N1QL Language Reference, DP4

Copyright

Chapter

1

Introduction

Topics:

The proliferation of internet commerce and activity coupled with the lower cost of computer processing and memory helped change the landscape of the database world. Relational SQL databases, so useful in the past, seem too clunky, rigid, and centralized to reflect new business requirements that, in real time, deal with huge sets of data, large numbers of users, and fluctuations in the size of those data and users. The very characteristics of relational databases that made them so useful turned into handicaps.

NoSQL databases solve many of the problems of this new world of big users, big data, and cloud computing. These databases are flexible (i.e. non-first normal form (N1NF)): data can be redundant and unstructured (as opposed to the rigid column/row/relationship structure of normalized relational database tables and schemas).

While NoSQL database solutions have entered the marketplace, the supporting high-level query languages have not been as quickly produced.

Enter N1QL. N1QL is a leading-edge query language created for use with Couchbase, a document-based database. N1QL provides a common query language and data model for JSON-document databases. N1QL aims to meet the query needs of distributed document-oriented databases.

N1QL is based on SQL. It has familiar data definition language (DDL), data manipulation language (DML), and query language (SELECT) statements, yet it works on NoSQL databases features such as key-value stores, multi-value attributes, and nested objects. If you know SQL, you can easily learn N1QL, in fact, you're more than halfway there.

Data in Couchbase

In Couchbase Server, data is stored in JSON documents. A document is a JSON object consisting of a number of fields that are defined. An object can be one or more documents and object attributes can be fields in the document. You can store data as key-value pairs, using the key as you would a primary key in SQL.

In Couchbase Server, you do not need to perform data modeling and establish relationships between tables the way you would in a traditional relational database. Every JSON document can have its own individual set of keys, although one or more informal schemas could be adopted for data, and each document has its own implicit schema that is represented in how you organize and nest information in your documents.Your document structure can evolve over time as your application grows and adds new features.

Unlike the relational model, attributes within a document do not have to have a predefined type. For instance, a zip code can be an integer or a string using the document model. Even documents belonging to the same class of application objects, such as a customer document can have different attributes than other customer documents. For example, some customers can have an extra field for shipping insurance, while other customer documents do not have that field at all.

N1QL Queries and Results

As mentioned, N1QL leverages and extends SQL.

In N1QL, queries are based on the SELECT statement. As with SQL, you can have nested subqueries. N1QL queries run on JSON documents. You can query over multiple documents using the JOIN clause.

Additionally, because the data can be nested, there are operators and functions that let you navigate through nested arrays. Because data can be irregular, you can use conditions specified in the WHERE clause to retrieve them.

N1QL provides paths to support nested data. A path uses a dot notation syntax and provides the logical location of an attribute within a document. For example, to get the street from a customer order, the path, `orders.billTo.street` is used. This path refers to the value for 'street' in the 'billTo' object. A path is used with arrays or nested objects to get to attributes within the data structure.

Array syntax in the path can also be used to get to information. For example, this path `orders.items[0].productId` evaluates to the productId value for the first array element under the order item, items.

Paths provide a method for finding data in document structures without having to retrieve the entire document or handle it within an application. Any document data can be requested and returned to an application. When only relevant information is returned to an application, querying bandwidth is reduced.

In N1QL, a query returns a set of documents. The returned document set is not required to be uniform, though it can be. Typically, specifying a SELECT with fixed set of attribute (column) names results in a uniform set of documents. SELECT with a wildcard (*) results in non-uniform result set. The only guarantee is that every returned document meets the query criteria.

You can also use standard GROUP BY, ORDER BY, LIMIT, and OFFSET clauses as well as a rich set of functions to transform the results as needed.

N1QL and SQL

The most important difference between traditional SQL and N1QL is the data model. In a traditional SQL database, data is constrained to tables with a uniform structure. Here is how a very simple Employee database might look in a relational database. It consists of two tables, Employee and Employers. Name is the primary key.

```


EMPLOYEE
```

```
Name | SSN | Wage
-----------------------------------------------------------------------
Jamie | 234 | 123
Steve | 123 | 456

SCHEMA:
Name -> String of width 100
SSN -> Number of width 9
Wage -> Number of width 10

EMPLOYERS:
-----------------------------------------------------------------------
 Name_Key  | Company   | Start | End
 Jamie     | Yahoo     | 2005  | 2006
 Jamie     | Oracle    | 2006  | 2012
 Jamie     | Couchbase | 2012  | NULL
```

In N1QL, the data exists as free-form documents, gathered as large collections called keyspaces. There is no uniformity and there is no logical proximity of objects of the same data shape in a keyspace. This is what the data would look like for N1QL:

```
 (HRData keyspace)
 {
     'Name': 'Jamie'
     'SSN': 234
     'Wage': 123
     'History':
      [
       ['Yahoo', 2005, 2006],
       ['Oracle', 2006, 2012],
     ]
 },

 {
     'Name': Steve
     'SSN':  123,
     'Wage': 456,
 }
```

Projection Differences

When you run a query in SQL, a set of rows, consisting of one or more columns each is returned. A header can be retrieved to obtain metadata about each column. It is generally not possible to get rowset where each row has a different set of columns.

```
SELECT Name, Company
    FROM Employee, Employers
    WHERE Name_Key = Name

 Name | Company
 ----------------
 Jamie | Oracle
 Jamie | Yahoo
 Jamie | Couchbase
 ----------------
```

Like SQL, N1QL allows renaming fields using the AS keyword. However, N1QL also allows reshaping the data, which has no analog in SQL. This is done by embedding the attributes of the statement in the desired result object shape.

```
 SELECT Name, History, {'FullTime': true} AS 'Status'
      FROM HRData

  {
     'Name': 'Jamie',
     'History':
     [
       ['Yahoo', 2005, 2006],
       ['Oracle', 2006, 2012],
       ['Couchbase', 2012, null]
     ],
     'Status': { 'FullTime': true }
 }
 {
     'Name': 'Steve',
     'Status': { 'FullTime': true }
 }
```

Selection Differences

In N1QL, the FROM clause is used to select between data sources, known as keyspaces. If HRData is a keyspace, the following statement selects the Name attribute from all documents in the HRData keyspace that have a Name attribute defined.

```
SELECT Name FROM HRData
```

A new twist is that each document can itself be regarded as a data source and the query run over its nested elements. Such nested elements are addressed using the dot (.) operator to descend a level, and the square bracket ([ ]) operator to index into an array element.

```
  SELECT FullTime FROM HRData.Status
{
     'FullTime': true
}
```

The selected fields can also be renamed using the AS operator, just like in SQL.

```
SELECT firstjob FROM HRData.History[0] AS firstjob
{
     'firstjob': ['Yahoo', 2005, 2006]
}

SELECT firstjob[2] FROM HRData.History[0] AS firstjob
{
     'firstjob[2]': 2006
}
```

Filtering Differences

N1QL supports the WHERE clause as does SQL with some slight differences The dot (.) and the square bracket ([])operators can be used to access nested elements as they are used in SELECT clauses.

Because N1QL data can be irregularly shaped, undefined values are recognized as distinct from null. A complementary set of operators like IS MISSING is added in addition to standard operators like IS NULL. New conversions, for example from non-zero integer values to Boolean value true, are supported as well.

Most standard SQL functions (e.g. LOWER()) are defined. In addition to standard filtering predicates, three new operators are introduced: ANY, SOME, and EVERY. These operators help in dealing with arrays in documents. ANY and SOME evaluate a condition for each element, and return true if any element meets the condition. EVERY also evaluates a condition for each element, except it returns true only if all elements matched the condition.

Key Features of N1QL

N1QL provides a rich set of features that let you retrieve, manipulate, transform, and create JSON document data. Here is a summary of the key features:

- **SELECT Statement**: N1QL's SELECT statement has the functionality of the SQL SELECT statement, but extends it to work with JSON documents. This lets you use your knowledge of SQL to work with the powerful NoSQL features of Couchbase that let you work with big data, big users, and cloud computing.
- **Data Manipulation Language (DML)**: N1QL provides DELETE, INSERT, UPDATE, and UPSERT statements. These statements allow you to create, delete, and modify the data stored in JSON documents by specifying and executing simple commands.
- **Indexes**: You can use the CREATE and DROP INDEX statements to easily create, and delete indexes.
- **Primary Key Access**: Data can be stored in key-value pairs, providing primary key access to any data.
- **Aggregation**: N1QL provides standard SQL aggregation operators, such as MIN, MAX, COUNT as well as grouping operators, the GROUP BY clause, and the group filter, HAVING.
- **Joins**: N1QL lets you retrieve data from multiple documents with joins specified in the FROM clause.
- **Nesting**: Nests let you retrieve sub-records for a record. That is, for each left hand input, the matching right hand inputs are collected into an array, which is then embedded in the result. For example, you can nest each item in a customer's order. Nests can be performed on multiple documents.
- **Unnesting**: Unnesting is the opposite of nesting. It extracts nested sub-records from a record, making each extraction a separate record. Unnesting is performed on a single document. Here is an example:

```
   SELECT Name, Job
 FROM HRData
 UNNEST HRData.History AS Job

{
     'Name': Jamie
     'Job': ['Yahoo', 2005, 2006]
 }
 {
     'Name': Jamie
     'Job': ['Oracle', 2006, 2012]
 }
 {
     'Name': Jamie
     'Job': ['Couchbase', 2012]
 }
```

- **Subqueries**: N1QL lets you ask multiple-part queries by using subqueries. A subquery is a query that is nested inside a SELECT, INSERT, or UPSERT statement. It can also be contained inside another subquery. When used with the SELECT statement or nested within another subquery, subqueries let you continue to refine the results of preceding queries.

- **UNION, INTERSECT, and EXCEPT**: N1QL provides the UNION, INTERSECT, and EXCEPT operations that let you compare two or more queries. UNION combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union. EXCEPT returns any distinct values from the left query that are not found on the right query. INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand.

Conventions

Conventions used to specify general syntax notation.

Convention	Notation	Example	Description
Upper case		SELECT	Indicates keywords.
Single or double quotes	' or "	city = 'London'	Surrounds expressions that are specific query items. Either single or double quotes can be used.
Square brackets	[ ]	[NOT] BETWEEN	Indicates optional parts.
Pipe	|	AND | OR	Separates alternatives.
Parentheses	( )	COUNT(*)	Functions to perform an operation.
Bold square, curly brackets and parentheses	**[], {}, ()**	**[**`element`**]**	Indicate that the brackets or parentheses are required in the syntaxs when a N1QL statement is constructed.

Chapter 2

Quick Start

Topics:

Welcome to N1QL. Here are some things to get you started quickly.

- Installation instructions describe how to install the N1QL query engine, query shell, and query tutorial.
- Query Tutorial is an interactive online tutorial that gets you up and querying quickly.
- Query Server is the N1QL query engine.
- Query Shell is the N1QL interface through which you can run queries.

Installation

To get started, navigate to the downloads page and download N1QL.

To run N1QL on your local system:

1. Expand the package archive.
2. On the command line, navigate to your local N1QL directory.
3. Run one of the following commands.

   For Unix:

   `./start_tutorial.sh`

   For Windows:

   `./start_tutorial.bat`

Running the Query Tutorial

The online tutorial is a good way to get started and play with N1QL in an easy, fun environment.

1. Make sure you've followed the instructions in the Installation section.
2. Open `>http://localhost:8093/tutorial` in your browser to use the tutorial on your local server.
3. To connect N1QL with your Couchbase Server:

   ```
   ./cbq-engine -datastore=http://[server_name]:8091/
   ```
4. To use the command-line interactive query tool, run the following command:

   ```
   ./cbq -engine=http://[couchbase-query-engine-server-name]:8093/
   ```
5. Before issuing queries against a Couchbase keyspace, run the following command from the query command line:

   ```
   CREATE PRIMARY INDEX ON [keyspace-name]
   ```

Running the Query Server

The Query Server is the N1QL query engine. You can run the Query Server through the Couchbase Server or locally.

1. To connect N1QL with your Couchbase Server:

   ```
   ./cbq-engine -datastore=http://[server_name]:8091/
   ```
2. Alternatively, you can connect N1QL to a local datastore, by running this command:

   ```
   ./cbq-engine -datastore=$PATH_TO_DATASTORE -namespace $DEFAULT_NAMESPACE
   ```

   For example, `./cbq-engine -datastore "$HOME/sample_build/tutorial" -namespace data`

Running the Query Shell

To use the command-line interactive query tool, run this command:

```
```

```
./cbq -engine=http://[couchbase-query-engine-server-name]:8093/
```

Chapter 3

System Information

Topics:

N1QL has a system catalog that stores metadata about a database. The system catalog is a namespace. The namespace is called *system.*

There is a keyspace for each type of artifact. The keyspace names are plural in order to avoid conflicting with N1QL keywords.

Logical Hierarchy

N1QL has the following artifacts.

Datastores

Datastores are analagous to sites. A datastore is a database deployment, e.g. a server cluster, cloud service, or mobile installation. Analogous to a RDBMS instance.

Namespaces

Namespaces are analagous to pools. A namespace is a unit of authorization, resource allocation, and tenancy. It is analogous to a RDBMS database or schema

Keyspaces

Keyspaces are analagous to buckets. A keyspace is a set of documents that are allowed to vary in structure. It is a unit of authorization and resource allocation. Analogous to a relational database table.

Indexes

An index on a keyspace. It is analogous to a relational database index. Types of indexes include tree, view, fulltext, hash, and other indexes.

Duals

The dual keyspace has been added for evaluating constant expresssions. It contains a single entry with no attributes.

Querying the System Catalog

Datastores

Datastores can be queried as follows:

```
SELECT * FROM system:datastores
```

It will return the following attributes:

- id: string (id for the datastore)
- url: string (URL for the datastore instance)

Namespaces

Namespaces can be queried as follows:

```
SELECT * FROM system:namespaces
```

It will return the following attributes:

- id: string (id for the namespace)
- name: string (name for the namespace)
- store_id: string (id for the datastore to which the namespace belongs)

Keyspaces

Keyspaces can be queried as follows:

```
SELECT * FROM system:keyspaces
```

It will return the following attributes:

- id: string (id for the keyspace)
- name: string (name for the keyspace)
- namespace_id: string (id for the namespace to which the keyspace belongs)
- store_id: string (id for the datastore to which the keyspace belongs)

Indexes

Indexes can be queried as follows:

```
SELECT * FROM system:indexes
```

It will return the following attributes:

- id: string (id for the index)
- name: string (name for the index)
- index_key: array of strings (list of index keys)
- index_type: string (type of index, for example, view index)
- keyspace_id: string (id for the keyspace to which the index belongs)
- namespace_id: string (id for the namespace to which the index belongs)
- store_id: string (id for the datastore to which the index belongs)

Dual

Dual can be used to evaluate constant expressions.

```
SELECT 2+5 FROM system:dual
```

It will return 7.

Chapter

4

Language Structure

Topics:

Statements

N1QL statements fall into three categories: data definition (DDL), data manipulation (DML), and queries. Data definition statements let you create, modify, and delete indexes. Data manipulation statements let you delete, insert, update, and upsert data into JSON documents. Queries let you retrieve data.

Expressions

Here are the types of N1QL expressions:

- literal-value
- identifier
- nested
- case
- logical
- comparison
- arithmetic
- concatenation
- construction
- function call
- subquery
- collection

Comments

N1QL supports block comments. Here is the syntax:

```
/* [[text] | [newline]]+ */
```

Reserved Words

The following keywords are reserved and cannot be used as unescaped identifiers. All keywords are case-insensitive.

Escaped identifiers can overlap with keywords.

Some of these keywords are not currently implemented but are reserved for future use.

ALL

ALTER

ANALYZE

AND

ANY

ARRAY

AS

ASC

BEGIN

BETWEEN

BINARY

BOOLEAN

BREAK

BUCKET

BY

CALL

CASE

CAST

CLUSTER

COLLATE

COLLECTION

COMMIT

CONNECT

CONTINUE

CREATE

DATABASE

DATASET

DATASTORE

DECLARE

DELETE

DERIVED

DESC

DESCRIBE

DISTINCT

DO

DROP

EACH

ELEMENT

ELSE

END

EVERY

EXCEPT

EXCLUDE

EXECUTE

EXISTS

EXPLAIN

FALSE

FIRST

FLATTEN

FOR

FROM

FUNCTION

GRANT

GROUP

GSI

HAVING

IF

IN

INCLUDE

INDEX

INLINE

INNER

INSERT

INTERSECT

INTO

IS

JOIN

KEY

KEYS

KEYSPACE

LAST

LEFT

LET

LETTING

LIKE

LIMIT

LSM

MAP

MAPPING

MATCHED

MATERIALIZED

MERGE

MINUS

MISSING

NAMESPACE

NEST

NOT

NULL

NUMBER

OFFSET

ON

OPTION

OR

ORDER

OUTER

OVER

PARTITION

PASSWORD

PATH

POOL

PREPARE

PRIMARY

PRIVATE

PRIVILEGE

PROCEDURE

PUBLIC

RAW

REALM

REDUCE

RENAME

RETURN

RETURNING

REVOKE

RIGHT

ROLE

ROLLBACK

SATISFIES

SCHEMA

SELECT

SELF

SET

SHOW

SOME

START

STATISTICS

STRING

SYSTEM

THEN

TO

TRANSACTION

TRIGGER

TRUE

TRUNCATE

UNDER

UNION

UNIQUE

UNNEST

UNSET

UPDATE

UPSERT

USE

USER

USING

VALUE

VALUED

VALUES

VIEW

WHEN

WHERE

WHILE

WITH

WITHIN

WORK

XOR

Chapter 5

Data types

Topics:

N1QL supports the following data types:

- Booleans
- Numbers
- Strings
- Arrays
- Objects
- NULL
- MISSING

These types are described in detail in their respective sections. While N1QL does not support the Date data type, it does provide a robust set of functions to work with dates.

Booleans

This type can have a value of TRUE and FALSE. The values are case insensitive.

Numbers

Numbers can be a signed decimal number that can contain a fractional part. Numbers can also use E-notation.

Strings

Strings include all Unicode characters and backslash escape characters. They are delimted by single quotation marks ('') or double quotation marks ("").

Arrays and Objects

Arrays

Arrays are an ordered list of zero or more values. The values can be of any type. Arrays are enclosed in square brackets ([ ]). Commas separate each value.

Here are some array examples: [“one”, “two”, “three”] and [1, 2, 3], and ["one", 2, "three"].

Objects

Objects consist of name-value pairs. The name must be a string, and the value can be any supported JSON data type.

Objects are enclosed in curly braces ({ }). Commas separate each pair. The colon (:) character separates the name from its value within each pair.

Here are some object examples: {"age": 17}, {"fname": "Jane", "lname": "Doe"}

All names must be strings and should be distinct from each other within that object.

NULL and MISSING

NULL

Nulls represent empty values using the keyword NULL. For example, a developer might initially set a field value to null by default until a user enters a value.

Null values are also generated by certain operations, for example, when dividing by zero or passing arguments of the wrong type.

Note that NULL is case insensitive. For example, null, NULL, Null, and nUll are all equivalent.

MISSING

Missing represents a missing name-value pair in a JSON document. If the referenced field does not exist, an empty result value is returned by the query. Missing is added as a literal expression, although it is not returned in the final results. It is omitted from objects, and is converted to null in result arrays.

Because N1QL is not constrained by a fixed schema, some objects and documents can contain fields that others do not contain; so a field can be present in one document and MISSING in another. (MISSING is not present in SQL, because every record in a table follows an identical schema.)

Collation

Here is the collation order used for N1QL data types:

- MISSING
- NULL
- boolean
- number
- string
- array
- object
- binary (non-JSON)

Date

N1QL does not support the Date data type, however N1QL does provide a full range of functions with which you can manipulate dates. See the “Date Functions” section for more information.

Chapter 6

Literals

Topics:

Literal values include strings, numbers, TRUE, FALSE, NULL, and MISSING.

N1QL supports the same literals as JSON, as defined by *json.org* with three exceptions:

- In N1QL, "true", "false," and "null" are case-insensitive to be consistent with other N1QL keywords. In standard JSON, "true", "false," and "null" are case-sensitive.
- "missing" is added as a literal expression, although it is not returned in final results. Missing is omitted from objects, and is converted to null in result arrays.
- In N1QL single and double quotation marks can be used for strings. JSON supports only double quotation marks.

The following query returns emails from contacts where the contact name is 'dave'.

```
SELECT email
   FROM contacts
   AS contact

WHERE contact.name = 'dave'
```

Returns:

```
{   "email": "dave@gmail.com" }
```

Booleans

```
 TRUE | FALSE
```

Boolean propositions evaluate to TRUE and FALSE. These values are case-insensitive.

Numbers

```
[-] [ digits] [fraction] [exponent]
```

Fraction:

```
 . digits
```

Exponent:

```
e | E  [ + | - ]  digits
```

Numbers can be either signed or unsigned digits with an optional fractional component and an optional exponent. The numbers should not start with a zero.

Strings

```
 "  characters  "
```

Characters:

```
 Unicode Character or an Backslash Escape character
```

Backslash Escaped characters:

```
 \  ( \ | / | b | f | n | r | t | u hex-digit hex-digit hex-digit hex-
digit )
```

Strings can be either unicode characters or escaped characters.

NULL and MISSING

NULL literal:

```
NULL
```

MISSING literal:

```
MISSING
```

Chapter 7

Identifiers

Topics:

An identifier is a symbolic reference to a value in the current context of a query. Identifiers can include keyspace names, fields within documents, and aliases. N1QL identifiers are case-sensitive.

An identifier can either be escaped or unescaped.

```
unescaped-identifier | escaped-identifier
```

Example

If the current context is the document:

```
{"name": "n1ql"}
```

Then the identifier `name` would evaluate to the value n1ql.

Unescaped Identifiers

Unescaped identifiers cannot support the full range of identifiers allowed in a JSON document, but do support the most common ones with a simpler syntax.

```
[a-zA-Z_] ( [0-9a-zA-Z_$] )*
```

Escaped Identifiers

Escaped identifiers are surrounded by back-ticks and support all identifiers in JSON. You can use the back-tick character within an escaped identifier by specifying two consecutive back-tick characters. Keywords cannot be escaped; therefore, escaped identifiers can overlap with keywords.

```
` characters `
```

For example, if you have a hyphen in an attribute name, it can be referred to as `first-name`.

Chapter

8

Operators

Topics:

Operators perform a specific operation on the input values or expressions.

N1QL provides a full set of operators that you can use within its statements. Here are the categories of N1QL operators:

- arithmetic
- collection
- comparison
- conditional
- construction
- logical
- nested
- string

Arithmetic Operators

Arithmetic operations perform arithmetic methods within an expression or any numerical value retrieved as part of query clauses. These operations include basic mathematical operations such as addition, subtraction, multiplication, divisions, and modulo. In addition, a negation operation changes the sign of a value.

These arithmetic operators only operate on numbers. If either operand is not a number, it will evaluate to NULL.

Operator	Description
+	Add values.
-	Subtract right value from left.
*	Multiply values.
/	Divide left value by right.
%	Modulo. Divide left value by right and return the remainder.
-*value*	Negate value.

arithmetic-term:

```
expression + expression | expression - expression |
  expression * expression | expression / expression |
  expression % expression | - expression
```

Example

This query selects the document where name is 'dave' and returns the name, age, and age times 12.

```
SELECT name, age, age*12
 AS age_in_months
             FROM tutorial
                  WHERE name = 'dave'

Returns:
{
    "age": 45,
    "age_in_months": 540,
    "name": "dave"
 }
```

Collection Operators

Collection operators allow you to evaluate expressions over collections or object(s). These include ANY, EVERY, ARRAY, FIRST, EXISTS , IN and WITHIN.

```
exists-expression | in-expression | within-expression | ANY | EVERY | ARRAY
 | FIRST
```

ANY

```
ANY variable ( IN  | WITHIN ) expression
    [  ,  variable ( IN | WITHIN ) expression  ]*
    [ SATISFIES condition ]  END
```

Any is a range predicate that allows you to test a Boolean condition over the elements or attributes of a collection, object, or objects. It uses the IN and WITHIN operators to range through the collection.

If at least one item in the array satisfies the ANY expression, then returns TRUE, otherwise returns FALSE.

Example

In the following query, all contacts who have one or more children over the age of 14 are retrieved. This query returns one item for 'dave' because one of his children is 17.

```
SELECT name
  FROM contacts
  WHERE ANY child IN children
  SATISFIES child.age > 14 END
```

Results:

```
"results": [ { "name": "dave" }, .... ]
```

SATISFIES

Sometimes the conditions you want to filter need to be applied to the arrays nested inside the document. The SATISFIES keyword is used to specify the filter condition. The expression after the ANY clause allows us to assign an identifier to an element in the array that we are searching through.

Example

```
SELECT fname, children
   FROM tutorial
       WHERE ANY child IN tutorial.children SATISFIES child.age > 10  END
```

Returns

```
{
 "resultset": [
   {
     "children": [
       {
         "age": 17,
         "fname": "Aiden",
         "gender": "m"
       },
       {
         "age": 2,
         "fname": "Bill",
         "gender": "f"
       }
     ],
     "fname": "Dave"
   },
   {
     "children": [
        {
          "age": 17,
          "fname": "Xena",
          "gender": "f"
        },
        {
          "age": 2,
          "fname": "Yuri",
          "gender": "m"
        }
      ],
      "fname": "Earl"
    },
    {
      "children": [
        {
          "age": 17,
          "fname": "Abama",
          "gender": "m"
        },
        {
          "age": 21,
          "fname": "Bebama",
          "gender": "m"
        }
      ],
      "fname": "Ian"
    }
  ]
}
```

EVERY

```
EVERY variable ( IN  | WITHIN ) expression
   [  ,  variable ( IN | WITHIN ) expression  ]*
   [ SATISFIES condition ]  END
```

EVERY is a range predicate that allows you to test a Boolean condition over the elements or attributes of a collection, object, or objects. It uses the IN and WITHIN operators to range through the collection.

If every array element satisfies the EVERY expression, returns TRUE. Otherwise returns FALSE. If the array is empty, returns TRUE.

Example

EVERY returns TRUE if all items meet the condition. For example, this query is almost identical to the preceding one except EVERY is used instead of ANY. This query scans all contacts and returns the name of any contact that has children over the age of 10. The result tells us that out of all of the contacts only 'ian' has children who are all over the age 10.

```
 SELECT name
    FROM contacts
    WHERE EVERY child IN children
    SATISFIES child.age > 10 END
```

Results

```
  { "name": "ian" }
```

SATISFIES

Sometimes the conditions you want to filter need to be applied to the arrays nested inside the document.

The SATISFIES keyword is used to specify the filter condition. The expression after the EVERY clause allows us to assign a name to an element in the array that we are searching through.

Example

```
SELECT fname, children
   FROM tutorial
       WHERE EVERY child
       IN tutorial.children SATISFIES child.age > 10  END
```

Returns

```
 {
 "resultset": [
   {
     "children": [
       {
         "age": 17,
         "fname": "Aiden",
         "gender": "m"
       },
       {
         "age": 2,
         "fname": "Bill",
         "gender": "f"
       }
     ],
     "fname": "Dave"
   },
   {
     "children": [
       {
         "age": 17,
         "fname": "Xena",
         "gender": "f"
       },
       {
         "age": 2,
         "fname": "Yuri",
         "gender": "m"
       }
     ],
     "fname": "Earl"
   },
   {
     "children": [
       {
         "age": 17,
         "fname": "Abama",
         "gender": "m"
       },
       {
         "age": 21,
         "fname": "Bebama",
         "gender": "m"
```

```
        }
      ],
      "fname": "Ian"
    }
  ]
}
```

ARRAY

array-expression:

```
ARRAY expression FOR variable ( IN |  WITHIN ) expression
   [ ,  variable ( IN | WITHIN ) expression ]* [ ( WHEN  condition) ] END
```

The ARRAY operator lets you map and filter the elements or attributes of a collection, object, or objects. It evaluates to an array of the operand expression, that satisfies the WHEN clause, if provided.

FIRST

first-expression:

```
FIRST expression FOR variable ( IN |  WITHIN )
  expression [ ,  variable ( IN | WITHIN ) expression]*
  [ ( WHEN  condition) ] END
```

The FIRST operator lets you map and filter the elements or attributes of a collection, object, or objects. It evaluates to a single element based on the operand expression that satisfies the WHEN clause, if provided..

EXISTS

exists-expression:

```
EXISTS expression
```

EXISTS evaluates to TRUE if the value is an array and contains at least one element.

IN

in-expression:

```
expression [ NOT ] IN expression
```

IN evaluates to TRUE if the right-side value is an array and directly contains the left-side value. NOT IN evaluates to TRUE if the right-side value is an array and does not directly contain the left-side value..

WITHIN

within-expr:

```
expression [NOT] WITHIN expression
```

WITHIN evaluates to TRUE if the right-side value contains the left-side value as a child or descendant. NOT WITHIN evaluates to TRUE if the right-side value does not contain the left-side value as a child or descendant.

Comparison Operators

Comparison operators allow two expressions to be compared.

The following table describes each comparison operator and its return values.

Comparison	Description	Returns
=	Equals to. Functionally equivalent to == for compatibility with other languages.	TRUE or FALSE
==	Equals to. Functionally equivalent to = for compatibility with other languages.	TRUE or FALSE
!=	Not equal to. Functionally equivalent to <> for compatibility with other languages.	TRUE or FALSE
<>	Not equal to. Functionally equivalent to != for compatibility with other languages.	TRUE or FALSE
>	Greater than.	TRUE or FALSE
>=	Greater than or equal to.	TRUE or FALSE
<	Less than.	TRUE or FALSE
<=	Less than or equal to.	TRUE or FALSE
BETWEEN	Search criteria for a query where the value is between two values, including the end values specified in the range. Values can be numbers, text, or dates.	TRUE or FALSE
NOT BETWEEN	Search criteria for a query where the value is outside the range of two values, including the end values specified in the range. Values can be numbers, text, or dates.	TRUE or FALSE
LIKE	Match string with wildcard expression. % for zero or more wildcards, _ to match any character at this place in a string. The wildcards can be escaped by preceding them with a backslash (\). Backslash itself can also be escaped by preceding it with another backslash.	TRUE or FALSE

NOT LIKE	Inverse of LIKE. Return TRUE if string is not similar to given string.	TRUE or FALSE
IS NULL	Field has value of NULL.	TRUE or FALSE
IS NOT NULL	Field has value or is missing.	TRUE or FALSE
IS MISSING	No value for field found.	TRUE or FALSE
IS NOT MISSING	Value for field found or value is NULL.	TRUE or FALSE
IS VALUED	Value for field found. Value is neither missing nor NULL	TRUE or FALSE
IS NOT VALUED	Value for field not found. Value is NULL.	TRUE or FALSE

Strings

String comparison is done using a raw-byte collation of UTF8 encoded strings (sometimes referred to as binary, C, or memcmp). This collation is case sensitive. Case-insensitive comparisons can be performed using the UPPER() or LOWER() functions. See the "String Functions" section for more information.

Arrays and Objects

Arrays are compared element-wise. Objects are first compared by length; objects of equal length are compared pair-wise, with the pairs sorted by name.

NULL and MISSING

The IS/IS NOT NULL/MISSING family of operators lets you specify conditions based on the existence (or absence) of attributes in a data set.

Additionally, these comparison situations produce the following results.

- If either operand in a comparison is MISSING, the result is MISSING.
- If either operand in a comparison is NULL, the result is NULL.
- If either operand is MISSING or NULL, the result is MISSING or NULL.

Example

```
SELECT fname, children
   FROM tutorial
      WHERE children IS NULL
```

Returns:

```
{
  "resultset": [
    {
      "children": null,
      "fname": "Fred"
    }
  ]
}
```

Example

```
    SELECT fname, children
       FROM tutorial
          WHERE children IS MISSING
```

Returns

```
    {
  "resultset": [
    {
      "fname": "Harry"
    },
    {
      "fname": "Jane"
    }
  ]
}
```

Conditional Operators

Case expressions evaluate conditional logic in an expression.

case-expr:

```
simple-case-expression | searched-case-expression
```

Simple Case Expression

simple-case-expr:

```
  CASE expression  ( WHEN expression THEN expression)
     [ ( WHEN expression THEN expression) ]*
     [  ELSE expression ]  END
```

Simple case expressions allow for conditional matching within an expression. The evaluation process is as follows:

- The first WHEN expression is evaluated. If it is equal to the search expression, the result of this expression is the THEN expression.
- If it is not equal, subsequent WHEN clauses are evaluated in the same manner.
- If none of the WHEN expressions are equal to the search expression, then the result of the CASE expression is the ELSE expression.
- If no ELSE expression was provided, the result is NULL.

Searched Case Expression

searched-case-expression:

```
 CASE  ( WHEN  condition THEN expression)
```

```
        [( WHEN  condition THEN expression ) ]*
        [ ELSE  expression ] END
```

Searched case expressions allow for conditional logic within an expression. The evaluation process is as follows:

- The first WHEN expression is evaluated.
- If TRUE, the result of this expression is the THEN expression.
- If not TRUE, subsequent WHEN clauses are evaluated in the same manner.
- If none of the WHEN clauses evaluate to TRUE, then the result of the expression is the ELSE expression.
- If no ELSE expression was provided, the result is NULL.

Example

The following example uses a CASE clause to handle documents that do not have a ship date. This scans all orders. If an order has a shipped-on date, it is provided in the result set. If an order does not have a shipped-on date, default text appears.

```
 SELECT
   CASE WHEN `shipped-on`
   IS NOT NULL THEN `shipped-on`
   ELSE "not-shipped-yet"
   END
   AS shipped
   FROM orders
```

Returns

```
{ "shipped": "2013/01/02" },
{ "shipped": "2013/01/12" },
{ "shipped": "not-shipped-yet" },
```

Construction Operators

There are two types of construction operators: array and object.

Array construction

```
 [ elements [, elements ]* ]
```

Elements

```
expression
```

Arrays are ordered lists with 0 or more values. These values can be one of the defined types. Arrays are enclosed in square brackets ([ ]). Commas separate each value. For example: [“one”, “two”, “three”] and [1, 2, 3] .

Objects construction

```
{ members [ , members ]* }
```

members:

```
string : expression
```

Objects contain name value members. Objects are enclosed in curly braces {}. Commas separate each pair (member). The colon (:) character separates the key or name from its value within each pair. All keys must be strings and should be distinct from each other within that object. The value can be any supported JSON data type.

Logical Operators

Logical terms let you combine other expressions using Boolean logic. N1QL provides three logical operators:

- AND
- OR
- NOT

AND

```
condition AND condition
```

AND evaluates to TRUE only if both conditions are TRUE.

OR

```
condition OR condition
```

OR evaluates to TRUE if one of the conditions is TRUE.

NOT

```
NOT condition
```

NOT evaluates to TRUE if the expression does not match the condition.

Nested Operators

Fields:

```
expression . ( identifier  | escaped identifier [ i ])
```

Elements:

```
expression [ expression ]
```

Array Slicing

```
expression [expression : [ expression ] ]
```

Two special operators are needed to access the data because Couchbase documents can have nested elements and embedded arrays. The '.' operator is used to refer to children down one level, and the '[ ]' is used to refer to an element in an array. You can use a combination of these operators to access nested data at any depth in a document.

Field Selection

The '.' operator is used to refer to children down one level in a nested expression.

Nested expressions support using the dot (.) operator to access fields nested inside of other objects. The form .[expression] is used to access an object field named by evaluating the expression contained in the brackets.

field-expression

```
expression . ( identifier  | escaped identifier [ i ])
```

By default, field names are case sensitive. To access a field case-insensitively, include the trailing *i*.

Example:

If you have the following data:

```
{
  "address": {
    "city": "Mountain View"
  },
  "revisions": [2013, 2012, 2011, 2010]
}
```

The following expressions all evaluate to `"Mountain View"`.

`address.city` , `address.`CITY`i`, `address.["ci" || "ty"]`, and `address.["CI" || "TY"]i`

Element Selection

Nested expressions also support using the bracket notation ([position]) to access elements inside an array. The [ ] operator is used to refer to an element in an array. Negative positions are counted backwards from the end of the array.

element-expression

```
expression [ expression ]
```

```
 {
    "address": {
    "city": "Mountain View"
    },
    "revisions": [2013, 2012, 2011, 2010]
}}
```

In our example, the expression `revisions[0]` evaluates to `2013`. The expression `revision[-1]` evaluates to `2010`.

Array Slicing

You can get subsets or segments of an array; this is called array slicing. Here is the syntax for array slicing:

```
source-array [ start : end ]
```

It returns a new a subset of the source array, containing the elements from position `start` to `end-1`. The element at `start` is included, while the element at `end` is not. The array index starts with 0.

If `end` is omitted, all elements from `start` to the end of the source array are included.

Negative positions are counted backwards from the end of the array.

This is the general nested expression syntax.

```
expression [expression : [ expression ] ]
```

Examples

The following shows a document and expressions with equivalent values.

```
{
  "address": {
       "city": "Mountain View"
  },
  "revisions": [2013, 2012, 2011, 2010]
}
```

The expression `address.city` evalutes to the value, Mountain View.

The expression `revisions[0]` evaluates to the value, 2013.

The expression `revisions[1:3]` evaluates to the array value, [2012, 2011].

The expression `revisions[1:]` evaluates to the array value [2012, 2011, 2010].

String Operators

N1QL provides the concatentation string operator.

```
expression || expression
```

The following example shows concatenation of two strings.

Example

```
SELECT fname || " " || lname AS full_name
    FROM tutorial
```

Returns:

```
  {
   "resultset": [
     {
       "full_name": "Dave Smith"
     },
     {
       "full_name": "Earl Johnson"
     },
     {
       "full_name": "Fred Jackson"
     },
     {
       "full_name": "Harry Jackson"
     },
     {
       "full_name": "Ian Taylor"
     },
     {
       "full_name": "Jane Edwards"
     }
   ]
 }
```

Operator Precedence

N1QL supports the following operators. Parentheses let you group operators and expressions. The items enclosed in parentheses are evaluated first.

The following table shows operator precedence level. An operator at a higher level is evaluated before an operator at a lower level.

Evaluation Order	Operator
1	CASE
2	. (period)
3	[ ] (left and right bracket)
4	- (unary)
5	* (multiply), / (divide), % (modulo)
6	+, - (binary)
7	IS
8	IN
9	BETWEEN
10	LIKE

11	< (less than, <= (less than or equal to, > (greater than), and => (equal to or greater than)
12	= (equal to) , == (equal to), <> (less than or greater than), != (not equal to)
13	NOT
15	AND
16	OR

Chapter 9

Functions

Topics:

Function names are used to apply a function to values, to values at a specified path, or to values derived from a DISTINCT clause. Function names are case-insensitive.

Aggregate Functions

Aggregate functions take multiple values from documents, perform calculations, and return a single value as the result. The function names are case insensitive.

You can only use aggregate functions in SELECT, LETTING, HAVING, and ORDER BY clauses. When using an aggregate function in a query, the query operates as an aggregate query.

Aggregate functions take one argument, the value over which to compute the aggregate function. The COUNT function can also take a wildcard (*) or a path with a wildcard (path.*) as its argument.

If there is no input row for the group, COUNT functions return 0. All other aggregate functions return NULL.

ARRAY_AGG(expression)

Returns array of the non-MISSING values in the group, including NULLs.

ARRAY_AGG(DISTINCT expression)

Returns array of the distinct non-MISSING values in the group, including NULLs.

AVG(expression)

Returns arithmetic mean (average) of all the number values in the group.

AVG(DISTINCT expression)

Returns arithmetic mean (average) of all the distinct number values in the group.

COUNT(*)

Returns count of all the input rows for the group, regardless of value.

COUNT(expression)

Returns count of all the non-NULL and non-MISSING values in the group.

COUNT(DISTINCT expression)

Returns count of all the distinct non-NULL and non-MISSING values in the group.

MAX(expression)

Returns the maximum non-NULL, non-MISSING value in the group in N1QL collation order.

MIN(expression)

Returns the minimum non-NULL, non-MISSING value in the group in N1QL collation order.

SUM(expression)

Returns sum of all the number values in the group.

SUM(DISTINCT expression)

Returns arithmetic sum of all the distinct number values in the group.

Array Functions

You can use array functions to evaluate arrays, perform computations on elements in an array, and to return a new array based on a transformation.

ARRAY_APPEND(expression, value)

Returns new array with value appended.

ARRAY_AVG(expression)

Returns arithmetic mean (average) of all the non-NULL number values in the array, or NULL if there are no such values.

ARRAY_CONCAT(expression1, expression2)

Returns new array with the concatenation of the input arrays.

ARRAY_CONTAINS(expression, value)

Returns true if the array contains value.

ARRAY_COUNT(expression)

Returns count of all the non-NULL values in the array, or zero if there are no such values.

ARRAY_DISTINCT(expression)

Returns new array with distinct elements of input array.

ARRAY_IFNULL(expression)

Returns the first non-NULL value in the array, or NULL.

ARRAY_LENGTH(expression)

Returns the number of elements in the array.

ARRAY_MAX(expression)

Returns the largest non-NULL, non-MISSING array element, in N1QL collation order.

ARRAY_MIN(expression)

Returns smallest non-NULL, non-MISSING array element, in N1QL collation order.

ARRAY_POSITION(expression, value)

Returns the first position of value within the array, or -1. Array position is zero-based, i.e. the first position is 0.

ARRAY_PREPEND(value, expression)

Returns new array with value pre-pended.

ARRAY_PUT(expression, value)

Returns new array with value appended, if value is not already present, otherwise returns the unmodified input array.

ARRAY_RANGE(start, end [, step])

Returns new array of numbers, from start until the largest number less than end. Successive numbers are incremented by step. If step is omitted, the default is 1. If step is negative, decrements until the smallest number greater than end.

ARRAY_REMOVE(expression, value)

Returns new array with all occurrences of value removed.

ARRAY_REPEAT(value, n)

Returns new array with value repeated n times.

ARRAY_REPLACE(expression, value1, value2 [, n])

Returns new array with all occurrences of value1 replaced with value2. If n is given, at most n replacements are performed.

ARRAY_REVERSE(expression)

Returns new array with all elements in reverse order.

ARRAY_SORT(expression)

Returns new array with elements sorted in N1QL collation order.

ARRAY_SUM(expression)

Sum of all the non-NULL number values in the array, or zero if there are no such values.

Comparison Functions

Comparison functions determine the greatest or least value from a set of values.

GREATEST(expression1, expression2, ...)

Largest non-NULL, non-MISSING value if the values are of the same type; otherwise NULL.

LEAST(expression1, expression2, ...)

Returns smallest non-NULL, non-MISSING value if the values are of the same type, otherwise returns NULL.

Conditional Functions for Unknowns

Conditional functions evaluate expressions to determine if the values and formulas meet the specified condition.

IFMISSING(expression1, expression2, ...)

Returns the first non-MISSING value.

IFMISSINGORNULL(expression1, expression2, ...)

Returns first non-NULL, non-MISSING value.

IFNULL(expression1, expression2, ...)

Returns first non-NULL value. Note that this function might return MISSING if there is no non-NULL value.

MISSINGIF(expression1, expression2)

Returns MISSING if expression1 = expression2, otherwise returns expression1. Returns MISSING or NULL if either input is MISSING or NULL.

NULLIF(expression1, expression2)

Returns NULL if expression1 = expression2, otherwise returns expression1. Returns MISSING or NULL if either input is MISSING or NULL.

Conditional Functions for Numbers

Conditional functions evaluate expressions to determine if the values and formulas meet the specified condition.

IFINF(expression1, expression2, ...)

Returns first non-MISSING, non-Inf number. Returns MISSING or NULL if a non-number input is encountered first.

IFNAN(expression1, expression2, ...)

Returns first non-MISSING, non-NaN number. Returns MISSING or NULL if a non-number input is encountered first.

IFNANORINF(expression1, expression2, ...)

Returns first non-MISSING, non-Inf, or non-NaN number. Returns MISSING or NULL if a non-number input is encountered first.

NANIF(expression1, expression2)

Returns NaN if expression1 = expression2, otherwise returns expression1. Returns MISSING or NULL if either input is MISSING or NULL.

NEGINFIF(expression1, expression2)

Returns NegInf if expression1 = expression2, otherwise returns expression1. Returns MISSING or NULL if either input is MISSING or NULL.

POSINFIF(expression1, expression2)

Returns PosInf if expression1 = expression2, otherwise returns expression1. Returns MISSING or NULL if either input is MISSING or NULL.

Date Functions

Date functions return the system clock value or manipulate the date string.

CLOCK_MILLIS()

Returns system clock at function evaluation time, as UNIX milliseconds. Varies during a query.

CLOCK_STR([fmt])

Returns system clock at function evaluation time, as a string in a supported format. Varies during a query. Supported formats:

- "2006-01-02T15:04:05.999Z07:00": Default format. (ISO8601 / RFC3339)
- "2006-01-02T15:04:05Z07:00" (ISO8601 / RFC3339)
- "2006-01-02T15:04:05.999"
- "2006-01-02T15:04:05"
- "2006-01-02 15:04:05.999Z07:00"
- "2006-01-02 15:04:05Z07:00"
- "2006-01-02 15:04:05.999"
- "2006-01-02 15:04:05"
- "2006-01-02"
- "15:04:05.999Z07:00"
- "15:04:05Z07:00"
- "15:04:05.999"
- "15:04:05"

DATE_ADD_MILLIS(expression, n, part)

Performs Date arithmetic, and returns result of computation. *n* and *part* are used to define an interval or duration, which is then added (or subtracted) to the UNIX timestamp, returning the result. Parts:

- "millenium"
- "century"
- "decade"
- "year"
- "quarter"
- "month"
- "week"
- "day"
- "hour"
- "minute"
- "second"
- "millisecond"

DATE_ADD_STR(expression, n, part)

Performs Date arithmetic. *n* and *part* are used to define an interval or duration, which is then added (or subtracted) to the date string in a supported format, returning the result.

DATE_DIFF_MILLIS(expression1, expression2, part)

Performs Date arithmetic. Returns the elapsed time between two UNIX timestamps as an integer whose unit is *part*.

- "millenium"
- "century"
- "decade"
- "year"
- "week"
- "day"
- "hour"
- "minute"
- "second"
- "millisecond"

DATE_DIFF_STR(expression1, expression2, part)

Performs Date arithmetic. Returns the elapsed time between two date strings in a supported format, as an integer whose unit is *part*.

DATE_PART_MILLIS(expression, part)

Returns Date part as an integer. The date expression is a number representing UNIX milliseconds, and part is one of the following date part strings.

- "millenium"
- "century"
- "decade": Floor(year / 10)
- "year"
- "quarter": Valid values: 1 to 4.
- "month": Valid values: 1 to 12.
- "day": Valid values: 1 to 31.
- "hour": Valid values: 0 to 23.
- "minute": Valid values: 0 to 59.
- "second": Valid values: 0 to 59.
- "millisecond": Valid values: 0 to 999.
- "week": Valid values: 1 to 53; ceil(day_of_year / 7.0)
- "day_of_year", "doy": Valid values: 1 to 366.
- "day_of_week", "dow": Valid values: 0 to 6.
- "iso_week": Valid values: 1 to 53. Use with "iso_year".
- "iso_year": Use with "iso_week".
- "iso_dow":> Valid values: 1 to 7.
- "timezone": Offset from UTC in seconds.
- "timezone_hour": Hour component of timezone offset.
- "timezone_minute": Minute component of timezone offset. Valid values: 0 to 59.

DATE_PART_STR(expression, part)

Returns Date part as an integer. The date expression is a string in a supported format, and part is one of the supported date part strings.

DATE_TRUNC_MILLIS(expression, part)

Returns UNIX timestamp that has been truncated so that the given date part string is the least significant.

DATE_TRUNC_STR(expression, part)

Returns ISO 8601 timestamp that has been truncated so that the given date part string is the least significant.

MILLIS(expression), STR_TO_MILLIS(expression)

Returns date that has been converted in a supported format to UNIX milliseconds.

MILLIS_TO_STR(expression [, fmt])

Returns the string in the supported format to which the UNIX milliseconds has been converted.

MILLIS_TO_UTC(expression [, fmt])

Returns the UTC string to which the UNIX timestamp has been converted in the supported format.

MILLIS_TO_ZONE_NAME(expression, tz_name [, fmt])

Converts the UNIX timestamp to a string in the named time zone, and returns the string.

NOW_MILLIS()

Returns statement timestamp as UNIX milliseconds; does not vary during a query.

NOW_STR([fmt])

Returns statement timestamp as a string in a supported format; does not vary during a query.

STR_TO_MILLIS(expression), MILLIS(expression)

Converts date in a supported format to UNIX milliseconds.

STR_TO_UTC(expression)

Converts the ISO 8601 timestamp to UTC.

STR_TO_ZONE_NAME(expression, tz_name)

Converts the supported timestamp string to the named time zone.

Example

The following query example retrieves purchase information for an e-commerce report. The report lists unique customers that purchased something on the company website in the last month. This information is used to identify user activity and growth.

```
SELECT distinct count(purchases.customerId)
  FROM purchases
  WHERE STR_TO_MILLIS(purchases.purchasedAt)
    BETWEEN STR_TO_MILLIS("2014-02-01") AND STR_TO_MILLIS("2014-03-01")
```

Returns:

```
{
   "resultset": [
     {
       "$1": 3831
     }
   ]
 }
```

JSON Functions

DECODE_JSON(expression)

Unmarshals the JSON-encoded string into a N1QL value. The empty string is MISSING.

ENCODE_JSON(expression)

Marshals the N1QL value into a JSON-encoded string. MISSING becomes the empty string.

ENCODED_SIZE(expression)

Number of bytes in an uncompressed JSON encoding of the value. The exact size is implementation-dependent. Always returns an integer, and never MISSING or NULL. Returns 0 for MISSING.

POLY_LENGTH(expression)

Returns length of the value after evaluating the expression. The exact meaning of length depends on the type of the value:

- MISSING: MISSING
- NULL: NULL
- String: The length of the string.
- Array: The number of elements in the array.
- Object: The number of name/value pairs in the object
- Any other value: NULL

Meta and UUID Functions

Meta functions retrieve information about the document or item as well as perform Base64 encoding.

BASE64(expression)

Returns Base64-encoding of *expression*.

META(expression)

Returns Meta data for the document *expression*.

UUID()

Returns a version 4 Universally Unique Identifier(UUID)

Number Functions

Number functions are functions that are performed on a numeric field.

ABS(expression)

Returns absolute value of the number.

ACOS(expression)

Returns arccosine in radians.

ASIN(expression)

Returns arcsine in radians.

ATAN(expression)

Returns arctangent in radians.

ATAN2(expression1, expression2)

Returns arctangent of expression2/expression1.

CEIL(expression)

Returns smallest integer not less than the number.

COS(expression)

Returns cosine.

DEGREES(expression)

Returns radians to degrees.

E()

Base of natural logarithms.

EXP(expression)

Returns $e^{expression}$.

LN(expression)

Returns log base e.

LOG(expression)

Returns log base 10.

FLOOR(expression)

Largest integer not greater than the number.

PI()

Returns PI.

POWER(expression1, expression2):

Returns $expression1^{expression2}$.

RADIANS(expression)

Returns degrees to radians.

RANDOM([expression])

Returns pseudo-random number with optional seed.

ROUND(expression [, digits])

Rounds the value to the given number of integer digits to the right of the decimal point (left if digits is negative). Digits is 0 if not given.

SIGN(expression)

Valid values: -1, 0, or 1 for negative, zero, or positive numbers respectively.

SIN(expression)

Returns sine.

SQRT(expression)

Returns square root.

TAN(expression)

Returns tangent.

TRUNC(expression [, digits])

Truncates the number to the given number of integer digits to the right of the decimal point (left if digits is negative). Digits is 0 if not given.

Example

```
SELECT
    AVG(reviews.rating) / 5 AS normalizedRating,
    ROUND((avg(reviews.rating) / 5), 2) AS roundedRating,
    TRUNC((avg(reviews.rating) / 5), 3) AS truncRating
    FROM reviews AS reviews
    WHERE reviews.customerId = "customer62"
```

Returns

```
{
   "resultset": [
     {
       "normalizedRating": 0.42000000000000004,
       "roundedRating": 0.42,
       "truncRating": 0.42
     }
   ]
 }
```

Object Functions

OBJECT_LENGTH(expression)

Returns number of name-value pairs in the object.

OBJECT_NAMES(expression)

Returns array containing the attribute names of the object, in N1QL collation order.

OBJECT_PAIRS(expression)

Returns array containing the attribute name and value pairs of the object, in N1QL collation order of the names.

OBJECT_VALUES(expression)

Returns array containing the attribute values of the object, in N1QL collation order of the corresponding names.

Pattern-Matching Functions

Pattern-matching functions allow you to work determine if strings contain a regular expression pattern. You can also find the first position of a regular expression pattern and replace a regular expression with another.

REGEXP_CONTAINS(expression, pattern)

Returns True if the string value contains the regular expression pattern.

REGEXP_LIKE(expression, pattern)

Returns True if the string value matches the regular expression pattern.

REGEXP_POSITION(expression, pattern)

Returns first position of the regular expression pattern within the string, or -1.

REGEXP_REPLACE(expression, pattern, repl [, n])

Returns new string with occurrences of pattern replaced with *repl*. If n is given, at most n replacements are performed.

String Functions

String functions perform operations on a string input value and returns a string or other value.

CONTAINS(expression, substring)

True if the string contains the substring.

INITCAP(expression), TITLE(expression)

Converts the string so that the first letter of each word is uppercase and every other letter is lowercase.

LENGTH(expression)

Returns length of the string value.

LOWER(expression)

Returns lowercase of the string value.

LTRIM(expression [, characters])

Returns string with all leading chars removed. Whitespace by default.

POSITION(expression, substring)

Returns the first position of the substring within the string, or -1. The position is zero-based, i.e. the first position is 0.

REPEAT(expression, n)

Returns string formed by repeating expression n times.

REPLACE(expression, substring, repl [, n])

Returns string with all occurrences of substr replaced with repl. If n is given, at most n replacements are performed.

RTRIM(expression, [, characters])

Returns string with all trailing chars removed (whitespace by default).

SPLIT(expression [, sep])

Splits the string into an array of substrings separated by sep. If sep is not given, any combination of whitespace characters is used.

SUBSTR(expression, position [, length])

Returns substring from the integer position of the given length, or to the end of the string. The position is zero-based, i.e. the first position is 0. If position is negative, it is counted from the end of the string; -1 is the last position in the string.

TITLE(expression), INITCAP(expression)

Converts the string so that the first letter of each word is uppercase and every other letter is lowercase.

TRIM(expression [, characters])

Returns string with all leading and trailing chars removed. Whitespace by default.

UPPER(expression)

Returns uppercase of the string value.

Example

The following example shows the use of a LOWER string function.

```
SELECT product
 FROM product
   UNNEST product.categories as categories
```

```
        WHERE LOWER(categories) = "appliances"
```

Type Functions

Type functions perform operations that check or convert expressions.

Type-Checking Functions

ISARRAY(expression)

Returns True if expression is an array, otherwise returns MISSING, NULL or false.

ISATOM(expression)

Returns True if expression is a Boolean, number, or string, otherwise returns MISSING, NULL or false.

ISBOOLEAN(expression)

Returns True if expression is a Boolean, otherwise returns MISSING, NULL or false.

ISNUMBER(expression)

Returns True if expression is a number, otherwise returns MISSING, NULL or false.

ISOBJECT(expression)

Returns True if expression is an object, otherwise returns MISSING, NULL or false.

ISSTRING(expression)

Returns True if expression is a string, otherwise returns MISSING, NULL or false.

TYPE(expression)

Returns one of the following strings, based on the value of expression:

- "missing"
- "null"
- "boolean"
- "number"
- "string"
- "array"
- "object"
- "binary"

Type-Conversion Functions

TOARRAY(expression)

Returns array as follows:

- MISSING is MISSING.
- NULL is NULL.
- Arrays are themselves.
- All other values are wrapped in an array.

TOATOM(expression)

Returns atomic value as follows:

- MISSING is MISSING.
- NULL is NULL.

- Arrays of length 1 are the result of TOATOM() on their single element.
- Objects of length 1 are the result of TOATOM() on their single value.
- Booleans, numbers, and strings are themselves.
- All other values are NULL.

TOBOOLEAN(expression)

Returns Boolean as follows:

- MISSING is MISSING.
- NULL is NULL.
- False is false.
- Numbers +0, -0, and NaN are false.
- Empty strings, arrays, and objects are false.
- All other values are true.

TONUMBER(expression)

Returns number as follows:

- MISSING is MISSING.
- NULL is NULL.
- False is 0.
- True is 1.
- Numbers are themselves.
- Strings that parse as numbers are those numbers.
- All other values are NULL.

TOOBJECT(expression)

Returns object as follows:

- MISSING is MISSING.
- NULL is NULL.
- Objects are themselves.
- All other values are the empty object.

TOSTRING(expression)

Returns string as follows:

- MISSING is MISSING.
- NULL is NULL.
- False is "false".
- True is "true".
- Numbers are their string representation.
- Strings are themselves.
- All other values are NULL.

Chapter 10

Subqueries

A subquery is an expression that is evaluated by executing an inner SELECT query. When an expression is contained in a N1QL statement, the expression is evaluated once for every input document to the statement. In the case of a subquery expression, the inner SELECT is executed once for every input document to the outer statement. A subquery expression returns an array every time it is evaluated; the array contains the results of the inner SELECT query. A subquery can refer to variables in the outer statement.

A subquery is just another expression, so it can appear in most places where an expression can appear; typically, a SELECT list or a WHERE clause. Subqueries are mostly used in SELECT, UPDATE and DELETE statements. Many queries that use subqueries can be made to run faster by rewriting them to use JOINs instead of subqueries.

In N1QL, if a subquery has a FROM clause, then USE KEYS is mandatory for the primary keyspace of the subquery.

Here's an example of a subquery breakdown. Let's say you want to find all purchases in April 2014 where at least one product costs over $500. You'd break this down into two queries.

- What products were purchased in April, 2014?
- Of those products, what products cost more than $500?

Each query builds on the results of the previous one. The first query finds the products purchased in April, 2014. The second query is a subquery. It takes the result set from the first query, and finds the products that cost more than $500.

The final result set contains those products purchased in April, 2014 with a price over $500.

Example:

List all purchases in April 2014 having at least one product costing over $500.

```
SELECT purchases.purchaseId, l.product
FROM purchases UNNEST purchases.lineItems l
WHERE
DATE_PART_STR(purchases.purchasedAt,"month") = 4
     AND
DATE_PART_STR(purchases.purchasedAt,"year") =
2014
     AND EXISTS (SELECT product.productId
FROM product USE KEYS l.product WHERE
product.unitPrice > 500);
```

Chapter 11

Boolean Logic

Topics:

Some clauses, such as WHERE, WHEN, and HAVING, require values to be interpreted as Booleans. In these cases, the following rules apply:

- MISSING, NULL, and false are false
- numbers +0, -0, and NaN are false
- empty strings, arrays, and objects are false
- all other values are true

Four-valued Logic

In N1QL, Boolean propositions can evaluate to NULL or MISSING (as well as to True and False). The following table describes how these values relate to the logical operators:

A	B	A and B	A or B
FALSE	FALSE	FALSE	FALSE
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE
NULL	FALSE	NULL	
TRUE	NULL	NULL	TRUE
NULL	NULL	NULL	NULL
FALSE	MISSING	FALSE	MISSING
TRUE	MISSING	MISSING	TRUE
NULL	MISSING	MISSING	NULL
MISSING	MISSING	MISSING	MISSING

A	not A
FALSE	TRUE
TRUE	FALSE
NULL	NULL
MISSING	MISSING

Comparing NULL and Missing Values

Operator	Non-NULL Value	NULL	MISSING
IS NULL	FALSE	TRUE	MISSING
IS NOT NULL	TRUE	FALSE	MISSING
IS MISSING	FALSE	FALSE	TRUE
IS NOT MISSING	TRUE	TRUE	FALSE
IS VALUED	TRUE	FALSE	FALSE
IS NOT VALUED	FALSE	TRUE	TRUE

Chapter 12

Statements

Topics:

N1QL statements fall into three categories: data definition (DDL), data manipulation (DML), and queries. Data definition statements let you create, modify, and delete indexes. Data manipulation statements let you delete, insert, update, upsert, and merge data into JSON documents. Queries let you retrieve data.

CREATE INDEX

The CREATE INDEX statement allows you to create a view index on the specified fields. It allows you to optimize the data access path for queries. All index names are unique per keyspace, and case sensitive matching is used to match the field names and paths.

create-index:

```
CREATE INDEX index-name ON named-keyspace-ref ( expression [, expression]* )
  [where-clause] [using]
```

index-name:

```
identifier
```

Specifies the name of the index to create.

where-clause:

```
WHERE condition
```

using:

```
USING VIEW
```

Specifies the type of index to be created (in this case, view index, which is the only type of index currently available.

The example shows how to create a view index on the attribute “firstname.” This index can be used to speed up queries like the following: `SELECT * from contacts WHERE firstname LIKE “%marty%”;`

Note : Namespace in all queries is optional for "default" namespace.

Example

```
CREATE INDEX firstnameIndex ON contacts (firstname)
```

CREATE PRIMARY INDEX

The CREATE PRIMARY INDEX statement allows you to create a primary index. Index names are unique per keyspace, and case sensitive matching is used to match the field names and paths.

create-primary-index :

```
CREATE PRIMARY INDEX ON named-keyspace-ref
```

named-keyspace-ref:

```
 [ namespace-name :]  keyspace-name
```

keyspace-name:

```
identifier
```

Specifies the keyspace as source for which the index needs to be created. You can add an optional namespace name to the keyspace name in this way:

```
namespace-name : keyspace-name
```

For example, `main:customer` indicates the customer keyspace in the main namespace. If the namespace name is omitted, the default namespace in the current session is used.

Example

This example creates a primary index on the contacts keyspace. You can prefix the keyspace-name with the namespace-name, for example, `people:contacts`.

```
CREATE PRIMARY INDEX ON contacts
```

DELETE [Experimental]

[This feature is experimental; it's not for use in production.]

DELETE immediately removes the specified document from your keyspace.

delete:

```
DELETE FROM keyspace-ref [use-keys-clause] [where-clause]
    [limit-clause] [returning-clause]
```

keyspace-ref:

```
[ (namespace-name :)] keyspace-name
```

use-keys-clause:

```
USE [PRIMARY] KEYS expression
```

where-clause:

```
WHERE condition
```

limit-clause:

```
LIMIT expression
```

returning-clause:

```
RETURNING (result-expression [, result-expression]* | [RAW | ELEMENT ]
 expression)
```

*keyspace-ref:*Specifies the data source from which to delete the document. You can add an optional namespace name to the keyspace name in this way:

```
namespace-name:keyspace-name
```

For example, main:customer indicates the customer keyspace in the main namespace. If the namespace name is omitted, the default namespace in the current session is used.

*use-keys-clause:*Specifies the keys of the data items to be deleted. Optional. Keys can be any expression.

*where-clause:*Specifies the condition that needs to be met for data to be deleted. Optional.

*limit-clause:*Specifies the greatest number of objects that can be deleted. This clause must have a non-negative integer as its upper bound. Optional.

*returning-clause:*Returns the data you deleted as specified in the result_expression.

Examples:

The following statement deletes product10.

```
DELETE FROM product p USE KEYS "product10" RETURNING p

"results": [
    {
        "p": {
            "categories": [
                "Luggage"
            ],
            "color": "sky blue",
            "dateAdded": "2014-05-06T15:52:18Z",
            "dateModified": "2014-05-06T15:52:18Z",
            "description": "This product is available on
                \u003ca target=\"_blank\"
                href=\"http://www.amazon.com/gp/product/
                B005HNKFSM/ref=s9_hps_bw_g198_ir011?pf_rd_m=ATVPDKIKX0DER\
                u0026pf_rd_s=merchandised-search-5\u0026pf_
                rd_r=D182EDFE2F434403B401\u0026pf_rd_t=101\
                u0026pf_rd_p=1486061902\u0026pf_rd_i=15743161
                \"\u003eAmazon.com\u003c/a\u003e.",
            "imageURL": "http://ecx.images-amazon.com/
                images/I/51KiHy-Y-2L._SY220_.jpg",
            "name": "Briggs \
                u0026 Riley Luggage Executive Clamshell Backpack",
            "productId": "product10",
            "reviewList": [
                "review47",
                "review873",
                "review1224",
                "review2203",
                "review2242",
                "review6162",
                "review6825",
                "review7300",
                "review9934"
            ],
            "type": "product",
            "unitPrice": 231.2
        }
    }
]
```

The following statement deletes the product that is priced at 5.25.

```
DELETE FROM product p WHERE p.unitPrice = 5.25 RETURNING p.productId

"results": [
        {
            "productId": "product99"
        }
    ]
```

```

```

DROP INDEX

The DROP INDEX statement allows you to drop an Index, that was created using the CREATE INDEX statement. Uses the dot character (.) to specify the keyspace-ref on which the index named index-name was created. All index names are unique per keyspace, and case sensitive matching is used to match the field names and paths.

drop-index:

```
DROP INDEX named-keyspace-ref . index-name
```

index-name:

```
identifier
```

Specifies the name of the index to drop.

Example

The following statement drops the index, `contact_name`.

```
DROP INDEX contacts.contact_name
```

DROP PRIMARY INDEX

The DROP PRIMARY INDEX statement allows you to drop the primary index created on the keyspace specified by named-keyspace-ref.

drop-primary-index:

```
DROP PRIMARY INDEX ON named-keyspace-ref
```

named-keyspace-ref:

```
[ namespace-name : ]  keyspace-name
```

keyspace-name:

```
identifier
```

Specifies the keyspace from which the index needs to be dropped. You can add an optional namespace-name to the keyspace-name in this way:

```
namespace-name : keyspace-name.
```

For example, main:customer indicates the customer keyspace in the main namespace. If the namespace is omitted, the default namespace in the current session is used.

Example:

This example drops the primary index on the keyspace. You can prefix the keyspace-name with the namespace-name, for example, `people:contacts`.

```
DROP PRIMARY INDEX ON contacts
```

EXPLAIN

The EXPLAIN statement can be used before any N1QL statement to obtain the information about how N1QL executed the statement.

Example:

```

EXPLAIN SELECT title, genre, runtime FROM catalog.details ORDER BY title

"results": [
   {
     "#operator": "Sequence",
     "~children": [
         {
            "#operator": "Sequence",
            "~children": [
                {
                   "#operator": "PrimaryScan",
                   "index": "#primary",
                   "keyspace": "catalog",
                   "namespace": "default"
                },
                {
                   "#operator": "Parallel",
                   "~child": {
                        "#operator": "Sequence",
                        "~children": [
                             {
                                "#operator": "Fetch",
                                "keyspace": "catalog",
                                "namespace": "default",
                                "projection": "`details`"
                             },
                             {
                                "#operator": "InitialProject",
                                "result_terms": [
                                    {
                                      "expr": "(`details`.`title`)"
                                    },
                                    {
                                      "expr": "(`details`.`genre`)"
                                    },
                                    {
                                       "expr": "(`details`.`runtime`)"
                                    }
                                    ]
                             }
                       ]
                }
                        }
                    ]
                },
                {
                     "#operator": "Order",
                     "sort_terms": [
                         {
                             "expr": "(`details`.`title`)"
                         }
                     ]
                 },
                 {
                     "#operator": "Parallel",
                     "~child": {
                         "#operator": "FinalProject"
                     }
         }
      ]
    }
]
```

INSERT [Experimental]

[This feature is experimental; it's not for use in production.]

Inserts new records into a keyspace .

You can create a document by using the INSERT INTO. Performing an insert succeeds only if no document with that name exists. If the document exists, the INSERT fails.

insert:

```
INSERT INTO keyspace-ref [insert-values | insert-select] [returning-clause]
```

keyspace-ref:

```
[(namespace-name :)]  keyspace [ [AS] alias]
```

keyspace:

```
identifier
```

insert-values:

```
 [ (  [ PRIMARY ] KEY , VALUE ) ]  values-clause [, values-clause]*
```

values-clause:

```
VALUES ( expression , expression )
```

insert-select

```
( [PRIMARY] KEY expression [ , VALUE expression ] ) select
```

returning-clause:

```
RETURNING (result-expression [, result-expression]* | [RAW | ELEMENT ]
 expression)
```

result-expression:

```
[path . ]  * | expression [ [ AS ] alias ]
```

path:

```
identifier [ [ expression ] ] [ . path ]
```

keyspace-ref: Specifies the keyspace into which to insert the document.

You can add an optional namespace-name to the keyspace-name in this way:

namespace-name:keyspace-name.

For example, `main:customer` indicates the customer keyspace in the main namespace. If the namespace name is omitted, the default namespace in the current session is used.

insert-values: Specifies the values to be inserted.

*insert-select*Specifies the values to be inserted as a SELECT statement.

*returning-clause:*Returns the data you inserted as specified in the result_expression.

Example:

This statement inserts the value Odwalla for the key ID into the products document.

```
INSERT INTO product (KEY, VALUE) VALUES ("odwalla-juice1”,
      { "productId": "odwalla-juice1”, "unitPrice": 4.40, "type": "product",
 "color":"green"}) RETURNING *

"results": [
        {
            "color": "green",
            "productId": "odwalla-juice1”,
            "type": "product",
            "unitPrice": 4.4
        }
    ]
```

SELECT

SELECT statements let you retrieve data from specified keyspaces. A simple query in N1QL has three parts to it:

- SELECT - Parts of the document to return in an array.
- FROM - The kespace, or datastore with which to work.
- WHERE - Conditions the data must satisfy to be retrieved.

Only a SELECT clause is required in a query.

Because the data is stored in documents rather than rigidly structured tables, queries can return a collection of different document structures or fragments. The one requirement is that the retrieved data will match the conditions in the WHERE clause, if one is provided.

Exercising the full power of the SELECT statement is more complex. With the SELECT statement, you can retrieve any data from a keyspace. Once retrieved you can specify how you want the data returned. You can also manipulate the data in the result set, and, using Data Manipulation statements, you can modify the data in the actual keyspace.

SELECT Statement Processing

The SELECT statement queries a keyspace. A JSON array is returned that contains zero or more result objects. SELECT behaves as a sequence of steps in a process. Each step in the process produces result objects that are then used as inputs in the next step until all steps in the process are complete. The possible elements and operations in a query include:

- **Specifying the Keyspace:** You specify the keyspace that is queried. This is the from-path parameter in a FROM clause. You can also provide a path as keyspace.
- **Filtering the Results:** You can filter result objects from the SELECT by specifying a condition in the WHERE clause.
- **Generating a Result Set:** You can generate a set of result objects with GROUP BY or HAVING clauses along with a result expression list, result-expression-list.
- **Removing Duplicates:** You can remove duplicate result objects from the result set by using the DISTINCT clause.
- **Ordering Items in the Result Set:** You can sort items in the order specified by the ORDER BY expression list.
- **Skipping Results:** You can skip the first *n* items as specified by the OFFSET clause
- **Limiting the Number of Results:** You can specify the maximum number of items returned with the LIMIT clause.

SELECT Statement Syntax

select:

```
subselect [ set-op ([ALL] )  subselect]* [order-by-clause] [limit-clause]
 [offset-clause]
```

set-op:

```
UNION | INTERSECT | EXCEPT
```

subselect:

```
select-from | from-select
```

select-from:

```
select-clause [from-clause] [let-clause] [where-clause] [group-by-clause]
```

from-select:

```
from-clause [let-clause] [where-clause] [group-by-clause] select-clause
```

SELECT Clause

select-clause:

```
SELECT ( [ALL | DISTINCT]  ( result-expression [, result-expression]*) |
    (RAW | ELEMENT) expression)
```

result-expression:

```
[(path .)] * | expression [ ([AS] alias) ]
```

path:

```
identifier [[ expression ]]  [. path]
```

alias:

```
identifier
```

FROM clause

from-clause:

```
FROM from-term
```

from-term:

```
from-path [([AS] alias)] [use-keys-clause] | from-term join-clause | from-
term nest-clause | from-term unnest-clause
```

from-path:

```
[(namespace :)] path
```

namespace:

```
identifier
```

use-keys-clause:

```
USE [PRIMARY] KEYS expression
```

join-clause:

```
[join-type] JOIN from-path [( [AS] alias) ] on-keys-clause
```

join-type:

```
INNER | LEFT [OUTER]
```

on-keys-clause:

```
ON [PRIMARY] KEYS expression
```

nest-clause:

```
[join-type] NEST from-path [ ( [AS] alias) ] on-keys-clause
```

unnest-clause:

```
[join-type] [UNNEST | FLATTEN] expression [ ( [AS] alias) ]
```

LET clause

```
LET (alias = expression) [, (alias = expression)]*
```

WHERE clause

where-clause:

```
WHERE condition
```

cond:

```
expression
```

GROUP BY clause

group-by-clause:

```
GROUP BY expression [ , expression ]* [ letting-clause ]
    [having-clause] ] | [letting-clause]
```

letting-clause:

```
LETTING  alias = expression [(, alias = expression) ]*
```

having-clause:

```
HAVING condition
```

ORDER BY clause

order-by-clause:

```
ORDER BY ordering-term [, ordering-term]*
```

ordering-term:

```
expression [ASC | DESC]
```

LIMIT clause

```
LIMIT expression
```

OFFSET clause

```
OFFSET expression
```

SELECT/Subselect

SELECT statements can begin with either SELECT or FROM. The behavior is the same in either case.

subselect:

```
select-from | from-select
```

select-from:

```
select-clause [from-clause] [let-clause] [where-clause]  [group-by-clause]
```

from-select:

```
from-clause [let-clause] [where-clause] [group-by-clause] select-clause
```

SELECT Clause

select-clause:

```
SELECT ( [ALL | DISTINCT]  ( result-expression [, result-expression]*) |
    (RAW | ELEMENT) expression)
```

result-expression:

```
[(path .)] * | expression [ ([AS] alias) ]
```

path:

```
identifier [ [ expression ] ] [. path]
```

alias:

```
identifier
```

ALL

SELECT ALL retrieves all of the data specified. ALL will display "all" of the specified columns including all of the duplicates. The ALL keyword is the default if nothing is specified.

DISTINCT

The DISTINCT clause removes duplicate result objects from the query's result set. If the DISTINCT clause is not used, the query returns all objects that meet the query conditions in a result set, which might include duplicates.

Example

```
SELECT DISTINCT orderlines[0].productId FROM orders
```

Results

```

 {
  "resultset": [
    {
      "productId": "coffee01"
    },
    {
      "productId": "tea111"
    }
  ]
}
```

If the query had run with no DISTINCT clause against orders with this SELECT statement, the result set would have been as follows:

```
SELECT  orderlines[0].productId
      FROM orders
```

Returns

```
 {
  "resultset": [
    {
      "productId": "coffee01"
    },
    {
      "productId": "coffee01"
    },
    {
      "productId": "tea111"
    },
    {
      "productId": "coffee01"
    }
  ]
}
```

RAW/ELEMENT

Specifies a raw expression.

FROM Clause

The FROM clause defines the keyspaces and input objects for the query.

This is an optional clause for your query. If this clause is omitted, the input for the query is a single empty object. You can perform calculations with the SELECT statement if the FROM clause is omitted.

Every FROM clause specifies one or more keyspaces. The first keyspace is called the primary keyspace.

```
FROM from-term
```

where *from-term* has the following syntax:

```
from-path [([AS] alias)] [use-keys-clause] | from-term join-clause | from-
term nest-clause | from-term unnest-clause
```

from-path:

```
[namespace :] path
```

namespace:

```
identifier
```

use-keys-clause:

```
USE [PRIMARY] KEYS expression
```

join-clause:

```
[join-type] JOIN from-path [[AS] alias ] on-keys-clause
```

join-type:

```
INNER | LEFT [OUTER]
```

on-keys-clause:

```
ON [PRIMARY] KEYS expression
```

nest-clause:

```
[join-type] NEST from-path [ ( [AS] alias) ] on-keys-clause
```

unnest-clause:

```
[join-type] [UNNEST | FLATTEN] expression [ ( [AS] alias) ]
```

Omitted FROM clause

If the FROM clause is omitted, the data source is equivalent to an array containing a single empty object. This allows you to evaluate expressions that do not depend on stored data.

Evaluating an expression:

```
SELECT 10 + 20
```

produces the following result:

```
[ { "$1" : 30 } ]
```

Counting the number of inputs:

```
SELECT COUNT(*) AS input_count
```

produces the following result:

```
[ { "input_count" : 1 } ]
```

Getting the input contents:

```
SELECT *
```

produces the following result:

```
[ { } ]
```

Keyspaces (buckets)

The simplest type of FROM clause specifies a keyspace (bucket):

```
SELECT * FROM customer
```

This returns every value in the customer keyspace.

The keyspace can be prefixed with an optional namespace (pool):

```
SELECT * FROM main:customer
```

This queries the customer keyspace in the main namespace.

If the namespace is omitted, the default namespace in the current session is used.

USE KEYS

Specific primary keys within a keyspace (bucket) can be specified. Only values having those primary keys will be included as inputs to the query.

To specify a single key:

```
SELECT * FROM customer USE KEYS "acme-uuid-1234-5678"
```

To specify multiple keys:

```
SELECT * FROM customer USE KEYS [ "acme-uuid-1234-5678", "roadster-
uuid-4321-8765" ]
```

In the FROM clause of a subquery, USE KEYS is mandatory for the primary keyspace.

Nested Paths

Nested paths within keyspaces can be specified using the period [.] as a level designation.

For each document in the keyspace, the path is evaluated and its value becomes an input to the query. For a given document, if any element of the path is NULL or MISSING, that document is skipped and does not contribute any inputs to the query.

For example, if some customer documents contain a primary_contact object, the following query can retrieve them:

```
SELECT * FROM customer.primary_contact
```

Here is the result set:

```
[
{"name" : "John Smith", "phone" : "+1-650-555-1234",
      “address" : { ... } },
{"name" : "Jane Brown", "phone" : "+1-650-555-5678",
      "address" : { ... } }
]
```

Nested paths can have arbitrary depth and can include array subscripts. For example, the following query:

```
SELECT * FROM customer.primary_contact.address
```

returns this result set:

```
[
  { "street" : "101 Main St.", "zip" : "94040" },
  { "street" : "3500 Wilshire Blvd.", "zip" : "90210" }
]
```

Joins

Joins allow you to create new input objects by combining two or more source objects. The joins-clause is optional, and follows the FROM clause; it allows you to combine two or more source objects to use as input objects. The KEYS clause is required after each JOIN. It specifies the primary keys for the second keyspace in the join.

Here is the syntax for the JOIN clause:

```
[ join-type ] JOIN from-path [ [ AS ] alias ] keys-clause
```

where `join-type [ LEFT ] is [ INNER | OUTER ]` and from-path is as discussed in the “from-path” section.

Joins can be chained. By default, an INNER join is performed. This means that for each joined object produced, both the left- and right-hand source objects must be non-missing and non-null.

If LEFT or LEFT OUTER is specified, then a left outer join is performed. At least one joined object is produced for each left-hand source object. If the right-hand source object is NULL or MISSING, then the joined object's right-hand side value is also NULL or MISSING (omitted), respectively.

The KEYS clause is required after each JOIN. It specifies the primary keys for the second keyspace in the join.

For example, if our customer objects were:

```

  {
     "name": ...,
     "primary_contact": ...,
     "address": [ ... ]
   }
```

And our invoice objects were:

```

   {
      "customer_key": ...,
      "invoice_date": ...,
      "invoice_item_keys": [ ... ],
      "total": ...
    }
```

And the FROM clause was:

```
FROM invoice inv JOIN customer cust ON KEYS inv.customer_key
```

Then each joined object would be:

```

    {
        "inv" : {
            "customer_key": ...,
            "invoice_date": ...,
            "invoice_item_keys": [ ... ],
            "total": ...
        },
        "cust" : {
            "name": ...,
            "primary_contact": ...,
            "address": [ ... ]
        }
    }
```

If our invoice_item objects were:

```
    {
        "invoice_key": ...,
        "product_key": ...,
        "unit_price": ...,
        "quantity": ...,
        "item_subtotal": ...
    }
```

And the FROM clause was:

```
FROM invoice JOIN invoice_item item ON KEYS invoice.invoice_item_keys
```

Then our joined objects would be:

```
    {
        "invoice" : {
            "customer_key": ...,
            "invoice_date": ...,
            "invoice_item_keys": [ ... ],
            "total": ...
        },
        "item" : {
            "invoice_key": ...,
            "product_key": ...,
            "unit_price": ...,
            "quantity": ...,
            "item_subtotal": ...
        }
    },
    {
        "invoice" : {
            "customer_key": ...,
            "invoice_date": ...,
            "invoice_item_keys": [ ... ],
            "total": ...
        },
        "item" : {
            "invoice_key": ...,
            "product_key": ...,
            "unit_price": ...,
            "quantity": ...,
            "item_subtotal": ...
        }
    },
    ...
```

```
```

ON KEYS is required after each JOIN. It specifies the primary keys for the second keyspace (bucket) in the join.

Joins can be chained.

By default, an INNER join is performed. This means that for each joined object produced, both the left and right hand source objects must be non-missing and non-null.

If LEFT or LEFT OUTER is specified, then a left outer join is performed. At least one joined object is produced for each left hand source object. If the right hand source object is NULL or MISSING, then the joined object's right-hand side value is also NULL or MISSING (omitted), respectively.

Unnests

If a document or object contains a nested array, UNNEST conceptually performs a join of the nested array with its parent object. Each resulting joined object becomes an input to the query. Unnests can be chained.

Here is the syntax for an UNNEST join:

```
[ join-type ] UNNEST path [ [ AS ] alias ]
```

where join-type is `[ INNER | [ LEFT ] OUTER ]`

The first path element after each UNNEST must reference some preceding path.

By default, an INNER unnest is performed. This means that for each result object produced, both the left-hand and right-hand source objects must be non-missing and non-null.

If LEFT or LEFT OUTER is specified, then a left outer unnest is performed. At least one result object is produced for each left source object. If the right-hand source object is NULL, MISSING, empty, or a non-array value, then the result object's right side value is MISSING (omitted).

Example

If some customer documents contain an array of addresses under the address field, the following query retrieves each nested address along with the parent customer's name.

```
SELECT c.name, a.* FROM customer c UNNEST c.address a
```

Here is the result set:

```
 [
    { "name" : "Acme Inc.", "street" : "101 Main St.",
        "zip" : "94040" },
    { "name" : "Acme Inc.", "street" : "300 Broadway",
 "zip" : "10011" },
    { "name" : "Roadster Corp.", "street" : "3500 Wilshire Blvd.",
        "zip" : "90210" },
    { "name" : "Roadster Corp.", "street" : "4120 Alamo Dr.",
 "zip" : "75019" }
]
```

In the following example, The UNNEST clause iterates over the reviews array and collects the reviewerName and publication from each element in the array. This collection of objects can be used as input for other query operations.

```
SELECT review.reviewerName, review.publication
   FROM beers AS b
      UNNEST review IN b.reviews
```

Here is the result set:

```
 {"id": "7983345",
 "name": "Takayama Pale Ale",
 "brewer": "Hida Takayama Brewing Corp.",
 "reviews" : [
   {"reviewerName" : "Takeshi Kitano",
         "publication" : "Outdoor Japan Magazine","date": "3/2013"},
   {"reviewerName" : "Moto Ohtake", "publication" : "Japan Beer Times",
         "date" : "7/2013"}
            ]
}
```

Nests

```
[ join-type ] NEST from-path [ [ AS ] alias ] keys-clause
```

where join-type is `[ INNER | [ LEFT ] OUTER ]`

Nesting is conceptually the inverse of unnesting. Nesting performs a join across two keyspaces. But instead of producing a cross-product of the left and right inputs, a single result is produced for each left input, while the corresponding right inputs are collected into an array and nested as a single array-valued field in the result object.

Nests can be chained with other nests, joins, and unnests. By default, an INNER nest is performed. This means that for each result object produced, both the left and right source objects must be non-missing and non-null. The right-hand side result of NEST is always an array or MISSING. If there is no matching right source object, then the right source object is as follows:

- If the ON KEYS expression evaluates to MISSING, the right value is also MISSING.
- If the ON KEYS expression evaluates to NULL, the right value is MISSING.
- If the ON KEYS expression evaluates to an array, the right value is an empty array.
- If the ON KEYS expression evaluates to a non-array value, the right value is an empty array.

If LEFT or LEFT OUTER is specified, then a left outer nest is performed. One result object is produced for each left source object.

Example

This example shows the NEST clause using invoice and invoice_item_ objects.

Recall our invoice objects:

```
{
     "customer_key": ...,
     "invoice_date": ...,
     "invoice_item_keys": [ ... ],
     "total": ...
 }
```

And our invoice_item objects:

```
{
         "invoice_key": ...,
         "product_key": ...,
         "unit_price": ...,
         "quantity": ...,
         "item_subtotal": ...
}
```

If the FROM clause was:

```
FROM invoice inv NEST invoice_item items ON KEYS inv.invoice_item_keys
```

The results would be:

```
{
       "invoice" : {
           "customer_key": ...,
           "invoice_date": ...,
           "invoice_item_keys": [ ... ],
           "total": ...
       },
       "items" : [
           {
               "invoice_key": ...,
               "product_key": ...,
               "unit_price": ...,
               "quantity": ...,
               "item_subtotal": ...
           },
           {
               "invoice_key": ...,
               "product_key": ...,
               "unit_price": ...,
               "quantity": ...,
               "item_subtotal": ...
           }
       ]
   },
   {
       "invoice" : {
           "customer_key": ...,
           "invoice_date": ...,
           "invoice_item_keys": [ ... ],
           "total": ...
       },
       "items" : [
           {
               "invoice_key": ...,
               "product_key": ...,
               "unit_price": ...,
               "quantity": ...,
               "item_subtotal": ...
           },
           {
               "invoice_key": ...,
               "product_key": ...,
               "unit_price": ...,
               "quantity": ...,
               "item_subtotal": ...
           }
       ]
   },
   ...
```

Arrays

If an array occurs along a path, the array can be subscripted to select one element.

Example

For each customer, the entire address array is selected. The following statement

```
SELECT a FROM customer.address a
```

produces the following result:

```
    [
       {
          "a": [
                 { "street" : "101 Main St.", "zip" : "94040" },
                 { "street" : "300 Broadway", "zip" : "10011" }
               ]
         },
         {
          "a": [
                 { "street" : "3500 Wilshire Blvd.", "zip" : "90210" },
                 { "street" : "4120 Alamo Dr.", "zip" : "75019" }
               ]
         }
   ]
```

Subscripting Example

For each customer, the first element of the address array is selected. The following statement

```
SELECT * FROM customer.address[0]
```

produces the following result:

```
     [
         { "street" : "101 Main St.", "zip" : "94040" },
         { "street" : "3500 Wilshire Blvd.", "zip" : "90210" }
     ]
```

AS Keyword

Like SQL, N1QL allows renaming fields using the AS keyword. However, N1QL also allows reshaping the data, which has no analog in SQL. To do this, you embed the attributes of the statement in the desired result object shape.

Aliases

Aliases in the FROM clause create new names that can be referred to anywhere in the query. When an alias conflicts with a keyspace or field name in the same scope, the identifier always refers to the alias. This allows for consistent behavior in scenarios where an identifier only conflicts in some documents.

LET Clause

```
LET (alias = expression) [, (alias = expression)]*
```

LET stores the result of a sub-expression in order to use it in subsequent clauses.

Aliases in the LET clause create new names that can be referred to anywhere in the query.

WHERE Clause

where-clause:

```
WHERE condition
```

condition:

```
expression
```

The WHERE clause allows you to filter result objects based on a condition. The WHERE condition is evaluated for each input object. Only objects evaluating to TRUE are retained.

Example

```
SELECT * FROM tutorial WHERE fname = 'Ian'
```

Returns:

```

   {
   "resultset": [
{
       "age": 56,
       "children": [
         {
           "age": 17,
           "fname": "Abama",
           "gender": "m"
         },
         {
           "age": 21,
           "fname": "Bebama",
           "gender": "m"
         }
       ],
       "email": "ian@gmail.com",
       "fname": "Ian",
       "hobbies": [
         "golf",
         "surfing"
       ],
       "lname": "Taylor",
       "relation": "cousin",
       "title": "Mr.",
       "type": "contact"
     }
   ]
 }

```

GROUP BY Clause

group-by-clause:

```
GROUP BY expression [ , expression ]* [ letting-clause ]
    [having-clause] ] | [letting-clause]
```

letting-clause:

```
LETTING  alias = expression [(, alias = expression) ]*
```

having-clause:

```
HAVING condition
```

GROUP BY allows you to organize a result set by one or more expressions. It can be used with aggregate functions to group the result set. The optional LETTING and HAVING clauses follow the GROUP BY clause.

Example

```
SELECT relation, COUNT(*) AS count
    FROM tutorial
        GROUP BY relation
```

Results

```
{
  "resultset": [
    {
      "count": 2,
      "relation": "friend"
    },
    {
      "count": 1,
      "relation": "coworker"
    },
    {
      "count": 1,
      "relation": "parent"
    },
    {
      "count": 2,
      "relation": "cousin"
    }
  ]
}
```

Example

```
SELECT title, type, COUNT(*) AS count FROM catalog GROUP BY type
```

LETTING Clause

```
LETTING  alias = expression [(, alias = expression) ]*
```

Aliases in the LETTING clause create new names that can be referred to in the HAVING, SELECT, and ORDER BY clauses.

HAVING Clause

```
HAVING condition
```

The HAVING clause optionally follows the GROUP BY clause. It filters result objects from the GROUP BY clause using the given condition.

UNION, INTERSECT, and EXCEPT

UNION, INTERSECT, and EXCEPT combine results from multiple subselects.

UNION returns values from the first and second subselects.

INTERSECT returns values that are present in both the first and second subselects.

EXCEPT returns values from the first subselect that are absent from the second subselect.

UNION, INTERSECT, and EXCEPT return distinct results, such that there are no duplicates.

UNION ALL, INTERSECT ALL, and EXCEPT ALL return all applicable values, including duplicates. These queries are faster, because they do not compute distinct results.

ORDER BY Clause

order-by-clause:

```
ORDER BY ordering-term [, ordering-term]*
```

ordering-term:

```
expression [ASC | DESC]
```

The ORDER BY clause lets your order the result set based on an expression. The order of items in the result set is determined by the expression specified in this clause. Objects are sorted first by the left-most expression in the list of expressions. Any items with the same sort value will be sorted with the next expression in the list. This process repeats until all items are sorted and all expressions in the list are evaluated. You can specify whether to order the results in ascending or descending order.

If no ORDER BY clause is specified, the order in which the result objects are returned is undefined.

As ORDER BY expressions can evaluate to any JSON value, we define an ordering when comparing values of different types. The following list describes the order by type (from lowest to highest):

- missing value
- null
- false
- true
- number
- string (string comparison is done using a raw byte collation of UTF8 encoded strings)
- array (element by element comparison is performed until the end of the shorter array; if all the elements so far are equal, then the longer array sorts after)
- object (larger objects sort after; for objects of equal length, key/value by key/value comparison is performed; keys are examined in sorted order using the normal ordering for strings)

The following example shows the ORDER BY clause.

```
SELECT fname, age
      FROM tutorial
          ORDER BY age
```

Returns:

```
 {
   "resultset": [
     {
       "age": 18,
       "fname": "Fred"
     },
     {
       "age": 20,
       "fname": "Harry"
     },
     {
       "age": 40,
       "fname": "Jane"
     },
     {
       "age": 46,
       "fname": "Dave"
     }
   ]
 }
```

LIMIT Clause

```
LIMIT expression
```

The LIMIT clause specifies the maximum number of objects that can be returned in a result set by SELECT. This clause must have a non-negative integer as its upper bound.

Name and age list: limit by 2

```
SELECT fname, age
     FROM tutorial
          ORDER BY age
              LIMIT 2
```

Returns:

```

 {
   "resultset": [
     {
       "age": 18,
       "fname": "Fred"
     },
     {
       "age": 20,
       "fname": "Harry"
     }
   ]
 }
```

OFFSET Clause

```
OFFSET expression
```

The OFFSET clause specifies a number of objects to be skipped. If a LIMIT clause is also present, the OFFSET is applied prior to the LIMIT. The OFFSET value must be a non-negative integer.

The OFFSET clause optionally follows the LIMIT clause. If an offset is specified, the specified number of objects is omitted from the result set before enforcing a specified LIMIT. This clause must be a non-negative integer.

Examples

The following examples show the LIMIT and OFFSET clauses.

Name and age list: limit and offset by 2

```
SELECT fname, age
     FROM tutorial
          ORDER BY age
              LIMIT 2 OFFSET 2
```

Returns:

```

{
   "resultset": [
     {
       "age": 40,
       "fname": "Jane"
     },
     {
       "age": 46,
       "fname": "Dave"
     }
   ]
 }
```

Golf products list: limit and offset by 10

```
SELECT *
 FROM product
      UNNEST product.categories AS cat
         WHERE LOWER(cat) IN ["golf"] LIMIT 10 OFFSET 10
```

UPDATE [Experimental]

[This feature is experimental; it's not for use in production.]

UPDATE replaces a document that already exists with updated values.

update:

```
UPDATE keyspace-ref [use-keys-clause] [set-clause] [unset-clause] [where-
clause] [limit-clause] [returning-clause]
```

set-clause:

```
SET path = expression [update-for] [ , path = expression [update-for] ]*
```

update-for:

```
FOR variable (IN | WITHIN) path  (, variable (IN | WITHIN) path)* [WHEN
 condition ] END
```

unset-clause:

```
UNSET path [update-for] (, path [ update-for ])*
```

keyspace-ref: Specifies the keyspace for which to update the document.

You can add an optional namespace-name to the keyspace-name in this way:

namespace-name:keyspace-name.

*use-keys-clause:*Specifies the keys of the data items to be updated. Optional. Keys can be any expression.

*set-clause:*Specifies the value for an attribute to be changed.

unset-clause: Removes the specified attribute from the document.

update-for: The update for clause uses the FOR statement to iterate over a nested array and SET or UNSET the given attribute for every matching element in the array.

*where-clause:*Specifies the condition that needs to be met for data to be updated. Optional.

*limit-clause:*Specifies the greatest number of objects that can be updated. This clause must have a non-negative integer as its upper bound. Optional.

*returning-clause:*Returns the data you updated as specified in the result_expression.

Example:

The following statement changes the "type" of the product, "odwalla-juice1" to "product-juice".

```
UPDATE product USE KEYS "odwalla-juice1” SET type = "product-juice"
 RETURNING product.type

"results": [
        {
            "type": "product-juice"
        }
    ]
```

This statement removes the "type" attribute from the "product" keyspace for the document with the "odwalla-juice1" key.

```
UPDATE product USE KEYS "odwalla-juice1" UNSET type RETURNING product.*

"results": [
        {
            "productId": "odwalla-juice1",
            "unitPrice": 5.4
        }
    ]
```

This statement unsets the "gender" attribute in the "children" array for the document with the key, "dave" in the "tutorial" keyspace.

```
UPDATE tutorial t USE KEYS "dave" UNSET c.gender FOR c IN children END
 RETURNING t

"results": [
        {
            "t": {
                "age": 46,
                "children": [
                    {
                        "age": 17,
                        "fname": "Aiden"
                    },
                    {
                        "age": 2,
                        "fname": "Bill"
                    }
                ],
                "email": "dave@gmail.com",
                "fname": "Dave",
                "hobbies": [
                    "golf",
                    "surfing"
                ],
                "lname": "Smith",
                "relation": "friend",
                "title": "Mr.",
                "type": "contact"
            }
        }
    ]
```

UPSERT [Experimental]

[This feature is experimental; it's not for use in production.]

Used to insert a new record or update an existing one. If the document doesn’t exist it will be created. UPSERT is a combination of INSERT and UPDATE.

upsert:

```
UPSERT INTO keyspace-ref [insert-values | insert-select]  [ returning-
clause]
```

keyspace-ref: Specifies the keyspace into which to upsert the document.

You can add an optional namespace-name to the keyspace-name in this way:

namespace-name:keyspace-name.

For example, `main:customer` indicates the customer keyspace in the main namespace. If the namespace name is omitted, the default namespace in the current session is used.

insert-values: Specifies the values to be upserted.

*insert-select*Specifies the values to be upserted as a SELECT statement.

*returning-clause:*Returns the data you upserted as specified in the result_expression.

Example:

The following statement upserts values for odwalla-juice1 into the product document.

```
UPSERT INTO product (KEY, VALUE) VALUES ("odwalla-juice1", { "productId":
 "odwalla-juice1",
      "unitPrice": 5.40, "type": "product", "color":"red"}) RETURNING * ;

"results": [
        {
            "color": "red",
            "productId": "odwalla-juice1",
            "type": "product",
            "unitPrice": 5.4
        }
    ]
```

Chapter

13

Programs

Topics:

Query Server

The Query Server is the N1QL query engine. The Query Server needs to run whenever you use N1QL or any of its programs, such as the Query Shell (CBQ) or, if you need to run the online Query Tutorial locally.

Query Shell

The Query Shell is a command-line interactive interface that allows you to run queries and display the query's results. You can run queries against any data stores you choose.

Query Tutorial

The Query Tutorial is an interactive online tutorial that gets you up and querying quickly. It guides you through some simple queries that give you an introduction to N1QL's power and simplicity. You can use the Query Tutorial to learn about N1QL basics.

Chapter

14

REST API

Topics:

N1QL provides a REST API for submitting N1QL queries and statements to a N1QL query engine. Its REST context is `http://<node>:8093/query` and `http://<node>:18093/query` where <*node*> is the hostname or IP address of a computer running the N1QL query engine.

The N1QL REST API query services allows you to execute a N1QL statement.

Execute N1QL

This method enables the client to execute a N1QL statement. It runs synchronously, so once execution of the statement in the request is started, results are streamed back to the client, terminating when execution of the statement finishes.

Request Specification

The following sections describe how to specify requests.

Request Format

Here is the format for GET and POST requests.

GET /query/service

POST /query/service

Request Parameters

This table contains details of all the parameters that can be passed in a request to the /query/service endpoint:

Parameter Name	Value	Description
`statement`	string	Required if `prepared` is not provided. Any valid N1QL statement for a POST request, or a read-only N1QL statement (such as SELECT or EXPLAIN) for a GET request. If both the `prepared` and `statement` parameters are present and non-empty, an error is returned.
`prepared`	string	Required if statement is not provided. The prepared form of the N1QL statement to be executed. If both the prepared and statement parameters are present and non-empty, an error is returned.
`timeout`	string (duration format)	Optional. Maximum time to spend on the request before timing out. The default value is "0s", which means no timeout is applied, that is, the request runs until completion regardless of how long it takes. Its format includes an amount and a unit. Valid units are “ns”, “us”, “ms”, “s”, “m”, “h”, for nanoseconds, microseconds, milliseconds, seconds, minutes and hours, respectively. The unit is mandatory. Examples are “10ms” (which means 10 milliseconds), “1m” (which means 1 minute), and “0.5s” (which means half a second). There is also a server-wide timeout parameter. If both the server-wide timeout parameter and this timeout parameter are specified,

Parameter Name	Value	Description
		the minimum of the two timeouts is applied.
`readonly`	boolean	Optional. Always true for GET requests. By default, false for POST requests. If the server-wide readonly parameter is set to true, its setting supersedes the request readonly parameter.
`metrics`	boolean	Optional. Specifies that metrics should be returned with query results. Default value is true.
`$<`*`identifier`*`>`	json_value	Optional. If the `statement` has one or more *named parameters*, there should be one or more named parameters in the request. A named parameter consists of the $ character followed by an identifier. An identifier is an alphabetical character followed by one or more alphanumeric characters. Named parameters apply to `prepared` also.
`args`	list	Optional. If the statement has one or more positional parameters, this parameter is required in the request. This is a list of JSON values, one for each positional parameter in the statement. Positional parameters apply to `prepared` also.
`signature`	boolean	Optional. Include a header for the results schema in the response. Default value is true.
`creds`	list	Optional. A list of credentials in the form of user/password objects. If credentials are supplied in the request header (that is, using HTTP basic authentication), then `creds` is ignored. See Authentication Parameters section for full specification.
`client_context_id`	string	Optional. A piece of data supplied by the client that is echoed in the response, if present. N1QL makes no assumptions about the meaning of this data and just logs and echoes it. Maximum allowed size is 64 characters. A `client_context_id` longer than 64 characters is cut off at 64 characters.

Named Parameters

If the statement in a request contains named parameters, the request should contain the parameters described in the following table.

Here is an example of a statement containing named parameters:

```
SELECT detail FROM emp WHERE name = $nval AND age > $aval
```

Parameter Name	Value
statement	SELECT detail FROM emp WHERE name = $nval AND age > $aval
$nval	"smith"
$aval	45

There should be a named parameter in the request for each query parameter in the request's statement parameter.

Positional Parameters

If the statement in a request contains positional parameters, the request should contain the parameters described in the following table.

Here is an example of a statement containing positional parameters:

```
SELECT detail FROM emp WHERE name = $1 AND hiredate > $2
```

Parameter Name	Value
statement	SELECT detail FROM emp WHERE name = $1 AND age > $2
args	["smith", 45]

Positional parameters can also be specified in a statement using the question mark (?), so the following statement is an alternative way to specify the same query:

Parameter Name	Value
statement	SELECT detail FROM emp WHERE name = ? AND age > ?
args	["smith", 45]

Request Content Type

For POST requests, you can specify the parameters in the request body in URL-encoded format or JSON format. For GET requests, you specify the parameters in the request URL in URL-encoded format. For URL-encoded parameters, the format is consistent with the syntax for variables according to the RFC 6570.

Response

This section has two subsections: Response HTTP Status Codes and Response Body.

Response HTTP Status Code

Normal Status Code:

200 OK- The request completed with or without errors. Any errors or warnings that occurred during the request will be in the response body.

Possible Error Codes:

400 Bad Request- The request cannot be processed for one of the following reasons:

- There is a N1QL syntax error in the statement.
- There is a missing or unrecognized HTTP parameter in the request.
- The request is badly formatted (e.g. there is a JSON syntax error in request body).

401 Unauthorized- The credentials provided with the request are missing or invalid.

403 Forbidden- There is a read-only violation. Either there was an attempt to create or update in a GET request or a POST request where readonly is set or the client does not have the authorization to modify an object (index, keyspace or namespace) in the statement.

404 Not Found- The statement in the request references an invalid namespace or keyspace.

405 Method Not Allowed- The REST method type in the request is unsupported.

409 Conflict- There is an attempt to create an object (keyspace or index) that already exists.

410 Gone- The server is shutting down gracefully. Previously made requests are being completed, but no new requests are being accepted.

500 Internal Server Error- There was an unforeseen problem processing the request.

503 Service Unavailable- There is an issue (that is possibly temporary) preventing the request being processed; the request queue is full or the data store is not accessible.

Response Body

The response body has the following structure:

```
{
    "requestID": UUID,
    "clientContextID": string,
    "signature":
       {
          *.* |
          ( field_name:    field_type,
          ...
          )
       },

       "results":
       [
          json_value,
          …
       ],

     "errors":
       [
           { "code": int, "msg":  string }, ...
        ],

      "warnings":
       [
           { "code": int, "msg": string }, …
       ],

      "status":  "success",

     "metrics":
      {
           "elapsedTime": string,
           "executionTime": string,
           "resultCount": unsigned int,
           "resultSize": unsigned int,
           "mutationCount": unsigned int,
           "errorCount": unsigned int,
           "warningCount": unsigned int
       }
}
```

Parameter Name	Value	Description
`requestID`	UUID	A unique identifier for the response.
`clientContextID`	string	The clientContextID of the request, if one was supplied (see client_context _id in Request Parameters).
`signature`	object	The schema of the results. Present only when the query completes successfully.
`results`	list	A list of all the objects returned by the query. An object can be any JSON value.
`status`	enum	The status of the request. Possible values are: success, running, errors, completed, stopped, timeout, fatal.
`errors`	list	A list of 0 or more error objects. If an error occurred during processing of the request, it will be represented by an error object in this list.
`error.code`	int	A number that identifies the error.
`error.msg`	string	A message describing the error in detail.
`warnings`	list	A list of 0 or more warning objects. If a warning occurred during processing of the request, it is represented by a warning object in this list.
`warning.code`	int	A number that identifies the warning.
`warning.msg`	string	A message describing the warning in full.
`metrics`	object	An object containing metrics about the request.
`metrics.elapsedTime`	string	The total time taken for the request, that is the time from when the request was received until the results were returned.
`metrics.executionTime`	string	The time taken for the execution of the request, that is the time from

Parameter Name	Value	Description
		when query execution started until the results were returned.
`metrics.resultCount`	unsigned int	The total number of objects in the results.
`metrics.resultSize`	unsigned int	The total number of bytes in the results.
`metrics.mutationCount`	unsigned int	The number of mutations that were made during the request.
`metrics.errorCount`	unsigned int	The number of errors that occurred during the request.
`metrics.warningCount`	unsigned int	The number of warnings that occurred during the request.

Request Error/Warning Format

Errors and warnings have the following format:

```
{
     "code" : int,
     "msg" : string,
     "name": string,
     "sev" : int,
     "temp" : bool
}
```

code: A unique number for the error/warning. The code ranges are partitioned by component. The codes can also include parts that indicate severity and transience. **code** is always present in every condition returned in the Query REST API or captured in a log.

msg:A detailed description of the condition. **msg** is always present in every condition returned in the Query REST API or captured in a log.

The following elements are optional and can be present in a condition returned in the Query REST API or captured in a log. Additional elements not listed here might also be present. Clients and consumers of the REST API or the logs must accommodate any additional elements.

name: Unique name that has a 1:1 mapping to the **code**. Uniquely identifies the condition. **name** is helpful for pattern matching and can have meaning making it more memorable than the code). The name should be fully qualified. Here are some examples:

- `indexing.scan.io_failure`
- `query.execute.index_not_found`

sev: One of the following N1QL severity levels (listed in order of severity):

1. Severe
2. Error
3. Warn
4. Info

temp: Indicates if the condition is transient (e.g. queue is full). If the value is **false**, it tells clients and users that a retry without modification produces the same condition.

Examples

Here are some examples of the REST API.

Successful Request

```
 $ curl -v http://localhost:8093/query/service \
-d "statement=SELECT text FROM tweets LIMIT 1"
< HTTP/1.1 200 OK
{
   "requestID": "11ed1981-7802-4fc2-acd6-dfcd1c05a288",
   "signature": {
       "text": "json"
   },
   "results": [
       {
           "text": "Couchbase is my favorite database"
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "3.455608ms",
       "executionTime": "3.116241ms",
       "resultCount": 1,
     "resultSize": 65,
     "mutationCount": 0,
     "errorCount": 0,
     "warningCount": 0
   }
}
$
```

The same request can be run as a GET request:

```
 $ curl http://localhost:8093/query?statement=SELECT%20text%20from%20tweets
%20limit%201
```

Request with Positional Parameters

```
 $ curl -v http://localhost:8093/query/service \
-d 'statement=SELECT text FROM tweets WHERE rating = $1 AND when >
 $2&args=[ 9.5, "1-1-2014"]'
< HTTP/1.1 200 OK
{
   "requestID": "11ed1984-7802-4fc2-acd6-dfcd1c05a288",
   "signature": {
       "text": "json"
   },
   "results": [
       {
           "text": "Couchbase is my favorite database"
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "3.455608ms",
       "executionTime": "3.116241ms",
       "resultCount": 1,
     "resultSize": 65,
     "mutationCount": 0,
     "errorCount": 0,
     "warningCount": 0
   }
}
$
```

Request with Named Parameters

```
  $ curl -v http://localhost:8093/query/service \
-d 'statement=SELECT text FROM tweets WHERE rating = $r AND when >
 $date&args=[ 9.5, "1-1-2014"]'
$lt; HTTP/1.1 200 OK
{
    "requestID": "11ed1984-7802-4fc2-acd6-dfcd1c05a288",
    "signature": {
        "text": "json"
    },
    "results": [
        {
            "text": "Couchbase is my favorite database"
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "3.455608ms",
        "executionTime": "3.116241ms",
        "resultCount": 1,
      "resultSize": 65,
      "mutationCount": 0,
      "errorCount": 0,
      "warningCount": 0
    }
}
$
```

Request Timeout

The request timed out because it could not be completed in the time given in the request (or the query engine timeout, if one was specified when starting the query engine).

```
  $ curl -v http://localhost:8093/query/service \
-d "statement=SELECT text FROM tweets LIMIT 1&timeout=1ms"
< HTTP/1.1 200 OK
{
    "requestID": "cb7b070d-3faa-4ee3-a9ed-bfa97a965d6b",
    "signature": {
        "text": "json"
    },
    "errors": [
        {
      "code": $lt;int$gt;,
      "msg": "The request resulted in timeout"
    }],
    "status": "timeout",
    "metrics": {
        "elapsedTime": "1.347944ms",
        "executionTime": "1.305518ms",
        "resultCount": 0,
      "resultSize": 0,
      "mutationCount": 0,
      "errorCount": 0,
      "warningCount": 0
    }
}
$
```

Request Error

A request error happens when there is a problem with the REST request itself, e.g. missing a required parameter.

```
$ curl -v http://localhost:8093/query/service
$lt; HTTP/1.1 400 Bad Request
{
   "requestID": "5c0a6a81-5fc8-4a33-a035-ed7fb1512710",
   "errors": [
       {
     "code": <int>,
     "msg": "Either statement or prepared must be provided"
   }],
   "status": "fatal",
   "metrics": {
       "elapsedTime": "134.7944us",
       "executionTime": "130.5518us",
       "resultCount": 0,
     "resultSize": 0,
     "mutationCount": 0,
     "errorCount": 1,
     "warningCount": 0
   }
}
$
```

Service Error

A service error means there is a problem that prevents the request being fulfilled:

```
$ curl -v http://localhost:8093/query/service \
-d "statement=SELECT text FROM tweets LIMIT 1"
$lt; HTTP/1.1 503 Service Unavailable
{
   "requestID": "5c0a6a81-2fc8-4a33-a035-ed7fb1512710",
   "errors": [
{
     "code": <int>,
     "msg": "Request queue full"
   }],
   "status": "errors",
   "metrics": {
       "elapsedTime": "134.7944us",
       "executionTime": "130.5518us",
       "resultCount": 0,
     "resultSize": 0,
     "mutationCount": 0,
     "errorCount": 1,
     "warningCount": 0
   }
}
$
```

N1QL Error

A N1QL error happens when there is an error processing the N1QL statement in a request:

```
$ curl -v http://localhost:8093/query/service \
-d "statement=SLECT text FROM tweets LIMIT 1"
< HTTP/1.1 400 Bad Request
{
    "requestID": "922edd9a-23d7-4053-8d60-91f7cbc22c83",
    "errors": [
      {
      "code": <int>,
        "msg": "Syntax error at token: SLECT"
    }],
    "status": "fatal",
    "metrics": {
        "elapsedTime": "134.7944us",
        "executionTime": "130.5518us",
        "resultCount": 0,
      "resultSize": 0,
      "mutationCount": 0,
      "errorCount": 1,
      "warningCount": 0
    }
}
$
$ curl -v http://localhost:8093/query/service \
-d "statement=SELECT text FROM weets LIMIT 1"
< HTTP/1.1 404 Not Found
{
    "request_id": "2ac080ba-6419-4905-a20b-c881966b6402",
    "errors": [
        {
      "code": <int>,
      "msg": "Keyspace weets does not exist"
    }],
    "status": "fatal",
    "metrics": {
        "elapsedTime": "134.7944us",
        "executionTime": "130.5518us",
        "resultCount": 0,
      "resultSize": 0,
      "mutationCount": 0,
      "errorCount": 1,
      "warningCount": 0
    }
}
```

```
$
```

Unsupported HTTP Method

For a REST method type that is not supported:

```
$ curl -v http://localhost:8093/query/service -X PUT \
-d "statement=SELECT text FROM tweets LIMIT 1"
< HTTP/1.1 405 Method Not Allowed
{
    "requestID": "6e0143ed-4657-4c9d-a184-703c930c7401",

   "errors": [
{
     "code": <int>,
     "msg": "PUT is not supported"
   }],
   "status": "fatal",
   "metrics": {
       "elapsedTime": "134.7944us",
       "executionTime": "130.5518us",
       "resultCount": 0,
     "resultSize": 0,
     "mutationCount": 0,
     "errorCount": 1,
     "warningCount": 0
   }
}
$
```

Request with Authentication - HTTP Header

In this example, the credentials (user="local:tweets", pass="pAss1")are given in the request header using basic authentication.

```
$ curl -v http://localhost:8093/query/service \
-d "statement=SELECT text FROM tweets LIMIT 1" \
-H "Authorization: Basic bG9jYWw6dHdlZXRzOnBBc3Mx"

< HTTP/1.1 200 OK
{
   "requestID": "11ed1981-7802-4fc2-acd6-dfcd1c05a288",
   "signature": {
       "text": "json"
   },
   "results": [
       {
           "text": "Couchbase is my favorite database"
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "3.455608ms",
       "executionTime": "3.116241ms",
       "resultCount": 1,
     "resultSize": 65,
     "mutationCount": 0,
     "errorCount": 0,
     "warningCount": 0
   }
}
$
```

Request with Authentication - Request Parameter

If a request requires more than one set of credentials, the creds parameter must be used, as in this example.

```
  $ curl -v http://localhost:8093/query/service \
      -d 'statement=SELECT t.text FROM tweets t
      JOIN users u KEY t.uid LIMIT 1
      &creds=[{"user": "local:tweets", "pass":"pAss1"}", {"user":
 "local:users", "pass":"pAss2}"]'

< HTTP/1.1 200 OK
{
    "requestID": "11ed1981-7802-4fc2-acd6-dfcd1c05a288",
    "signature": {
        "text": "json"
    },
    "results": [
        {
            "text": "Couchbase is my favorite database"
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "3.455608ms",
        "executionTime": "3.116241ms",
        "resultCount": 1,
      "resultSize": 65,
      "mutationCount": 0,
      "errorCount": 0,
      "warningCount": 0
    }
}
```

Chapter 15

Release Notes

Topics:

This is the fourth developer preview edition of Couchbase Query Engine and Language, released January 2015.

New Features

- Complete query language
- UPDATE, DELETE, INSERT, and UPSERT statements
- Subqueries
- UNION, INTERSECT, and EXCEPT operators
- Regular expressions (regexp) functions
- Array functions
- Many other functions
- Multi-line shell input, terminated by semicolon
- Result signature (schema)
- Named and positional parameters
- Prepared statements
- Complete REST API
- Parallel query execution

Changed Features

- Identifiers with hyphens must now be escaped: beer-sample must be `beer-sample`
- BASE64_VALUE() is changed to BASE64()
- System:sites is changed to system:datastores
- System:pools is changed to system:namespaces
- System:buckets is changed to system:keyspaces
- Syntax change -- :pool-name.bucket-name is changed to namespace:keyspace
- Invocation parameters for cbq-engine are changed

Fixes

Key	Summary
MB-9240	Operator precedence
MB-9259	REST runtime stats
MB-9571	Subqueries
MB-9579	EXISTS operator
MB-9807	Change :pool-name to pool-name:
MB-10100	Add DISTINCT to SUM() and AVG()
MB-10128	Notify user to create primary index
MB-10132	Engine crashes when prepending LIKE wildcard on index
MB-10920	Unable to start tuq if there are no buckets
MB-11050	Provide API for getting internal stats

Key	Summary
MB-11139	Ability to abort a long-running query from client side
MB-11165	TRUNC() should always truncate towards zero
MB-11812	Need a read-only mode to startup the query server
MB-12220	Add unique id generation functions to N1QL
MB-12418	Limit stack trace to the panicking goroutine
MB-12678	Fix output of explain
MB-12743	Unable to start cbq-engine with timeout option
MB-12784	DMLupdate: show mutation count

Known Issues

This release is a Developer Preview and is not meant to be used in production.