File: DM32.2-2014-00024R1_JSON-SQL-Proposal-1

Printed: Thursday, 06 March, 2014 at 13:44

Number of pages: 51

WG3:PEK-___

DM32.2-2014-00024R1

2014-03-06 — 06 March, 2014

Authoritative Version: Adobe Acrobat Portable Document Format (PDF)

Project: ANSI: 1234D — ISO: 1.32.03.05

Title: SQL/JSON, Part 1

Status: Change proposal

Author: Jim Melton, Fred Zemke, Beda Hammerschmidt, Krishna Kulkarni, Zhen Hua Liu, Jan-Eike Michels, Doug McMahon, Fatma Özcan, Hamid Pirahesh

Abstract: JSON provides a data model that is becoming increasingly important in the data management world, particularly in Web- and Cloud-based communities. NoSQL "database" systems are beginning to gain market attention in those communities and several of them use JSON as their guiding data model. It is important that SQL respond to the requirement to support JSON data by providing facilities for storage, retrieval, querying, and manipulation of JSON data in the context of SQL.

References:

[ACIDtxns]	*Distributed Transaction Processing: Concepts and Techniques*, Gray, Jim, and Reuter, Andreas; Morgan Kaufmann, 1993; ISBN 1-55860-190-2
[Avro]	http://avro.apache.org/
[BSON]	http://bsonspec.org/
[ECMAscript]	*ECMAScript Language Specification*, http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf

[FoundCD]	INCITS DM32.2-2012-153 = WG3:USN-003, *(Committee Draft) Foundation for SQL (SQL/Foundation)*, 23 October 2012
[Hadoop]	http://hadoop.apache.org/
[HDFS]	Hadoop Distributed File System, http://wiki.apache.org/hadoop/ProjectDescription
[JDIF]	The JSON Data Interchange Format, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[JAQL]	http://code.google.com/p/jaql/wiki/JaqlOverview
[JPath]	http://projects.plural.cc/jsonij/wiki/JPath
[JSONintro]	*Introducing JSON*; http://www.json.org/
[JSONiq]	http://www.jsoniq.org
[JSONPath]	http://goessner.net/articles/JsonPath/
[JSONschema]	http://json-schema.org
[MEDCD]	INCITS DM32.2-2012-156 = WG3:USN-006, *(Committee Draft) Management of External Data (SQL/MED)*, 23 October 2012
[Mongo]	http://www.mongodb.org/
[NoSQLDB]	http://nosql-database.org/
[NoSQLdef]	http://en.wikipedia.org/wiki/NoSQL
[RDFmodel]	*Resource Description Framework (RDF) Model and Syntax Specification*, World Wide Web Consortium, http://www.w3.org/TR/rdf-syntax/
[RFC1951]	RFC 1951, *DEFLATE Compressed Data Format Specification*, P. Deutsch, May 1996; http://tools.ietf.org/html/rfc1951
[RFC4627]	RFC 4627, *The application/json Media Type for JavaScript Object Notation (JSON)*, D. Crockford, July 2006; http://tools.ietf.org/html/rfc4627
[SQLJSON2]	DM32.2-2014-00025 = WG3:PEK=___, *SQL/JSON, Part 2*
[Unicode]	*The Unicode Standard*, http://unicode.org
[XDM]	(Recommendation) XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/xpath-datamodel/
[XMLcast]	INCITS H2-2004-020r1 = WG3-SIA-041, *XMLCast*, 4 April 2004 (Zemke, *et al*)
[XMLCD]	INCITS DM32.2-2012-161 = WG3:USN-010, *(Committee Draft) XML-Related Specifications (SQL/XML)*, 23 October 2012
[XMLquery]	INCITS H2-2004-021r1 = WG3-SIA-042, *XMLQuery*, 4 April 2004 (Zemke, *et al*)
[XMLtable]	INCITS H2-2004-039r1 = WG3-SIA-051, *XMLTable*, 10 May 2004 (Zemke, *et al*)
[XPath]	(Candidate Recommendation) XML Path Language (XPath) Version 3.0. http://www.w3.org/TR/2012/CR-xpath-30-201212xx/
[XQuery]	(Candidate Recommendation) XML Query Language (XQuery) Version 3.0. http://www.w3.org/TR/2012/CR-xquery-30-201212xx/

[XSD]	(Recommendation) XML Schema Part 1: Structures. http://www.w3.org/TR/xmlschema-1/ and (Recommendation) XML Schema Part 2: Datatypes. http://www.w3.org/TR/xmlschema-2/
[OtherRefs]	(Other references will be supplied as required.)

Note to reviewers: In this paper, I occasionally use boxed paragraphs like this one to signify notes to reviewers. These notes are *not* to be applied to the referenced documents.

Note to reviewers: In this paper, I use the following conventions:

Red bold face text	Indicates new text to be **inserted** into existing text
~~Blue strikeout text~~	Indicates text to be ~~deleted~~ from existing text
Plain text	Indicates existing text that is to be retained unchanged or new text that is part of a new Subclause
Green boldface text	Text in a revision of the present paper that differs from the previous version
[Note to proposal reader: …]	Indicates notes to proposal readers that are not to be placed into the document being modified
[Note to Editor: …]	Indicates notes to the editor of the document being modified; these notes are not to be placed into the document being modified
Boxed text	Indicates “Editing/Merger instructions” that are placed into incremental parts

1. Discussion

1.1 Introduction

1.1.1 Ballot Comment Addressed

The present proposal, if accepted, will partially resolve the following ballot comment:

257.	P02-USA-950	1-Major Technical	JavaScript Object Notation (popularly known as JSON) [RFC4627] is becoming increasingly important in Web- and Cloud-based communities for exchanging data between Web-based applications. This has in turn led to the demands for storage, retrieval, querying, and manipulation of JSON data in the context of SQL It is important that SQL respond to these requirements by providing appropriate extensions for handling JSON data. **Solution** None provided with comment.

1.1.2 What is JSON?

JSON [RFC4627, JSONintro] (an acronym for "JavaScript Object Notation") is both a *notation* (that is, a syntax) for representing data *and* a[n implied] data model. JSON is not an *object-oriented* data model in the classic sense; that is, it does not define sets of classes and methods, type inheritance, or data abstraction. Instead, JSON "objects" are simple data structures, including arrays. [RFC4627] says that JSON is a *serialization* of structured data. Its initial intended use was as a data transfer syntax. (NOTE: The present proposal references [RFC4627] normatively and places other relevant references into the Bibliography.)

[JDIF] is a recently-published (late 2013) ECMA standard that specifies the syntax of JSON very concisely. There appear to be very minor differences between the syntax defined in [JDIF] and that in [RFC4627], but they appear to be inadvertent and there are plans to eliminate them. It is not known whether ECMA[1] intends to submit [JDIF] for fast-track processing as an ISO/IEC standard.

The first-class objects in the JSON data model are *objects* and *arrays*. A JSON object is zero or more name-value pairs and is enclosed in curly braces — {…}. A JSON array is an ordered sequence of zero or more values and is enclosed in square brackets — […].

Here is an example of a JSON object:

```
{ "Name" : "Isaac Newton",
  "Weight" : 80,
  "Famous" : true,
  "Phone" : null }
```

The name-value pairs are separated by commas, and the names are separated from the values by colons. The names are always strings and are enclosed in (double) quotation marks. The values may be strings, numbers, Booleans (represented by the JSON literals `true` and `false`), and JSON nulls (represented by the JSON literal `null`).

[1] The name "ECMA" was once an acronym for "European Computer Manufacturer's Association", but is no longer an acronym at all.

Here is an example of a JSON array:

```
[ "Robert J. Oppenheimer", 98, false, "Beechwood 45789" ]
```

JSON arrays and objects are fully nestable. That is, any JSON value is permitted to be an “atomic” value (a string, number, or literal), a JSON object, or a JSON array.

JSON is sometimes used to represent *associative arrays* — arrays whose elements are addressed by content, not by position. An associative array can be represented in JSON as a JSON object whose members are name-value pairs; the name is used as the “index” into the “array” — that is, to locate the appropriate member in the object — and the value is used as the content of the appropriate member. Here is such an associative array:

```
{ "Isaac Newton" : "apple harvester" ,
  "Robert J. Oppenheimer": "security risk" ,
  "Albert Einstein" : "patent clerk" ,
  "Stephen Hawking" : "inspiration" }
```

An extremely important part of JSON’s design is that it is inherently schema-less. Any JSON object can be modified by adding new name-value pairs, even with names that were never considered when the object was initially created…or designed. Similarly, any JSON array can be modified by changing the number of values in the array. (Note that there is no inherent reason why a schema language could not be defined for JSON; the JSON culture and the principal applications that use JSON do not provide impetus for such a schema language.)

The reason, its proponents claim, that JSON is (and should remain) schemaless, is that schemas tend to impose restrictions on the structure of the data that would limit the ways in which JSON is principally used. (The authors of the present paper are skeptical about such claims, in part because the same was also said about XML and proven false in that domain. Similar arguments are made about the atomic type system of JSON — numbers, strings, Booleans, and null. Note in particular that JSON does not recognize any atomic type associated with date and time data. We believe that JSON is insufficiently mature for its proponents and users to recognize the great value of data design and strong typing.) We discuss this subject a bit more in section 1.1.4.1 below.

JSON is, in many ways, isomorphic with XML. That is, JSON provides a syntax for representing semi-structured data, just as does XML, and defines a semantic for that data, just as does XML. Of course, XML elements can have attributes, for which JSON has no corresponding concept, and there are important detailed differences (such as the exact set of characters usable in each language)[2]. But JSON is a convenient notation for representation of semi-structured data, as is XML. JSON, it is true, is a bit more compact than XML, as it does not provide “closing tags”. The reader of the present proposal must not presume that JSON can, or should, replace XML in any sense.

1.1.3 Representations of JSON data

Before delving much deeper into the primary topic of the present proposal, readers should understand that JSON data can be represented in several widely-acknowledged and -used forms. The most obvious and most easily recognizable is its “character string” representation, in which JSON data is represented in Unicode characters as plain text. More specifically, explicit characters such as the left square brace, comma, right curly brace, quotation mark, and letters and digits are all used in their native Unicode representation (UTF-8, UTF-16, UTF-32).

[2] In addition, XML allows for elements with mixed content (that is, elements whose children are a mixture of ordinary text and other elements); JSON provides no similar capability.

However, for a variety of reasons, JSON data is sometimes stored and exchanged in one of several *binary* representations. For example, a binary representation of JSON data may be significantly smaller (fewer octets) than the character representation of the same data, so a reduction in network bandwidth and or storage media can be achieved by transferring or storing that data in a binary representation.

Readers should note that there are no published standards, either from traditional *de jure* standards organizations nor from consortia or other *de facto* standards groups, for any binary representations of JSON. The two described in the current paper are frequently used and may be representative of binary JSON representations generally. However, they are not proposed in the facilities described in the current paper. The following discussion is intended only to illustrate the use of and issues with binary serializations of JSON. No proposal is being made to specify Avro, BSON, or any other binary representation of JSON for incorporation into the SQL standard. It will be implementation-defined whether or not such representations are supported in any particular implementation.

1.1.3.1 Avro [Avro]

Avro[3] is described as a “data serialization system”. As such, its use is not limited to a binary representation or as a compression representation of JSON data. However, a number of JSON environments have chosen Avro as their preferred binary, compressed representation. Avro was designed in the scope of Hadoop [Hadoop].

Avro has a number of important characteristics that affect its choice as a JSON representation.

- Data is represented in a variable-length, “shortest” form; *e.g.*, the integer 2 and the integer 2000 occupy a different number of octets in an Avro string.
- Numbers are represented using a rather arcane “zig-zag” encoding (this notation “moves” the sign bit from its normal position as the first bit to the last bit; doing so permits removing leading zeros from the numbers, thus making their representation occupy fewer octets).
- There is not a one-to-one mapping between JSON atomic types and Avro atomic types.
- Avro data is *always* associated with an Avro *schema*. An Avro schema describes the physical structure of the corresponding Avro data and is needed merely to “unpack” Avro data because of the variable-length fields and the various encoding rules. An Avro schema may accompany each individual Avro data string, or it may be specified separately and applied to all Avro data strings in, say, an Avro data file. Avro schemas tell almost nothing about the data other than how it is packed into a data string. Avro schemas are described in more detail in section 1.1.4.1 below.
- Avro strings can be encoded using JSON notation (which sort of contradicts its use as a different representation for JSON data) or using a binary notation. Either notation can be compressed; the preferred compression “codec[4]” is *deflate* [RFC1951]. (If the encoded string is not compressed, the Avro specification says that it is “encoded using the null codec”.)

Readers should recognize that Avro is not a different kind of data at all. It is, instead, merely another way of representing the same data that the JSON character string format represents. (It should be noted that not all possible Avro strings can be treated as JSON data; similarly, not every character string is a valid bit of JSON

[3] The term “Avro” does not appear to be an acronym, nor to mean anything in particular.

[4] The term “codec” is an abbreviation of “en<u>cod</u>er/<u>dec</u>oder”

data.) In the present proposal, we refer to Avro as one possible *serialization* of JSON data; the character string format is another serialization of the same data.

1.1.3.2 BSON [BSON]

BSON[5] (some people seem to pronounce this term as though it were spelled "bison", others as "beeson") is another data serialization system. [BSON] says that it is "a binary format in which zero or more key/value pairs are stored as a single entity", called a "document." BSON is used by — and apparently was created for — MongoDB [Mongo].

BSON strings are no less difficult for human beings to read than Avro strings are, but the design of BSON is significantly different than Avro's. A BSON document is roughly a sequence of *elements*, each of which is introduced by a single-octet code (for example, the hexadecimal value '01' identifies the element as a double precision floating point value, which is always eight octets in length, and '0D' identifies the element as a string containing JavaScript code), followed by an optional element name, followed by the (mandatory) element value.

It appears that element codes do not correspond to valid Unicode characters other than the so-called "control characters" and similar restricted codepoints. This makes it possible to scan through a BSON string to find the starting point of each element. As with Avro, there is no ability to "index" directly into a BSON string to locate the start of the third or tenth element. Element names, for example, are defined to be "C strings", meaning strings terminated with an octet with the value zero. String values are defined to be a 4-octet value that specifies the number of octets (not characters!) in the string, followed by characters occupying the specified number of octets, followed by an octet with the value zero (apparently using both an explicit string length *and* a "null-terminated" value).

BSON, like Avro, is not a different kind of data, but merely provides yet another way of representing JSON data. (Also like Avro, not all BSON strings represent valid JSON data.)

1.1.3.3 Other JSON serializations

There are almost certainly other "binary" serializations of JSON defined by parties somewhere in the world, but we do not believe that any of them have received the acceptance that Avro and BSON have. Until one or more formats have wide adoption and a standardized specification, it is inappropriate to incorporate any of these into the SQL standard. Implementations, of course, are free to support any such formats as implementation extensions to the standard.

1.1.4 Schemas

1.1.4.1 JSON schemata

The present proposal is independent of any notion of schemata for JSON, primarily because the JSON community generally rejects the need for such descriptions of "valid data". JSON text is, of course, sufficiently "self-describing" that data encoded in JSON is easily parsed and can often be used in application components that have no specific knowledge of the data contained therein. This last fact explains why JSON is so successful in the broader data management community, in spite of the lack of a standard way to document its structure. The authors of the present proposal believe that there will eventually be a demand for standardized ways of

[5] We have heard that the name "BSON" is intended to evoke the notion of "Binary JSON".

describing JSON data structure for particular applications — that is, a schema language — but that time has not arrived. This presents a key problem that will confront application developers.

[RFC4627] does not provide any mechanism by which JSON values can be tested for validity other than strict syntactic validity, and neither does [JSONintro] or [ECMAscript]. By contrast, [XQuery] permits the specification of an XML Schema against which XML values can be checked for structural (and, to an extent, semantic) validity.

The authors of the present proposal have become aware of at least one effort to define a schema for JSON [JSONschema] that would describe the structure of JSON data to be considered "valid" for some given application. However, the most recent available draft of the specification is dated more than two years ago and we have been unable to ascertain any significant amount of interest in the JSON community.

While a schema language for JSON data could be useful in some circumstances (just as an XML Schema language is valuable in some circumstances), it does not appear that such a language is widely used, and it is beyond the scope of the present proposal to create such a language.

1.1.4.2 Validity

In the XML world, the word "valid" is used to describe the state in which an XML document satisfies the criteria given by an XML Schema (or Document Type Definition, DTD). A (much) less rigorous requirement on XML documents is identified by the phrase "well-formed". Well-formedness in XML is a syntactic condition in which characters that are invalid in XML (*e.g.*, "&" and "<") are not present, elements are correctly nested (*e.g.*, no element "overlap" exists), attribute values do not contain instances of the specific quotation marks (single or double) that are used to enclose the attribute values, *etc.*

There is no term in common use to describe such syntactic requirements on JSON data instances. The present proposal uses the word "valid" to describe data instances that satisfy all JSON syntactic requirements (corresponding to "well-formed" in the XML world). If future proposals define JSON schema facilities, the use of "valid" would likely be extended to incorporate structural/semantic validity in addition to syntactic validity.

In order to make the concept of validity useful, the present proposal specifies a new SQL predicate, IS JSON, to test the (syntactic) validity of JSON data instances. Additional syntax (analogous to that defined in Subclause 11.6 of [XMLCD]) would be specified to test schema-style validity should that be later proposed.

1.1.4.3 Avro schemata

Because Avro is a representation in which each "field" (a bit of JSON data) occupies no more octets than is required, using the particular encoding method for data of each type, it is not possible to simply *index* to a specific field in each Avro string. In fact, because of the way that Avro encodes its fields, it is not possible to scan an Avro string and identify the start of the second, third, or twenty fifth field in that string.

Consequently, Avro specifies that each Avro value be described by an Avro *schema*. Avro schemata are expressed in the character representation of JSON. The schema that represents an entire Avro string is composed of "smaller" schemata that represent each field in the Avro string. The schema describes each field by its name, data type, and (if not already unambiguous) length. Thus, an application wishing to access the fields in an Avro string must first parse the schema of that string, then use that information to locate the desired fields in the string and to "decode" the field contents into a value of a JSON data type.

Because each JSON text can be of a different size or contain different components, one might wish to provide a different schema for each JSON text…a schema that uniquely describes that text and not (necessarily) any other JSON text. This approach necessarily creates an increase in size of the JSON texts associated with those schemata. For small JSON texts and/or JSON texts with a great many fields, the overhead (in octets) of providing a schema for each such text in the Avro representation becomes unacceptable very quickly. For very large JSON texts, particularly with small numbers of (very large) fields, the presence of a per-text schema may be perfectly reasonable.

Avro does not require that a schema be provided along with each JSON text, individually. It does permit that approach, but it also allows for a single schema to describe all of the Avro-represented JSON texts in a "container file." As long as all of the texts in that container file are sufficiently alike, a single schema is adequate — and, of course, produces much-reduced overhead. In the context of the present proposal, an entire column containing the Avro representation of JSON texts acts as the "container file." The Avro schema associated with such a column is part of the metadata describing the column.

In the preceding paragraph, the phrase "sufficiently alike" was used. By that phrase, we mean that each object/array in the rows of a column contain the same number of members/elements, each having the same name (for objects) and data type[6]. But, we also mean that the lengths (number of octets) of each member/element must also be respectively the same, and that's not always easy to ensure.

In order to eliminate the requirement that the lengths of each corresponding member/element in all objects/arrays in the column be equal, the present proposal provides the ability for a column-level schema to describe the objects/arrays in the column in general, but that each object/array also be allowed to contain an object/array-specific schema that augments the column-level schema.

1.1.4.4 BSON schemata

Because the BSON representation of JSON contains a one-octet code as the first octet of every field, it is possible to scan a BSON value and uniquely identify each field and its data type. Consequently, no sort of schema is required for BSON-represented JSON data.

However, BSON — like Avro — uses variable-length fields, so that corresponding members/elements in different objects/arrays can have significantly different lengths. Scanning BSON strings to locate those field codes (and thus the fields themselves) requires CPU cycles. BSON might benefit from having a schema capability somewhat similar to Avro's, but it is certainly not necessary.

1.1.5 Why does JSON matter in the context of SQL? What is JSON's relationship to NoSQL?

It is unclear that JSON and SQL [FoundCD] have any inherent relationship, but it is equally clear that the technical, business, and government worlds are increasingly using both kinds of data (as well as XML data) in their environments. Individual applications are required to access and manipulate all of these kinds of data, often concurrently. As we have discovered with the use of SQL and XML data in individual applications, there are great benefits when a single data management system can concurrently handle all of the data. Among the benefits are: reduced administrative costs, improved security and transaction management, better performance and greater optimizability, better resource allocation, and increased developer productivity.

We are convinced that incorporation of JSON into the SQL umbrella will offer implementers and users alike the benefits described above. That fact easily justifies the relatively small increase in size and complexity of the SQL standard, especially when the approach of the present proposal is used.

[6] Therefore, the JSON objects "**`{ "name" : "Joe", "salary" : 85000 }`**" and "**`{ "name" : "Bob", "salary" : 78000 }`**" are "sufficiently alike, but "**`{ "name" : "Ann", "bonus" : 85000 }`**" is not.

Most readers of the present proposal will have heard the term “NoSQL” [NoSQLdef], variously interpreted as “no SQL” (meaning that SQL is not needed or used) or “Not only SQL” (meaning that SQL may be needed and used, but other data models are of similar importance).

NoSQL[7] database systems (we were tempted to put the latter two words into quotes, but resisted) [NoSQLDB] are generally characterized by the following attributes:

- They do not use the relational model (they also do not use a number of other data models).
- They tend to be focused on “big data” and on applications for which “approximate” answers are often adequate.
- They are optimized for data retrieval, not for data creation or update, nor on the relationships between data.
- They usually do not use ACID [ACIDtxns] transactions; instead, they may offer transactional models that result in “eventual consistency”.
- They tend to be designed using distributed, fault-tolerant, highly parallel hardware and software architectures.

NoSQL database systems come in a very wide variety of kinds, base on their targeted marketplaces, data models, and application requirements. They have been crudely taxonomized into:

- Key-value stores
- “BigTable” stores
- Document stores
- Graph databases

Key-value stores provide exactly the capability that the name implies: applications provide a key and are given a value in return. Key-value stores may manage only “flat” pairs, or they may manage hierarchical pairs.

BigTable stores implement multi-dimensional arrays such that applications provide one or more index values (often strings used as key values, instead of numeric indexes) that together provide the location of a cell, the value of which is returned to the application.

Document stores, contrary to what the name many suggest to many, do not *necessarily* store textual documents such as books, specifications, or magazines; instead, they store data that may be traditional textual documents or organized collections of structured (and semi-structured) data.

Graph databases provide a way to store data that is generally linked together into graphs (often directed graphs, sometimes trees in particular).

Many, but hardly all, NoSQL database systems manage data represented in JSON, especially key-value stores and document stores. Graph databases often manage data that is isomorphic with RDF [RDFmodel] data.

[7] The coiners of the term “NoSQL” did so for reasons of public relations rather than for any inherent bias against the SQL database language. The term was first used to advertise a technical conference addressing open-source, non-relational database systems and was thought to provide an interesting “hook” that would attract attendees.

We have observed that the rapidly increasing use of JSON to interchange data between Web applications has caught the attention of academics, technologists, developers, and even management types. SQL database implementers are increasingly convinced that they must support JSON data "natively" or risk losing market share to implementations of NoSQL database systems. The present proposal offers a way by which such implementers can provide that support in a common manner that will not further fragment the "real" DBMS marketplace.

1.1.6 JSON terminology

Before we discuss the relationship between JSON and SQL, we must settle on specific terminology that the present proposal (and the SQL standard) will use to reference the various "things" incorporated into the JSON data model. JSON is, of course, taken from JavaScript, which has been standardized under the name "ECMAScript" [ECMAscript]. [ECMAscript] defines the terminology for its objects, but [RFC4627] and [JDIF] use terminology that is significantly different. Other specifications associated with JSON use still different terminology.

The present specification proposes to stick as closely as possible to the notation in [RFC4627] in the belief that most users of JSON will be familiar and comfortable with that terminology. To be clear, the following terms and their definitions are used:

Term[8]	**Definition**
JSON text	A sequence of **token**s, which *must* be encoded in Unicode [Unicode] (UTF-8 by default); insignificant white space may be used anywhere in JSON text except within strings (where all white space is significant), numbers, and literals — note that JSON text is a single object or array
Token	One of six structural characters ("{", "}", "[", "]", ":", ","), **string**s, **number**s, and **literal**s
Value	An **object**, **array**, **number**, **string**, or one of three literals
Type	A **primitive type** or a **structured type**
Primitive type	A string, a number, a **Boolean**, or a **null**
Primitive value	A value that is a string, number, Boolean, or null
Structured type	An **object** or an **array**
Structured value	A value that is an object or an array
String	A sequence of Unicode characters; some characters must be "escaped" by preceding them with a reverse solidus ("\"), while any or all characters can be represented in "Unicode notation" comprising the string "\u" followed by four

[8] Each term is implicitly qualified with the prefix "JSON", *e.g.*, JSON value, JSON object. The present proposal uses the qualification whenever the use of the term would risk ambiguity in context and eschews it otherwise.

Term[8]	**Definition**
	hexadecimal digits or two such strings representing the UTF-16 surrogate pairs representing characters not on the Basic Multilingual Plane (strings are surrounded by double-quote characters, which are not part of the value of the strings)
Number	A sequence comprising an integer part, optionally followed by a fractional part and/or an exponent part (non-numeric values, such as infinity and NaN are not permitted)
Boolean	The literal "`true`" or the literal "`false`"
Null	The literal "`null`"
Object	A structure represented by a "{", zero or more **member**s[9] separated by ",", and "}"
Member	A string followed by a colon followed by a value in an object (a member is also known as a "name-value pair"; the name is sometimes called a "key" and the second value is sometimes called a "bound value")
Array	A structure represented by a "[", zero or more **element**s separated by ",", and "]"
Element	A value in an array
Field	A member in an object, an element in an array, a name in a member, or a value in a member
Data model (general)	A definition of what kinds of data belong to a particular universe of discourse, including the operations on those kinds of data
JSON data model	The (implicit[10]) data model associated with JSON
SQL/JSON data model	The data model created for operating on JSON data within the SQL language

1.1.7 Use cases for JSON support in SQL

Even if all readers of the present proposal were in complete agreement about the abstract necessity of supporting JSON data in the SQL environment, we must still identify the particular use cases that such support must satisfy. Our work convinces us that there are three primary use cases:

[9] [RFC4627] states that objects contain *unordered* sequences of name-value pairs, while arrays contain *ordered* sequences of values.

[10] [JSON] does not specify a data model; instead, it specifies a syntax for representation of data. A data model (actually, several possible data models) can be inferred from the syntax specification, hence "implicit data model."

- JSON data ingestion and storage
- JSON data generation from relational data
- Querying JSON as persistent semi-structured data model instances

The following sections discuss these use cases in greater detail.

1.1.7.1 JSON data ingestion and storage

The question posed by this use case is "How can we acquire JSON data using SQL?" For example, great quantities of JSON data already exist in many enterprises and are stored in something like HDFS [HDFS]. It might be possible to treat such "external" stores as though they were a kind of "external table" (*e.g.*, an SQL view mapped appropriately to HDFS), in which case the JSON data could be queried just as though it were in an SQL table.

There are a number of technical issues to be addressed with such an approach, such as the fact that JSON data is nowhere near as structured as SQL table data. The present proposal avoids those technical issues entirely by *not* pursuing this approach.

Instead, the present proposal focuses on ingesting JSON data as character strings that are then stored in ordinary SQL columns of some string type. When such data is retrieved from those columns for use in JSON-based functions, it is transformed (parsed) into instances of an internal SQL/JSON data model that is never directly exposed to the application author.

1.1.7.2 JSON data generation from relational data

This use case asks "How can we (declaratively) generate JSON data instances from relational tables for data export?" Applications that are based on JSON data not only want to store and retrieve such data upon demand, but they typically want their queries against such data to provide results in the same form — JSON. Although it is trivial to provide procedural mechanisms by which applications can laboriously (and with many likely errors) construct JSON data, SQL's declarative nature suggests that JSON objects and arrays should be generated instead of potentially lengthy character strings that represent such objects and arrays. (Readers are cautioned not to misinterpret this use case as requiring provision of a "bulk JSON data export" facility.)

The present proposal addresses this use case by providing several functions that transform the data stored in SQL tables into instances of the internal SQL/JSON data model that can , if needed, be serialized back into character string form.

1.1.7.3 Querying JSON as persistent semi-structured data model instances

With this use case, we explore how we can query JSON data that is stored directly in SQL tables. As will be seen later in the present proposal, we do not propose directly mapping entire SQL tables into single (or, necessarily, multiple) JSON objects or arrays, although we do provide support for such mappings when needed by applications. Instead, we are proposing to store JSON data within an opaque data type (specifically an SQL string or Large Object) that can be manipulated through the functional interface specified in the present proposal.

1.1.8 "Non-Use cases" and other "non-goals"

1.1.8.1 Direct access to external JSON data

All access to JSON data in the present proposal requires that the JSON data first be imported into the SQL environment. This extends to the various built-in functions specified in the present proposal, none of which have parameters that reference external JSON data. Applications are required to access external JSON data themselves, then insert those data into SQL strings (character or binary, as appropriate) and INSERT (or UPDATE) rows in SQL tables containing those strings.

Although the present proposal does not propose facilities to directly query JSON data that is not within the SQL environment, implementations will undoubtedly provide such facilities as extensions.

1.1.8.2 Generation of JSON results containing only atomic values

In the present proposal, all JSON atomic values exposed outside of JSON instances are automatically (by default) cast to corresponding SQL scalar values. Of course, SQL's **`<cast expression>`**s can be used to cause those atomic values to be cast to SQL scalar values of different (that is, non-default) types. We believe that ambiguities will necessarily arise if, for example, we attempt to store JSON atomic values directly in SQL columns. Of course, some future proposal might reconsider this question if solutions are found to that problem.

1.1.8.3 Updating JSON data

The present proposal does not specify any mechanism for modifying JSON data, even if stored in an SQL column. While it may be desirable to support such a mechanism, that is beyond the scope of the present proposal and may be addressed in some future proposal. Readers should note, however, that the use of SQL UPDATE statements to completely *replace* the value of a column in a row is still appropriate, even if that column holds JSON data.

1.1.8.4 Specification of the details of the JSON query language

The syntax and semantics of the query/path language used to locate specific JSON data is not specified in the present proposal, but *is* specified in an accompanying proposal [SQLJSON2].

1.1.9 What features are needed to address those use cases?

Now that we have a better understanding of the use cases that we need to address in order to provide useful and robust support for JSON data in the SQL language, we may explore the specific language features needed to satisfy those use cases.

We express those features using the following notions:

- SQL query expressions must have access to JSON data according to its structure (*e.g.*, using the names of key-value pairs in JSON objects, positions in JSON arrays, *etc.*).
- SQL queries must be able to generate JSON data directly for return to invokers of those queries.
- SQL tables must be able to store JSON data persistently.

In the next sections, we explore each of these "macro-features" in turn.

1.1.9.1 Storing JSON data in an SQL table

The ability to store data — arbitrary data — in SQL tables is not in question. Any data can be massaged into some form that can be stored in plain old, traditional SQL tables…maybe not efficiently, but stored nonetheless. The requirement here is to store JSON data without massaging it into another form, which is an operation that can be both costly and irreversible. What is required is the ability to store *JSON data*, as JSON, in SQL tables, and that requirement can be met in either of two ways. First, we could modify the SQL language to support the JSON data model by making JSON objects and arrays first-class objects in the same way that tables are; this is very undesirable, of course, because it would require substantial modifications to the entire SQL language. Alternatively, we can store JSON data "natively" (more about that shortly) in existing SQL objects — specifically, columns in rows of tables. Readers will recall that the same question was raised by the decision to support XML data within the SQL environment.

The approach taken by the present proposal is to store JSON data into **character string columns or binary string columns** that are defined within ordinary SQL tables. That permits those JSON data to participate in SQL queries (and, importantly, SQL transactions) in the same manner as the data stored in other columns of the same tables. By choosing to use columns whose declared types are **string types**, we avoid the standardization (and implementation) overhead[11] of creating a new built-in SQL data type without losing any significant advantages of a built-in type.

Applications, however, are not expected to provide detailed code to manipulate JSON data in those **strings** directly in the form of character string operations. The present proposal provides a number of built-in SQL functions that access (query) JSON data stored in such columns. These functions are described in section 1.2.3 below.

1.1.9.2 Generating JSON in an SQL query

As discussed above, JSON *text* is one of two kinds of values: JSON objects and JSON arrays. The present proposal provides built-in functions that generate JSON objects and JSON arrays as the results of SQL queries (in the most general sense), whether the source of the data queried is JSON data or ordinary SQL data. These functions are also described in section 1.2.3 below.

1.1.9.3 Querying JSON data in SQL tables using SQL

The mechanisms for storing JSON data in the SQL environment and for generating JSON data out of the SQL environment are perhaps obvious and relatively noncontroversial. However, querying JSON data using the SQL language, whether that data is stored in SQL columns of SQL tables as described in section 1.1.9.1 above, raises an important — and interesting — question: How should those queries be expressed?

SQL queries are expressed using SQL's **`<query expression>`**s. Query expressions, in turn, comprise query specifications, which comprise table expressions, WHERE clauses, and so forth. Each component of a query expression ingests and produces an SQL (virtual) table, because the SQL model has only tables as first-class objects.

JSON's first class "objects" are JSON objects and JSON arrays. What is needed is a language that "understands" the JSON data model and can declaratively express queries against data represented in that model. Readers will recall that the same problem arose when the inclusion of XML into the SQL environment

[11] See Section 1.2.1 for more information about this decision.

was proposed; the solution in that case was to use the best-known and standardized language designed for querying XML — XQuery (see [XQuery]).

There is not currently an obvious analog to XQuery that applies to JSON. That is, there is not a well-known, standardized, and universally accepted language that queries data represented in the JSON data model. Instead, there are several languages that enjoy varying degrees of support in the JSON community, none of which have been adopted by a recognized standardization body. Naturally, their merits and disadvantages lie largely in the mind of each individual observer, and none are universally accepted as "the solution" to this particular problem. The present paper does not specify any query language. Instead, an accompanying proposal [SQLJOSN2] specifies a JSON query language (the SQL/JSON path language) that is best suited for the use cases outlined in Section 1.1.7 above.

1.2 Architecture and design

1.2.1 Lightweight

The present proposal deliberately takes a "lightweight" approach to specifying support for JSON in the context of the SQL language. The primary reason for taking this approach is to improve the chances of its rapid adoption into the SQL standard, as well as the rapid adoption by SQL implementers.

There are four specific facets (each discussed separately below) to the present proposal that represents decisions made in alignment with the lightweight approach:

— First, the present proposal does not provide a new data type for JSON data; this is the opposite of the approach taken for XML data in the SQL standard [XMLCD]. This is a key decision, made with the knowledge that defining new data types in the SQL standard have extensive (and expensive) ramifications to both the standard and its implementations.

— The present proposal makes the decision that JSON is JSON, whether it is represented in its character representation or some binary representation. See section 1.2.3 below for discussion of this point.

— All operations on JSON data (including conversion, creation, storing, and querying) are done through various standard-defined functions specified in the present proposal. Other functions might be proposed in the future as need becomes apparent.

— A standard JSON query language is implied by the present proposal and defined in accompanying proposal [SQLJSON2]), which takes the approach of requiring only a rather minimal language. The language can be extended, or other languages permitted, if and when the need becomes apparent.

1.2.2 No native JSON data type in SQL

The present proposal treats JSON as a sort of pseudo-type based on character strings (and binary strings). The primary reason for this decision is the costs associated with adding new data types to the SQL standard and, perhaps more importantly, to implementations of the standard.

When [XMLCD] was first being developed, the opposite decision — to create a new data type for XML data — was made. That choice was made palatable within the SQL standard by eschewing the definition of host language bindings for the new type in both module language and in embedded SQL. In fact, the grammar used in embedded SQL explicitly uses syntax such as:

```
SQL TYPE IS XML ... AS VARCHAR
SQL TYPE IS XML ... AS BLOB
SQL TYPE IS XML ... AS CLOB
```

(It's also worth noting that the XML data type incorporates modifiers that make a given site of XML type be specifically an XML document, XML content, or an XQuery Data Model [XDM] sequence, and that the values of XML type in that site are either untyped, any XML type, or only a particular XML Schema [XSD] type.)

The present proposal takes the position that the XML type in SQL is not a "real" data type, but a sort of synonym (or alias) for VARCHAR, BLOB, or CLOB. Instead of proposing the same for JSON data, partly because no corresponding modifiers are relevant, the present proposal avoids the pretense that JSON is inherently a new type of data in SQL and simply allows the application author to explicitly use CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, NATIONAL CHARACTER VARYING, NATIONAL CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT as she chooses.

This decision to not create an SQL type specifically to store JSON data implies that application authors must take responsibility for tracking which sites of string types store JSON and which do not. Those that store JSON data must be identified to the various functions provided for handling JSON data by marking them with AS JSON clauses.

1.2.3 JSON is JSON

As discussed in section 1.1.3 above, the present proposal takes the position that all JSON data, whether represented in its character string form or in some binary string form, is nothing other than JSON data. (That is, we do not discuss JSON data as distinguished from Avro data or BSON data[12]. All operations on JSON data in its character string form are equally applicable — with the identical semantics — if applied to JSON data in one of its binary forms.) Consequently, very little in the present proposal acknowledges the fact that multiple representations of JSON may exist.

The primary evidence of support for both plain-text representations of JSON and various binary representations is found in rules that cause a character string representation of JSON to be produced (by default) by a query when the JSON input to the query has a character string representation, and correspondingly for input with a binary string representation. In each case, the query author is free to use implementation-defined syntax (if any) that over-rides the default and produces some binary representation of JSON when the source JSON data is provided in its plain-text representation, and *vice-versa*. Plain-text representations of JSON are provided in both character data types (CHARACTER VARYING, CHARACTER LARGE OBJECT, NATIONAL CHARACTER VARYING, NATIONAL CHARACTER LARGE OBJECT) and binary data types (BINARY VARYING, BINARY LARGE OBJECT). Binary representations would likely be provided only in binary data types.

1.2.4 Handle JSON using built-in functions

In [XMLCD], operations on XML data are generally performed using a set of built-in functions specified explicitly for that purpose. The present proposal takes exactly the same approach. We refer to the functions specified in the present proposal as "the SQL/JSON functions" for convenience.

We partition the SQL/JSON functions into two groups: constructor functions and query functions. Constructor functions use ordinary SQL aggregates on values of SQL types and produce JSON values. Query functions

[12] The reader is reminded that the present proposal does not specify *any* binary representations of JSON.

evaluate SQL/JSON path language expressions against JSON values, producing values of SQL types. Constructor functions may also be called “publishing functions” as [XMLCD] calls its analogous functions.

The four query functions (JSON_VALUE, JSON_TABLE, JSON_EXISTS, and JSON_QUERY) are similar in two respects: their second arguments are “path” expressions used to locate values within the JSON texts passed as the value of their first arguments; and they return SQL values. The four constructor functions (JSON_OBJECT, JSON_OBJECTAGG, JSON_ARRAY, and JSON_ARRAYAGG) return JSON values (objects or arrays).

To illustrate the use of the SQL/JSON constructor functions, we will use the following two ordinary SQL tables and one additional table that incorporates a single column of JSON data:

```
CREATE TABLE DEPTS (
  dept_no     INTEGER,
  dept_name   CHARACTER VARYING(30) )

CREATE TABLE JOBS (
  job_seq     INTEGER,
  job_attrib  CHARACTER(5),
  job_attval  CHARACTER VARYING(64) )

CREATE TABLE employees (
  emp_id      INTEGER,
  department  INTEGER,
  json_emp    CHARACTER VARYING (5000) )
```

1.2.4.1 JSON_OBJECT

SQL applications working with JSON data will often need to construct new JSON objects, either for use within the applications themselves, for storage in the SQL database, or to return to the application program itself. The present proposal specifies a built-in function, JSON_OBJECT, that constructs JSON objects from explicit name/value pairs; the names in those name/value pairs must be SQL identifiers, while the values may be specified as SQL literals or as any other SQL expressions — including subqueries.

JSON_OBJECT is somewhat analogous to XMLELEMENT in [XMLCD]. It is typically used in **`<select list>`**s.

The following example illustrates the usage of JSON_OBJECT:

```
SELECT
  JSON_OBJECT( 'deptno' : d.deptno,
               'deptname' : d.deptname )
FROM depts AS d
```

This query returns one row for each department recorded in the DEPTS table; that row contains a single column, which contains a serialization of a JSON object having the department number and name. Visually, the returned JSON object in the only column of the first row of the table would look something like this:

(unnamed) [VARCHAR(id[13])]
{ "deptno" : 314, "deptname" : "Engineering" }

1.2.4.2 JSON_OBJECTAGG

Often, it is inappropriate or even impossible to construct a JSON object by explicitly specifying the names of the contained name/value pairs (*e.g.*, because the names are not known *a priori*). Instead, an application developer may wish to construct a JSON object as an aggregation of information in an SQL table. Presuming that the SQL table actually contains a column with JSON names and another column with corresponding values, the built-in function JSON_OBJECTAGG ("object aggregate") performs this function.

The following example will create a JSON object containing a sequence of name/value pairs in which the name is a department name and the value is the department number:

```
SELECT JSON_OBJECTAGG ( dept_name, dept_no )
FROM depts
```

The result of this query is a table containing a single row of one column, which contains a serialization of a JSON object. That object would look something like this:

```
{ "Engineering" : 314, "Architecture" : 113, "Accounting" : 12,
  "Sales" : 7, "Executives" : 13 }
```

The reader of the present proposal will observe that this is actually a kind of *pivot* of the DEPTS table.

The JSON_OBJECTAGG function can also be used in grouped queries to good effect. Consider the JOBS table with the following contents:

JOB_SEQ	JOB_ATTRIB	JOB_ATTVAL
101	Leader	155566
101	Duration	00:30:00
101	Description	Design the new tables for the web site
234	Duration	01:00:00
234	Description	Load the tables with existing data
492	Leader	129596
17	Description	Design the look-and-feel of the web site

The SQL query:

[13] "id" represents an implementation-defined length.

```
SELECT  j.job_seq,
   JSON_OBJECTAGG( j.job_attrib, j.job_attval RETURNING VARCHAR(80) )
     AS attributes
FROM jobs j
GROUP BY j.job_seq
```

will produce a table containing four rows, each containing two columns. The first column of the table contains the job sequence numbers, while the second column contains a serialization of a JSON object that is a *pivot* of the information in all of the rows associated with the corresponding job sequence number. The result will look something like this:

JOB_SEQ	**ATTRIBUTES [VARCHAR(id[13])]**
101	{ "Leader" : "155566", "Duration" : "00:30:00", "Description" : " Design the new tables for the web site " }
234	{ "Duration" : "01:00:00", "Description" : "Load the tables with existing data" }
492	{ "Leader" : "129596" }
17	{ "Description" : " Design the look-and-feel of the web site" }

1.2.4.3 JSON_ARRAY

Just as an application developer might wish to construct a JSON object from an explicit list of data, she might wish to construct a JSON array from a similar list of data. The proposed built-in function JSON_ARRAY provides that capability. Its syntax and semantics are very much like those of JSON_OBJECT, excepting (of course) that the result is a JSON array instead of a JSON object.

JSON_ARRAY constructs a JSON array, each element of which is taken from the rows selected in the containing SQL query.

Unlike JSON_OBJECT, JSON_ARRAY has two variants: One variant produces its result from an explicit list of values, not name/value pairs; the values are SQL values (literals or computed values, including subqueries); the second variant produces its results from an SQL query expression invoked within the function.

The following query illustrates the use of JSON_ARRAY:

```
SELECT
  JSON_ARRAY( 'deptno', d.deptno, 'deptname', d.deptname )
FROM depts AS d
```

This query returns one row for each department recorded in the DEPTS table; that row contains a single column, which contains a serialization of a JSON array having the department number and name, as well as another object with the identifier and name of the employee whose name comes first in alphabetical order. Visually, the returned JSON object in the only column of the first row of the table would look something like this:

```
[ "deptno", 314, "deptname", "Engineering" ]
```

1.2.4.4 JSON_ARRAYAGG

Just as an SQL application might need to construct a JSON object as an aggregation of SQL data, so might it need to construct a JSON array as an aggregate. This function's design was based on XMLAGG in [XMLCD].

For example:

```
SELECT JSON_ARRAYAGG( j.job_attval RETURNING CLOB(8K) )
         AS attributes
FROM jobs j
```

The result of this query is a table of one row and one column, which would look something like this:

ATTRIBUTES
["155566", "00:30:00", "Design the new tables for the web site", "01:00:00", "Load the tables with existing data", "129596", "Design the look-and-feel of the web site"]

JSON_ARRAYAGG supports an optional ORDER BY clause that allows the results of the query to be ordered before the selected data is extracted to be placed in the resulting JSON array.

1.2.5 Additional SQL syntax enhancements

In addition to the built-in functions discussed in section 1.2.4 above, there are a few enhancements to SQL syntax that are required for meaningful JSON support. These include syntax for a predicate to test the validity of JSON data instances, as well as syntax to specify that input arguments or results are to be JSON values instead of ordinary SQL values.

1.2.5.1 JSON input clause

Whenever JSON data is being passed as an argument into a function, the application author must specify that the value of the argument is, in fact, JSON data. The syntax used to specify that fact is:

```
<JSON input clause> ::=
    FORMAT <JSON input representation>

<JSON input representation> ::=
    JSON
  | <implementation-defined JSON representation option>
```

The <implementation-defined JSON representation option> might, for example, specify some binary representation of JSON, such as Avro or BSON.

1.2.5.2 JSON output clause

Whenever JSON data is returned as the result of a function, the application author will normally wish to control the form in which that JSON data is returned. The syntax used to specify that form is:

```
<JSON output clause> ::=
    RETURNING <data type> [FORMAT <JSON output representation> ]
```

```
<JSON output representation> ::=
    JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
  | <implementation-defined JSON representation option>
```

1.2.5.3 IS JSON predicate

Applications will frequently want to ensure that the data they expect to consume as JSON data is, indeed, JSON data. The IS JSON predicate determines whether the value of a specified string does or does not conform to the structural rules for JSON. The syntax of the IS JSON predicate is:

```
<JSON predicate> ::=
    <string value expression>
    [ FORMAT <JSON input representation> ]
    IS [ NOT ] JSON
    [ <JSON predicate uniqueness constraint> ]

<JSON predicate uniqueness constraint> ::=
    WITH UNIQUE [ KEYS ]
  | WITHOUT UNIQUE [ KEYS ]
```

1.2.6 Handling of JSON nulls and SQL nulls

SQL (correctly) distinguishes between data such as zero-length strings and the special pseudo-value known as "the null value". The semantics of those two things are quite different and those differences affect a great many SQL operations. The differences are an important part of the semantics of the SQL language.

The present proposal, by adding support for JSON data in the context of SQL, adds yet another related difference: the JSON null. In JSON, null is an actual value, represented by a JSON literal ("**`null`**"). We must be able to distinguish JSON nulls from SQL null values and that distinction will be an important part of the semantics of JSON handling in the SQL context.

To illustrate the situation, consider the JSON object stored in a column of an SQL table:

`{ "a" : null, "b" : "null", "c" : "" }`

JSON_VALUE, evaluated against that JSON object, returning the result as an SQL scalar value, would return, for each respective name/value pair, the following: the SQL null value, the SQL scalar value "**`'null'`**", and the SQL scalar value "**`''`**"; if JSON_VALUE were used to retrieve the value associated with the name "d", it would return the SQL null value. Note that, when retrieving the value of the first name/value pair, the JSON value "null" is automatically transformed into an SQL null value.

The JSON constructor functions have to deal with situations in which the SQL data that is being queried are SQL null values. The present proposal supplies optional syntax to allow the application author to select whether SQL null values are included in the JSON object or JSON array being constructed, or whether object members or array elements whose (bound) values are SQL null values are omitted from the JSON object or JSON array being constructed.

1.2.7 Conformance to JSON constructor functions

There are three conformance features defined for the JSON constructor functions:

- Tx11 — Basic SQL/JSON constructor functions (everything in the present proposal except the two items covered by the other two features)
- Tx12 — SQL/JSON: JSON_OBJECTAGG
- Tx13 — SQL/JSON: JSON_ARRAYAGG with ORDER BY

2. Proposal

Unless otherwise specified, all changes proposed in this section of the present paper are to [FoundationCD].

2.1 Changes to Clause 2, "Normative references"

2.1.1 Changes to Subclause 2.2, "Other international standards"

2.1.1.1 INSERT the following normative references in the appropriate places

[ECMAscript] ISO/IEC 16262:2011, Information technology -- Programming languages, their environments and system software interfaces -- ECMAScript language specification; also available as *ECMAScript Language Specification*, http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf

[RFC4627] RFC 4627, The application/json Media Type for JavaScript Object Notation (JSON), D. Crockford, July 2006; http://tools.ietf.org/html/rfc4627

[Unicode] The Unicode Standard, http://unicode.org

2.2 Changes to Clause 3, "Definitions, notations, and conventions"

2.2.1 Changes to Subclause 3.1.6, "Definitions provided in Part 2"

2.2.1.1 INSERT the following definitions in the appropriate places

3.1.6.x Data model (general)

definition of what kinds of data belong to a particular universe of discourse, including the operations on those kinds of data

3.1.6.x JSON array

structure represented by a "[", zero or more elements separated by ",", and "]"

3.1.6.x JSON Boolean

JSON literal "**`true`**" or JSON literal "**`false`**"

3.1.6.x JSON element

JSON text fragment that is a JSON value in a JSON array

3.1.6.x JSON data model

(implicit) data model associated with JSON

3.1.6.x JSON member

JSON string followed by a colon followed by a JSON value in a JSON object

NOTE nnn — A member is also known as a "name-value pair"; the name is sometimes called a "key" and the second value is sometimes called a "bound value"

3.1.6.x JSON null

JSON literal "`null`"; a JSON null is distinct from an SQL null value and from an SQL/JSON null

3.1.6.x JSON number

Unicode character string comprising an integer part, optionally followed by a fractional part and/or an exponent part

3.1.6.x JSON object

structure represented by a "{", zero or more members separated by ",", and "}"

3.1.6.x JSON string

Unicode character string; some characters must be "escaped" by preceding them with a reverse solidus ("\"), while any or all characters can be represented in "Unicode notation" comprising the string "\u" followed by four hexadecimal digits or two such strings representing the UTF-16 surrogate pairs representing characters not on the Basic Multilingual Plane (strings are surrounded by double-quote characters, which are not part of the value of the strings)

NOTE nnn — This definition applies only to JSON tokens in JSON text.

3.1.6.x JSON text

sequence of JSON tokens, which must be encoded in Unicode [Unicode] (UTF-8 by default); insignificant white space may be used anywhere in JSON text except within strings (where all white space is significant), numbers, and literals

NOTE nnn — JSON text is a single JSON object or JSON array

3.1.6.x JSON text fragment

substring of a JSON text that conforms to any BNF non-terminal in [RFC4627]

3.1.6.x JSON token

one of six structural characters ("{", "}", "[", "]", ":", ","), JSON strings, JSON numbers, and JSON literals

3.1.6.x JSON value

JSON object, JSON array, JSON number, JSON string, or one of three JSON literals

3.1.6.x SQL/JSON data model

data model created for operating on JSON data within the SQL language

2.3 Changes to Clause 4, "Concepts"

2.3.1 INSERT a new Subclause:

4.x JSON data handling in SQL

4.x.1 Introduction

JSON (an acronym for "JavaScript Object Notation") is both a *notation* (that is, a syntax) for representing data *and* a[n implied] data model. JSON is not an *object-oriented* data model in the classic sense; that is, it does not define sets of classes and methods, type inheritance, or data abstraction. Instead, JSON "objects" are simple data structures, including arrays. Some sources say that JSON is a *serialization* of structured data. Its initial intended use was as a data transfer syntax. The syntax of JSON is specified very concisely in [JDIF]. However, for reasons related to nomenclature, this International Standard uses terminology and concepts from [RFC4627].

The first-class components of the JSON data model are *objects* and *arrays*. A JSON object is zero or more name-value pairs and is enclosed in curly braces — {…}. A JSON array is an ordered sequence of zero or more values and is enclosed in square brackets — […].

In a JSON object, the name-value pairs are separated by commas, and the names are separated from the values by colons. The names are always strings and are enclosed in (double) quotation marks. In a JSON array, the values are also separated by commas. The values in both JSON objects and JSON arrays may be JSON strings, JSON numbers, JSON Booleans (represented by the JSON literals **`true`** and **`false`**), JSON nulls (represented by the JSON literal **`null`**), JSON objects, or JSON arrays. JSON arrays and JSON objects are fully nestable. That is, any JSON value is permitted to be an "atomic" value (a string, number, or literal), a JSON object, or a JSON array.

JSON is sometimes used to represent *associative arrays* — arrays whose elements are addressed by content, not by position. An associative array can be represented in JSON as a JSON object whose members are name-value pairs; the name is used as the "index" into the "array" — that is, to locate the appropriate member in the JSON object — and the value is used as the content of the appropriate member.

Part of JSON's design is that it is inherently schema-less. Any JSON object can be modified by adding new name-value pairs, even with names that were never considered when the object was initially created or designed. Similarly, any JSON array can be modified by changing the number of values in the array. One consequence of JSON's schema-less nature is that it is not possible to determine the *validity* of JSON data, except by application programs. However, a JSON processing system can determine by direct inspection whether or not a given bit of JSON data is well-formed (that is, whether it obeys the syntax defined in [JDIF]).

NOTE nnn — In the context of SQL, JSON objects and JSON arrays cannot be modified *in situ*. Instead, new JSON objects or JSON arrays are constructed that may strongly resemble an existing JSON object or JSON array and then used to replace the existing JSON object or JSON array.

NOTE xxx — For all SQL/JSON functions, JSON data that acts either as an argument or as the result is represented as character strings or binary strings.

4.x.2 Implied JSON data model

The implied JSON data model comprises JSON text and certain kinds of values represented as JSON text fragments.

NOTE nnn — While [JDIF] specifies only the syntax used by JSON to represent data, other documents (*e.g.*, [RFC4627]) imply a data model derived from that syntax. That implied data model is insufficiently complete or precise to form the basis for standardization of JSON use in an SQL environment. Consequently, this International Standard specifies an SQL/JSON data model.

The components of the implied JSON data model are:

— A *JSON text* is a character string or binary string that conforms to the definition of "JSON-text" in [RFC4627].

— A *JSON text fragment* is a substring of a JSON text that conforms to any BNF non-terminal in [RFC4627].

— A *JSON literal* is a JSON text fragment that is any of the key words **`true`**, **`false`** , or **`null`**.

— A *JSON member* is a JSON text fragment that conforms to the definition of member in [RFC4627] section 2.2 "Objects". If *M* is a JSON member, then *M* matches the BNF **`member = string name-separator value`** ; the *key* of *M* is the JSON fragment matching **`string`** in this production, and the *bound value* of *M* is the JSON fragment matching **`value`** in this production.

— A *JSON object* is a JSON text fragment that conforms to the definition of object in [RFC4627] section 2.2 "Objects".

— A *JSON array* is a JSON text fragment that conforms to the definition of array in [RFC4627] section 2.3 "Arrays".

— A *JSON number* is a JSON text fragment that conforms to the definition of number in [RFC4627] section 2.4 "Numbers".

— A *JSON string* is a JSON text fragment that conforms to the definition of string in [RFC4627] section 2.5 "Strings".

— The *value* of a *JSON string* is the Unicode character string enclosed in the delimiting <double quote>s of a JSON string.

NOTE nnn — Within JSON strings, certain characters are represented using an "escaped notation" that comprises a <reverse solidus> followed by the desired character. For example, a <double quote> in a JSON string is represented by the sequence <reverse solidus><double quote> (**`\"`**). (The complete list of Unicode characters that can be represented in a JSON string *only* by such an escape sequence is: ", \, /, backspace, form feed, line feed, carriage return, and tab; in addition arbitrary Unicode characters can be included by using "\u" followed by four hexadecimal digits.) The value of a JSON string is determined after replacing all such escaped sequences with their equivalent Unicode values.

NOTE nnn — The implied JSON data model is specified in terms of the Unicode character strings that contain JSON text. The facilities specified in this International Standard do not utilize the implied JSON data model except when parsing JSON text into the SQL/JSON data model or when serializing values of the SQL/JSON data model into JSON text.

— A *JSON value* is a JSON object, JSON array, JSON number, JSON string, or one of three JSON literals.

4.x.3 SQL/JSON data model

The SQL/JSON data model comprises SQL/JSON items and SQL/JSON sequences. The components of the SQL/JSON data model are:

— An *SQL/JSON item* is defined recursively as any of the following:

- An *SQL/JSON scalar*, defined as a non-null value of any of the following predefined (SQL) types: character string with character set Unicode, numeric, Boolean, or datetime.
- An *SQL/JSON null*, defined as a value that is distinct from any value of any SQL type.

  NOTE nnn — An SQL/JSON null is distinct from the null value of any SQL type.
- An *SQL/JSON array*, defined as an ordered list of zero or more SQL/JSON items, called the *SQL/JSON elements* of the SQL/JSON array.
- An *SQL/JSON object*, defined as an unordered collection of zero or more SQL/JSON members, where an *SQL/JSON member* is a pair whose first value is a character string with character set Unicode and whose second value is an SQL/JSON item. The first value of an SQL/JSON member is called the *key* and the second value is called the *bound value*.

  NOTE nnn — [RFC 4627] section 2.2 "Objects" says "The names within an object SHOULD be unique". Thus non-unique keys are permitted but not advised. The user may use the WITH UNIQUE KEYS clause in the <JSON predicate> to check for uniqueness if desired.

— An *SQL/JSON sequence* is an ordered list of zero or more SQL/JSON items.

The maximum number of SQL/JSON elements in an SQL/JSON array is implementation-defined.

The maximum number of SQL/JSON members in an SQL/JSON object is implementation-defined.

The maximum length of the key in an SQL/JSON member is implementation-defined.

If the declared type of an SQL/JSON element or of the key of an SQL/JSON member is a string type, the maximum length is implementation-defined.

NOTE nnn — There is no SQL <data type> whose value space is SQL/JSON items, or SQL/JSON sequences.

Two SQL/JSON items are *comparable* if one of them is the SQL/JSON null, or if both are in one of these types: character string, numeric, Boolean, DATE, TIME, TIMESTAMP.

Two SQL/JSON items *SJI1* and *SJI2* are said to be *equivalent*, defined recursively as follows:

— If *SJI1* and *SJI2* are non-null values of a predefined type, then *SJI1* and *SJI2* are equivalent if they are equal.

— If *SJI1* and *SJI2* are the SQL/JSON null, then *SJI1* and *SJI2* are equivalent.

— If *SJI1* and *SJI2* are SQL/JSON arrays, then *SJI1* and *SJI2* are equivalent if they are of the same length *N*, and corresponding elements of *SJI1* and *SJI2* are equivalent.

  NOTE nnn — "Corresponding elements" in two arrays are elements that have the same index position in both arrays.

— If *SJI1* and *SJI2* are SQL/JSON objects, then *SJI1* and *SJI2* are equivalent if they have the same number of members, and there exists a bijection *B* from *SJI1* to *SJI2* mapping each SQL/JSON member *M* of *SJI1* to an SQL/JSON member *B*(*M*) of *SJI2* such that the key and bound value of *M* are equivalent to the key and bound value of *B*(*M*), respectively, for all members *M* of *SJI1*.

4.x.4 SQL/JSON functions

All manipulation (*e.g.*, retrieval, creation, testing) of SQL/JSON items is performed through a number of SQL/JSON functions.

There are 11 such functions, categorized as SQL/JSON retrieval functions and SQL/JSON construction functions. The SQL/JSON retrieval functions are characterized by operating on JSON data and returning an SQL value (possibly a Boolean value) or a JSON value. The SQL/JSON construction functions return JSON data created from operations on SQL data or other JSON data.

The SQL/JSON retrieval functions are:

— <JSON value function>: extracts an SQL value of a predefined type from **a JSON text**.

— <JSON query>: extracts **a JSON text** from **a JSON text**.

— <JSON table>: converts **a JSON text** to an SQL table.

— <JSON predicate>: tests whether a string value is or is not properly formed JSON text.

— <JSON exists predicate>: tests whether an SQL/JSON path expression returns any SQL/JSON items.

The SQL/JSON construction functions are:

— <JSON object constructor>: generates a string that is a serialization of an SQL/JSON object.

— <JSON array constructor>: generates a string that is a serialization of an SQL/JSON array.

— <JSON object aggregate constructor>: generates, from an aggregation of SQL data, a string that is a serialization of an SQL/JSON object.

— <JSON array aggregate constructor>: generates, from an aggregation of SQL data, a string that is a serialization of an SQL/JSON array.

A *JSON-returning function* is an SQL/JSON construction function or JSON_QUERY.

4.x.5 OTHER POSSIBLE SECTIONS???

Draft whatever additional text is required.

2.3.2 Changes to Subclause 4.16.4, “Aggregate functions”

2.3.2.1 REPLACE the fourth paragraph (“Every other aggregate function…”) with:

Every other aggregate function may be classified as a *unary group aggregate function*, a *binary group aggregate functions*, an *inverse distribution*, ~~or~~ a *hypothetical set function*, or a *JSON aggregate function*.

2.3.2.2 APPEND after the last paragraph (“The hypothetical set functions…”):

The JSON aggregate functions transform information in rows of SQL tables into JSON objects (JSON_OBJECTAGG) and JSON arrays (JSON_ARRAYAGG).

2.4 Changes to Clause 5, “Lexical elements”

2.4.1 Changes to Subclause 5.2, “<token> and <separator>”

2.4.1.1 **INSERT** the following new <reserved word>s in the appropriate locations:

- JSON_ARRAY
- JSON_ARRAYAGG
- JSON_OBJECT
- JSON_OBJECTAGG

2.4.1.2 **INSERT** the following new <non-reserved word>s in the appropriate locations:

- ENCODING
- FORMAT
- JSON
- RETURNING
- UTF16
- UTF32
- UTF8

2.5 Changes to Clause 6, “Scalar expressions”

2.5.1 Modify Subclause 6.31, “<string value function>”

2.5.1.1 **INSERT** the following new alternative to the production for <string value function>:

```
  | <JSON value constructor>
```

2.5.1.2 **MODIFY** Syntax Rule 1), “The declared type of…”

09 The declared type of <string value function> is the declared type of the immediately contained <character value function>, ~~or~~ <binary value function>, or <JSON value constructor>.

2.5.1.3 **MODIFY** General Rule 1), “The result of…”

The result of <string value function> is the result of the immediately contained <character value function>, ~~or~~ <binary value function>, or <JSON value constructor>.

2.5.2 INSERT a new Subclause

6.x <JSON value constructor>

Function

Generate a JSON text fragment.

Format

```
<JSON value constructor> ::=
    <JSON object constructor>
  | <JSON array constructor>

<JSON object constructor> ::=
    JSON_OBJECT <left paren>
    [ <JSON name and value> [ { <comma> <JSON name and value> }... ]
    [ <JSON object null clause> ] ]
    [ <JSON output clause> ]
    <right paren>

<JSON name and value> ::=
    <JSON name> <colon> <JSON value expression>

<JSON name> ::=
    <character value expression>

<JSON object null clause> ::=
    NULL ON NULL
  | ABSENT ON NULL

<JSON array constructor> ::=
    <JSON array constructor by enumeration>
  | <JSON array constructor by query>

<JSON array constructor by enumeration> ::=
    JSON_ARRAY <left paren>
    [ <JSON value expression> [ { <comma> <JSON value expression> }... ]
    [ <JSON array null clause> ] ]
    [ <JSON output clause> ]
    <right paren>

<JSON array null clause> ::=
    NULL ON NULL
  | ABSENT ON NULL

<JSON array constructor by query> ::=
    JSON_ARRAY <left paren>
    <query expression> [ <JSON input clause> ]
    [ <JSON array null clause> ]
    [ <JSON output clause> ]
    <right paren>
```

Syntax Rules

1) The declared type of a <JSON value constructor> is the declared type of its immediately contained <JSON object constructor> or <JSON array constructor>.

2) If <JSON output clause> is not specified, then RETURNING *ST* FORMAT JSON is implicit, where *ST* is an implementation-defined string type.
3) Let *JVCFDT* be the <data type> immediately contained in the explicit or implicit <JSON output clause> *JOC* and let *JVCFFO* be the explicit or implicit <JSON output representation> of *JOC*.
4) The Syntax Rules of Subclause 9.x "Serializing an SQL/JSON item" are applied with *JVCFFO* as the *FORMAT OPTION* and *JVCFDT* as the *TARGET TYPE*.
5) If <JSON object constructor> is specified, then:
   a) If <JSON object null clause> is not specified, then NULL ON NULL is implicit.
   b) The declared type of <JSON object constructor> is *JVCFDT*.
6) If <JSON array constructor> is specified, then:
   a) If <JSON array null clause> is not specified, then ABSENT ON NULL is implicit.
   b) The declared type of <JSON array constructor> is *JVCFDT*.
   c) If <JSON array constructor by query> is specified, then the <query expression> *QE* shall be of degree 1 (one).

Access Rules

None.

General Rules

1) The value of a <JSON value constructor> is the value of its immediately contained <JSON object constructor> or <JSON array constructor>.
2) If <JSON object constructor> *JOC* is specified, then:
   a) If the length of the value of any <JSON name> simply contained in *JOC* exceeds its implementation-defined maximum length, then an exception condition is raised: *data exception — string data, right truncation*.
   b) If any <JSON value expression> simply contained in *JOC* contains a <string value expression> *SVE* and the length of the value of *SVE* exceeds its implementation-defined maximum length, then an exception condition is raised: *data exception — string data, right truncation*.
   c) Let *NNV* be the number of <JSON name and value>s immediately contained in *JOC*.
   d) Case:
      i) If *NNV* = 0, then let *CJO* be a JSON object with no members.
      ii) Otherwise:
         1) For each i, 1 (one) ≤ i ≤ *NNV* :
            A) Let JNV_i be the i-th <JSON name and value> immediately contained in *JOC*.
            B) Let JN_i be the <JSON name> immediately contained in JNV_i.
            C) Let VJN_i be the value of JN_i.
            D) If VJN_i is the null value, then an exception condition is raised: *data exception — null value not allowed*.

E) Let JVE_i be the <JSON value expression> immediately contained in JNV_i and let $VJVE_i$ be the value of JVE_i.

F) Case:

I) If $VJVE_i$ is a null value, then let JBV_i be the SQL/JSON null.

II) If JVE_i specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.x "Parsing a JSON text" are applied, with JVE_i as the *JSON TEXT*, *JFO* as the *FORMAT OPTION*, and an implementation-defined <JSON predicate uniqueness constraint> as the *UNIQUENESS CONSTRAINT*. Let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* that are returned by that Subclause.

Case:

1) If *ST* is an exception condition, then the exception condition *ST* is raised.

2) Otherwise, let JBV_i be *SJI*.

III) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JBV_i be $VJVE_i$.

2) Otherwise, let JBV_i be the result of

CAST ($VJVE_i$ AS *SDT*)

where *SDT* is an implementation-defined character string type with character set Unicode.

G) Let M_i be the SQL/JSON member whose key is VJN_i and whose bound value is JBV_i.

2) If, for any i, 1 (one) $\leq i \leq NNV$, and any j, $i < j \leq NNV$, $VJN_i = VJN_j$, then an exception condition is raised: *data exception — duplicate JSON object key value*.

3) Case:

A) If the implicit or explicit <JSON object null clause> specifies NULL ON NULL, then

Case:

I) If *NNV* is greater than the implementation-defined maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) $\leq i \leq NNV$.

B) Otherwise,

Case:

I) If the number of M_i, 1 (one) $\leq i \leq NNV$, whose bound values are not the SQL/JSON null is greater than the implementation-defined maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) ≤ *i* ≤ *NNV*, whose bound values are not the SQL/JSON null.

NOTE nnn — There is no implied order of the members of the constructed JSON object.

e) Let *JVCF* be *CJO*.

3) If <JSON array constructor by enumeration> *JAC* is specified, then:

a) If any <JSON value expression> simply contained in <JSON array constructor by enumeration> contains a <string value expression> *SVE* and the length of the value of *SVE* exceeds its implementation-defined maximum length, then an exception condition is raised: *data exception — string data, right truncation*.

b) Let *NJVE* be the number of <JSON value expression>s immediately contained in *JAC*.

c) Case:

i) If *NJVE* is 0 (zero), then let *CJA* be a JSON array with no elements.

ii) Otherwise:

1) For each *i*, 1 (one) ≤ *i* ≤ *NJVE*:

A) Let JVE_i be the *i*-th <JSON value expression> immediately contained in *JAC* and let $VJVE_i$ be the value of JVE_i.

B) Case:

III) If $VJVE_i$ is a null value, then let JE_i be the SQL/JSON null.

IV) If JVE_i specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.x "Parsing a JSON text" are applied, with JVE_i as the *JSON TEXT*, *JFO* as the *FORMAT OPTION*, and an implementation-defined <JSON predicate uniqueness constraint> as the *UNIQUENESS CONSTRAINT*. Let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* that are returned by that Subclause.

Case:

1) If *ST* is an exception condition, then the exception condition *ST* is raised.

2) Otherwise, let JE_i be *SJI*.

V) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JE_i be $VJVE_i$.

2) Otherwise, let JE_i be the result of

CAST ($VJVE_i$ AS *SDT*)

where *SDT* is an implementation-defined character string type with character set Unicode.

4) Case:

   A) If the implicit or explicit <JSON array null clause> specifies NULL ON NULL, then

      Case:

      I) If *NJVE* is greater than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.

      II) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ *i* ≤ *NJVE*.

   B) Otherwise,

      Case:

      I) If the number of JE_i, 1 (one) ≤ *i* ≤ *NJVE*, that are not the SQL/JSON null is greater than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.

      II) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ *i* ≤ *NJVE*, that are not the SQL/JSON null.

   NOTE nnn — The elements of constructed JSON arrays are ordered and array element indices starts with 0 (zero).

d) Let *JVCF* be *CJA*.

4) If <JSON array constructor by query> *JACQ* is specified, then:

a) *QE* is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

b) Case:

   i) If *N* is 0 (zero), then let *CJAQ* be a JSON array with no elements.

   ii) Otherwise:

      1) For each *i*, 1 (one) ≤ *i* ≤ *N*:

         A) Let JVE_i be the *i*-th row of *T* and let $VJVE_i$ be the value of the column of JVE_i.

         B) Case:

            III) If $VJVE_i$ is a null value, then let JE_i be the SQL/JSON null.

            IV) If ***JACQ* contains a <JSON input clause> *JFO*,** then the General Rules of Subclause 9.x "Parsing a JSON text" are applied, with $VJVE_i$ as the *JSON TEXT*, *JFO* as the *FORMAT OPTION*, and an implementation-defined <JSON predicate uniqueness constraint> as the *UNIQUENESS CONSTRAINT*. Let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* that are returned by that Subclause.

               Case:

               1) If *ST* is an exception condition, then the exception condition *ST* is raised.

               2) Otherwise, let JE_i be *SJI*.

V) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JE_i be $VJVE_i$.

2) Otherwise, let JE_i be the result of

CAST ($VJVE_i$ AS *SDT*)

where *SDT* is an implementation-defined character string type with character set Unicode.

5) Case:

A) If the implicit or explicit <JSON array null clause> specifies NULL ON NULL, then

Case:

I) If *N* is greater than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.

II) Otherwise, let *CJAQ* be a JSON array whose elements are, in order, JE_i, 1 (one) $\leq i \leq N$.

B) Otherwise,

Case:

I) If the number of JE_i, 1 (one) $\leq i \leq N$, that are not the SQL/JSON null is greater than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.

II) Otherwise, let *CJAQ* be a JSON array whose elements are, in order, JE_i, 1 (one) $\leq i \leq N$, that are not the SQL/JSON null.

NOTE nnn — The elements of constructed JSON arrays are ordered and array element indices starts with 0 (zero).

c) Let *JVCF* be *CJAQ*.

5) The General Rules of Subclause 9.x "Serializing an SQL/JSON item" are applied with *JVCF* as the *SQL/JSON ITEM*, *JVCFFO* as the *FORMAT OPTION*, and *JVCFDT* as the *TARGET TYPE*. Let *ST* be the *STATUS* and let *CJV* be the *JSON TEXT* that are returned by that Subclause.

Case:

a) If *ST* is an exception condition, then the exception condition *ST* is raised.

b) Otherwise, the result of <JSON value constructor> is *CJV*.

Conformance Rules

1) Without Feature Tx11, conforming SQL language shall not contain <JSON value constructor>.

2) Without Feature Tx11, conforming SQL language shall not contain <JSON object constructor>.

3) Without Feature Tx11, conforming SQL language shall not contain <JSON array constructor>.

2.6 Changes to Clause 9, “Additional common rules”

2.6.1 INSERT a new Subclause

9.x Parsing JSON text

Subclause Signature

```
“Parsing a JSON text” [General Rules] (
  Parameter: "JSON TEXT",
  Parameter: "FORMAT OPTION",
  Parameter: "UNIQUENESS CONSTRAINT"
) Returns: "SQL/JSON ITEM", "STATUS"
```

Function

Convert a JSON text to an SQL/JSON item.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *JV* be the *JSON TEXT*, let *FO* be the *FORMAT OPTION*, and let *UC* be the *UNIQUENESS CONSTRAINT* in an application of this Subclause. The result of the application of this Subclause is *SJI*, returned as *SQL/JSON ITEM*, and *ST*, returned as *STATUS*.

2) Let *ST* be the condition: *successful completion*.

3) Case:

   a) If *FO* is JSON, then:

      i) Case:

         1) If *JV* is a character string, then let *ENC* be the Unicode encoding of *JV*.

         2) Otherwise, let *ENC* be the encoding determined by Section 3 “Encoding” in [JSON].

      ii) *JV* is parsed according to the grammar of Section 2 “JSON grammar” in [JSON], using the encoding *ENC*.

      iii) Case:

         1) If *JV* is not a JSON text, then let *ST* be the exception condition: *data exception — invalid JSON text*.

         2) If *UC* is WITH UNIQUE KEYS and *JV* contains a JSON object that has two JSON members whose keys are equivalent Unicode character strings, then let *ST* be the exception condition: *data exception — non-unique keys in a JSON object*.

3) Otherwise:

   A) The function *F* transforming a JSON text fragment *J* to an SQL/JSON item is defined recursively according to the grammar of Section 2 “JSON grammar” in [JSON], as follows:

      I) If *J* is the JSON literal `false`, then *F*(*J*) is the **truth value** *False*.

      II) If *J* is the JSON literal `true`, then *F*(*J*) is the **truth value** *True*.

      III) If *J* is the JSON literal `null`, then *F*(*J*) is the SQL/JSON null value.

      IV) If *J* is a JSON number, then *F*(*J*) is the value of the <signed numeric literal> whose characters are identical *J*.

      V) If *J* is a JSON string, then *F*(*J*) is an SQL character string whose character set is Unicode and whose characters are the ones enclosed by quotation marks in *J* after replacing any escape sequences by their unescaped equivalents.

      VI) If *J* is a JSON array, then *F*(*J*) is the SQL/JSON array whose elements are obtained by applying the transform *F* to each element of *J* in turn.

      VII) If *J* is a JSON member *K*:*V* , then *F*(*J*) is the SQL/JSON member whose key is *F*(*K*) and whose bound value is *F*(*V*).

      VIII) If *J* is a JSON object, then:

         1) Let M_1, . . ., M_m be the members of *J*, enumerated in an implementation-dependent order. For all *i*, 1 (one) $\leq i \leq m$, let K_i be the key and let V_i be the bound value of M_i. Let SJM_i be the SQL/JSON member whose key is $F(K_i)$ and whose bound value is $F(V_i)$.

         2) It is implementation-dependent whether members with redundant duplicate keys are removed. If the implementation-dependent choice is to delete members with redundant duplicate keys, then for all *i*, 1 (one) $\leq i \leq m$, and all *j*, 1 (one) $\leq j \leq m$, $i \neq j$, if $K_i = K_j$, an implementation-dependent choice of M_i or M_j is removed from the list of members of *J*.

         3) *F*(*J*) is the SQL/JSON object whose members are SJM_1, . . ., SJM_m.

   B) Let *SJI* be *F*(*JV*)

b) Otherwise, let *SJI* be the SQL/JSON **object or SQL/JSON array** obtained using implementation-defined rules for parsing *JV* according to format *FO* and uniqueness constraint *UC*. If there is an error during this conversion, let *ST* be an implementation-defined exception condition.

4) *SJI* is the *SQL/JSON ITEM* and *ST* is the *STATUS* that is the result of the application of this Subclause.

Conformance Rules

None.

2.6.2 INSERT a new Subclause

9.y Serializing an SQL/JSON item

Subclause Signature

```
"Serializing an SQL/JSON item" [Syntax Rules] (
  Parameter: "FORMAT OPTION",
  Parameter: "TARGET TYPE"
)
"Serializing an SQL/JSON item" [General Rules] (
  Parameter: "SQL/JSON ITEM",
  Parameter: "FORMAT OPTION",
  Parameter: "TARGET TYPE"
) Returns: "JSON TEXT", "STATUS"
```

Function

Serialize an SQL/JSON item as a JSON text.

Syntax Rules

1) Let *FO* be the *FORMAT OPTION* and let *TT* be the *TARGET TYPE* in an application of this Subclause.

2) Case:

   a) If *FO* **contains** JSON, then *TT* shall be either a character string type or a binary string type. If *TT* is a character string type, then the character set of *TT* shall be a Universal Character Set.

   b) Otherwise, *TT* shall be an implementation-defined data type appropriate to the format identified by *FO*.

Access Rules

None.

General Rules

1) Let *SJI* be the *SQL/JSON ITEM*, let *FO* be the *FORMAT OPTION* and let *TT* be the *TARGET TYPE* in an application of this Subclause. The result of this Subclause is a JSON text *JV* of type *TT* returned as *JSON TEXT*, and a completion condition *ST* returned as *STATUS*.

2) Case:

   a) If *FO* **contains** JSON then:

      i) Case:

         1) If *TT* is a character string type, then let *ENC* be the Unicode encoding of *TT*.

         2) If *TT* is a binary string type, then let *ENC* be UTF8, UTF16, or UTF32, as specified in the <JSON output representation> contained in *FO*.

      ii) Let *JV* be an implementation-dependent value of type *TT* and encoding *ENC* such that these two conditions hold:

         1) *JV* is a JSON text.

            NOTE nnn — It follows that it is an error if *SJI* is not an SQL/JSON array or SQL/JSON object.

         2) When applying the General Rules of Subclause 9.x, “Parsing a JSON text” with *JV* as the *JSON TEXT*, *FO* as the *FORMAT OPTION*, and WITHOUT UNIQUE KEYS as the *UNIQUENESS CONSTRAINT*, the returned *STATUS* is *successful completion* and the returned *SQL/JSON ITEM* is an SQL/JSON item that is equivalent to *SJI*.

      If there is no such *JV*, then let *ST* be the exception condition: *data exception — invalid JSON text*.

iii) If *JV* is longer than the length or maximum length of *TT*, then an exception condition is raised: *data exception — string data, right truncation*.

b) Otherwise, let *JV* be an implementation-defined value such that, when applying the General Rules of Subclause 9.x, "Parsing a JSON text" with *JV* as the *JSON TEXT*, *FO* as the *FORMAT OPTION*, and **an implementation-defined <JSON predicate uniqueness constraint>** as the *UNIQUENESS CONSTRAINT*, the returned *STATUS* is *successful completion* and the returned *SQL/JSON ITEM* is an SQL/JSON item that is equivalent to *SJI* according to an implementation-defined definition of this equivalence. If there is no such *JV*, then let *ST* be the exception condition: *data exception — invalid JSON text*.

2) *JV* is the *JSON TEXT* and *ST* is the *STATUS* that are the result of the application of this Subclause.

Conformance Rules

None.

2.7 Changes to Clause 10, "Additional common elements"

2.7.1 Changes to Subclause 10.9, "<aggregate function>"

2.7.1.1 In the Format, INSERT a new alternative in the production for <aggregate function>

```
  | <JSON aggregate function> [ <filter clause> ]
```

2.7.1.2 INSERT a new Conformace Rule

n) Without Feature Txx1, conforming SQL language shall not contain <JSON aggregate function>.

2.7.2 INSERT a new Subclause

10.x <JSON aggregate function>

Function

Generate a JSON object or a JSON array from an aggregation of SQL data.

Format

```
<JSON aggregate function> ::=
    <JSON object aggregate constructor>
  | <JSON array aggregate constructor>

<JSON object aggregate constructor> ::=
    JSON_OBJECTAGG <left paren>
    <JSON name> <comma> <JSON value expression>
    [ <JSON object aggregate null clause> ]
    [ <JSON output clause> ]
    <right paren>

<JSON object aggregate null clause> ::=
    NULL ON NULL
  | ABSENT ON NULL

<JSON array aggregate constructor> ::=
    JSON_ARRAYAGG <left paren>
    <JSON value expression>
    [ <JSON array aggregate order by clause> ]
    [ <JSON array aggregate null clause> ] ]
    [ <JSON output clause> ]
    <right paren>

<JSON array aggregate order by clause> ::=
    ORDER BY <sort specification list>

<JSON array aggregate null clause> ::=
    NULL ON NULL
  | ABSENT ON NULL
```

Syntax Rules

1) If <JSON output clause> is not specified, then RETURNING *ST* FORMAT JSON is implicit, where *ST* is an implementation-defined string type.

2) Let *JACFDT* be the <data type> immediately contained in the explicit or implicit <JSON output clause> *JOC* and let *JACFFO* be the explicit or implicit <JSON output representation> of *JOC*.

3) The Syntax Rules of Subclause 9.x, "Serializing an SQL/JSON item", are applied with *JACFFO* as the *FORMAT OPTION* and *JACFDT* as the *TARGET TYPE*.

4) If <JSON object aggregate constructor> is specified, then:

   a) If <JSON object aggregate null clause> is not specified, then ABSENT ON NULL is implicit.

   b) The declared type of <JSON object aggregate constructor> is *JACFDT*.

5) If <JSON array aggregate constructor> is specified, then:

   a) If <JSON array aggregate null clause> is not specified, then NULL ON NULL is implicit.

   b) The declared type of <JSON array aggregate constructor> is *JACFDT*.

Access Rules

None.

General Rules

1) Let *AF* be the <JSON aggregate function>.

2) Let *T* be the argument source of *AF*, as defined in the General Rules of Subclause 10.9, “<aggregate function>”.

3) Case:

   a) If <filter clause> is specified, then the <search condition> is effectively evaluated for each row of *T*. Let *T1* be the collection of rows of *T* for which the result of the <search condition> is <u>True</u>.

   b) Otherwise, let *T1* be *T*.

4) If <JSON object aggregate constructor> is specified, then:

   a) Let *TN* be the cardinality of *T1*.

   b) Case:

      i) If *TN* is 0 (zero), then the result of the <JSON object aggregate constructor> is the null value and no further General Rules of this Subclause are evaluated.

      ii) Otherwise:

         1) Let *JVE* be the <JSON value expression>.

         2) For each i, 1 (one) $\leq i \leq TN$, let R_i be the i-th row of *T1*:

            A) Let VJN_i be the value of <JSON name> **evaluated** for R_i.

            B) If VJN_i is the null value, then an exception condition is raised: *data exception — null value not allowed*.

            C) Let $VJVE_i$ be the value of *JVE* evaluated for R_i.

            D) Case:

               I) If $VJVE_i$ is a null value, then let JBV_i be the SQL/JSON null.

               II) If ***JVE*** specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.x "Parsing a JSON text" are applied, with JVE_i as the *JSON TEXT*, *JFO* as the *FORMAT OPTION*, and an implementation-defined <JSON predicate uniqueness constraint> as the *UNIQUENESS CONSTRAINT*. Let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* that are returned by that Subclause.

                  Case:

                  1) If *ST* is an exception condition, then the exception condition *ST* is raised.

                  2) Otherwise, let JBV_i be *SJI*.

               III) Otherwise,

                  Case:

                  1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JBV_i be $VJVE_i$.

                  2) Otherwise, let JBV_i be the result of

                     CAST ($VJVE_i$ AS *SDT*)

                     where *SDT* is an implementation-defined character string type with character set Unicode.

            E) Let M_i be the SQL/JSON member whose key is VJN_i and whose bound value is JBV_i.

         3) If, for any i, 1 (one) $\leq i \leq TN$, and any j, $i < j \leq TN$, $VJN_i = VJN_j$, then an exception condition is raised: *data exception — duplicate JSON object key value*.

         4) Case:

A) If the implicit or explicit <JSON object null clause> specifies NULL ON NULL, then

Case:

I) If *TN* is greater than the implementation-defined maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) ≤ *i* ≤ *TN*.

B) Otherwise,

Case:

I) If the number of M_i, 1 (one) ≤ *i* ≤ *TN*, whose bound values are not the SQL/JSON null is greater than the implementation-defined maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) ≤ *i* ≤ *TN*, whose bound values are not the SQL/JSON null.

NOTE nnn — There is no implied order of the members of the constructed JSON object.

c) Let *JACF* be *CJOA*.

5) If <JSON array aggregate constructor> is specified, then:

a) If <sort specification list> is specified, then let *K* be the number of <sort key>s; otherwise, let *K* be 0 (zero).

b) Let *TXA* be the table of *K*+1 columns obtained by applying the <value expression> immediately contained in the <JSON value expression> ***JVE*** simply contained in the <JSON array aggregate constructor> to each row of *T1* to obtain the first column of *TXA*, and, for all *i*, 1 (one) ≤ *i* ≤ *K*, applying the <value expression> simply contained in the *i*-th <sort key> to each row of *T1* to obtain the (*i*+1)-th column of *TXA*.

c) Let *TXA* be ordered according to the values of the <sort key>s found in the second through (*K*+1)-th columns of *TXA*. If K is 0 (zero), then the ordering of *TXA* is implementation-dependent.

d) Let *NTXA* be the number of rows in *TXA*.

e) Let R_i, 1 (one) ≤ *i* ≤ *NTXA*, be the rows of *TXA* according to the ordering of *TXA*.

f) Case:

i) If *NTXA* is 0 (zero), then the result of the <JSON array aggregate constructor> is the null value and no further General Rules of this Subclause are evaluated.

ii) Otherwise:

1) For each *i*, 1 (one) ≤ *i* ≤ *NTXA*:

A) Let $VJVE_i$ be the value of the first column of $\boldsymbol{R_i}$.

B) Case:

III) If $VJVE_i$ is a null value, then let JE_i be the SQL/JSON null.

IV) If ***JVE*** specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.x "Parsing a JSON text" are applied, with $VJVE_i$ as the *JSON TEXT*, *JFO* as the *FORMAT OPTION*, and an implementation-defined <JSON predicate uniqueness constraint> as the *UNIQUENESS CONSTRAINT*. Let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* that are returned by that Subclause.

Case:

1) If *ST* is an exception condition, then the exception condition *ST* is raised.
2) Otherwise, let JE_i be *SJI*.

V) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JE_i be $VJVE_i$.
2) Otherwise, let JE_i be the result of

   CAST ($VJVE_i$ AS *SDT*)

   where *SDT* is an implementation-defined character string type with character set Unicode.

d) Case:

i) If the <JSON array null clause> specifies NULL ON NULL, then

   Case:

   1) If *NJVE* is greater than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.
   2) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ *i* ≤ *NTXA*.

ii) Otherwise,

   Case:

   1) If the number of JE_i, 1 (one) ≤ *i* ≤ *NJVE*, that are not the SQL/JSON null is greater than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.
   2) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ *i* ≤ *NTXA*, that are not the SQL/JSON null.

NOTE nnn — The elements of constructed JSON arrays are ordered and array element indices starts with 0 (zero).

c) Let *JACF* be *CJAA*.

6) The General Rules of Subclause 9.x "Serializing an SQL/JSON item" are applied with *JACF* as the *SQL/JSON ITEM*, *JACFFO* as the *FORMAT OPTION*, and *JACFDT* as the *TARGET TYPE*. Let *ST* be the *STATUS* and let *CJV* be the *JSON TEXT* that are returned by that Subclause.

Case:

a) If *ST* is an exception condition, then the exception condition *ST* is raised.

b) Otherwise, *CJV* is the result of <JSON aggregate function>.

Conformance Rules

1) Without Feature Tx11, conforming SQL language shall not contain <JSON array aggregate constructor>.

2) Without Feature Tx12, conforming SQL language shall not contain <JSON object aggregate constructor>.

3) Without Feature Tx13, conforming SQL language shall not contain <JSON array aggregate constructor> that specifies a <JSON array aggregate order by clause>.

2.7.3 INSERT a new Subclause

10.x <JSON value expression>

Function

Specify a value to be used as input by an SQL/JSON function.

Format

```
<JSON value expression> ::=
    <value expression> [ <JSON input clause> ]

<JSON input clause> ::=
    FORMAT <JSON input representation>

<JSON input representation> ::=
    JSON
  | <implementation-defined JSON representation option>
```

Syntax Rules

1) FORMAT JSON specifies the data format specified in [**RFC4627**].

2) FORMAT <implementation-defined JSON **representation option**> specifies an implementation-defined data format.

   NOTE nnn: For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON **representation option**> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of Subclause 9.x, “Parsing a JSON text”, and Subclause 9.y, “Serializing an SQL/JSON item”, respectively.

3) The declared type *DT* of the <value expression> *VE* simply contained in <JSON value expression> *JVE* either shall be a character string type, a binary string type, a numeric type, a datetime type, or Boolean, or shall be a <data type> the values of which can be cast to a character string type according to the Syntax Rules of Subclause 6.13, “<cast specification>”.

4) If *VE* is a JSON-returning function *JRF*, and <JSON input clause> is not specified, then

   Case:

   a) If the explicit or implicit <JSON output clause> simply contained in *JRF* contains FORMAT JSON, then the implicit <JSON input clause> of *JVE* is FORMAT JSON.

   b) Otherwise, the implicit <JSON input clause> of *JVE* is FORMAT <implementation-defined JSON representation option>.

5) If an explicit or implicit <JSON input clause> is specified, then *DT* shall be a string type.

6) If *DT* is a binary string type, then an explicit or implicit <JSON input clause> shall be specified.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

2.7.4 INSERT a new Subclause

10.x <JSON output clause>

Function

Specify the data type, format, and encoding of the JSON text created by a JSON-returning function.

Format

```
<JSON output clause> ::=
    RETURNING <data type> [ FORMAT <JSON output representation> ]

<JSON output representation> ::=
    JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
  | <implementation-defined JSON representation option>
```

Syntax Rules

1) If FORMAT is not specified, then FORMAT JSON is implicit.

2) If FORMAT JSON is specified or implicit, then the <data type> *DT* shall identify a string type *ST*.

   Case:

   a) If *DT* identifies a character string type, then *DT* shall have a Universal Character Set, ENCODING shall not be specified, and an implicit choice of UTF8, UTF16, or UTF32 is determined by the character encoding form of *DT* (*i.e.*, the keywords UTF8, UTF16, and UTF32 denote the UTF8, UTF16, and UTF32 character encoding forms, respectively).

   b) If *DT* is a binary string type and ENCODING is not specified, then it is implementation-defined whether UTF8, UTF16, or UTF32 is specified.

3) FORMAT JSON specifies the data format specified in [**RFC4627**].

4) FORMAT <implementation-defined JSON format> specifies an implementation-defined data format.

   NOTE nnn: For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON format> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of Subclause 9.x, “Parsing a JSON text”, and Subclause 9.y, “Serializing an SQL/JSON item”, respectively.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

2.8 Changes to Clause 13, “SQL-client modules”

2.8.1 Changes to Subclause 13.3, “<externally-invoked procedure>”

2.8.1.1 **ADD** the following lines to the Ada package SQLSTATE_CODES defined in Syntax Rule 10)e):

```
DATA_EXCEPTION_DUPLICATE_JSON_OBJECT_KEY_VALUE:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_SQL_JSON_DATETIME_FUNCTION:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_INVALID_JSON_TEXT:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_INVALID_SQL_JSON_SUBSCRIPT:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_MORE_THAN_ONE_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_NON-NUMERIC_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_NON-UNIQUE_KEYS_IN_A_JSON_OBJECT:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_SINGLETON_SQL_JSON_ITEM_REQUIRED:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_SQL_JSON_ARRAY_NOT_FOUND:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_SQL_JSON_MEMBER_NOT_FOUND:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_NO_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_SQL_JSON_NUMBER_NOT_FOUND:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_SQL_JSON_OBJECT_NOT_FOUND:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_TOO_MANY_JSON_ARRAY_ELEMENTS:
  constant SQLSTATE_TYPE := "22---";
DATA_EXCEPTION_TOO_MANY_JSON_OBJECT_MEMBERS:
  constant SQLSTATE_TYPE := "22---";
```

2.9 Changes to Clause 24, “Status codes”

2.9.1 Changes to Subclause 24.1, “SQLSTATE”

2.9.1.1 **ADD** the following entries to Table 33, “SQLSTATE class and subclass values”

Category	Condition	Class	Subcondition	Subclass
X	*data exception*	22	*duplicate JSON object key value*	
			invalid argument for SQL/JSON datetime function	
			invalid JSON text	
			invalid SQL/JSON subscript	
			more than one SQL/JSON item	
			no SQL/JSON item	
			non-numeric SQL/JSON item	
			non-unique keys in a JSON object	
			singleton SQL/JSON item required	
			SQL/JSON array not found	
			SQL/JSON member not found	
			SQL/JSON number not found	
			SQL/JSON object not found	
			too many JSON array elements	
			too many JSON object members	

2.10 Changes to Clause 25, “Conformance”

TO BE SUPPLIED IF NEEDED

2.11 Changes to Annex A, “SQL conformance summary”

THE CONTENTS OF THIS ANNEX ARE PRODUCED AUTOMATICALLY

2.12 Changes to Annex B, “Implementation-defined elements”

2.12.1 INSERT the following list elements

n) **Subclause 4.x.3 SQL/JSON data model**

a) The maximum number of SQL/JSON members in an SQL/JSON object is implementation-defined.

b) The maximum length of the key in an SQL/JSON member is implementation-defined.

c) If the declared type of an SQL/JSON element or of the key of an SQL/JSON member is a string type, the maximum length is implementation-defined.

d) If the declared type of an SQL/JSON element or of the key of an SQL/JSON member is a string type, the maximum length is implementation-defined.

n) **Subclause 6.x <JSON value constructor>**

a) If <JSON output clause> is not specified, then an implementation-defined string type is implicit.

b) If the length of the value of any <JSON name> exceeds its implementation-defined maximum length, then an exception condition is raised: *data exception — string data, right truncation*.

c) If any <JSON value expression> contains a <string value expression> *SVE* and the length of the value of *SVE* exceeds its implementation-defined maximum length, then an exception condition is raised: *data exception — string data, right truncation*.

d) An implementation-defined <JSON predicate uniqueness constraint> is used when serializing JSON text.

e) If the explicit or implicit data type specified for the output of JSON text is not a character string type, a numeric type, or a Boolean type, then the generated JSON text is cast to an implementation-defined string type.

f) If a JSON object is constructed that has more members than the implementation-defined maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members*.

g) If a JSON array is constructed that has more elements than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.

n) 9.x Parsing JSON text

h) If JSON text is contained in a string the format of which is an implementation-defined format, the rules for parsing that JSON text are implementation-defined.

i) If an error occurs during parsing JSON text contained in a string the format of which is an implementation-defined format, an implementation-defined exception condition is raised.

n) 9.x Parsing JSON text

j) When serializing JSON items to JSON text, if the target data type is not a string type, the target type shall be an implementation-defined data type appropriate to the format identified by the specified or implicit format.

k) The actual value of the target type of serialization that is JSON text is implementation-dependent, but must be parseable as JSON text.

10.x <JSON aggregate function>

l) If <JSON output clause> is not specified, then an implementation-defined string type is implicit.

m) An implementation-defined <JSON predicate uniqueness constraint> is used when serializing JSON text.

n) If the explicit or implicit data type specified for the output of JSON text is not a character string type, a numeric type, or a Boolean type, then the generated JSON text is cast to an implementation-defined string type.

o) If a JSON object is constructed that has more members than the implementation-defined maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members*.

p) If a JSON array is constructed that has more elements than the implementation-defined maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements*.

10.x <JSON value expression>

a) FORMAT <implementation-defined JSON format> specifies an implementation-defined data format.

10.x <JSON output clause>

a) FORMAT <implementation-defined JSON format> specifies an implementation-defined data format.

b) If the target data type is a binary string type and ENCODING is not specified, then it is implementation-defined whether UTF8, UTF16, or UTF32 is specified.

2.13 Changes to Annex C, "Implementation-dependent elements"

n) **9.x Parsing JSON text**

a) Whether JSON object members with redundant duplicate keys are included in an object is implementation-dependent. If the implementation-dependent choice is to omit members with redundant duplicate keys, then when a JSON object is constructed in which two or more members have the same key value, an implementation-dependent member is selected for retention and all other such members are removed.

n) 9.x Serializing an SQL/JSON item

a) The result of serializing an SQL/JSON item is implementation-dependent.

10.x <JSON aggregate function>

a) The elements of a JSON array constructed without a <JSON array aggregate order by clause> are in an implementation-dependent order.

Changes to Annex E, “Incompatibilities with ISO/IEC 9075-2:2011”

2.13.1 In list element 1), INSERT the following <reserved word>s in proper alphabetic order:

— JSON_ARRAY

— JSON_ARRAYAGG

— JSON_OBJECT

— JSON_OBJECTAGG

2.14 Changes to Annex F, “SQL feature taxonomy”

Feature ID	Feature Name
Tx11	Basic SQL/JSON constructor functions
Tx12	SQL/JSON: JSON_OBJECTAGG
Tx13	SQL/JSON: JSON_ARRAYAGG with ORDER BY

2.15 Changes to “Bibliography”

2.15.1 INSERT the following bibliography references in the appropriate places

[Avro] http://avro.apache.org/

[BSON] http://bsonspec.org/

[JDIF] The JSON Data Interchange Format, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[JSONintro] Introducing JSON; http://www.json.org/

3. Comments Resolved

The present proposal, if accepted, would partially resolve CD ballot comment sequence P02-USA-950.

4. Coverage

The author of the present proposal believes that it covers certain required areas of concern as reflected in the following table:

For All Proposals		
1	Concepts	Y

2	Access Rules	N/A
3	Conformance Rules, including the relevant Annexes	Y
4	Lists of SQL-statements by category	N/A
5	Table of identifiers used by diagnostics statements	N/A
6	Collation coercibility determination for changes related to character strings	N/A
7	Closing Possible Problems when a proposal resolves them	N/A
8	Any new Possible Problems clearly identified	N/A
9	Reserved and non-reserved keywords	Y
10	SQLSTATE tables and Ada package	Y
11	Information and Definition Schemas	N/A
12	Implementation-defined and –dependent Annexes	Y
13	Incompatibilities Annex	N/A
14	Embedded SQL bindings and host language implications	N/A
15	Dynamic SQL issues: including Dynamic descriptor areas	N/A
16	CLI issues	N/A

End of paper