unnoticed altogether.

Database recovery and transaction rollback provides another area of concern. Suppose a user wires a funds transfer, then instigates a rollback that returns the database to the condition prior to the funds transfer, would it be easy to trace the funds transfer? The ability of a user to instigate certain kinds of rollbacks must be protected.

Reorganization is an area that needs to be specially guarded. Logical reorganization causes column, tables, and relationship types and associated subclauses to be added, changed, or deleted. In a static relationship DBMS, the MANDATORY clause could be changed to OPTIONAL. In a dynamic relationship DBMS, the referential action could be changed from SET NULL to CASCADE DELETE, which is no small change.

Physical reorganization usually locks the entire database to reclaim unused space, optimize the indexes, and resort rows into an efficient order. In many respects, physical database is an unload and then a load. Physical database reorganization could well take hours and hours, so its instigation should be well planned and accomplished only by those in authority.

The extent to which a mixture of run-units operate concurrently depends on the types of run-units executing, and the installation parameters established for the DBMS. If an executing DBMS has eight threads with a queue of twenty for each thread, then the operations certainly will be disrupted if the DBMS crashes and is brought back up with only two threads and five users allowed in each thread. Another way to disrupt an environment through a processing slowdown is to start a series of jobs, one in each thread that touched every row.

It should go without saying that the facility by which security and privacy is established should also be very well protected. The areas to be protected are all those discussed in this section.



The final area of system control security concern is DBMS installation and maintenance. DBMS vendors usually provide computer programs to perform various installation activities like applying bug fixes, establishing threads, lengths of queues, storage structure component absolute sizes, and the number and allocation of buffers. Since all these parameters dramatically affect the operation of the DBMS, access to the programs that change these parameters needs to be carefully guarded.

### 6.8.3 Definition Alternatives

There are two main alternatives for the definition of security and privacy: DBMS oriented and user oriented. The DBMS orientation views the DBMS as a fortress that needs perimeters of defenses. SYSTEM 2000 illustrates one example of this, and ADABAS illustrates another. With SYSTEM 2000, each column has three types of security: read, write, and select. A password is assigned to a matrix of these securities for all the columns in the database. Figure 6.31 illustrates the types of securities applied to four columns. The password ABLE provides access to components 1, 2, and 4. For the EMPLOYEE\_ID, the password ABLE enables reading (display) of these columns. However, only the EMPLOYEE\_ID column is used for selection. No permissions are given for updating any of the four columns. The ADABAS orientation to security is to establish concentric rings of ever-increasing power. The outside ring allows total control over all operations. As the rings get smaller, so do the permitted options.

The second alternative, user profiles, is typified by ANSI/SQL. In its implementation, specific privileges to facilities are granted to a specific user. In ANSI/SQL, the basic security privilege commands are GRANT and REVOKE. The ANSI/SQL combines these commands with other components--for example, the commands SELECT, INSERT, and DELETE can be authorized for users such as FORD and SMITH on tables such as EMPLOYEE and PROJECT. Tables exist as either base or derived. A base table is one that is defined to be derived from no other. Thus, a base table holds the rows of data when they are first entered into a database. A derived table, called a view, materializes as a consequence of an operation on one or more base tables. Privileges are granted on either base tables or views (derived). Since a view can contain fewer columns (columns) than a base table from which it is derived, the ANSI/SQL view can be used to provide column (column) security. On the UPDATE and INSERT commands, a column list can be contained within parentheses to narrow the set of columns affected.

COLUMN	PASSWORDS			
	ABLE	BAKER	CHAD	DOG
1 EMPLOYEE ID	r.w	ruwv	r.w.	R.wv
2 FIRST NAME	r.w	ruwv	r.w.	r.wv
3 MIDDLE INITIAL	r.w	ruwv	r.w.	r.wv
4 LAST NAME	r.w	ruwv	r.w.	r.wv





r = output authority  
 u = update authority  
 w = selection for purposes of retrieval authority  
 v = selection for purposes of update authority

**Figure 6.31.** System 2000 Security Illustration

Additionally, a view can have fewer rows (rows) than its parent base table through the use of a WHERE clause. Consequently, ANSI/SQL also provides value based security. When views and select clauses are combined with the security commands, user names, and activity commands, ANSI/SQL contains sufficient security and privacy for most situations.

ANSI/SQL additionally allows privileges to be granted. That is, a user is granted the authority to grant security privileges to others. The process can become quite entangled. It might be well to have all the privileges stored in one database and centrally administered. Examples of ANSI/SQL security are

GRANT SELECT ON TABLE *employee* TO *ford*

GRANT INSERT, DELETE ON TABLE *stats* TO *smith*

GRANT SELECT ON TABLE *myrek* TO *public*

GRANT SELECT, UPDATE (*salary*, *tax*) ON TABLE *stats* TO *nash*

#### 6.8.4 Trapping and Reporting Security Violations

Enforcement of security and privacy is accomplished in a variety of ways. Some facilities are under DBMS control, others are integrated into the operating system. While the latter is more secure, it implies that the DBMS must be an integral component of the hardware vendor's system software.

Whenever a security violation is detected the most secure environments immediately terminate a covered function. Some facilities let the user immediately attempt a restart, while others prohibit restart for a period of time. A most secure facility prohibits the offending run-unit from being restarted until the run-unit is *manually* revalidated.

When a violation occurs, some DBMSs log various information concerning the attempt such as the user-ID, run-unit-ID, date and time, etc. A DBA can then review reports from this security log.

#### 6.8.5 Static and Dynamic Differences

Over the years, the approach to security and privacy has changed from one of syntax clauses contained in a data definition language or clauses providing simple shielding to a security



strategy of defining, deleting, and maintaining user profiles that grant specific privileges on specific database components to specific users. Since the majority of early DBMSs were static, and the majority of recent DBMSs are dynamic, the *older* DBMSs (static) are characterized by the former approach, and the *newer* DBMSs (dynamic) are characterized by the latter approach.

### 6.8.6 Security and Privacy Summary

Security and privacy is very easy to require, easy to formulate, but difficult to implement--effectively. Clearly, the weak link in any DBMS security scheme is the operating system. Through the operating system, the O/S files comprising the database can often be viewed through system editors, etc. Thus, the operating system must enable the DBMS to declare these O/S files secure from any type of access, except the DBMS.

The most useful security and privacy facilities are those that enable a profile to be defined, deleted, and maintained for specific users. Even though ANSI/SQL allows delegation of the granting and revoking of privileges to others, it is clear from the various ANSI/H2 papers on this topic that the delegation capability has problems. It is therefore best to avoid delegating delegations to delegators who. . .

Companies must publish their security and privacy policies in company handbooks and on bulletin boards just like notices of equal employment opportunity, sex harassment, and racial discrimination. Finally, there must be a sophisticated logging capability to capture various types of information about attempted violations. Once captured, the violators must be severely punished. For employees of a company, the penalties could range from personnel review for the first attempt to dismissal for a repeated attempt. For those outside the company, civil and criminal suits must be vigorously pursued. If security and privacy policies are not recorded and enforced, then it is all a farce.

## 6.9 Installation and Maintenance

A sophisticated DBMS vendor provides well developed procedures for the DBMS's installation and maintenance. The activities included in this area are

- Initial installation and testing
- Special version configuration
- Interfacing with the operating environment
- Maintaining DBMS versions and upgrades
- DBMS--Telecommunications Monitor Interface
- Managing DBMS bugs

To help isolate DBMS bugs, the sophisticated DBMS vendor provides well-defined procedures that users can employ to determine whether the problem is with the user's program or with the DBMS.



A final component of any DBMS installation and maintenance activity is the inclusion of well-documented mechanisms for incorporating new versions of the DBMS and for transferring a database in the DBMS's old storage structure format to a new one.

### **6.9.1 Initial Installation and Testing**

To accomplish DBMS installation, there should be complete instructions on its installation. These instructions should include

- Procedures for validating its correct operation
- A complete suite of test programs

With these procedures and test programs, an installation team can invoke these test programs to make sure that the installation was successful.

### **6.9.2 Special Versions**

Since most DBMS installations are large and diverse, a set of well defined procedures for creating special versions of the DBMS that improve the efficiency of certain applications is needed. For example, dramatic changes in application performances can be achieved by manipulating

- The number of buffer pools and their sizes
- DBMS subroutine overlay configurations
- Placement of storage structure components on different computer channels and drives, etc

As might be expected, DBMS changes that dramatically benefit one application often cause degradations to other applications.

When DBMSs are installed on multiple user systems, there are occasions when a special version of the DBMS needs to be created for just one very large job. For example, if an ANSI/SQL database is being transferred from one hardware manufacturer's machine to another, for example, IBM to DEC, the loaded database and the load modules created by the operating system from the application programs are not transferable.

The source statements of the schema, the data (in an unloaded format), and the source language statements of the application programs are transferable. Once these are moved to the DEC environment, the schema has to be redefined to the DBMS resident on DEC. If the application programs are written in COBOL, then certain parts of the application programs (e.g., the environment division) have to be converted. Then, if the DBMS vendor on DEC did not provide a generalized loader program, such a program has to be written.



The alternatives to accomplish the massive data loading required for a complete database transfer include

- A job within the multiple user environment, or
- A specially created single user environment, or
- A single user DBMS load job along with others on the computer, or
- Asking that the computer be reserved for this job alone.

The first alternative does not require a special version of the DBMS. The other three alternatives require DBMS versions specially configured for just one job, with special overlays and special buffer configurations.

For the purposes of this one very large job, the single user environment is preferable as well as exclusive use of the computer. Experience has shown that arranging for the exclusive use of the computer on the weekend, or a holiday, or late into the night is well worth the effort.

Another use of special versions is to remove certain sets of load modules to absolutely prevent an activity. For example, if the modules that permit security and privacy changes are deleted from the run-time copy of the DBMS, it becomes quite difficult to change these access controls.

Whenever special versions are created, procedures are needed for their maintenance, for example, the ability to segregate these versions by different names, or differently named libraries, so that they are invoked and maintained unambiguously.

### 6.9.3 Operating Environment Interfaces

Once installed, DBMSs interface with existing system software. This software includes the operating system, teleprocessing monitors, various access methods, job control languages, sort and print utilities, and the like. The specifications for these interfaces should be provided in sufficient detail that the DBMS's impact is adequately understood and normal everyday operations are carried out.

Many installations consist of combinations of mainframes and PCs, along with their disks and printers. It is not uncommon for an ad hoc query to be formulated on the PC, issued to the mainframe, and the response shipped back to the PC for printing or incorporating into other PC based processes.

Since PC-mainframe connections are quite common, it is important to know how the DBMS vendor handles these connections. If these are not adequately handled, another DBMS site may have accomplished it. Thus, membership in a DBMS vendor's user group often provides answers not available from vendors.



### 6.9.4 DBMS Versions and Upgrades

Throughout the years, DBMS vendors have traditionally issued new releases of the DBMS. On occasion these new releases require the saving of existing databases and restoring these saved databases through the new version of the DBMS in order to change the storage structure of the loaded database. One such change might be to install the count of the instances of a multiple occurrence array (indexing) at the head of the array. Such a change might be needed by a WHERE clause optimizer.

Whenever a new release of the DBMS is issued, the amount of time required for its installation is not just the transfer of a tape to disk. It also includes whatever saving and restoring has to be done to the existing databases. This activity also has to occur to all the archived databases that were created by the DBMS version that is being replaced.

Most DBMS vendors have a policy of repairing problems that are found only in the current released version and *one* prior version. That means if a site chooses not to upgrade as new DBMS versions are released, within one or two years there will be no support from the vendor--even if the maintenance contracts are in force!

If a site retires versions of a database, say the 1977 copy of its accounting data, through a DBMS SAVE, then that retired copy will have to be marched forward whenever a new release of the DBMS is issued to guarantee that the data is still accessible in 1991.

As a final precaution, every site should have one final backup of every database, and that backup should be in source (unloaded) data form. Included in this source form should be the

- Source language versions of the schema and all views
- Data in source format, preferably in third normal form
- The source language form of an unload and a load program

With these three sources, no database is ever permanently stranded. The data should be in third normal form because that forces all the relationships to be value based. If, for example, the relationship is arbitrary, then its loaded order is manifest through a column that serves as a sequence number.

When the DBMS software is re-released, sometimes all the software modules are released and other times only the changed modules are re-released. Accompanying each release, comprehensive documentation should be provided that has step-by-step instructions for modifying existing special versions. After the first installation of a mainframe or mini-computer DBMS, subsequent releases should be able to be installed by customer personnel, rather than DBMS vendor personnel. These releases should be installable in less than a week once the procedures for applying a new release are understood. Greatly assisting the process of installation is the use of a program that guides the installer through the process. This is quite common on PC products. Once installed, this program can be used to change installation parameters.

Between major releases of the DBMS, the vendor often provides temporary fixes to specific problems, often called program temporary fixes (PTFs). Whenever there are interim releases of PTFs, their accompanying documentation should instruct customers which PTFs are mandatory to apply and which are optional to apply.



Vendors of the more sophisticated and complicated DBMSs often offer classes and/or seminars geared to the needs of DBMS software installers and maintainers. The vendors also issue special newsletters and bulletins.

### 6.9.5 TP Monitor Restrictions

Whenever a DBMS is procured, there will always be the requirement to interface with one or more telecommunication monitors. Some DBMS vendors have these monitors as part of the operating system, and other vendors, such as IBM, allow multiple choices. Each of these choices has to be examined to determine the restrictions placed on specific DBMS features and on any multiple-user operations of the DBMS. A most critical issue relating to the interface between the TP monitor and the DBMS is the recovery process. That is, does either have to be brought down during the recovery of the other?

### 6.9.6 Managing DBMS Bugs

Every DBMS has software errors, popularly known as bugs. As an aside, the word *bug* is thought to have originated when one of the earliest computers (late 1940s) stopped working because a wasp got trapped in one of the electro-mechanical gates.

Included in the installed DBMS environment should be a series of vendor supplied test databases that are used to help isolate a bug. With this suite of databases, a member of the DBA staff should be able to duplicate the problem and then report it to the vendor.

A list of known problems should be provided with every release of the DBMS. These problems should be explained so they are understandable to the DBA staff, not just the DBMS vendor staff.

Whenever a bug is isolated at the client site, it should be centrally registered, and called into the vendor along with whatever information is required by the vendor. The vendor should then assign the bug an identification number to track its analysis and resolution.

It is important to keep sufficient information about each bug so that when new releases of the DBMS are received, the DBA staff can determine whether the bugs have been fixed.

### 6.9.7 Installation and Maintenance Summary

The installation and maintenance procedures for a DBMS only seem to get noticed when they are not working well, or when they are used too often because there are a large number of bugs that never get fixed.

It is important to know how to create special versions of the DBMS to achieve specialized performances, accomplish one-time jobs, and exclude unwanted features.

As new releases of the DBMS are issued it is very important to know how long it will really take to install because the DBMS's installation might require saving and restoring all the existing and archived databases, not just copying the load modules to disk.



Finally, since DBMS vendors typically only service the current release and one previous release, it is important to keep current on the DBMS releases because vendor service might only be forthcoming after several weeks to a month of backing-up, installing, and restoring databases.

## **6.10 Application Optimization**

Application optimization is the process of analyzing the operational characteristics of

- Databases
- Computer hardware support
- Systems software support
- The applications software

These analyses enable design modifications that can achieve:

- Performance enhancements
- Greater user satisfaction
- Improved corporate command and control

These performance improvements are achieved a number of different ways.

### **6.10.1 Logical Database Analysis**

The logical database component of a database application is the database's design. There are a number of different areas to review when attempting performance improvements, for example:

- Adding derived data
- Denormalizing database designs
- Centralization versus decentralization
- Column specification

#### **6.10.1.1 Derived Data**

The initial design of a database should not have any derived data, and its tables should all be in at least third normal form. This type of design contains the minimum amount of data, and has the most simple strategy for updating. Figure 6.32 illustrates the design of a database for a very large corporation that tracked product sales and competitor penetration within a specific product set. Within this design there are five main categories of data:

- Corporate headquarters
- Marketing headquarters
- Field unit
- National account



- Account

Within each main data category there is identification and address information, and for the national account and account tables there are subordinate descriptive information. For NATIONAL ACCOUNT, there are:

- Discount Plans
- Key contacts

Within the ACCOUNT there are:

- Service contracts
- Competitive hardware
- Company hardware
- Organizational characteristics
- Product use characteristics
- Detailed sales data
- Shipments

A MAIL CODES table was created to both standardize and centralize the type of mailings that each of the five main tables is to receive. Each of the main tables had a mail code that indicated which set of the mailings were to be received.

This design enabled the organization to know all the competitive hardware for a particular national organization by reviewing the competitive hardware for each of its members (accounts). The organization could know how each of its products is doing in sales by processing all the detailed sales data for each of the accounts. Finally, the organization could track the sales performance of its field units (regions, districts, and representatives) through the relationship between the field units and the accounts.

However, when the number of row accesses were counted for reports that were more than listings of rows by account, the counts went into the millions. The creation of a third normal form database design caused retrieval efficiency to be sacrificed in the name of data non-redundancy and update efficiency. But since the database application was report intensive rather than update intensive, the trade-off was judged not acceptable.

The database design progressed through different redesign scenarios to achieve the maximum retrieval efficiency while still maintaining an acceptable level of update efficiency.

When the first scenario was completed, the redesign looked like the one in Figure 6.33. This redesign achieved a number of changes:

- Recognizing that FIELD UNIT is actually the nest of tables: REGION, DISTRICT, and REPRESENTATIVE. This break out was done partially because the types of information kept at each node were different, and also because the periodicity of the data is different.





- Installing the table COMPETITIVE HARDWARE within NATIONAL ACCOUNT.
- Installing the ADDRESS table and relating it to ACCOUNT.
- Installing HQ ORGANIZATION PERSON and relating it to the CORPORATE HQ table.
- Installing MAIL CODES at each node. Multiple sets of mail codes were needed because each node needed to receive different types of materials.

These changes achieved two different benefits:

- Queries were created more simply as the table dealt with only one topic. As originally designed, only a sophisticated language could be used.
- The number of rows involved in each table was less as the tables were partitioned.

These changes established new tables and also expanded the scope of the data update programs from a few tables to a larger number of tables. The first set of derived data was from a summary set of COMPETITIVE HARDWARE information from the individual account COMPETITIVE HARDWARE table.

These changes did not, however address the millions of rows needed to be accessed to track sales by REGIONS, DISTRICTS, representatives, or by product.

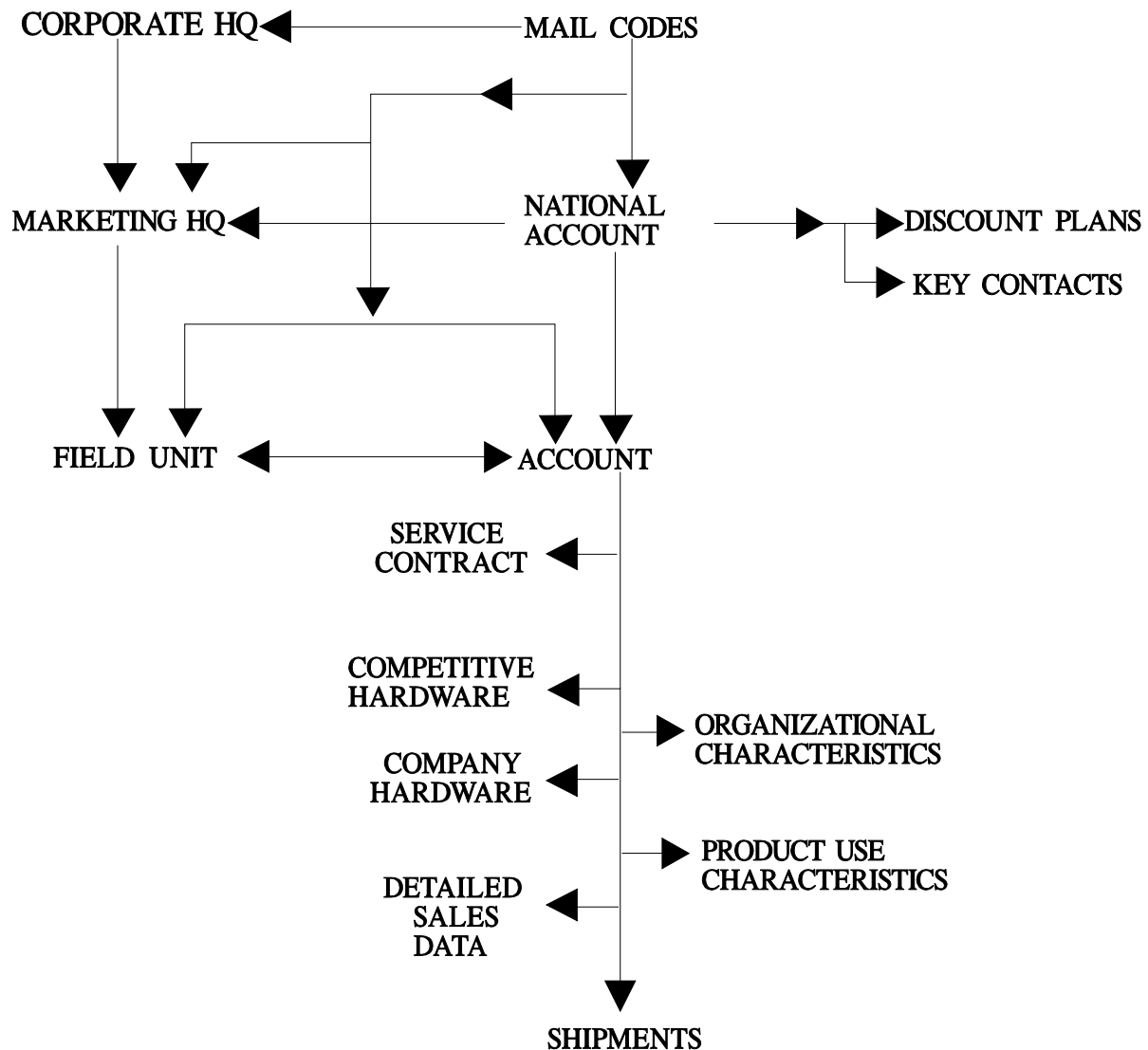
The second major set of database design changes involved the creation of a SUMMARY STATISTICS matrix, which is composed entirely of derived data. These changes are in Figure 6.34, and are used to store a set of data about the sales data across a number of years. Included in this data are yearly, quarterly, and monthly sales by product for sixty months. Since there is a variable number of products, the row is based on product, with the time periods as the columns. The number of columns is 85; that is, 60 for the months, 20 for the quarters, and 5 for the years. The data for these statistics tables initially entered the database as new rows for the DETAILED SALES DATA. After this data is entered, it is sorted different ways, totaled, and appropriately entered in the various SUMMARY STATISTICS tables. The benefit of these changes is that sales progress on a product by product basis can be viewed within a relevant context. That is, within MARKETING HQ, REGION, DISTRICT, or REPRESENTATIVE, etc. Not addressed with this change, however, is the ability to track the progress of a particular product--directly. With the second set of changes, product sales performance tracking became possible, but only within the context of an ACCOUNT or REPRESENTATIVE.

The third design change specifically addressed the ability to track product sales performance. The database's design was required to view the description of data from three different vantage points:

- Account and national account
- Marketing organization
- Product organization

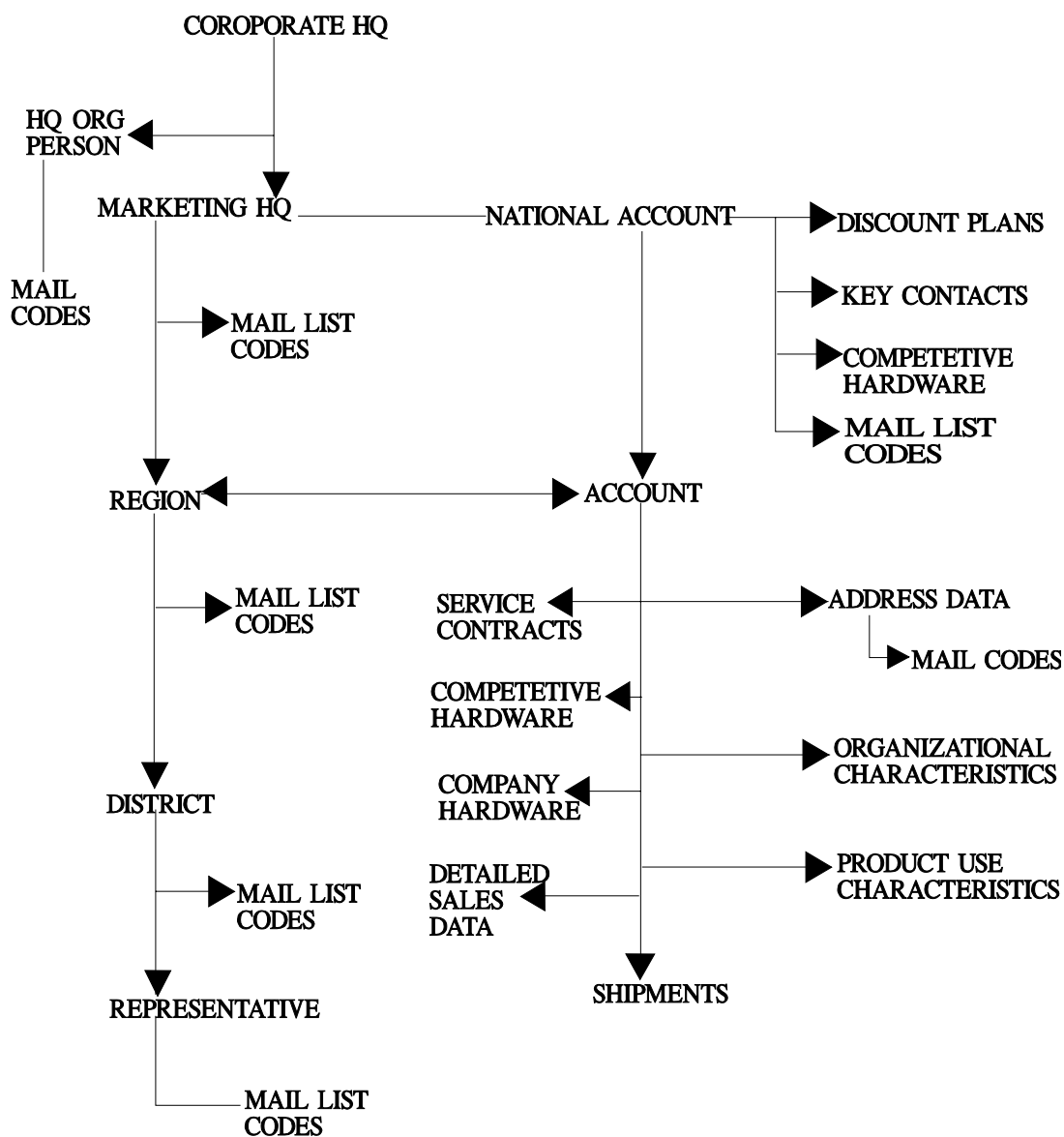


The first two view points are certainly handled by the database design presented in Figure 6.34. What is missing is the recognition of the existence of a third set of *clients* for the database: product developers. The first two clients, marketing and account managers, were handled but the third was forgotten. Serving the needs of this third client required an analysis of its retrieval needs, and the incorporation of the appropriate set of tables. In the analysis of the clients, it became obvious that product is divided into hardware and consumables, and that these two sets of products had to be tracked independently.



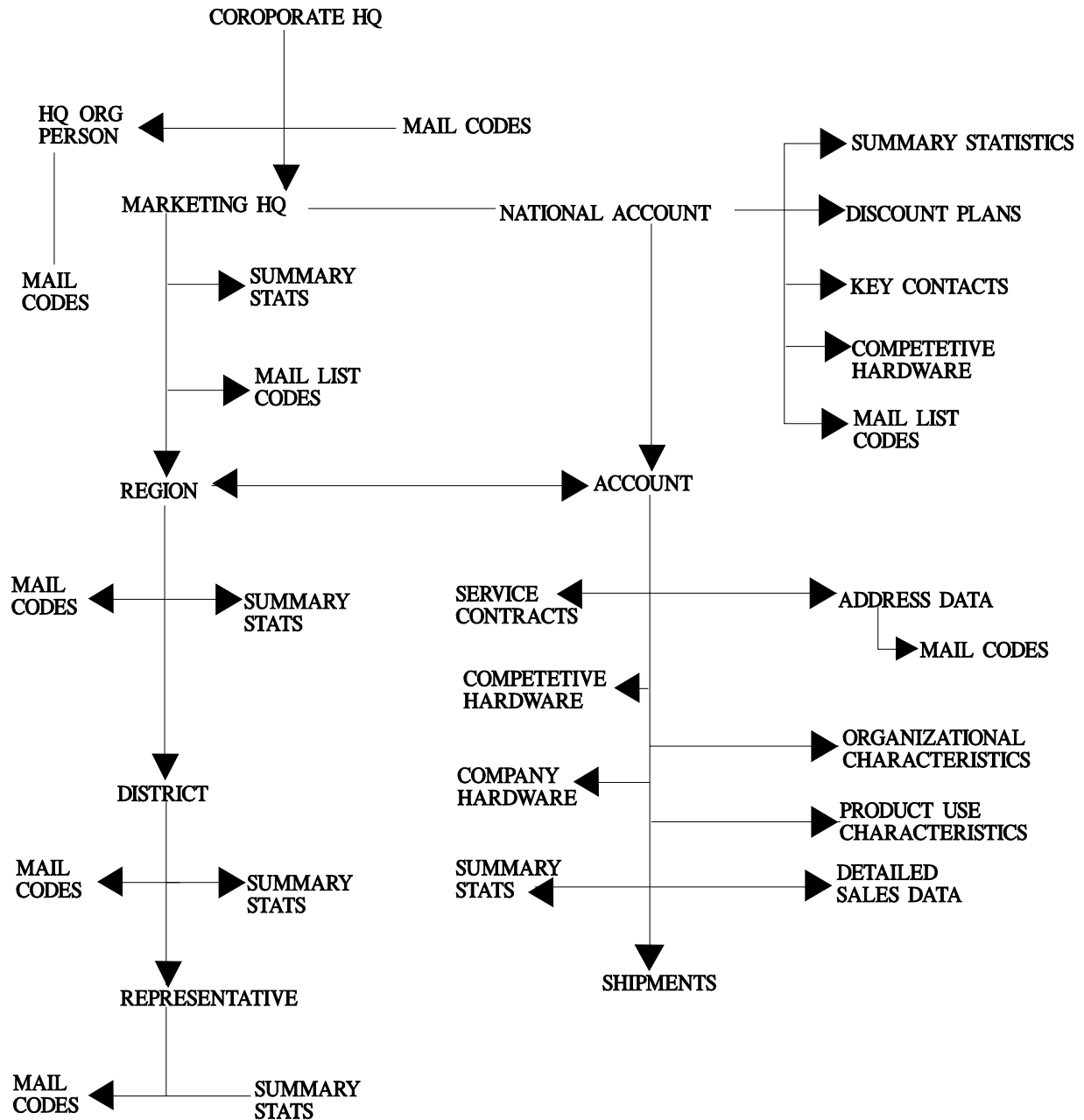
**Figure 6.32.** Sales and marketing database design–initial





**Figure 6.33.** Sales and Marketing database design—second design





**Figure 6.34.** Sales and marketing database design—third design





The set of tables in Figure 6.35 was created within the tree structure: COMPANY PRODUCT. In large measure, all the data contained in this tree is derived as it came from the DETAILED SALES DATA from the prior month.

These three sets of changes to the initial database design solved the majority of the retrieval problems. Managers, product, and market researchers were able to use natural languages efficiently to obtain the majority of the data needed for reports. These solutions were not free, however. The costs were of two types: data redundancy and increased update time. The amount of storage doubled. The updating time increased to the extent that the results of one month's sales were not available until the end of the second week in the following month. Previously, this data (Figure 6.34) was available at the end of the first week after the end of the month.

In summary, whenever a database application is designed and implemented, certain of its functions are apt to run slower than others. Most often, if the database is designed in third normal form, updating has benefitted at the expense of reporting. Bringing a balance between updating and reporting usually requires some level of database redesign of the kind described above.

There are other ways of tuning a database application. That requires a detailed understanding of the DBMS, the database, and the application.

Some DBMSs have special sets of utilities to produce statistics about the performance of their own operations. Included are input/output (I/O) counts, central processing unit (CPU) seconds, overlay swaps, structure navigation, etc. The purpose of these utilities is to learn how the DBMS reacts to a specific application configuration. This information can then be used to optimize the application code that processes the database, the design of the database, and the use of the DBMS.

Assessing the need for database application optimization requires that the DBMS have utilities to assess the performance of the database so as to identify bottlenecks, inefficient file structures, access methods selected, and database usage patterns.

Once a bottleneck is found, it is important that the DBMS be able to assess performance impacts on current applications resulting from proposed database revisions or on changes in application workloads.

### **6.10.1.2 Denormalization**

Normalization is the process of designing table structures that are the most simple. Except for a redundancy of keys, when a database structure is third normal form, the updating process is minimized. There are additional benefits derived from fourth and fifth normal forms, but they are not addressed here. However, when third normal form structures are used for various types of reporting, the number of accesses are maximized. For example, in an order entry application an invoice table is likely have the following columns for each line item:

- Product item number
- Product weight
- Product unit price



- Product quantity ordered
- Extended price
- Extended weight

If this information is placed in an invoice line item table then its primary key is a combination of these three additional columns

- Customer number
- Order date
- Order number within date

If there were 10,000 orders each day and an average of three invoice line items per invoice, then the 120,000 accesses needed to create the invoices are:

- One access to obtain the product information for each product
- One to write the product, product and prices information for each invoice line item
- One to write the before image for recovery
- One to write the after image for recovery

There are also 30,000 accesses to obtain the invoice information to generate the invoices. For this application there are about 150,000 accesses. If the organization wanted to reduce the number of accesses, it might discover, through analysis that there has never been an invoice with more than 8 invoice line items. If the invoice line item were redesigned to include:

- Customer number
- Order date
- Order number within date
- Quantity of invoice line items ordered
- Product item number-1
- Product weight-1
- Product unit price-1
- Product quantity ordered-1
- Extended price-1
- Extended weight-1
- Product item number-2
- Product weight-2
- Product unit price-2
- Product quantity ordered-2
- Extended price-2
- Extended weight-2



through to

- Product item number-8
- Product weight-8
- Product unit price-8
- Product quantity ordered-8
- Extended price-8
- Extended weight-8

Then the number of invoices is greatly reduced. The total number of accesses is 60,000, as follows:

- An average of three accesses to obtain the product information for each product
- One access to write the product, product and prices information for each invoice
- One access to write the before image for recovery
- One access to write the after image for recovery

The number of accesses to print the invoices is an additional 10,000, bringing down the total number of accesses to 70,000 rather than 150,000.

This redesign process achieves significant performance improvements, but at the cost of denormalization. Because of the denormalization process the following problems arise:

- Reduced ability to use natural languages for simple queries and reports
- Having to add an additional invoice when there are orders having more than eight invoice order items
- The cost of having space for eight order items for each order when there is--on average--only three
- The additional programming logic that has to be installed to process a specific invoice line item

In short, the saved 80,000 accesses are not without cost. All the problems listed above have to be identified, enumerated, and analyzed to determine the real savings. If after the analysis there is still a net savings, then denormalization is beneficial to the organization.

### **6.10.1.3 Centralization Versus Decentralization**

One of the largest factors in database application failure is the quest to build THE database. It is far better to design and build a database that has a too narrow scope than one that cannot ever be successful. In database, success is principally achieved through consensus and compromise. As the scope enlarges, the probability of consensus decreases. There are too many conflicts in the





critical policies required for database success. Thus, it is better to restrict a database's scope to a well bounded area of corporate policy, for example, product engineering within one class of product lines. To attempt a single database across very different product lines is to court failure.

In the example contained at the beginning of this section, the product manager user was ignored in the drafting of the original database design. Until the needed reports were analyzed, the required changes in database design were not uncovered. The database project methodology must clearly identify all the relevant users, and capture their information needs in a comprehensive and correct manner.

#### 6.10.1.4 Column Specification

Column specification is often over done. Database principles must apply to database application specification and building. That is, each fact must be defined only once.

A good example of over specification is telephone number. While there might be a home telephone number, an office telephone number, and a mobile telephone number, must all three be defined, or should telephone number be defined only once and then employed whenever relevant? There needs to be only one definition, not three. However, there are three columns: HOME TELEPHONE NUMBER, OFFICE TELEPHONE NUMBER, and MOBILE TELEPHONE NUMBER.

However, when columns have different semantics, they are different columns and their definitions must reflect those differences. For example, *order date*, *shipping date*, *invoice date*, and *payment due date* are really four different columns, each requiring a very precise definition because corporate policy behind each is different and the difference is critical to database applications.

Finally, column names should not be bounded by any artificial restrictions such as 10 characters, etc. Rather, they should be sufficient to convey to all knowledgeable users the same meaning.

#### 6.10.1.5 Relationships

Relationships among sets of rows are methods of accessing specific sets faster than serial searching. In a dynamic relationship DBMS, relationships are implemented through indexes and are always value based. In static relationship DBMSs, relationships are value based or arbitrary, and are implemented through DBMS generated relationship mechanisms.

Establishing a relationship should occur only after a cost benefit analysis. To determine whether a relationship is cost effective, its creation and maintenance costs must be balanced against the savings derived from its use. The savings are the differences between the cost of finding a set of rows using a serial search versus using the relationship. The total savings are the savings for one use of the relationship multiplied by the frequency of its use. When the savings exceed the cost of creation and maintenance then the relationship is cost effective.

When there are too few relationships, performance improvements are achieved by creating new relationships. While the load and update program costs rise, the cost of retrievals,



and the cost to find rows for the purpose of updating, lessen by a greater amount. Users of the update programs will have to be alerted that these programs take longer to execute.

Incorporating new relationships in the database does not automatically cause them to be used. In static relationship DBMSs, the language clauses within views that take advantage of relationships have to be changed. For example, if there is a serial search programmed into the view, that search language has to be changed to retrieve the rows according to the mechanisms provided by the relationship instances. In dynamic relationship DBMSs, the view has to be changed to select rows according to just one indexed column rather than having that column contained in a list of columns in the WHERE clause.

When there are too many relationships, that is, when the cost of creation and maintenance exceeds the access cost savings, then the relationship mechanisms have to be removed. In the case of static relationships, the programs that AUTOMATICALLY insert rows into relationships will not have to be changed because the operation is DBMS controlled and AUTOMATIC. If the mechanism is MANUAL, then the update programs have to be changed, otherwise the program logic will fail because the relationship mechanism is not present. All the retrieval views will have to be changed because they will attempt to use a facility that has been removed. In its place, serial search mechanisms have to be installed.

For dynamic relationship DBMSs that employ indexes, update programs do not have to be changed as indexes are usually automatically updated with every row update. The views from the retrieval programs have to be changed to install the specially required logic to find the rows without the aid of the relationship.

When the relationships are too transitive, the volume of accesses can be too high. For example, to find a particular type of customer within a particular territory within a particular district within a particular region requires the following sequence of processing:

- Process the relationship region to find the particular region
- Process the various districts within the region to find the particular district
- Process the various territories within the district to find the particular territory
- Process the various customers within the territory to find the particular type of customer

If that processing occurs often, the relationships are said to be too transitive. All those relationship instances have to be processed to find the particular customer belonging to the required customer type. Rather, make the relationship more direct by placing all the customers of a certain type within a relationship of customer types.

As shown in Chapter 4, the mechanism of relationship processing differs widely between static and dynamic relationship DBMSs. In general, static relationship DBMSs process relationships much faster than do dynamic relationship DBMSs. Because of that significant performance difference, it should always be an alternative to consider changing DBMSs when all other performance improvement measures have been exhausted.

When an application is *young*, its design is more likely to change than when an application is *mature*. Dynamic relationship DBMSs enable relationship type changes much more easily and quickly than do static relationship DBMSs. Thus, it is almost always much *safer* to implement a database application with a dynamic relationship DBMS than with a static



relationship DBMS. As the application matures, the quantity of relationship changes between and among tables lessens, and so does the need to make these changes. When the application quiesces with respect to relationship type changes, the benefits of having a dynamic relationship DBMS disappear. Reimplementing the application with a static relationship DBMS at that time makes sense as the application will almost always run faster.

#### 6.10.1.6 Model Transformation

Whenever a database is initially designed, it should be in third normal form. Once designed, it can be implemented relationally, or with an ILF or a network data model DBMS, and possibly even a hierarchical data model DBMS. Under relational all the relationships must be dynamic. With ILF, hierarchical, and network, some of the relationships are static and others dynamic. As the number of static relationships increases, the flexibility for relationship specification decreases. But, as the number of static relationships increases the performance associated with relationship processing increases. A database is transformed too much when it is both *specified* and *implemented* relationally even though the dynamic relationship benefits available--inherently--by the relational data model are not required by the application. Rather, what is required is implementation through one of the other three data models to accomplish performance requirements. A database that is under transformed is one that is specified relationally but is implemented through a static relationship data model DBMS in spite of needing the relationship specification and change benefit.

An important component of transformation is the ability to represent multiple third normal form tables within one complex table. For example, in a personnel application there might be an EMPLOYEE table that has a series of subordinate tables for

- Telephone numbers
- Skills and abilities
- Degrees, institutions, and dates
- Sick and annual leave transactions

If its third normal design is implemented relationally then the four sets of data subordinate to EMPLOYEE have to be implemented as four different tables. In contrast, if the DBMS supports complex tables then these four types of data can be implemented as one multiple valued column and three different repeating groups. Assuming that the candidate DBMS allows this data to be acceptably accessed through the natural languages (POL, query, and report writer) then it becomes difficult not to justify the use of complex tables in this case. Complex tables typically provide many of the static relationship DBMS benefits with the flexibility of dynamic relationships across different tables.

The current ANSI/SQL language (SQL 1986 or SQL 1992) cannot, however, comprehend complex tables. To overcome this, the DBMS vendor has to provide mapping between two dimensional tables and these more complex structures. Such a capability could be provided through the use of views where the user perceives a two dimensional table, and the



SQL view language maps the view to an underlying set of base tables, where one base table is the apex of a set of tables, and the nested repeating groups are represented as underlying joins.

If only a relational DBMS is available then the database's design becomes overly fractured. To bring the database design back into an understandable scheme, denormalization might be the only solution. That is less preferable, however, than to change to an ILF data model DBMS that allows complex tables.

#### **6.10.1.7 Views**

Views have one main value: they shield the application program from having to know the data model of the DBMS. If the view allows inclusion of the WHERE clause in the view's definition, the view facility additionally allows for a set of security that is not normally available from many DBMSs.

Views are defined at either extreme, one view for all, or a view for each. To retain complete control over the user-database interface, there should be a one-to-one relationship between the view and its run-units, and between the run-units and its users. In such a situation the database administrator, who presumably is the one who defines the views, could well become a bottleneck. Thousands of views might have to be created, maintained, and administered in a short amount of time.

The other extreme is to have one or a few views serve all the different users, run-units and applications. That alternative is undesirable because there is too little accountability and security.

A preferred approach is to allow decentralization in the authority to define views, and to allow wherever possible the users' inclusion of their own WHERE clauses. These WHERE clauses are in addition to those in the nested view logic that serve the needs of value based security. Without the ability to include WHERE clauses in the run units, the number of views grows with no real justification.

### **6.10.2 Physical Database Analysis**

Database application performance can be improved without having to change the design of the application. The changes that are accomplished address the database's

- Storage structure
- Data loading
- Data updating
- Database maintenance

#### **6.10.2.1 Storage Structure**

Changes in the database's storage structure can affect its



- Dictionary
- Indexes
- Relationships
- Data

#### **6.10.2.1.1 Dictionary**

The dictionary typically contains the compiled schema, views, and other dictionary oriented information such as edit tables. Some DBMSs allow multiple schemas--one at a time--to control the operation of a given database. Performance improvements are achieved if different schemas are allowed to operate for different--exclusive use--jobs. For example, if a large volume of data is to be loaded, the presence of an extensive set of editing and validation rules exercised by the DBMS will consume significant computer time when compared to the same job without those rules. To have fast, large volume data loads requires external data editing and validation.

If the DBMS has an option to have multiple schemas, great care should be exercised in their use lest an uncontrolled schema remain operational after the data loading job is completed and the on-line access job queues have been reopened.

Alternative view constructions are also a source of performance improvements. The ANSI/SQL language provides several different techniques for accomplishing the same task. If each of these different techniques is encoded in the select statement contained in a view, then the different views are likely to perform differently.

#### **6.10.2.1.2 Index Design**

If a DBMS's DBA can choose among different index construction techniques, then an application might benefit from an alternative index design. For example, if the primary key column is solely for the purposes of unique access, then an index construction that enables both unique access and range searching is of marginal value. A hierarchical index construction for the unique value portion serves no purpose. It is better to switch to a hash/random unique value portion.

If an application's analysis shows that four secondary indexes are always used in combination to find a unique row, then combining those values for the development of a candidate key (unique values guaranteed) certainly enhances performance. This change, however, requires changes in the views servicing the application programs.

An analysis of applications might show that many rows are being serially searched even though an index is used. An index cost benefit analysis should be performed to determine whether installation of another index is appropriate. This change also requires changes to the views servicing the application programs.



### **6.10.2.1.3 Relationships**

The only relationships that can be changed without affecting run-units are the relationships that are stored as separated pointers. The only changes possible in this would be to change the relationship file's blocking factor or physical device location.

Other types of relationship changes such as reducing the length of chains by introducing intermediate tables all require database design and application program changes.

### **6.10.2.1.4 Data**

There are a number of changes that affect the rows. These changes include:

- Choosing fixed or variable formats
- Choosing fixed or variable length rows
- Clustering logically related data onto the same DBMS row instance
- Changing the blocking factors on the O/S files that store the rows

### **6.10.2.2 Access Strategy**

Access strategy changes include many areas. Changes to the use of the dictionary, indexes, relationships, and data all affect the actual access strategy that is employed. The following are but a few of the access strategy changes that affect performance.

- If a schema table is assigned to different DBMS-required O/S files, there might be a difference in the locking mechanisms that can be installed.
- If multiple schema tables are assigned to the same DBMS-required O/S file, then rows from different tables are clustered to reduce the I/Os for certain retrievals.
- If a DBMS-required O/S file is assigned to only one memory buffer, then conflicts in the use of that memory buffer can be prevented.
- If a DBMS-required O/S file is assigned to multiple memory buffers, then the DBMS rows from these files can occupy more of the buffers than is otherwise possible, increasing the application's performance.
- If the instances from a schema table are stored on different channels, there can be overlapping I/Os, thus increasing performance.



### **6.10.2.3 Data Loading**

Changes in data loading affect performance. If an edit and validation intensive, large database load is in progress, all on-line programs will certainly suffer. Performance improvements are realized if those data loading efforts are isolated to just the night hours, or by keeping the on-line queues locked out during these loads so that already validated data is speed-loaded without the slowing effects of editing and validation.

If the DBMS supports an option that allows index adjustment at the end of a job rather than at the end of each transaction, performance can be greatly improved if all index maintenance is performed after all rows are loaded, rather than after each row is loaded.

### **6.10.2.4 Data Updates**

When a database is initially loaded, its organization is probably optimal. As updates occur, new rows are added, deleted, and modified, causing adjustments to the database's dictionary, indexes, relationships, and data. In short, the database's physical organization becomes different from its logical organization. As this difference increases, performance decreases, leading to a needed database reorganization.

A set of standard updates can be created and run periodically so that the degradation in the database's performance is tracked. At the point where the cost of degraded performance over a period of time exceeds the cost of reorganization, then it is time to reorganize.

### **6.10.2.5 Database Maintenance**

Database backups are an integral cost of data processing. If there has not been a crash of a database environment for a long period of time, it is tempting to reduce the frequency of backups. A better approach is to vary the time between backups to reflect the immediate need for recovery. If during the month there are only ad hoc updates then maybe the backups could be every week or two. But towards the end of the month there may be a large number of updates reflecting end of month processing. During that critical period several backups might have to be performed in order to keep the recovery costs in line.

## **6.10.3 Interrogation**

Natural languages offer an allure that is hard to resist: quick development and maintenance. This benefit is not without cost. If an entire organization develops hundreds to thousands of these natural language based run-units and if this language executes interpretively, then the overall performance of database applications will degrade rapidly in contrast to the same set of application run-units developed through a compiler language.



Natural languages are ideally suited for application run-units that have a short life span, or for components of an application that are not yet fixed in design.

For those applications that have a long life span it is an ideal strategy to develop these run-units with a natural language to take advantage of the quick development and easy maintenance, and then translate them to a compiler based language over time when the program design changes become lessened.

The cost of a run-unit consists of four parts:

- Cost of design
- Cost of development
- Cost of operation
- Cost of maintenance

Natural languages can dramatically affect the second and fourth components. As an application's life-span becomes longer the effect of the cost of development and maintenance becomes less in comparison to the cost of operation.

The cost of operation for an individual run-unit is basically the cost of a single execution times the frequency of execution. The cost of execution includes a number of factors, including:

- Data selection clauses
- Navigation logic
- Update effects on indexed columns
- The modification of relationships binding rows

Each of these four factors should be examined individually and then in combinations. For example, there are often a number of different ways to conform a select clause that employs different types of relationship logic while achieving the same effect. All the computed performances, however, will be different depending on the quantity of indexed columns.

The DBA group within each organization should maintain test databases that are of different sizes and designs. These databases can be used to test different combinations of select clauses, index combinations, and navigation logic.

#### **6.10.4 System Control**

System control performance assessment and improvements generally revolve around using or not using a particular facility.

##### **6.10.4.1 Audit Trails**

Audit trails identify the types of transactions that are captured. To improve a DBMS's performance only the update requests should be journaled. While such a change will improve performance, lost will be knowledge of when and for whom reporting is accomplished.





Performance can be improved also by disabling journaling during a large data loading program. If the job fails then the job would be rerun from a database backup. That benefit would require that the database be exclusively used by the large data loading program.

#### **6.10.4.2 Backup and Recovery**

Backup and recovery costs can be affected by changing the method employed. If backups are performed only once every two weeks and if only after images are kept, then the recovery preparation cost allocated to the individual update transaction is lower than if the backup is twice a week and both before and after transaction images are captured. The cost of recovery, however, is much greater in the first scenario than in the second. If the number of update transactions is low, then the overall cost of the first scenario might be lower, all factors considered. The goal is to determine the most cost effective risk scenario.

#### **6.10.4.3 Reorganization**

Reorganization can dramatically affect overall database performances. If an index is badly disorganized the cost of reorganization is many times overshadowed by the savings realized in interrogation. The rows organization is affected through continued updates. If the access strategy is hash/random, and the distribution of the rows is by primary key value, the overall performance can fall dramatically as the volume of adds and deletes increases. If charts are carefully plotted, the DBA group can usually predict when database reorganizations should be performed. A sophisticated DBMS should have utilities to assist in determining the most opportune time to perform physical database reorganization.

#### **6.10.4.4 Security and Privacy**

Security and privacy is an important application optimization issue. Passwords should be centrally administered and changed on a regular basis.

While there is probably not a direct link between database efficiency and the quantity of passwords, there is likely to be real organizational efficiency in keeping these passwords well organized and centrally administered.

#### **6.10.4.5 Multiple Database Processing**

Multiple database processing generally provides faster access to the users of multiple, smaller databases than to the same users of a combined database. The three costs of multiple database processing are:

- The cost of the cross processing itself



- The cost of multiple database locks when a single logical update affects multiple databases
- The cost of maintaining more than one copy of some data

If these costs exceed the savings received by the users then the databases should be combined.

#### **6.10.4.6 Concurrent Operations**

The specification of the level of concurrency can affect database application costs. Locks are typically on the following levels:

- Row (sometimes called row locking)
- DBMS row instance
- Table (that is, all its rows)
- O/S file
- Database

The cost of concurrency is examined with two cases. The first is the update whose select clause finds just one row to change. The second is the update that finds thousands of rows to change.

If concurrency is maintained at the row level, then for the first case, only one lock has to be posted. Under the second case, several thousand locks have to be posted.

If locking is at the DBMS row instance level, then in the first case, while only one lock is posted, rows from one or more tables are locked. Under the second case, if there are 10 rows per DBMS row then possibly 10 times fewer locks have to be posted.

If locking is at the O/S file level, then in the first case only one lock has to be posted. Under the second case only one lock has to be posted, assuming all the rows are stored in the same O/S file.

If locking level is at the database level, then in either case only one lock has to be posted.

In all the cases, if the table includes indexed columns, then locks have to be posted on the index storage structure components as well.

Posting is only one part of the cost of concurrent operations. When the next update request arrives, the rows required by the second request have to be identified and all their locks attempted. If any of the locks from the first request conflict with the second set of locks then a decision has to be made whether to reject the second transaction, hold it in the queue, or partially process it. The effort to determine whether there are conflicts is easiest at the database and O/S level, and increasingly expensive as the number of locks to examine increases. The cost of examining locks increases as the number of update transactions grows and as the granularity of the locks becomes finer (DBMS row, table, or row).

Careful tests have to be constructed to determine the costs (lock processing overhead) of concurrent operations as contrasted to the benefits (concurrent processing). These tests will produce different results depending upon the DBMS, the database, and the types of concurrent operations desired. When all these tests are performed, the tradeoff analysis can begin. For some



databases and applications the most cost effective locking level might be the row, while for other applications the locking level might be the DBMS row or even the O/S file.

#### **6.10.4.7 DBMS Installation and Maintenance**

Applications can be optimized by manipulating the configuration of the DBMS itself. The following are keys to DBMS performance:

- Whether the DBMS operates heavily overlaid, or flat
- Whether the DBMS's transactions receive appropriate priorities within the teleprocessing queues
- Whether there are sufficient numbers of appropriately sized buffers
- Whether the number, size and blocking factors of the various work files can be affected

#### **6.10.5 Static and Dynamic Differences**

As relational DBMSs attempt larger and more sophisticated production applications, their storage structures are becoming more complicated. Complicated storage structures are a fact of life for almost all the static relationship DBMSs, and many of the ILF data model dynamic relationship DBMSs.

The benefits of sophisticated and complicated storage structures are the performance improvements they bring to large database applications. However, these performance improvements typically require full time attention to database design, application configuration, and DBMS tuning. Many issues of database journals present articles that discuss tuning.

There used to be a great difference between static and dynamic relationship DBMSs in the area of application optimization. For static relationship DBMSs, tuning is a way of life for several professionals. For dynamic relationship DBMSs, the process is simple: there were no options. Now the options for tuning within dynamic relationship DBMSs are increasing dramatically, and that is supposed to represent progress.

#### **6.10.6 Application Optimization Summary**

If a database is designed in third normal form and there is only one user connected to an infinite size computer of blinding speed, there is no need for application optimization. Since the converse is true with respect to computer sizes and speed, tuning is a necessary fact of life. Accomplishing application optimization with the least disruption to existing applications is preferred. The following is a suggested sequence for attacking application optimization:



- Review and change the facilities that only affect the configuration of the DBMS.
- Review and change the database's storage structures for reblocking.
- Review and change the various system control facilities.
- Review and change the indexes, relationships, etc. and incorporate those changes into views.
- Review and change the database design through derived data and/or denormalization.

### 6.11 System Control Static and Dynamic Differences

The main difference between static and dynamic relationship DBMSs in the area of system control (see Figure 6.1) relates to the domain of the effects of the system control facilities. In a dynamic relationship DBMS, the domain is usually the table, while in a static environment, the domain is usually many interrelated tables.

Since most system control operations affect a single database, a dynamic logical table reorganization is possible without physically affecting other tables. In a static database environment, a logical table reorganization is likely to require the entire database operation to stop until the reorganization is completed.

In short, the principal difference between static and dynamic system control is the span of the various system control operations. The functions most affected by the difference are audit trails, backup and recovery, reorganization, and multiple database processing.

For example, if there is an operating system failure during the time an on-line, multi-user update DBMS environment is operating, a database recovery event recovers all databases that are active under a particular copy multi-user DBMS. In a static environment, all the tables in all the active databases are recovered.

In a dynamic environment, the recovery might neither be as effective nor as complete as the static environment. For example, suppose all tables are backed up over a weekend, and updates occurred on Monday and Tuesday to some dynamic tables. On Wednesday, these updated tables were removed from the on-line environment. A failure happens on Thursday. If the removed tables are not on-line, then the Monday recovery cannot occur against those off-line tables. In such a case, the DBMS has to issue a message indicating that the tables are missing from the on-line environment, and until they are brought on-line, the recovery cannot be completed.

### 6.12 System Control Summary

Even though the areas of system control have been presented separately, they almost always interact. For example, application optimization tests might indicate that the application is



slowing. In response, physical database reorganization may be invoked to bring the database back into optimum condition. To accomplish database reorganization, the concurrent operations capability of EXCLUSIVE USE must be invoked. If the security and privacy facility permits user profiles with specification of database operational verbs, then only the DBA should be allowed to activate the REORGANIZE verb.

With respect to effects on database application design, system control is probably the second most critical DBMS function. The first is static or dynamic relationships. Because of the DBMS differences in system control capabilities, an in depth assessment of each DBMS and each application must be accomplished to determine the difference between DBMS capability and application need. If the match is good, then the amount of extra database application programming to overcome DBMS capability short-falls is small. Otherwise, it is large. Remember the risks of not having system control, which restated are:

- Not having audit trails means that there is no accountability for database updates, reports, and other types of use.
- Not having sophisticated message processing will not stop errors. Rather users will become frustrated, and stop using the system.
- Not having backup and recovery results in total database loss as important update transactions are lost and critical databases are not recoverable after crashes.
- Not having reorganization ultimately causes the database to become antiquated and *freeze-up*. This is due to structural inefficiencies or because the database does not contain the right columns, relationships, and tables.
- Not having security and privacy enables valuable corporate data to be stolen.
- Not having multiple database processing causes a redundancy of facts, which in turn causes errors through time, and an eventual break down in the usefulness of database projects.
- Not having concurrent operations capabilities and guidelines will--at some time--prevent the generation of an extremely important report because of some trivial, but database locking, operation.
- Not having database application optimization causes the expenditures for hardware to increase as a consequence of unimproved database designs, inefficient physical structures, and inappropriate DBMS configurations.
- Not having good installation and maintenance procedures eventually causes special software libraries to be lost, different versions of modules to be linked together, and databases to be lost due to corruptions caused by misconnected DBMS modules.



*Although time consuming and seemingly nonproductive, the only thing more expensive than good system control is not having system control.*



## Index

Access control	31
Access languages	38, 259, 261, 279, 283, 328
Access method	236
Access path	108, 166
Access Strategy	44, 51, 163, 164, 179, 180, 186, 192, 204, 235, 237, 238, 240, 242, 244-248, 252, 257, 296, 390, 422, 425
ACM	34, 35, 57
After image recovery	358
Alias	159, 265
American National Standards Institute	iii, 1
ANSI	ii, iii, xxi, xxii, 1-5, 11-15, 18-23, 25, 26, 30-35, 46-51, 56, 57, 70, 79, 82, 85, 95, 99, 105-107, 133, 135-139, 150, 156, 157, 161, 163, 166, 236, 245, 255, 258, 259, 261, 263, 272, 275, 285-288, 291, 292, 301, 304, 305, 311, 312, 316, 318, 319, 328, 329, 332, 337, 364, 371, 373, 388, 393, 400-403, 419, 421
ANSI SQL	iii, 291
ANSI/NDL	xxi, 14, 15, 18, 20-23, 32, 33, 50, 56, 70, 82, 95, 105, 106, 133, 136-138, 157, 161, 236, 255, 275, 287, 305, 371, 388, 393
Application optimization	49, 52, 53, 339-342, 407, 414, 425, 427-429
Architecture	ii, 2-4, 11, 30, 163
Assertion	3, 39
Audit trail	47, 283, 284, 342-344, 352, 383, 392, 396, 398, 399
Audit Trails	xxi, 37, 41, 42, 49, 52, 55, 56, 269, 283, 284, 308, 327, 332, 336, 339-344, 392, 393, 398, 424, 428, 429
Backout	306
Backup and Recovery	37, 42, 49, 52, 53, 55, 56, 256, 283, 284, 307, 308, 327, 328, 332, 339, 341, 352, 364, 367, 368, 392, 393, 398, 425, 428, 429
Backus naur form	20, 214
Benchmark	73, 296
Binding	5, 25, 27, 81, 82, 94, 109, 125, 126, 134, 160, 212, 213, 245, 278, 365, 424
Blocking factor	175, 187, 188, 191, 235, 236, 422
BNF	20, 214, 215, 217-231
Boolean	27, 191, 245, 267-269, 295, 333, 334
Buffer management	247, 390
Buffers	56, 247, 248, 365, 390, 400, 422, 427
Business Event	279
Business Organizations	232
Business Policy	60
Calc	181, 182, 184, 185, 187, 189, 191, 235, 238
Call Level Interface	28
Candidate key	68, 72, 73, 125, 166, 201, 421
Centralized Semantics	299
Chain	96, 97, 104, 195, 197, 200, 208, 309, 364
Check clauses	26, 55, 78
Checkpoint	354, 357, 358, 360, 365, 368, 393



COBOL . . .	iii, 12-14, 17-19, 25, 37, 38, 40, 42, 44, 46, 48, 49, 52, 59, 156, 157, 226, 259, 261-263, 265, 266, 269, 270, 277, 288, 290, 292, 297, 299, 310, 311, 331, 337, 381, 403
CODASYL . . .	xxii, 1, 13, 17-20, 50, 70, 138, 139, 151, 238, 253, 289, 292, 304, 366, 367, 372, 393
Column . .	ii, xxi, 7, 8, 15, 16, 20, 26, 30, 39, 40, 42, 47, 50, 51, 53, 55, 59-69, 71-74, 76, 78, 88, 90, 94, 95, 97, 98, 101, 104, 105, 108, 111, 126, 129, 139-142, 145, 147, 152-156, 161, 164-166, 169-173, 175, 177, 178, 180-183, 186, 189-191, 193, 201, 211, 214, 217, 224-234, 236-239, 242, 243, 245, 246, 251-256, 263, 266, 267, 271, 275, 287, 295-297, 305, 306, 309, 312, 316, 319-321, 325, 329, 333, 339, 345, 366, 369-371, 373, 374, 388, 395-397, 399, 400, 405, 407, 417-419, 421
Conceptual schema . . . . .	2-4
Concurrency . . . . .	4, 375, 376, 379, 426
Concurrent Operations . . .	37, 49, 52, 53, 283, 285, 308, 327, 332, 339-342, 375, 376, 378, 379, 381, 383, 384, 392, 393, 398, 426, 429
Contract . . . . .	98, 102, 104, 105, 148, 205, 206, 246, 251, 280, 313, 314, 330, 334, 363
Cursor . . . . .	27, 300-305
Data administration . . . . .	30
Data Architecture . . . . .	3, 11
Data Conversion . . . . .	265
Data dictionary . . . . .	8, 31, 165, 166, 239, 352
Data element . . . . .	30, 31, 69, 213
Data Elements . . . . .	69
Data file . . . . .	102, 165, 212, 326, 399
Data integrity . . . . .	68, 156, 157, 306
Data integrity rule . . . . .	157, 306
Data Loading . . . . .	37, 41, 42, 44, 46, 49, 103, 163-165, 194, 240, 245, 248-251, 255, 257, 281, 337, 347, 390, 395, 396, 404, 420, 421, 423, 425
Data Models . . . . .	ii, xxi, 4, 8, 11-15, 49, 51, 56, 59, 125-127, 130-133, 136-138, 144, 149-152, 160, 161, 163, 232, 252, 259-261, 289, 291, 300, 419
Data Record Type . . . . .	ii
Data Semantics . . . . .	1
Data Structure . . . . .	126, 128, 138, 152, 198, 200, 243, 261, 270, 299
Data type . . . . .	27, 55, 61, 62, 64, 65, 74, 152, 155, 265, 293, 371
Data Update Subsystem . . . . .	44
Database Maintenance . . . . .	6, 163, 256, 257, 420, 423
Database Management System . . . . .	1, 2, 18, 25
Database Project . . . . .	55, 282, 417
Database view . . . . .	266
DBCS . . . . .	13, 14, 17, 20
DBKEY . . . . .	181, 187-191, 302, 366
DBMS . . . . .	ii, iii, xxi, xxii, 1, 2, 4-6, 10-15, 17, 18, 20-23, 28, 33, 34, 37, 39-43, 45-47, 49-56, 59, 64, 67-69, 73, 82, 85, 90, 91, 94-96, 98, 99, 102-107, 109, 125-127, 129, 130, 133, 134, 136-138, 145, 150, 151, 155, 156, 160, 161, 163-166, 168, 170, 172, 173,





- 
- 175, 177-179, 181, 183-194, 197, 201-204, 207-210, 212-214, 217, 219-226, 228,  
230, 232, 234-266, 269, 270, 278, 279, 283-289, 291, 292, 295-299, 301, 305-312,  
315, 316, 319, 325-329, 332-334, 336-346, 348, 350-355, 357, 360, 362, 364-369,  
371-407, 414, 417-423, 425-429
- DBMS table ..... 69
- DBMS view ..... 264
- DBTG ..... 17
- Deadly embrace ..... 286, 287, 383
- Derived data ..... 317, 325, 331, 336, 407, 409, 428
- Distributed database ..... 4
- Division ... 43, 52, 83, 84, 129, 139, 145, 175, 176, 187, 188, 288, 290, 298, 386, 388, 389, 403
- Document ..... iii, 5, 6, 13, 17, 19, 20, 30, 31
- Domain ..... 60-63, 74, 76, 149, 340, 360, 374, 428
- dpANS ..... 19, 20
- DSDL ..... 163, 215, 218-221, 223, 235-237, 239, 252-254
- Dynamic Data Model ..... 114, 130, 149, 297
- Enterprise ..... 2
- Exclusive control ..... 364, 393
- Extent ..... ii, 20, 41, 395, 396, 399, 414
- File .. ii, 12, 14, 17, 31, 42, 43, 45, 46, 50, 51, 56, 59, 96, 99, 102, 104, 127, 128, 130, 136, 137,  
144-147, 150-152, 159-161, 165, 175, 177-179, 190, 210-212, 214-223, 234-238,  
240, 253, 255, 256, 284, 285, 289, 291, 297, 298, 304, 309, 317, 318, 321, 322,  
326-328, 340, 342, 353, 354, 356, 357, 360, 365-367, 369, 370, 379, 384, 385,  
387, 390, 391, 396, 399, 414, 422, 426, 427
- Foreign key ..... 68, 73, 107, 175, 203, 251, 253, 254, 305
- Form . 6, 20, 26, 61, 63, 73, 95, 102, 161, 180, 181, 184, 191, 193, 204, 214, 228, 237, 281, 288,  
292, 310, 356, 359, 396, 405, 407, 408, 414, 419, 427
- FORTTRAN ..... iii, 12-14, 17, 25, 37, 44, 46, 52, 226, 261-263, 265, 266, 277, 292, 310, 337
- Fourth generation language ..... 311
- Free space ..... 235
- Function key ..... 327
- Hierarchical index ..... 167, 183, 184, 186-188, 421
- Hierarchy ..... 85, 127, 134, 135, 137, 141, 142, 145, 159, 250, 255, 296, 301, 303, 305, 375
- Independent Logical File .. ii, 12, 50, 51, 56, 59, 99, 127, 128, 130, 136, 137, 144-147, 150-152,  
159-161, 253, 255, 289, 291, 298, 304
- Information resource dictionary system ..... 1, 2, 12, 30, 165
- Information system ..... 38, 281
- Insertion ..... 13, 22, 68, 105, 106, 109, 249, 250, 255, 272, 291, 305
- Insertion options ..... 105, 106, 249
- Installation and maintenance ..... 49, 52, 53, 339-342, 398, 400, 402, 403, 406, 427, 429
- Internal schema ..... 3, 4, 163
- Interrogation ... ii, 38, 39, 41, 44, 46, 49, 52, 53, 55, 98, 165, 175, 194, 201, 203, 204, 207-209,  
234, 242, 245-247, 259, 260, 262, 264-267, 283-285, 299, 308, 332, 333, 336-339,  
343, 346, 347, 373, 376, 385, 391, 393, 395, 397, 423, 425



Inverted access .....	xxi
ISO .....	iii, iv, 2, 4-7, 10-12, 31, 32, 34, 35, 364
Job ..	45, 53, 67-69, 139, 142, 145, 166, 242, 243, 264, 312, 319, 320, 325, 337, 354-357, 365-368, 379, 384, 398, 403, 404, 421, 423, 425
Join .....	26, 52, 109, 114, 115, 117, 118, 125, 132, 136, 149, 155, 201, 291, 298, 398
Journal file .....	284, 356, 357, 360, 365-367, 369, 399
Life Cycle .....	5, 30, 156, 270, 369
List processing .....	51, 173, 179, 190, 192, 202, 204, 207-211, 240, 242, 243, 391
Lock .....	42, 286, 365, 366, 381-383, 426
Locking .....	257, 285, 286, 307, 342, 343, 353, 366, 367, 373, 382, 422, 426, 427, 429
Logging .....	307, 402
Logical Database ..	37, 42, 43, 45, 49, 51, 53, 55, 59, 67, 125-127, 155, 160, 163, 176, 247, 256, 259, 339, 369, 370, 372-374, 387, 389, 395, 407
Logical Database Reorganization .....	42, 247, 369, 370, 372, 373
Many-to-many relationship .....	88, 90, 98
Map .....	42, 181, 187-192, 228, 282, 307, 366
Message Processing .....	49, 52, 53, 283, 284, 308, 327, 332, 336, 339, 341, 345, 398, 399, 429
Metadata .....	xxii, 1, 4, 7, 11, 30, 31, 37, 166
Mission .....	19, 395
Model transformation .....	419
Multiple Database Processing ..	45, 49, 52, 56, 283, 287, 308, 327, 332, 337, 339, 341, 384-388, 390-394, 425, 428, 429
NDL ..	iii, xxi-xxiii, 1, 4-6, 11-23, 30-34, 50, 52, 56, 70, 82, 85, 95, 105, 106, 133, 136-138, 140, 150, 156, 157, 161, 203, 236, 238, 239, 255, 261, 270, 272-277, 287, 304, 305, 371, 388, 393
Network data language .....	iii
NIST .....	iii
Object .....	30, 90
ODBC .....	28, 312, 329
One-to-one relationship .....	51, 387
Open database connectivity .....	312
Operating Systems .....	151, 309, 379, 390
Ordered .....	82, 101, 147, 174, 192, 238, 296, 297, 334, 342, 372, 415, 416
Organization ...	iii, xxii, 1, 2, 4, 6, 14, 16, 43, 50, 52, 53, 59-61, 84-86, 104, 130, 132, 136, 137, 139, 153, 160, 161, 165, 167, 168, 171-173, 175, 176, 178, 182, 184, 215-217, 231, 238, 242, 251, 253, 286, 299, 316, 332, 355, 373, 384, 386, 388, 395, 408, 409, 415, 416, 423-425
Outer join .....	26, 298
Output report .....	325, 326
Overflow .....	44
Packed decimal .....	225, 265
Padding .....	372
Page .....	iii, 34, 35, 42, 57, 189, 258, 266, 316, 317, 320, 326, 329-331, 336, 356
Passive Repository .....	166



- 
- Physical Database . . 14, 44, 46, 49, 51-53, 55, 163, 164, 189, 256, 257, 259, 340, 372-374, 382, 386, 389, 395, 399, 420, 425, 429
  - Pointer . . . . . 42, 54, 96-98, 104, 183, 195, 197, 235, 254
  - Precision . . . . . 1, 63, 64, 265
  - Primary key 26, 38, 39, 50, 51, 67-69, 72, 73, 78, 93, 97, 98, 125, 129, 139, 140, 142, 145, 153, 164, 166-171, 175, 178, 180, 181, 183-186, 195, 198, 201, 203, 207, 214, 217, 222, 233, 240, 242, 243, 254, 299, 320, 332, 375, 377, 415, 421, 425
  - Procedure oriented language . . . . . 261-264, 312, 338
  - Project . 20, 25, 26, 30, 34, 35, 38, 51, 55, 74, 109, 111, 112, 132, 145, 149, 170, 192, 249, 250, 282, 326, 400, 417
  - Project Management . . . . . 30, 326
  - Prototype . . . . . 336, 337
  - Query update language . . . . . 336
  - Queue . . . . . 294, 301, 380, 381, 399, 426
  - RDA . . . . . 5
  - RDAP . . . . . 6, 364
  - RDL . . . . . 20, 32
  - Recursion . . . . . 27, 31, 145, 153
  - Recursive . . . . . 15, 16, 50, 75, 82-87, 125, 135, 138, 145, 151, 153, 156, 160, 289
  - Recursive relationship . . . . . 16, 50, 82-85, 87, 145
  - Referential integrity . iii, 8, 13, 17, 20, 25, 26, 32, 39, 42, 55, 56, 73, 75, 105-109, 150, 156, 193, 247, 251, 252, 256, 305, 306, 371, 396, 398
  - Relation . . . . . 14, 110, 112, 113, 115, 116, 118, 120, 122, 124, 132
  - relational data language . . . . . 20
  - Relational database . . . . . 13, 148, 271
  - Relational model . . . . . ii, 51, 136, 148
  - Relational operators . . . . . 245, 333
  - Reorganization . 10, 41, 42, 45, 47, 49, 52, 53, 105, 164, 219, 228, 247, 252, 262, 283, 288, 308, 327, 328, 332, 336, 339-341, 366, 367, 369-374, 376, 380, 382, 383, 398, 399, 423, 425, 428, 429
  - Report . xxii, 3, 11, 17, 34, 35, 37, 38, 40, 43-47, 52, 53, 56, 57, 78, 95, 163, 165, 174, 175, 177, 214, 235, 240, 246, 258, 260-264, 266, 269, 270, 278, 279, 287, 288, 297, 299, 306, 307, 309, 311, 315-317, 320, 325, 326, 328-339, 342, 353, 354, 364, 379, 387, 391-394, 396, 406, 408, 419, 429
  - Repository . . . . . 2, 30, 163, 165, 166
  - Resource . . . . . 1, 2, 12, 28, 30, 165, 286, 383, 398
  - Restore . . . . . 10, 164, 235, 382, 397
  - Retention . . . . . 13, 22, 105-107, 255, 272, 305
  - Retention clause . . . . . 22
  - Rings . . . . . 400
  - Robust . . . . . 59
  - Role . . . . . 3, 68, 72-74, 263, 354
  - Rollback . 10, 29, 280, 283, 285, 295, 298, 306, 328, 354, 355, 357-360, 362-365, 368, 375, 392, 399



Root segment .....	69
Row . . . ii, 7, 8, 13, 14, 16, 17, 26-28, 30, 31, 38, 39, 41, 42, 44, 47, 51, 52, 54, 55, 59, 63-65, 67-69, 71-74, 76, 78, 85, 91, 94-98, 101-107, 109, 111, 114, 125-127, 129, 130, 132, 134, 138-141, 144, 147, 149, 150, 152-154, 156, 163-170, 172, 173, 177-181, 183, 184, 186-189, 191-194, 197, 198, 201, 202, 204, 207-209, 211, 214, 217-231, 235-238, 240-245, 247-255, 257, 260, 263, 265, 270, 277-279, 281, 284, 285, 288, 289, 291, 292, 294, 295, 304-308, 312, 316, 319-321, 325-327, 329, 332-334, 357, 363, 364, 371, 372, 375-377, 381, 382, 385, 389, 396, 399, 408, 409, 418, 421-423, 426, 427	
Run unit .....	54, 286, 325, 349, 355, 360
Save .....	49, 156, 164, 187, 193, 265, 326, 388, 405
Schema . . . . 2-4, 6, 8, 10, 11, 20, 21, 23, 26, 30, 32, 41-43, 59, 68, 107, 127, 155-157, 163, 203, 264, 265, 270, 272, 275, 284, 285, 294, 296-300, 304, 306, 307, 311, 321, 343, 346, 353, 373, 385, 386, 388, 391, 393, 395, 403, 405, 421, 422	
Screen .....	xxii, 6, 34, 59, 72, 156, 279-282, 288, 299, 312, 315, 321, 326, 331, 391
Secondary key . . . . .	68, 72, 73, 125, 129, 166, 168, 180, 182-188, 203, 204, 210, 239, 242, 308
Security and Privacy . . . . .	4, 49, 52, 53, 55, 283, 287, 308, 327, 332, 333, 336, 339, 341, 376, 384, 392-394, 398-402, 404, 425, 429
Segment .....	51, 69, 70, 91, 127, 141, 198, 200, 201, 255, 257
Select clause .....	94, 182, 264, 297, 319, 373, 398, 424, 426
Semantics .....	1, 5, 13-15, 17, 19, 34, 60, 63, 74, 75, 98, 101, 133, 262, 299, 370, 417
Serial search .....	192, 417, 418
Serializability .....	376
Simple record .....	128
Single thread .....	378-380
Single user mode .....	348, 379
Singular set .....	15, 81
Sort clause .....	82, 104, 238, 245, 297, 333, 371
SPARC .....	xxii, 2-5, 11, 12, 18, 32, 34, 35, 47, 57, 163, 258
SQL . . . ii, iii, xxi-xxiii, 1, 4-7, 11-15, 19, 20, 24-26, 28-34, 46, 49, 51, 56, 99, 107, 112, 113, 115, 116, 118, 120, 122, 124, 127, 130, 133, 135-137, 150-156, 160, 161, 245, 261, 263, 271, 275-278, 287, 291, 292, 299, 301, 304, 305, 311, 312, 316-319, 328, 332, 337, 371, 373, 400-403, 419-421	
SQL 1992 .....	25, 26, 32, 33, 419
SQL 1999 .....	ii, 26, 29, 32, 51, 150-155
SQL 2003 .....	ii, 32, 151
Staff .....	1, 172, 176, 397, 406
Standards Planning and Requirements Committee .....	2, 3
Stored procedure .....	371
Table . . . ii, v, xxi, 7, 8, 14-16, 20, 26, 27, 30, 41, 42, 45-47, 50-53, 55, 67-74, 78, 81-83, 85, 88, 90, 91, 93, 95, 98-101, 105-108, 111, 113, 126, 127, 129, 130, 136, 138, 140, 141, 144, 145, 148, 149, 151-153, 155, 156, 160, 161, 164-166, 168, 170, 171, 175, 176, 180, 181, 184, 192, 198, 200-203, 207-212, 214-216, 219-224, 226, 228, 232, 234, 235, 237-243, 245-247, 251, 252, 255, 257, 260, 263, 265, 266, 275, 280,	



	281, 283, 284, 287, 288, 292, 294, 296-300, 304-307, 312, 315-318, 325, 328, 329, 334, 335, 339, 340, 342-344, 348, 351-353, 364, 366, 367, 369-371, 373-375, 384, 388, 390, 395, 400, 401, 408, 409, 414, 415, 419, 420, 422, 426, 428
Third normal form	73, 405, 407, 408, 414, 419, 427
TP	365, 406
Tree structure	414
Trigger	95
Tuning	248, 252, 414, 427
User	xxi, 6, 10, 11, 27, 29, 31, 39, 42, 43, 45-47, 49, 52-54, 94, 95, 98, 101, 104, 115, 126, 138, 150, 165, 170, 174, 177, 192, 194, 239, 240, 242, 245-248, 257, 259-264, 266, 270, 277-279, 284-287, 293, 296, 297, 310, 312, 317, 318, 325-328, 331, 335, 336, 339, 342, 344, 345, 348, 351, 352, 355-362, 364, 365, 368, 373, 374, 376- 381, 383-385, 390, 392, 393, 396, 399-404, 406, 407, 417, 419, 420, 427-429
User interface	6
User Training	352
Value-based	13, 82, 98, 105, 150, 172
Vector	64, 231, 232, 234
WG3	4, 34, 35, 364
X3	2, 3, 12, 13, 19, 20, 34, 35, 57, 258

