SQL:1999

A Tutorial

Jim Melton
jmelton@us.oracle.com
jim.melton@acm.org

Consulting Member of Technical Staff
Oracle Server Technologies

Today's Agenda

- 10,000 meter view of SQL:1999
- Drill down into some interesting features
- Brief look at other parts of standard
- Quick review of process and timetable

What is SQL:1999?

- “SQL3”: Third generation of SQL standard
- Significant enhancement over SQL-92
- Principle theme: object orientation, but…
- Other new features, too

What is SQL:1999?

- Multi-part standard — ISO/IEC 9075-n:1999
  - Part 1: SQL/Framework
  - Part 2: SQL/Foundation
  - Part 3: SQL/CLI
  - Part 4: SQL/PSM
  - Part 5: SQL/Bindings

Part 1: SQL/Framework

- Common definitions & concepts
- Structure of multi-part standard
- Basic conformance structure & statement
- About 75 pages

Part 2: SQL/Foundation

- The “meat” of the standard
- Omits host language bindings, dynamic SQL, call interface, and similar issues
- Traditional SQL *and*…
- Object-oriented SQL
- About 1100 pages

“Traditional” Features

- Data types
- Predicates
- Semantics
- Security
- Active Database

New Data Types

- LARGE OBJECT
  - CHARACTER LARGE OBJECT — CLOB
  - BINARY LARGE OBJECT — BLOB
- BOOLEAN
- ARRAY — **`datatype ARRAY [n]`**
- ROW — **`ROW (name type,…)`**
- *Distinct* user-defined types

New Data Types — Large Object

- CHARACTER/ BINARY LARGE OBJECT
- **`WAR_AND_PEACE CLOB(25M)`**
  `CHARACTER SET CYRILLIC`
- **`EMPLOYEE_PHOTO      BLOB(50K)`**
- Normal character string literals and hex string literals apply
- **`SUBSTRING`**, **`TRIM`**, **`||`**, *etc*., all apply

New Data Types — Large Object

- However:
  - Comparison only for = and **<>**
  - No **GROUP BY** or **ORDER BY**
  - No **DISTINCT**
  - No **PRIMARY KEY** or **FOREIGN KEY**

New Data Types — BOOLEAN

- **`POLITICIANS_LIE  BOOLEAN`**
- Boolean value expressions are, in effect, predicates (and *vice versa*)
- Boolean literals:
  `TRUE`, **`FALSE`**, **`UNKNOWN`**
- **`COL1 AND (COL2 OR NOT COL3)`**
  `IS NOT FALSE`

New Data Types — ARRAY

- Varying-length arrays of element having specified type
- **`COL1  INTEGER ARRAY [50]`**
- “**50**” is the *maximum* cardinality
- Actual cardinality determined by highest occupied element (even if null)
- No arrays of arrays or multi-dimension arrays

New Data Types — ROW

- Explicit rows of *fields* (name/type pairs)
- Implicit rows in SQL-86, *et seq*
-
  ```
  COL1  ROW (name VARCHAR(50),
             dept INTEGER)
  ```

New Predicates

- SIMILAR — UNIX®-like regular expression
- DISTINCT — accounts for null values
- Type Predicate…later

New Predicates — SIMILAR

- UNIX®-like regular expression
- More powerful than **`LIKE`**
- **`NAME SIMILAR TO '(SQL-(86|89|92|99))|(SQL(1|2|3))'`**
- **However, *not* identical to UNIX syntax**

New Predicates — DISTINCT

- Differs from equality test — accounts for null values
- **`(10, 'abc', null ) =`**
  `(10, 'abc', null )` is *unknown*
- **`(10, 'abc', null )`**
  `IS DISTINCT FROM`
  `(10, 'abc', null )` is *false*

New Semantics

- View update — functional dependencies
- Recursion
- Locators — Array, LOB, and UDT
- Savepoints — single-nested subtransactions

New Semantics — View Update

- Better view update semantics
  - \ more views can be updated
- **`PRIMARY KEY`**, **`FOREIGN KEY`**, **`UNIQUE`** constraints are used
- Application of relational model to SQL ☺

New Semantics — Recursion

- WITH clause & recursive query expressions
- Recursive views
- **WITH RECURSIVE**

```
WITH RECURSIVE
  Q1 AS SELECT...FROM...WHERE...,
  Q2 AS SELECT...FROM...WHERE...
SELECT...FROM Q1, Q2 WHERE…
```

- Omit **RECURSIVE** for query shorthand use

New Semantics — Locators

- A *locator* is a value that uniquely identifies an instance of a "thing" in the database
  - Array
  - LOB
  - UDT
- Allows operations (*e.g.*, **SUBSTR**) without moving value to the host program
- *Only* valid on client side!

New Semantics — Savepoints

- Behaves like single-nested subtransactions
- **`ROLLBACK TO SAVEPOINT`** allows "partial rollback" of transaction
- **`RELEASE SAVEPOINT`** acts like tentative commit of part of transaction

New Security Features — Roles

- Privileges assigned to authorization IDs
- Privileges assigned to roles
- Roles assigned to authorization IDs
- Roles assigned to other roles
- Improves manageability of databases

Active Database — Triggers

- Database object tightly bound to a table
- Fires when certain event happens on table
  - Per-statement activation
  - Per-row activation
  - Before or after statement or row action
  - Access to table or row values possible

Triggers

-
```
CREATE TRIGGER trig1
  BEFORE UPDATE OF col1,col2
    ON tbl1
    REFERENCING OLD ROW AS orow
  FOR EACH ROW
    WHEN orow.col3 > 100
  INSERT INTO audit VALUES
      (CURRENT_USER,'tbl1',
       orow.col1,orow.col2);
```

Object Orientation

- Structured user-defined types
- Attributes & behavior
- Encapsulated: functions & methods
- Observers & mutators
- Type hierarchies (single inheritance)
- User-defined CAST, ordering
- Typed tables & reference types

User-Defined Types

- Three major topics to cover:
  - Distinct types
  - Structured types
  - Reference types

Distinct Types

- Based on built-in type
  `CREATE TYPE IQ AS INT FINAL`
- Cannot mix source type and distinct type in expressions

```
DECLARE VARIABLE X INTEGER;
DECLARE VARIABLE Y IQ;
...X+Y  --INVALID EXPR!
...X+CAST(Y AS INTEGER)  --OK
```

User-defined CASTs (distinct)

- No implicit casts to/from structured types
- User-defined functions provide capability
- Example: cast from **`IQ`** type to **`INTEGER`**
- In **`CREATE TYPE`**:

```
  CAST (SOURCE AS DISTINCT)
    WITH int_to_iq
 CAST (DISTINCT AS SOURCE)
    WITH iq_to_int
```

Structured Types

- Once called “abstract data types”
- May have arbitrarily-complex structure
- Analogous to **`struct`** in C language
- Stored data ⇒ state ⇒ attributes
- Behavior ⇒ semantics ⇒ methods & functions & procedures
- Other characteristics, too…
- Oracle’s implemented & implementing this

Attributes

- “Stored data”
- Each attribute can be:
  - Built-in type, including collection type
  - User-defined type
- System generates one “get” function (observer) and one “set” function (mutator) for each attribute — *not* overloadable

Encapsulation

- Hide implementation from users
- Allows implementation to change without affecting applications — provided interface remains constant
- Application accesses *everything* through functional interface, even attributes (using observer and mutator functions)

Procedures, Functions, Methods

- Generic concept: routine ⇒ procedure, function, method — usually “stored”
- Procedure: input & output parameters; invoked using “CALL” statement
- Function: input parameters only (output returned as “value” of function); invoked using functional notation
- Method: Special case of function

Procedures, Functions, Methods

- Procedures
  - Can be overloaded
  - Same name, must have different number of parameters/arguments
  - Data types of arguments not useable for overloading
  - In any schema, not bound to structured type

Procedures, Functions, Methods

- Functions
  - Can be overloaded (except get/set functions — which are really methods, anyway!)
  - Multiple functions with same name, same number of parameters
  - Distinguish by data types of arguments
  - But…use only compile-time data types (“declared type”) for overloading — no runtime overloading
  - In any schema, not bound to structured type

Procedures, Functions, Methods

- Methods
  - Can be overloaded
  - Tightly bound to single structured type
  - Must be in same schema as type definition
  - First argument implicit, distinguished — argument type is associated structured type
  - All arguments but first use declared type for resolution; first argument uses most-specific (“run-time”) type

Procedures, functions, methods

- SQL routines
  - Written in SQL
  - Parameters of any SQL data type
- External routines
  - Written in Ada, C, COBOL, Fortran, M, Pascal, PL/I (and Java…later today!)
  - Parameters have impedance mismatch
  - Can “call back” into database
- Tutorial all on its own!

Dot *vs* functional notation

- Dot notation: **`a.b.c`**
- Functional notation: **`c(b(a))`**
- Two sides of the same coin!
- Functions *must* use functional notation, and methods *must* use dot notation
- Observer: **`SELECT EMP.AGE FROM...`**
- Mutator: **`SET EMP.AGE = 10`**

Dot *vs* Functional Notation

- Any number of levels deep
  `x.y.z.w.r`
- Does not reveal physical implementation (“syntactic sugar”)
  `x.y.z.w.r(s) Û r(w(z(y(x))),s)`

Encapsulation *redux*

- Consider:

```
CREATE TYPE rational AS
   ( numerator      INTEGER,
     denominator    INTEGER )
```

- Implicit functions (one pair of two)

```
CREATE FUNCTION numerator (rational)
   RETURNS INTEGER
CREATE FUNCTION numerator
   (rational, INTEGER)
  RETURNS rational
```

Constructor functions

- These are *not objects*, so no “new object” is created
- Instead, a *site* has values assigned
- **DECLARE VARIABLE ratvar rational;**
  SET ratvar = rational(5,7);
  INSERT INTO table1 (ratcol)
  VALUES (rational(13,131));

```
DECLARE VARIABLE ratvar rational;
SET ratvar = rational(5,7);
INSERT INTO table1 (ratcol)
  VALUES (rational(13,131));
```

Constructor functions

- System-generated default constructor

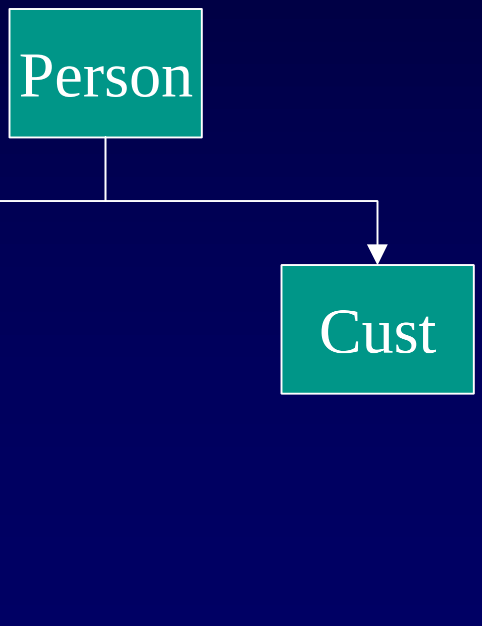
```
CREATE FUNCTION rational()
  RETURNS rational
```

- Overloadable: Any number of user-defined constructors:

```
CREATE FUNCTION rational(numer,denom)
  RETURNS rational
CREATE FUNCTION rational(denom)
  RETURNS rational
```

Type Hierarchies

- Allows specialization of existing types
- “Subtype” & “Supertype”



Inheritance

- Subtype *inherits* everything from supertype
- SQL:1999 supports only *single* inheritance (could be extended to multiple later)
- In subtype definition:
  - New attributes can be added
  - New methods can be added
  - Methods can be over-ridden

Inheritance

-
  ```
  CREATE TYPE emp UNDER person
    ( salary   DECIMAL(6,2),
      dept     department )
    METHOD give_raise(...)...,
    OVERRIDING METHOD
      address(...)...;
  ```

Inheritance

- Declared type

```
CREATE TYPE department (
  dept_name    CHARACTER(30),
  manager      employee, ...)
```

- Most-specific type

```
DECLARE VARIABLE x department;
SET x.manager =
  executive('Ortencio',...);
```

Structured Type Syntax

-
  ```
  CREATE TYPE name
   [ UNDER supertype-name ]
   AS ( attrib-name type,... )
   [ [ NOT ] INSTANTIABLE ]
   [ NOT ] FINAL
   [ REF ref-options ]
   [ method-spec,... ]
  ```

Structured Type Syntax

- **`REF ref-options`** Þ
  - User-defined:
    ```
    REF USING predefined-type
      [ ref-cast-option ]
    ```
  - Derived:
    ```
    REF ( attrib-name, ... )
    ```
  - System-generated:
    ```
    REF IS SYSTEM GENERATED
    ```

Structured Type Syntax

- **`method-spec`** Þ
  - Original method:
    ```
    [ INSTANCE | STATIC ] METHOD name
      ( param-name type,... )
      RETURNS type
    ```
  - Over-riding method:
    `OVERRIDING` original-method

User-defined CASTs (structured)

- Implicit casts to/from structured types
- User-defined functions provide capability
- Example: cast from **`rational`** to **`REAL`**
- Separate from **`CREATE TYPE`**:

```
CREATE CAST (rational AS REAL)
  WITH rational_to_real
  AS ASSIGNMENT
```

- Implicit casting with optional **`AS ASSIGNMENT`**

User-defined CASTs (structured)

-
  ```
  CREATE FUNCTION rational_to_real
      ( ratval rational )
    RETURNS REAL
    RETURN
      ratval.numer/ratval.denom;
  ```
- Usage:
  ```
  ...CAST (ratvar AS REAL)...
  ```

User-defined Ordering

- Required in order to have comparison of structured types
- Ordering forms:
  - **`EQUALS ONLY BY`** <category>
  - **`ORDER FULL BY <category>`**
- Ordering categories: **`RELATIVE WITH`**, **`MAP WITH`** , or **`STATE`**
- User-defined functions do the job

User-defined Ordering

-
  ```
  CREATE ORDERING FOR rational
    ORDER FULL BY MAP WITH
        rat_map
  ```

-
  ```
  CREATE FUNCTION rat_map
      ( param rational )
    RETURNS REAL
    RETURN
      param.numer/param.denom;
  ```

User-defined Ordering

- `...ratvar1 > ratvar2...`

  is equivalent to

  ```
  ...ratmap(ratvar1) >
     ratmap(ratvar2)...
  ```

Typed Tables

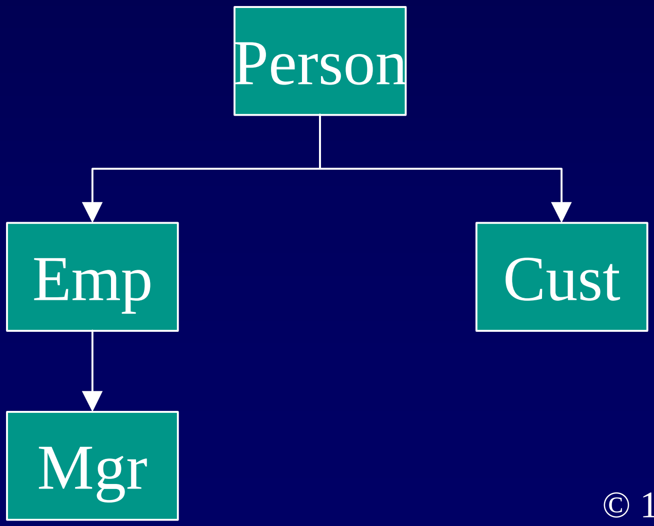
- Instances of type are rows in a table
- Behaves very much like objects
-
  ```
  CREATE TABLE rationals
     OF rational
     REF IS id_col ref-option
  ```
- Creates a base table with one column per attribute, plus one “self-referencing” column

Typed Tables

- **`ref-option`** must correspond to the **`ref-options`** in the type definition
- Can add additional columns if desired
- get/set functions operate on corresponding columns
- constructor must operate in context of **`INSERT`** or **`UPDATE`** statement

Table Hierarchies

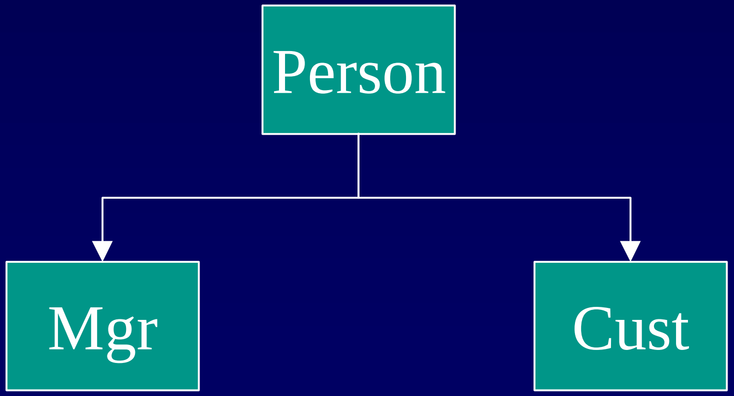
- Corresponds to type hierarchies
- Supertable must be “of” supertype
- Subtable must be “of” subtype
- However, allowed to “skip” types


Person

Emp

Cust

Mgr


Person

Mgr

Cust

Reference Types

- Allows one site to reference (identify, point to) another
- Only instances of structured types can be referenced…
- …but only if they are rows in a typed table

Reference Types

- Reference value of a row/instance explicitly represented in row:
  `REF IS col-name ref-option`
- Reference value never changes while row exists, never identifies a different row, unique in catalog (database)
- Object ID (OID)? I believe it!

Reference Types

- `REF(type) [SCOPE table-name]`
- A given REF type can only reference instances of a single, specified structured type
- If specified, the **`SCOPE`** clause restricts the references to the specified table: `REF(rational) SCOPE rationals`

Using References

- Form “path expressions”
-
  ```
  CREATE TABLE special_numbers
    ( name     CHARACTER(25),
      number   REF(rational)
               SCOPE rationals) )
  ```
-
  ```
  SELECT name
  FROM special_numbers
  WHERE number -> numer = 2
    AND number -> denom = 3;
  ```

Using References

- SQL statement’s privileges must be appropriate for operation on referenced table and columns (a/k/a attributes)
- `SELECT(rationals)` privilege required for

```
SELECT name
FROM special_numbers
WHERE number -> numer = 2
  AND number -> denom = 3;
```

Relationship to Java™ Object Model

- Several aspects of SQL:1999 object model were driven by Java object model
  - Methods with single distinguished parameter
  - Single inheritance
- What’s not in SQL:1999?
  - Explicit “interface” notion
  - Inheritance of interfaces

Information Schema

- Self-describing “catalog” of database
- Describes every object and relationship:
  - Tables and views; columns
  - UDTs; attributes
  - Data types
  - Routines
- Defined in terms of views based on tables in Definition Schema

Definition Schema

- Not required to be implemented: “Conforming SQL language shall not reference the Definition Schema”
- Base tables model the architecture of SQL

Conformance to SQL:1999

- Core SQL
  - Entry SQL-92
  - + Much of Intermediate SQL-92
  - + Some of Full SQL-92
  - + A few new SQL3 features
- Packages & Parts

Core SQL:1999

- Entry SQL-92, *plus*
  - CHARACTER VARYING, length > 0
  - UPPER/LOWER case conversion functions
  - AS keyword for correlation names: FROM EMP AS E
  - Qualified asterisk: SELECT E.* FROM EMP AS E
  - EXCEPT DISTINCT
  - UNION & EXCEPT on “compatible” data types
  - Expressions in VALUES clause
  - Value expressions in ORDER BY

Core SQL:1999 (continued)

- Holdable cursors
- PRIMARY KEY implies NOT NULL
- Column names in FOREIGN KEY different order than in PRIMARY KEY
- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
- SET TRANSACTION READ ONLY or READ WRITE
- Queries with subqueries may be updatable
- Minimal Information Schema, Documentation Schema
- More schema definition language statements

Core SQL:1999 (continued)

- LEFT & RIGHT OUTER JOIN
- DATE, TIME, & TIMESTAMP (but not time zones)
- Grouped operations
- CAST functions
- Explicit DEFAULT in VALUES and UPDATE…SET
- CASE expression
- Value expressions in NULL predicate
- Distinct data types
- Basic flagging

Packages

- Enhanced datetime facilities
- Enhanced integrity management
- OLAP facilities
- PSM & CLI
- Basic Object Support & Enhanced Object Support
- Active Database
- SQL/MM Support

Part 4: SQL / PSM

- PSM-96 specified:
  - functions & procedures
  - SQL-server modules
  - computational completeness
- PSM-99 specifies:
  - SQL-server modules
  - computational completeness
- Analogous to PL/SQL, Transact-SQL, *etc.*
- About 160 pages

SQL-server Modules

- Optional for conformance to PSM
- “Persistent modules”
- **`CREATE MODULE modname`**

```
    [options]
  routine-def; ...
```

- **EXECUTE** privilege on module ⇒ **EXECUTE** privilege on each routine

Computational Completeness

- Compound statement
- SQL variable declaration
- Assignment statement
- CASE statement
- IF statement
- ITERATE and LEAVE statements
- LOOP, WHILE, REPEAT, and FOR statements

Compound Statement

-
  ```
  [ label: ]
  BEGIN [ [ NOT ] ATOMIC ]
    [ local-declaration; … ]
    [ local-cursor-decl; ... ]
    [ local-handler-decl; ... ]
    [ SQL-statement; ... ]
  END [ label ]
  ```

Condition Declaration

- **`DECLARE cond-name CONDITION`**
  `[ FOR sqlstate-value ]`
- IF **`FOR sqlstate-value`** specified, then **`cond-name`** is an alias for that value
- Otherwise, **`cond-name`** is a “user-defined condition”

Handler Declaration

-
  ```
  DECLARE type HANDLER
      FOR cond-list
    SQL-statement
  ```
- **`type`** Þ **`CONTINUE`**, **`EXIT`**, or **`UNDO`**
- **`cond-list`** Þ list of conditions:
  - **`sqlstate-value`**
  - **`condition-name`**
  - **`SQLEXCEPTION`**, **`SQLWARNING`**, or **`NOT FOUND`**

SQL Variable Declaration

- `DECLARE var-name-list datatype [ default-value ]`
- Any data type, including structured types, REF types, ROW types, collection types, *etc*.
- Default value assigned when variable “site” is created (*e.g.*, on entry to routine in which declared)

Assignment Statement

- **`SET target = source`**
- **`target`** can be:
  - SQL variable, SQL parameter, host parameter
  - **`row-site-name.field-name`**
  - **`udt-side-name.method-name`**
- **`source`** can be:
  - Value expression of appropriate type
  - Contextually-typed value: **`NULL`**, **`EMPTY`**

Assignment Statement

- **`SET ratvar.numer = 10`**
  is equivalent to
  `SET ratvar = ratvar.numer(10)`
  is equivalent (in some sense!) to
  `ratvar.numer(ratvar,10)`

CASE Statement

- Simple CASE statement:

```
CASE value-expression-0
  WHEN value-expression-1
    THEN SQL-statement-list-1
  WHEN value-expression-2
    THEN SQL-statement-list-2
  ...
  ELSE SQL-statement-list-n
```

CASE Statement

- Searched CASE statement:

```
CASE
   WHEN search-condition-1
      THEN SQL-statement-list-1
   WHEN search-condition-2
      THEN SQL-statement-list-2
   ...
   ELSE SQL-statement-list-n
```

IF Statement

-
  ```
  IF search-condition-1
     THEN statement-list
     [ ELSEIF search-condition-2
           THEN statement-list ]
     [ ELSE statement-list ]
  END IF
  ```

LOOP Statement

-
  ```
  [ label: ]
  LOOP
    SQL-statement-list
  END LOOP [ label ]
  ```
- Loops “forever” or…
- …or until forced termination:
  - **`LEAVE`**
  - **`ITERATE`**

LEAVE and ITERATE Statements

- **`LEAVE label`**
- Immediately branches to the statement after the statement ending the containing statement (not just looping statements!)
- **`ITERATE label`**
- Immediately branches to the statement ending the containing looping statement

WHILE and REPEAT Statements

-
  ```
  [ label: ]
  WHILE search-condition DO
    SQL-statement-list
  END WHILE [ label ]
  ```

-
  ```
  [ label: ]
  REPEAT SQL-statement-list
    UNTIL search-condition
  END REPEAT [ label ]
  ```

FOR Statement

```
[ label: ]
FOR for-loop-variable-name AS
  [ cursor-name [sensitivity]
  CURSOR FOR ]
  cursor-specification
  DO SQL-statement-list
END FOR [ label ]
```

Part 5: SQL/Bindings

- Embedded SQL
- Dynamic SQL
- “Direct Invocation”
- About 250 pages

Embedded SQL

- SQL embedded in:
  - Ada
  - C
  - COBOL
  - Fortran
  - MUMPS
  - Pascal
  - PL/I
- Little new since SQL-92

Embedded SQL

- Embedded SQL DECLARE Sections
  - Declares host variables for use in SQL statements
  - Generates implicit conversions to minimize the impedance mismatch
- **`EXEC SQL…;`**
  `EXEC SQL…<newline>`
  `&SQL(…)`

Dynamic SQL

- Depends on “SQL descriptor areas”
- Most products (including Oracle) use SQLDA instead
- **`PREPARE`** statements for repeated execution
- **`EXECUTE`** prepared statements
- **`EXECUTE IMMEDIATE`** if only one use
- Little new since SQL-92

Direct Invocation

- “Interactive SQL”
- Most of the same statements, but…
- …“multi-row select statement” added
- Nothing new since SQL-92

Part 3: SQL / CLI

- Call-Level Interface
- Best-known implementation: ODBC
- CLI-95: Revision in progress
- Align with SQL:1999 features and ODBC 3.0 features
- About 400 pages

SQL / CLI

- Analogous to dynamic SQL, but…
- Better support for *shrink-wrapped* applications
  - No precompilation or even *re*compilation
  - Binary code works with multiple DBMSs
- Alternative to protocol-based interoperability such as RDA — doesn’t solve “network” problem
- Uses *handles* to manage resources
- CLI descriptor areas analogous to dynamic SQL’s system descriptor areas

SQL / CLI

- Multi-row fetch
- Multiple & parallel result set processing
- General SQL:1999 alignment, including support for new data types
  - unstructured row types
  - structured types
  - locators

SQL / CLI

- Environment functions
  - AllocHandle & AllocEnv
  - GetEnvAttr & SetEnvAttr
  - FreeHandle & FreeEnv
- Everything depends on the environment handle

SQL / CLI

- Connection functions
  - AllocHandle & AllocConnect
  - GetConnectAttr & SetConnectAttr
  - Connect & Disconnect
  - FreeHandle & FreeConnect
- Implicit set connection

SQL / CLI

- Statement functions
  - AllocHandle & AllocStmt
  - GetStmtAttr & SetStmtAttr
  - FreeHandle & FreeStmt
- Statement execution functions
  - Prepare & Execute & ExecDirect
  - StartTran & EndTran

SQL / CLI

- Descriptor functions (IPD, IRD; APD, ARD)
  - AllocHandle & FreeHandle
  - GetDescField & SetDescField
  - GetDescRec & SetDescRec
  - DescribeCol & ColAttribute & NumResultCols
  - CopyDesc
  - BindCol & BindParameter
  - GetData & GetParamData & PutData

SQL / CLI

- Cursor functions
  - GetCursorName & SetCursorName
  - Fetch & FetchScroll
  - CloseCursor
  - MoreResults & NextResult

SQL / CLI

- Diagnostic functions
  - GetDiagField & GetDiagRec
  - Error
  - RowCount
- General functions
  - DataSources
  - GetFunctions & GetInfo & GetFeatureInfo
  - Cancel
  - GetSessionInfo

SQL / CLI

- Locator functions
  - GetLength & GetPosition & GetSubstring
- Metadata functions
  - Tables & TablePrivileges
  - Columns & ColumnPrivileges
  - SpecialColumns
  - PrimaryKeys & ForeignKeys
  - GetTypeInfo

SQL / CLI Example

```
void main () {
   SQLHENV      henv;
   SQLHDBC      hdbc;
   SQLHSTMT     hstmt;
   SQLRETURN    rc;
   SQLINTEGER   runs, runs_ind, title_ind;
   SQLCHAR      title[101];
 // Allocate environment handle
 rc = SQLAllocHandle (SQL_HANDLE_ENV,
                      SQL_NULL_HANDLE, &henv);
 if (rc != SQL_SUCCESS) {
       report_and_exit (rc,
           "Allocate Environment Failed"); }
```

ORACLE®

SQL / CLI Example (continued)

```
// Create connection
rc = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
if (rc != SQL_SUCCESS) {
      report_and_exit (rc,
            "Allocate DBC handle Failed"); }
rc = SQLConnect (hdbc,
      (SQLCHAR *) "Movies", SQL_NTS,
      (SQLCHAR *) "dba", SQL_NTS,
      (SQLCHAR *) "sql", SQL_NTS );
if (rc != SQL_SUCCESS) {
      report_and_exit (rc,
            "Allocate Environment Failed"); }
```

SQL / CLI Example (continued)

```
rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
// execute the select statement
rc = SQLExecDirect (hstmt,
      (SQLCHAR *) "SELECT title, runs "
                  "FROM   movies "
                  "WHERE  year_introduced = '1980'",
      SQL_NTS );
if (rc != SQL_SUCCESS) {
       report_and_exit (rc,
             "Statement Execution Failed"); }
// bind the result columns to variables
SQLBindCol (hstmt, 1, SQL_C_CHAR, title, 100,
            &title_ind);
SQLBindCol (hstmt, 2, SQL_C_ULONG, &runs, 0,
            &runs_ind);
```

ORACLE®

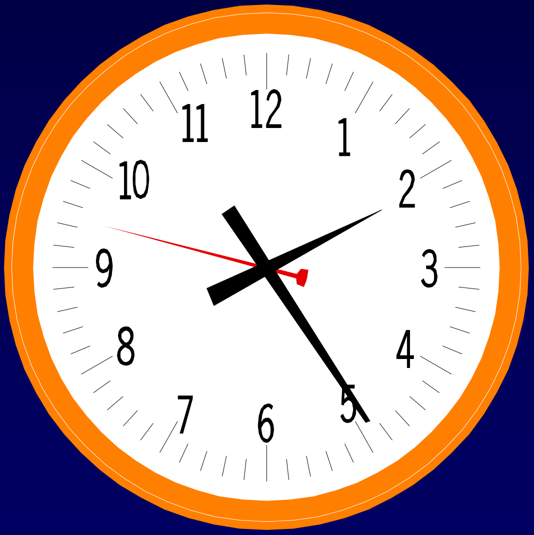
SQL / CLI Example (continued)

```
// get data from database
while ((rc = SQLFetch (hstmt)) != SQL_NO_DATA) {
   if (rc != SQL_SUCCESS) {
          report_and_exit (rc, "Fetch Failed"); }
   cout << "\"" << title << "\"";
   if (runs_ind >= 0)
      cout << ", " << runs << " minutes";
   cout << endl; }
// Cleanup
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
SQLDisconnect (hdbc);

}
```

New Parts of the SQL Standard

- Part 6: *obsolete*
- Part 7: SQL/Temporal



Part 7: SQL/Temporal

- Work temporarily suspended for SQL3 focusing
- Two strongly opposed philosophies
- Work expected to continue late 1999
- Distinguish from time-series data
  - Most vendors support time-series
  - Few vendors have market need for temporal

New Parts of the SQL Standard

- Part 6: *obsolete*
- Part 7: SQL/Temporal
- Part 8: *obsolete*
- Part 9: SQL/MED

Part 9: SQL / MED

- Management of External Data
- Seen as way to give SQL access to non-SQL data (*e.g.*, flat files, even sensors)
- Foreign tables, abstract LOBs(?): SQL API
- DataLink: SQL control, native API
- Federated database?
- (Non-final) Committee Draft late 1998

Foreign Servers

- SQL-aware or non-SQL-aware
- Mix-and-match: multiple foreign servers can be involved in single statement execution, along with local SQL-server
- Accessed via foreign-data wrapper
- Handle-based API (“light-weight CLI”)
- Create a market for 3rd-party “foreign-data wrappers” to control various foreign servers

Foreign Tables

- SQL API for non-SQL data, user-defined functions for semantics
- Foreign-data wrapper decides how to support table semantics for foreign tables
  - Possibly limited to “SELECT * FROM T”
  - Possibly unlimited SQL statement capabilities

Foreign servers, tables, *etc.*

- CREATE FOREIGN DATA WRAPPER
- CREATE FOREIGN SERVER
- CREATE FOREIGN TABLE
- CREATE USER MAPPING
- ALTER and DROP for each CREATE

Federated Database

- Still under consideration and discussion
- Foreign tables, foreign columns, *etc*.
- “Import” metadata or describe it
- Current products:
  - Oracle’s Transparent Gateway
  - IBM’s Data Joiner
  - Sybase’s Omni

Datalink

- Native API for non-SQL data, requires “hooks” to keep data source under database control (even transactional)
- For example, filesystem will deny DELETE of a file if a Datalink has attached it to a cell in a database

New Parts of the SQL Standard

- Part 7: SQL/Temporal
- Part 9: SQL/MED
- Part 10: SQL/OLB

Part 10: SQL / OLB

- Object Language Bindings
- SQLJ Part 0
- SQL embedded in Java
- ANSI publication of X3.135.10:1998 (oriented towards SQL-92)
- ISO FCD starting late 1998 (aimed at JDBC 2.0 and SQL:1999)

Part 10: SQL / OLB

- Based on JDBC paradigm/model; can share context with JDBC for mix-and-match
- Instead of cursor, uses strongly-typed iterators
- Provides “default” runtime using JDBC, or…
- …implementation-defined “customizations” for better alignment with products

SQL / OLAP

- Amendment 1 to SQL:1999
- Driven by Oracle and IBM, with participation from Informix and others
- Possible standardization in 2001

Part 11: SQL/Schemata

- Part 5: SQL/Bindings now merged with Part 2: SQL/Foundation
- Information and Definition Schema definitions moved to new Part 11: SQL/Schemata
- No functionality change, no conformance implications…merely an “editorial detail”

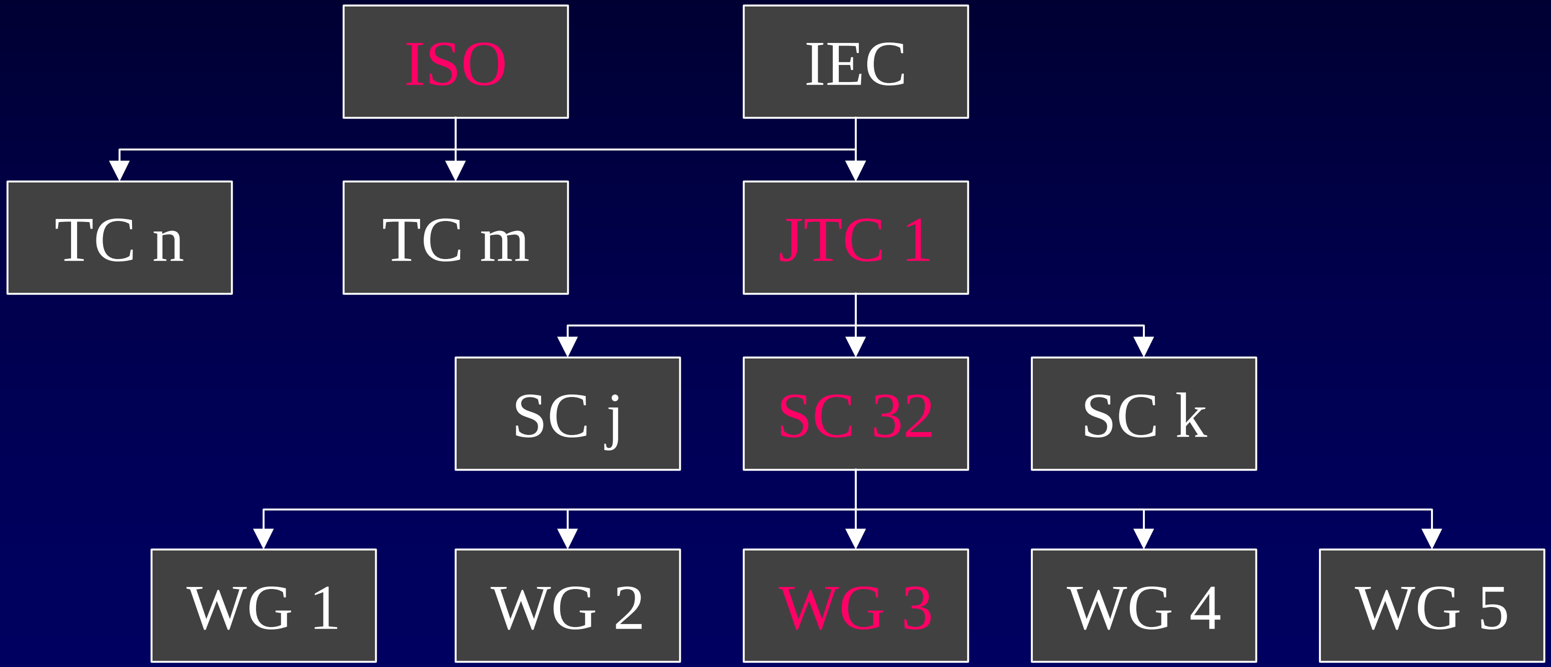
Implementations

- Most vendors say “about 2 or 3 product cycles to conform to Core SQL:1999”
- Suggests late 2001 for conforming products
- A few vendors claim they will conform sooner
- Vendors will choose packages based on their perception of marketplace needs
- Vendors will implement selected other features as needed

Process

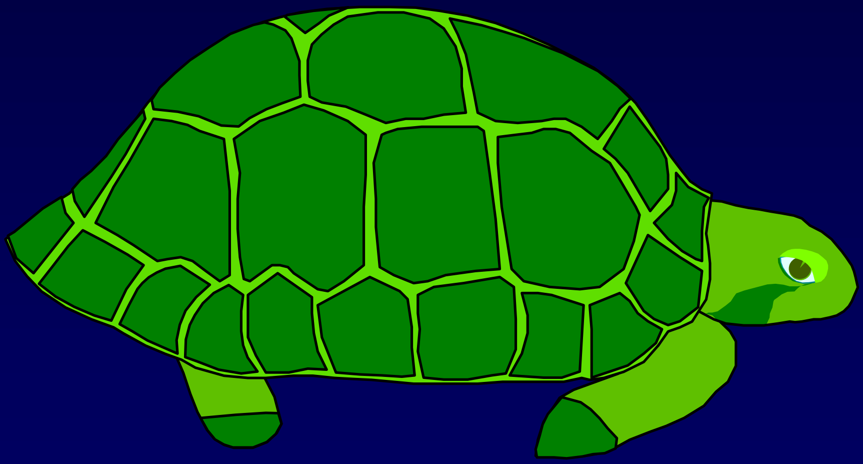
- ANSI — American National Standards Inst.
- NIST — National Inst. Of Stds & Technology
- ISO — Int’l Organization for Standardization
- X/Open (a/k/a The Open Group)
- SQL Access Group
- SQLJ (non)consortium — more later

ISO Organization



SQL:1999 — Did It Take Too Long?

- Yes ☺

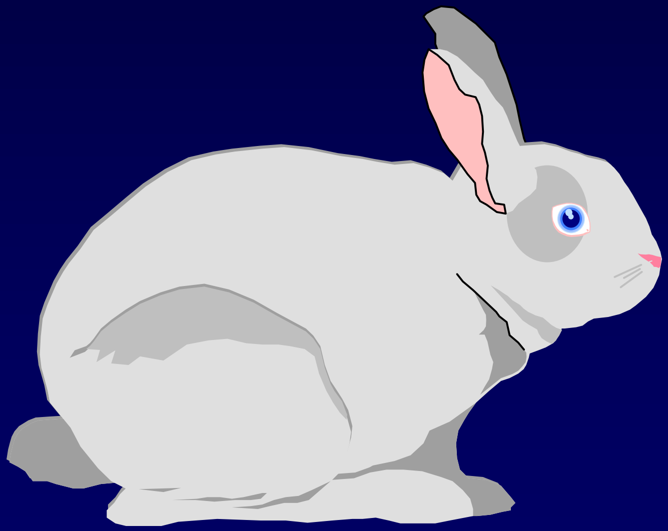


SQL:1999 — Did It Take Too Long?

- Yes
- But why?
  - Tried to do too much
  - Extreme controversy over object model
  - Distracted by parallel processing of CLI, PSM, and other work
  - Reduced resources, increased technology

SQL:1999 — Did It Take Too Long?

- Yes
- But why?
- Avoiding that error in future
  - Smaller increments
  - Plan for 3-year cycle (“SQL:200n”, *not* “SQL4”)
  - Depend more on “incremental” parts



Related Standards Efforts

- SQL/MM
- RDA
- RMDM
- Export/Import
- SQLJ
  - Part 1: SQL Routines Using the Java™ Programming Language (ANSI NCITS 331.1)
  - Part 2: SQL Types Using the Java™ Programming Language

SQL / MM — ISO / IEC 13249-n

- Multi-part standard
  - Part 1: Framework
  - Part 2: Full-text
  - Part 3: Spatial
  - Part 4: General Purpose Facilities
  - Part 5: Still Image
- Class libraries of SQL:1999 structured types

RDA — ISO / IEC 9579

- First version based on OSI…no interest (generic and specializations)
- Second version “tacked on” TCP/IP support
- Third version is SQL-only, transport-independent (but optimized for Internet)
- Little vendor interest…mostly Canadian and UK government interest
- New work on security, SQL/MED support, distributed transactions

RMDM — ISO / IEC 10032

- Reference Model for Data Management
- Provides a context for discussing issues surrounding data management, including metadata, *etc.*
- Widely ignored ☺

Export/Import — ISO/IEC 13238-n

- Multi-part standard
  - Framework
  - SQL Specialization
  - IRDS Specialization
- Generally ignored
- Vendors sometimes actually hostile ☺

SQLJ — ANSI NCITS 331.n

- Currently *not* part of SQL standard, may change later (if submitted to and adopted by ISO)
- Non-consortium, all major database vendors and some other participants
- Develop specs, leave publication to others
- NCITS 331.1 = SQLJ Part 1
- SQLJ Part 2 expected to be NCITS 331.2

Summary

- 10,000 meter view of SQL:1999
- Drilled down for some interesting features
- Brief look at other parts of standard
- Quick review of process and timetable

Questions?