



BSI Standards Publication

Information technology — Database languages — SQL Technical Reports

Part 6: SQL support for JavaScript Object
Notation (JSON)

National foreword

This Published Document is the UK implementation of ISO/IEC TR 19075-6:2017.

The UK participation in its preparation was entrusted to Technical Committee IST/40, Data management and interchange.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2017.

Published by BSI Standards Limited 2017

ISBN 978 0 580 93531 2

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 30 April 2017.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

**Information technology — Database
languages — SQL Technical Reports —**
Part 6:
**SQL support for JavaScript Object
Notation (JSON)**

*Technologies de l'information — Langages de base de données — SQL
rapport techniques —*

Partie 6: Support de SQL pour JavaScript Object Notation (JSON)



Contents

Page

Foreword.....	ix
Introduction.....	x
1 Scope.....	1
2 Normative references.....	3
2.1 ISO and IEC standards	3
2.2 Other international standards	3
3 JavaScript Object Notation (JSON).....	5
3.1 What is JSON?	5
3.2 Representations of JSON data	6
3.2.1 Avro.....	6
3.2.2 BSON.....	7
3.3 Schemas.....	7
3.3.1 JSON schemata and validity.....	7
3.3.2 Avro schemata.....	8
3.3.3 BSON schemata	8
3.4 Why does JSON matter in the context of SQL? What is JSON's relationship to NoSQL?.....	9
3.5 JSON terminology	10
3.6 Use cases for JSON support in SQL.....	11
3.6.1 JSON data ingestion and storage	11
3.6.2 JSON data generation from relational data.....	12
3.6.3 Querying JSON as persistent semi-structured data model instances	12
3.7 What features address those use cases?	12
3.7.1 Storing JSON data in an SQL table	12
3.7.2 Generating JSON in an SQL query.....	13
3.7.3 Querying JSON data in SQL tables using SQL	13
4 The SQL/JSON data model	15
4.1 SQL/JSON items.....	16
4.1.1 Atomic values.....	17
4.1.2 SQL/JSON arrays.....	18
4.1.3 SQL/JSON objects	18
4.2 SQL/JSON sequences.....	19
4.3 Parsing JSON	19
4.4 Serializing JSON.....	19
5 SQL/JSON functions	21
5.1 Handle JSON using built-in functions.....	21

5.2	JSON API common syntax	21
5.2.1	JSON value expression	22
5.2.2	Path expression	22
5.2.3	PASSING clause	22
5.2.4	JSON output clause	23
5.2.5	ON ERROR and ON EMPTY syntax	23
5.3	Query functions	24
5.3.1	JSON_EXISTS	24
5.3.2	JSON_VALUE	27
5.3.3	JSON_QUERY	32
5.3.4	JSON_TABLE	35
5.3.4.1	COLUMNS clause that is not nested	36
5.3.4.2	Nested COLUMNS clause	39
5.3.4.3	PLAN clause	40
5.3.5	Conformance features for query operators	44
5.4	Constructor functions and IS JSON predicate	46
5.4.1	JSON_OBJECT	47
5.4.2	JSON_OBJECTAGG	48
5.4.3	JSON_ARRAY	50
5.4.4	JSON_ARRAYAGG	50
5.4.5	IS JSON predicate	52
5.4.6	Handling of JSON nulls and SQL nulls	53
5.4.7	Conformance features for constructor functions	53
6	SQL/JSON path language	55
6.1	Overview of SQL/JSON path language	55
6.2	Objectives for the SQL/JSON path language	57
6.3	Modes	58
6.3.1	Example of strict vs lax	59
6.4	Lexical issues	61
6.5	Syntax summary	62
6.6	Formal semantics	63
6.6.1	Notational conventions	63
6.7	Primitive operations	64
6.7.1	Concatenation	64
6.7.2	unwrap()	64
6.7.3	wrap()	65
6.8	Mode declaration	65
6.9	<JSON path primary>	66
6.9.1	Literals	66
6.9.2	Variables	67
6.9.3	Parentheses	68
6.10	Accessors	68
6.10.1	Member accessor	69

6.10.2	Member wildcard accessor	72
6.10.3	Element accessor	73
6.10.4	Element wildcard accessor	75
6.10.5	Sequence semantics of the accessors	76
6.11	Item methods	77
6.11.1	type()	77
6.11.2	size()	78
6.11.3	Numeric item methods (double, ceiling, floor, abs)	78
6.11.4	datetime()	78
6.11.5	keyvalue()	79
6.12	Arithmetic expressions	80
6.12.1	Unary plus and minus	81
6.12.2	Binary operations	82
6.13	Filter expressions	82
6.13.1	true/false and <u>True/False</u>	83
6.13.2	null and <u>Unknown</u>	83
6.13.3	Error handling in filters	83
6.13.4	Truth tables	86
6.13.5	Comparison predicates	87
6.13.6	like_regex predicate	89
6.13.7	starts with predicate	90
6.13.8	exists predicate	90
6.13.9	is unknown predicate	92
6.14	Conformance features for SQL/JSON path language	93
Bibliography.....		95
Index.....		97

Tables

Table	Page
1 Terms and definitions.	10
2 JSON, SQL/JSON, and SQL (other than SQL/JSON values and counterparts).	15
3 Parallels between JSON text and SQL/JSON data model.	19
4 JSON_EXISTS sample data.	24
5 Result of the sample query.	25
6 Accessor example.	26
7 Result 1	28
8 Result 2	29
9 Result 3	29
10 Result 4	30
11 Result 5	31
12 Result 6	31
13 Comparison of wrapper options.	35
14 JSON_TABLE sample data in a book recommendation table.	36
15 Query result	38
16 Query result	39
17 Query result	42
18 Query result	42
19 Query result	44
20 DEPTS table.	46
21 JOBS table	47
22 EMPLOYEES table	47
23 The JSON object returned	48
24 Returned JSON object with the corresponding job sequence number.	49
25 Query result	51
26 Query Results.	51
27 Query Results.	52
28 Three aspects of path evaluation governed by modes.	58
29 Example of strict vs lax.	59
30 Features of the SQL/JSON path language	62
31 Data used by unwrap() example.	64
32 Data used by wrap() example.	65
33 Examples of atomic values in the SQL/JSON path language	66
34 Examples of the escaping rules	66
35 Evaluation of '\$.phones.type' in lax mode.	70
36 Intermediate step	70
37 Evaluation of '\$.phones[*].type'	71
38 Evaluation of '\$.phones[*] ? (exists (@.type)).type'	71
39 Evaluation of '\$.phones.*' in lax mode	72
40 Evaluation of '\$.phones[*].*'	73
41 Evaluation of 'lax \$.sensors.*[0, last, 2]'.	75
42 The step in the evaluation	76
43 Result of the query with the sample data	80

44	Evaluation of <code>lax -\$.readings.floor()</code>	81
45	Evaluation of <code>'lax (-\$.readings).floor()'</code>	81
46	Table T with two rows	84
47	Computation on row K=102	84
48	Modified table T	85
49	Computation on row K=102 in modified table T.....	85
50	Computation of <code>'lax \$? (@.hours > 9)'</code> on row K=102.....	86
51	Computation of <code>'strict \$? (@.hours > 9)'</code> in strict mode	86
52	Result of <code>&&</code>	87
53	Result of <code> </code>	87
54	Result of <code>!</code>	87
55	Supported comparisons	88
56	Final result	89
57	A table with JSON column	91
58	Evaluation of <code>'strict \$? (exists (@.name)).name'</code> on row K=201.....	91
59	Evaluation of <code>'strict \$? (exists (@.name)).name'</code> on row K=202.....	91

Figures

Figure	Page
1 Relationships between “JSON” and “SQL/JSON”	15
2 The SQL/JSON path language architecture	55

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 32, *Data management and interchange*.

A list of all parts in the ISO/IEC 19075 series can be found on the ISO website.

NOTE 1 — The individual parts of multi-part technical reports are not necessarily published together. New editions of one or more parts can be published without publication of new editions of other parts.

Introduction

The organization of this part of ISO/IEC 19075 is as follows:

- 1) **Clause 1, “Scope”**, specifies the scope of this part of ISO/IEC 19075.
- 2) **Clause 2, “Normative references”**, identifies additional standards that, through reference in this part of ISO/IEC 19075, constitute provisions of this part of ISO/IEC 19075.
- 3) **Clause 3, “JavaScript Object Notation (JSON)”**, introduces what is JSON.
- 4) **Clause 4, “The SQL/JSON data model”**, introduces the data model that is used by the SQL/JSON functions and the SQL/JSON path language.
- 5) **Clause 5, “SQL/JSON functions”**, introduces the SQL/JSON functions to query and construct JSON.
- 6) **Clause 6, “SQL/JSON path language”**, introduces the SQL/JSON path language.

Information technology — Database languages — SQL Technical Reports —**Part 6:****SQL support for JavaScript Object Notation (JSON)****1 Scope**

This Technical Report describes the support in SQL for JavaScript Object Notation.

This Technical Report discusses the following features of the SQL language:

- Storing JSON data.
- Publishing JSON data.
- Querying JSON data.
- SQL/JSON data model and path language.

(Blank page)

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 ISO and IEC standards

[ISO9075-2] ISO/IEC 9075-2:2016, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

2.2 Other international standards

[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>

[Unicode] The Unicode Standard,

<http://unicode.org>

[RFC7159] Internet Engineering Task Force, RFC 7159, *The JavaScript Object Notation (JSON) Data Interchange Format*, March 2014,

<https://tools.ietf.org/rfc/rfc7159.txt>

(Blank page)

3 JavaScript Object Notation (JSON)

3.1 What is JSON?

JSON (an acronym for “JavaScript Object Notation”) is both a notation (that is, a syntax) for representing data and an implied data model. JSON is not an object-oriented data model; 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. [RFC7159] says that JSON is a text format for the serialization of structured data. Its initial intended use was as a data transfer syntax.

The complete syntax of JSON is specified in [RFC7159].

The first-class components of the JSON data model are JSON values. A JSON value is one of the following: JSON object, JSON array, JSON string, JSON number, or one of the JSON literals: true, false, and null. 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.

Here is an example of a JSON array:

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

In a JSON array the values are separated by commas. JSON arrays and objects are fully nestable. That is, 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 can be 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

3.1 What is JSON?

initially created or designed. Similarly, any JSON array can be modified by changing the number of values in the array.

3.2 Representations of JSON data

Before delving much deeper into the primary topic of this Technical Report, 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).

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 this Technical Report are frequently used and may be representative of binary JSON representations generally. The following discussion is intended only to illustrate the use of and issues with binary serializations of JSON. The SQL standard leaves it implementation-defined whether or not such representations are supported in any particular implementation.

3.2.1 Avro

Avro [Avro] 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 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 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.

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 data.) In this Technical Report, Avro is referenced as one possible *serialization* of JSON data; the character string format is another serialization of the same data.

3.2.2 BSON

BSON [BSON] (variously pronounced as though it were spelled “bison” or 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 — a specific commercial product.

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.

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.)

3.3 Schemas

3.3.1 JSON schemata and validity

Neither [RFC7159] nor [ECMAScript] provide any mechanism by which JSON values can be tested for validity other than strict syntactic validity.

JSON text is 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.

There is 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, there does not appear to be any significant amount of interest from the JSON community for the rapid development of a schema definition language for JSON.

While a schema language for JSON data could be useful in some circumstances, it does not appear that such a language is widely used, and [ISO9075-2] neither relies on nor creates such a language.

[ISO9075-2] uses the word “valid” to describe data instances that satisfy all JSON syntactic requirements. [ISO9075-2] specifies a new SQL predicate, IS JSON, to test the (syntactic) validity of JSON data instances.

3.3.2 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 this Technical Report, 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. That phrase means 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. It also means that the lengths (number of octets) of each member/element must be respectively the same, and that’s not always easy to ensure.

NOTE 2 — Therefore, the JSON objects “{ “name” : “Joe”, “salary” : 85000 }” and “{ “name” : “Bob”, “salary” : 78000 }” are “sufficiently alike”, but “{ “name” : “Ann”, “bonus” : 85000 }” is not.

3.3.3 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.

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

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

It is unclear that JSON and SQL [ISO9075-2] have any inherent relationship, but it is equally clear that the technical, business, and government worlds are increasingly using both kinds of data in their environments. Individual applications are required to access and manipulate all of these kinds of data, often concurrently. 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.

Incorporation of JSON into the SQL umbrella offers 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 described in this Technical Report is used.

NoSQL database systems [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, based 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 may 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.

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

The rapidly increasing use of JSON to interchange data between Web applications has caught the attention of academics, technologists, developers, and even enterprise management. SQL database implementers are increasingly convinced that they must support JSON data “natively”. This Technical Report describes the approach taken in [\[ISO9075-2\]](#), which allows such implementers to provide that support in a common manner

3.5 JSON terminology

JSON is taken from JavaScript, which has been standardized under the name “ECMAScript” [\[ECMAScript\]](#). [\[ECMAScript\]](#) defines the terminology for its objects, but [\[RFC7159\]](#) uses terminology that is significantly different. Other specifications associated with JSON use still different terminology.

[\[ISO9075-2\]](#) and this Technical Report stick as closely as possible to the notation in [\[RFC7159\]](#). The following terms and their definitions are used:

Table 1 — Terms and definitions

Term	Definition
JSON text	A sequence of 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 that JSON text is a single object or array
Token	One of six structural characters (“{”, “}”, “[”, “]”, “:”, “,”), strings, numbers, and literals
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 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”

Term	Definition
Null	The literal “null”
Object	A structure represented by a “{”, zero or more members 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 elements 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) data model associated with JSON
SQL/JSON data model	The data model created for operating on JSON data within the SQL language

3.6 Use cases for JSON support in SQL

There are three primary use cases:

- 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.

3.6.1 JSON data ingestion and storage

Defining a new native data type is both costly to implement for SQL implementations and costly to adopt by database tools and applications alike. Thus, [ISO9075-2] took the approach to ingest JSON data as character strings or binary strings that are then stored in ordinary SQL columns of some existing 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.

3.6.2 JSON data generation from relational data

This use case asks “How can JSON data instances be (declaratively) generated 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.)

[ISO9075-2] 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. This Technical Report provides several examples for those functions.

3.6.3 Querying JSON as persistent semi-structured data model instances

This use case explores how JSON data that is stored directly in SQL tables can be queried. Direct mapping of entire SQL tables into single (or, necessarily, multiple) JSON objects or arrays has not been specified in [ISO9075-2], although support is provided for such mappings when needed by applications. Instead, JSON data is stored within an opaque data type (specifically an SQL string or Large Object) that can be manipulated through the functional interface specified in [ISO9075-2], as illustrated by examples in this Technical Report.

3.7 What features address those use cases?

The use cases are addressed as follows:

- SQL query expressions can access 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 can generate JSON data directly for return to invokers of those queries.
- SQL tables can store JSON data persistently.

In the next sections, each of these “macro-features” are explored in turn.

3.7.1 Storing JSON data in an SQL table

The approach taken by [ISO9075-2] and described in this Technical Report 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, the standardization (and implementation) overhead of creating a new built-in SQL data type is avoided 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. [ISO9075-2] provides a number of built-in SQL functions

that access (query) JSON data stored in such columns. These functions are described in [Clause 5, “SQL/JSON functions”](#).

3.7.2 Generating JSON in an SQL query

[\[ISO9075-2\]](#) provides built-in functions that generate JSON objects and JSON arrays as the results of SQL queries, whether the source of the data queried is JSON data or ordinary SQL data. These functions are also described in [Clause 5, “SQL/JSON functions”](#).

3.7.3 Querying JSON data in SQL tables using SQL

Since there is no well-known, standardized, and universally accepted language that queries data represented in the JSON data model, [\[ISO9075-2\]](#) defines the SQL/JSON path language, which is embedded in SQL operators that address the use cases outlined in [Subclause 3.6, “Use cases for JSON support in SQL”](#).

(Blank page)

4 The SQL/JSON data model

The SQL/JSON data model used by SQL/JSON path language can be summarized as “sequences of items”. The items are SQL scalar values with an additional SQL/JSON null value, and composite data structures using JSON arrays and objects.

To clearly distinguish between JSON values “outside” the SQL-environment and their analogs “inside” the SQL-environment, the following conventions are adopted:

- The modifier “JSON” refers to constructs within a character or binary string that conforms to [RFC7159](#).
- The modifier “SQL/JSON” refers to JSON constructs within the SQL-environment.

The relationship between “JSON” outside of and “SQL/JSON” within the SQL-environment is crudely illustrated in Figure 1, “Relationships between “JSON” and “SQL/JSON””.

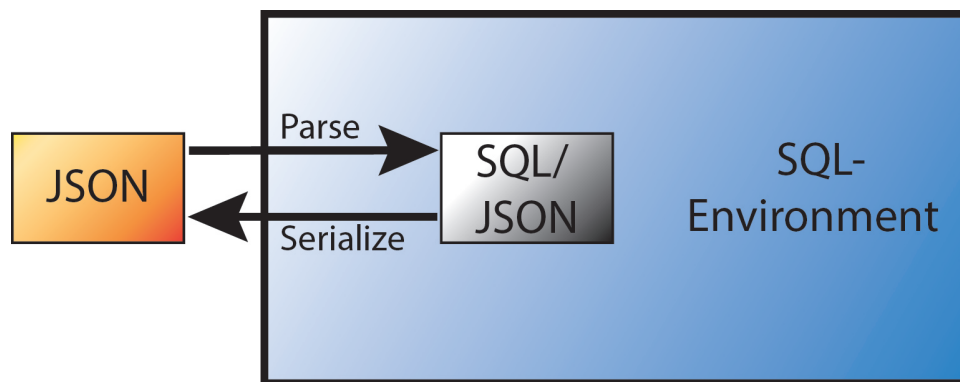


Figure 1 — Relationships between “JSON” and “SQL/JSON”

JSON data is parsed into SQL/JSON values, which can then be serialized back into JSON. All JSON data can be parsed into SQL/JSON values; however, not all SQL/JSON values can be serialized into JSON data. Parsing and serializing are addressed in more detail in Subclause 4.3, “Parsing JSON”, and Subclause 4.4, “Serializing JSON”, respectively.

Table 2, “JSON, SQL/JSON, and SQL (other than SQL/JSON values and counterparts)”, lists various terms used in [\[ISO9075-2\]](#) and this Technical Report related to JSON, SQL/JSON, and SQL (other than SQL/JSON values and their corresponding counterparts).

Table 2 — JSON, SQL/JSON, and SQL (other than SQL/JSON values and counterparts)

JSON	SQL/JSON	SQL (other than SQL/JSON)
JSON array	SQL/JSON array	
JSON object	SQL/JSON object	

4.1 SQL/JSON items

JSON	SQL/JSON	SQL (other than SQL/JSON)
JSON member	SQL/JSON member	
JSON literal null	SQL/JSON null	typed nulls
JSON literal true	<u>True</u>	<u>True</u>
JSON literal false	<u>False</u>	<u>False</u>
JSON number	number	non-null number
JSON string	character string	non-null character string
	datetime	non-null datetime
	SQL/JSON item	
	SQL/JSON sequence	

NOTE 3 — SQL/JSON and SQL other than SQL/JSON have precisely the same value spaces in the case of non-null scalars of boolean, numeric, string, or datetime types.

4.1 SQL/JSON items

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

- 1) 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.
- 2) An SQL/JSON null, defined as a value that is distinct from any value of any SQL type.

NOTE 4 — An SQL/JSON null is distinct from the null value of any SQL type.

- 3) 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.
- 4) 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 5 — [RFC7159], 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.

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

Two SQL/JSON items *SJ11* and *SJ12* are said to be equivalent, defined recursively as follows:

- 1) If *SJ11* and *SJ12* are non-null values of a predefined type, then *SJ11* and *SJ12* are equivalent if they are equal.

- 2) If *SJI1* and *SJI2* are the SQL/JSON null, then *SJI1* and *SJI2* are equivalent.
- 3) If *SJI1* and *SJI2* are both SQL/JSON arrays of the same length *N* and corresponding elements of *SJI1* and *SJI2* are equivalent, then *SJI1* and *SJI2* are equivalent.
- 4) If *SJI1* and *SJI2* are SQL/JSON objects with 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* then *SJI1* and *SJI2* are equivalent.

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

NOTE 6 — There is no SQL <data type> whose value space is SQL/JSON items or SQL/JSON sequences.

SQL/JSON items are typed values in the following categories:

- 1) Atomic values:

Atomic values are the non-null SQL values of the following types:

- a) Strings in a Unicode character set.
 - b) Numeric values:
 - i) Exact numeric values.
 - ii) Approximate numeric values.
 - c) Boolean values.
 - d) Datetime values.
 - e) The SQL/JSON null value, which is distinct from all SQL values, including the SQL null.
- 2) SQL/JSON arrays.
 - 3) SQL/JSON objects.

These categories are discussed in the following subsections.

4.1.1 Atomic values

Atomic values in the SQL/JSON data model are virtually a subset of the values of <predefined type> in [ISO9075-2], Subclause 6.1, “<data type>”. To recap the choices in Foundation:

```
<predefined type> ::=
    <character string type> [ CHARACTER SET <character set specification> ]
    [ <collate clause> ]
    | <national character string type> [ <collate clause> ]
    | <binary string type>
    | <numeric type>
    | <boolean type>
    | <datetime type>
    | <interval type>
```

The SQL/JSON data model differs from Foundation predefined types as follows:

4.1 SQL/JSON items

- 1) The data model does not support <binary string type> and <interval type>.
- 2) Only character strings of Unicode characters are supported. The only collation is the binary (codepoint) collation of Unicode.
- 3) The SQL/JSON null value is regarded as the sole value of its own type. That is, there is no null character string value, null numeric value, *etc.* In addition, JavaScript semantics are used for nulls in comparisons — the SQL/JSON null is equal to the SQL/JSON null. (See [ECMAScript], section 11.9.6, “The strict equality comparison algorithm”, and similar sections.)

In general, operations in the path language operate on atomic values in the SQL/JSON data model with the same semantics as the corresponding operation in SQL, but note that null semantics follows [ECMAScript].

Datetimes have no serialized representation in JSON. They are part of the SQL/JSON data model to support comparison predicates after converting JSON strings to datetime.

4.1.2 SQL/JSON arrays

An SQL/JSON array is an ordered list of zero or more SQL/JSON items in the SQL/JSON data model. When serialized, the list is separated by commas and enclosed in square brackets, for example:

```
[ 2.3, "bye bye" ]
```

SQL arrays are 1-relative, whereas [ECMAScript] arrays are 0-relative. SQL/JSON arrays are also 0-relative. This allows some path expression to be used in either the SQL/JSON path language or in JavaScript.

4.1.3 SQL/JSON objects

An SQL-JSON object is an unordered set of zero or more members.

NOTE 7 — Empty objects are permitted; see [RFC7159], section 2.2, “Objects”, and [ECMAScript], section 15.12.1.2, “The JSON syntactic grammar”.

A member is a pair of values:

- 1) The first value is a character string and is called the key of the member.
- 2) The second value is any SQL/JSON item in the SQL/JSON data model and is called the bound value of the member.

An SQL/JSON object is serialized enclosed in curly braces, with the members listed in a non-deterministic order separated by commas. Each member is serialized as its key (a character string, therefore enclosed in double quotes), a colon, and then the serialization of the (second) value of the member. For example:

```
{ "name": "Fido", "tag": 12345 }
```

4.2 SQL/JSON sequences

An SQL/JSON sequence is an ordered list of zero or more SQL/JSON items. SQL/JSON sequences do not nest; they can only be concatenated. One may think of an SQL/JSON item as equivalent to an SQL/JSON sequence of that one item, which is an acceptable mental model, but this Technical Report endeavors to always view an SQL/JSON sequence as a container of zero or more SQL/JSON items.

4.3 Parsing JSON

Parsing refers to the process of importing from some storage format into the SQL/JSON data model. The storage format may be JSON text stored in a Unicode character string, or it may be some binary format such as AVRO or BSON. Since no specifications of AVRO and BSON are formally referenceable, they have been left as implementation extensions in [ISO9075-2].

The conversion from JSON is straightforward, because SQL/JSON data model is basically a superset of JSON. In particular:

Table 3 — Parallels between JSON text and SQL/JSON data model

JSON text	SQL/JSON data model
true	<i>true</i>
false	<i>false</i>
null	null
string, e.g., "hello dolly"	string, e.g., 'hello dolly'
number	The format for numbers in [RFC7159] is the same as numeric literals in SQL. Consequently, a number from JSON can be transferred into SQL by simply parsing it as a <signed numeric literal>.
array	SQL/JSON array
object	SQL/JSON object

4.4 Serializing JSON

Serializing JSON refers to the process of exporting a value from the SQL/JSON data model back to some storage format. Serialization is specified only to JSON text; conversion to some other format, such as AVRO or BSON, is left as an implementation-defined extension in [ISO9075-2].

SQL/JSON datetimes cannot be serialized; neither can SQL/JSON sequences of length greater than one.

4.4 Serializing JSON

The result of serialization is an implementation-dependent string of Unicode characters that, if parsed, would restore the original value in the SQL/JSON data model.

The precise result of serialization is implementation-dependent because:

- Meaningless whitespace is not specified.
- Because of escape sequences, there are multiple ways to serialize a character string.
- [\[ECMAScript\]](#), Sections 5.1.3, “The numeric string grammar”, and 9.3.1, “BNumber applied to the string type”, provide some discretion in the formatting of numbers.
- Members of an SQL/JSON object are unordered, so there are many possible permutations of the members of an object.

5 SQL/JSON functions

In [ISO9075-2], operations on JSON data are generally performed using a set of built-in functions specified explicitly for that purpose. These functions are called “SQL/JSON functions”. [ISO9075-2] also specifies some supporting SQL syntax, such as the IS JSON predicate.

5.1 Handle JSON using built-in functions

SQL/JSON functions are partitioned into two groups: constructor functions (JSON_OBJECT, JSON_OBJECTAGG, JSON_ARRAY, and JSON_ARRAYAGG) and query functions (JSON_VALUE, JSON_TABLE, JSON_EXISTS, and JSON_QUERY). Constructor functions use values of SQL types and produce JSON values (JSON objects or JSON arrays) represented in SQL character or binary string types. Query functions evaluate SQL/JSON path language expressions against JSON values, producing values of SQL/JSON types, which are converted to SQL types. The SQL/JSON path language is described in Clause 6, “SQL/JSON path language”.

5.2 JSON API common syntax

The SQL/JSON query functions all need a path specification, the JSON value to be input to that path specification for querying and processing, and optional parameter values passed to the path specification. They use a common syntax:

```
<JSON API common syntax> ::=
  <JSON context item> <comma> <JSON path specification> [ AS <JSON table path name> ]
    [ <JSON passing clause> ]

<JSON context item> ::=
  <JSON value expression>

<JSON path specification> ::=
  <character string literal>

<JSON passing clause> ::=
  PASSING <JSON argument> [ { <comma> <JSON argument> } ]

<JSON argument> ::=
  <JSON value expression> AS <identifier>
```

The type of the <value expression> contained in the <JSON value expression> immediately contained in the <JSON context item> is a string type. If the <JSON context item> does not implicitly or explicitly specify a <JSON input clause>, then FORMAT JSON is implicit.

If <JSON API common syntax> is not contained in <JSON table>, then <JSON table path name> is not required. The <JSON table path name> is optional; if <JSON table path name> is not specified, then an implementation-dependent <JSON table path name> is implicit. The <JSON context item> is the JSON input on which the SQL/JSON path expression operates.

5.2.1 JSON value expression

<JSON context item> is defined as a <JSON value expression>, which is in turn defined as:

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

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

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

A <JSON value expression> may have an optional <JSON input clause>. This indicates that the values expression should be parsed as JSON. The standardized option is FORMAT JSON; implementations may also support syntax such as FORMAT AVRO or FORMAT BSON. When using the <JSON input clause>, the <value expression> may be either a character string or a binary string.

If the user does not specify the ENCODING for a <JSON value expression>, then the SQL-implementation will determine it as one of the three encoding alternatives: UTF8, UTF16, and UTF32.

5.2.2 Path expression

The context item is followed by a comma and the SQL/JSON path expression. <JSON path specification> is a character string literal. Requiring a literal enables the implementation to optimize the query by analyzing the SQL/JSON path expression and planning accordingly, for example, if there are indexes available on the JSON data.

The optional <JSON table path name> is syntax that is only used in JSON_TABLE, and then only if the user wants to write an explicit plan for processing nested COLUMNS clauses. Explicit plans for JSON_TABLE are explained later.

5.2.3 PASSING clause

SQL's JSON facilities include a PASSING clause that is used to pass parameters to the SQL/JSON path expression. For example, suppose that an SQL/PSM routine has parameters “upper” and “lower” and it is desired to find rows in which the JSON data has a member called “age” between these two values. The user might write:

```
SELECT *
FROM T
WHERE JSON_EXISTS (T.C,
    'lax $ ? ($lo <= @.age && @.age <= $up) '
    PASSING upper AS "up",
           lower AS "lo")
```

In this example there are two variables defined in the PASSING clause. In “SQL-land” they are the PSM variables upper and lower; in “SQL/JSON-land” they are \$up and \$lo.

Syntactically, the PASSING clause is a comma-separated list of <JSON argument>s, defined as:

```
<JSON argument> ::=
  <JSON value expression> AS <identifier>
```

The <JSON value expression> specifies the value to be passed into the SQL/JSON path engine. This may be of any type supported by the data model (Unicode character strings, numbers, booleans and datetimes) or any other type that can be cast to a Unicode character string (for example, almost all non-Unicode character strings, or user-defined types with a user-defined cast to a Unicode character string type). In the example above, the <value expression>s were SQL/PSM variables, but any <value expression> may be used. Thus, a join to another table can be constructed by passing a column from one table into path expression on an SQL/JSON value of another table.

The <identifier> in the <JSON argument> specifies the variable name by which the value can be referenced within the SQL/JSON path expression. In this example, these identifiers are “up” and “lo”. Within the path expression, these are referenced with a prefixed dollar sign (\$up and \$lo), since \$ marks the variables in a path expression.

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

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

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

If FORMAT is not specified, then FORMAT JSON is implicit. If the <JSON output clause> specifies or implies JSON, then the <data type> shall identify a string type. FORMAT JSON specifies the data format specified in [RFC7159]. FORMAT <implementation-defined JSON representation option> specifies an implementation-defined data format.

NOTE 8 — 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 “Parsing a JSON text”, and “Serializing an SQL/JSON item”, respectively.

5.2.5 ON ERROR and ON EMPTY syntax

The SQL/JSON query functions also have ON ERROR and ON EMPTY clauses in common. However, the details of these clauses vary depending on the query function, so they are not included in the <JSON API common syntax> shown above. They will be described individually for each query function.

5.3 Query functions

The SQL/JSON query functions are:

- JSON_EXISTS — determine whether an SQL/JSON value satisfies a path specification.
- JSON_VALUE — extract an SQL scalar value from an SQL/JSON value.
- JSON_QUERY — extract an SQL/JSON value from an SQL/JSON value.
- JSON_TABLE — extract a table from an SQL/JSON value.

5.3.1 JSON_EXISTS

JSON_EXISTS determines whether a JSON value satisfies a search criterion. The syntax for JSON_EXISTS is:

```
<JSON exists predicate> ::=
  JSON_EXISTS <left paren>
    <JSON API common syntax>
    [ <JSON exists error behavior> ON ERROR ]
  <right paren>
```

```
<JSON exists error behavior> ::=
  TRUE | FALSE | UNKNOWN | ERROR
```

The syntax for JSON_EXISTS is the shared syntax `<JSON API common syntax>`, plus an optional `ON ERROR` clause, defaulting to `FALSE ON ERROR`. If the value of the `<JSON context item>` simply contained in the `<JSON API common syntax>` is the null value, then the result of `<JSON exists predicate>` is *Unknown*.

JSON_EXISTS is a predicate that can be used to test whether an SQL/JSON path expression is fulfilled in an SQL/JSON value. JSON_EXISTS evaluates the SQL/JSON path expression; the result is *True* if the path expression finds one or more SQL/JSON items.

Consider the following sample data in the following table with two columns, *K* (the primary key) and *J* (a column containing JSON text):

Table 4 — JSON_EXISTS sample data

K	J
101	{ "who": "Fred", "where": "General Products", "friends": [{ "name": "Lili", "rank": 5 }, { "name": "Hank", "rank": 7 }] }
102	{ "who": "Tom", "where": "MultiCorp", "friends": [{ "name": "Sharon", "rank": 2 }, { "name": "Monty", "rank": 3 }] }
103	{ "who": "Jack", "friends": [{ "name": "Connie" }] }
104	{ "who": "Joe", "friends": [{ "name": "Doris" }, { "rank": 1 }] }
105	{ "who": "Mabel", "where": "Black Label", "friends": [{ "name": "Buck", "rank": 6 }] }