MySQL Document Store MySQL Pre-FOSDEM Days Tutorial

--Dave Stokes @Stoker David.Stokes@Oracle.com

Two Hours!

Overview

Traditionally relational databases including MySQL have required a schema to be defined before documents can be stored. This usually requires a Database Administrator (or someone with someone with DBA skills) to do this work before the data store can be used. Other issues keeping users from immediate use of MySQL include indexes and data normalization. In cases where there is imperfect knowledge of the data or the data is rapidly 'evolving' the need to restructure can become a choke point or can lead to bad practices as a development project progresses.

The MySQL Document Store is a schema-less (or schema-flexible) storage system for JSON documents. If you create documents describing products, you do not need to know and define all possible attributes of any products before storing and operating with the documents. This differs from working with a relational database and storing products in a table, when all columns of the table must be known and defined before adding any products to the database.

This tutorial is an introduction to the MySQL Document Store where using MySQL as a NoSQL JSON document store is stressed. And the Document Store also works on traditional relational tables or a combination of both traditional table data and NoSQL JSON documents. This flexibility of the document store model with the power of the relational model and provides you with more opens to handle your data. Please note there is only so much that can be covered in the limited amount of time provide but I hope to give you a broad overview of the capabilities of the MySQL Document Store. --Dave

Introduction

Example 1

So what does a NoSQL MySQL look like? This example uses the new MySQL Shell (also known as mysqlsh). A schema named 'fosdem' has been created previously.

```
 MySQL  localhost:33060+ ssl  fosdem  JS > \use fosdem
Default schema `fosdem` accessible through db.
 MySQL  localhost:33060+ ssl  fosdem  JS > db
<Schema:fosdem>
 MySQL  localhost:33060+ ssl  fosdem  JS > db.createCollection('a')
<Collection:a>
 MySQL  localhost:33060+ ssl  fosdem  JS > db.a
<Collection:a>
 MySQL  localhost:33060+ ssl  fosdem  JS > db.a.add( { foo: "bar" } )
Query OK, 1 item affected (0.0062 sec)
 MySQL  localhost:33060+ ssl  fosdem  JS > db.a.find()
{
    "_id": "00005df0ed910000000000000002",
    "foo": "bar"
}
1 document in set (0.0005 sec)
```

The first command **\use fosdem** sets the current session to use the foddem schema. The server responds with a notice the the fosdem schema can be referenced by an object named db.

If just '**db**' is entered, the system responds by telling us the current schema is named *fosdem*. This is a handy trick for when 'senior DBA moments' occur.

To create a document collection, roughly analogous to create table in the relation world, we can enter **db.createCollection('a')**.

Typing '**db.a**' the system will inform us that it is a Collection.

Now data can be entered -- all without having to set up tables, indexes, views, or other components of the relational world. To enter data type **db.a.add({ foo: "bar" })** at the prompt.

The data that was just entered can be quickly retrieved by typing **db.a.find()** and the server will display the full document. We will ignore the **_id** for about the next thirty minutes but it is easy to see that we entered a JSON document and were able to retrieve that document

without a) Structured Query Language (SQL), b) a relational table, or c) waiting on a DBA to set things up for our use.

Data has a life span and for this collection it is very short. The entire collection can be removed with a simple **db.dropCollection('a')**

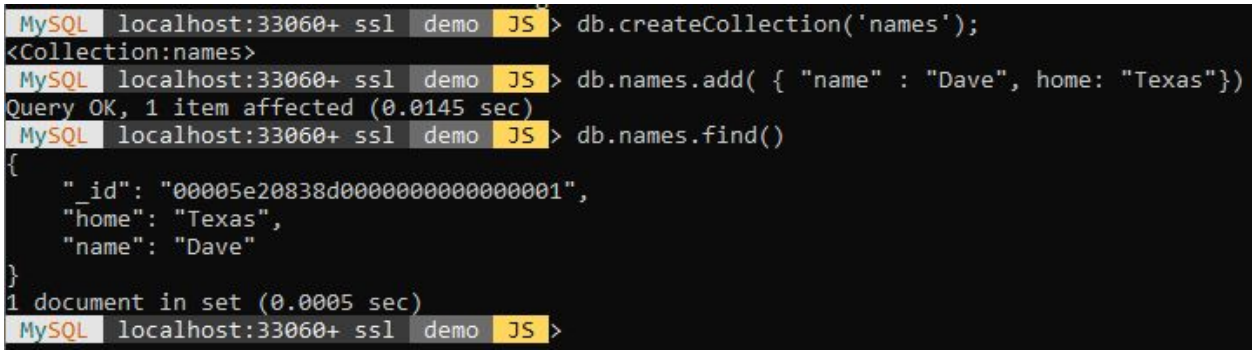
```
db.dropCollection('a')
```

Example 2

Next will create a schema named demo, create a document collection, and populate it with data.

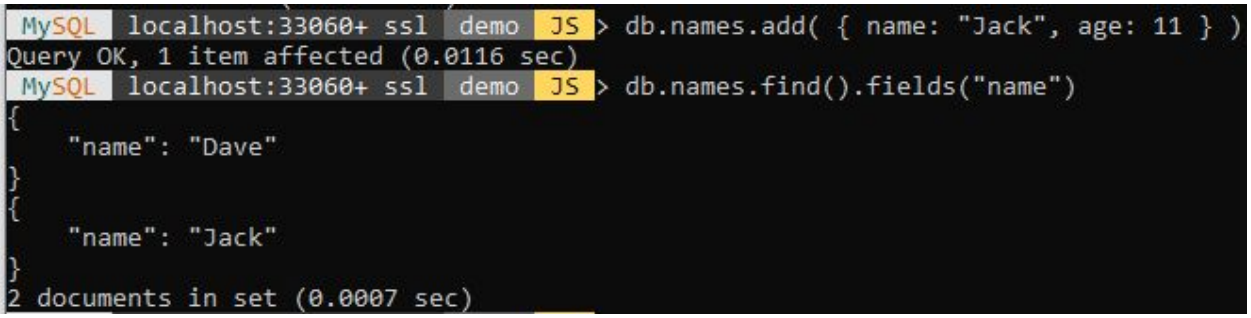
1. Once we login to the server we can create a new schema for our session. Since this is a session level operation, we will issue the command **session.createSchema('demo')**
2. Next **\use demo** is used to connect the demo schema to the db pointer.
3. Entering **db** by itself displays the name the schema in use.

4. Next a document collection is created with **db.createCollection('names')**
5. Data is then added by issuing **db.names.add({ "name": "Dave", home: "Texas" })**
6. Data is reviewed with the command **db.names.find()**



7. Another record is entered with **add()**

8. The data is reviewed with **find()** but this note only the name key/value pair is asked for in the query.



Things to note:

The above examples had *no* Structured Query Language or SQL involved.

No schemas, tables, indexes, views, or other traditional relational databases prerequisites were needed before we started. It was simple to connect to the server, create a schema, create a collection, and start storing data.

The data is free form and mutable. The two documents have different sets of data. The first record had a *name* and a *home* key while the second had a *name* and an *age*. You may have noticed the the first record had the key as “name” while the second one did not have the quotes.

Exercise 1 -- Create a schema named ‘fosdem’, a collection named ‘a’, and add your name and another piece of information. What is needed for quoting keys and what is needed for quoting values? Time - 10 minutes

A Very Quick Explanation of JSON objects and JSON arrays

Objects are bounded by {}, arrays are bounded by [].

You can have arrays in objects, objects in arrays, in any combo you want.

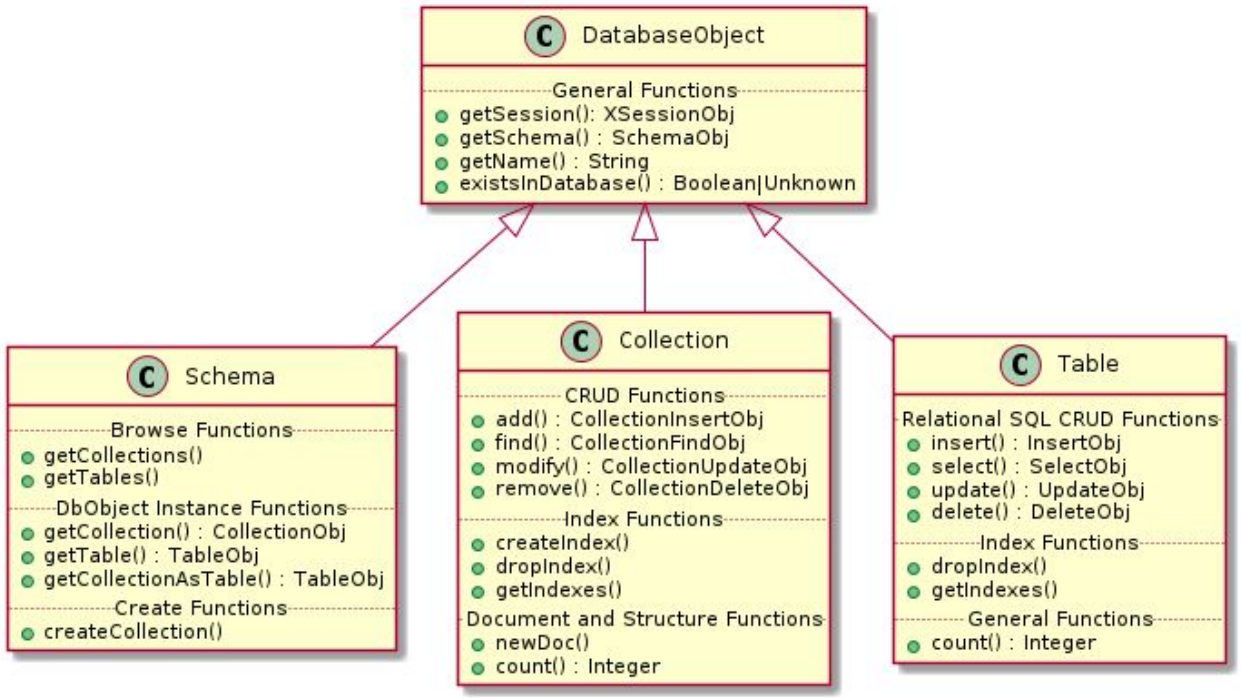
You can embed down as many levels as you want but discretion is advised as they get harder to read as you add more.

The MySQL JSON data type will hold roughly 1GB per column.

See https://www.json.org/json-en.html for more details on JSON.

SYNTAX, Tax, and Sin

The MySQL Document Store allows developers to work with SQL relational tables and schema-less JSON collections. To make that possible MySQL has created the X Protocol and the X Dev API which puts a strong focus on CRUD by providing a fluent API allowing you to work with JSON documents in a natural way. The X Protocol is a highly extensible and is optimized for CRUD as well as SQL API operations.The X DevAPI is implemented by MySQL Shell and MySQL Connectors that support X Protocol.



The X DevAPI wraps several very powerful concepts in a simple API.

- A new high-level session concept enables you to write code that can transparently scale from single MySQL Server to a multiple server environment with InnoDB Cluster
- Read operations are simple and easy to understand.
- Non-blocking, asynchronous calls follow common host language patterns.

The X DevAPI introduces a new, modern and easy-to-learn way to work with your data.

- Documents are stored in Collections and have their dedicated CRUD operation set.

- Work with your existing domain objects or generate code based on structure definitions for strictly typed languages.
- Focus is put on working with data via CRUD operations.
- Modern practices and syntax styles are used to get away from traditional SQL-String-Building.

An X DevAPI session is a high-level database session concept that is different from working with traditional low-level MySQL connections. Sessions can encapsulate one or more actual MySQL connections when using the X Protocol. Use of this higher abstraction level decouples the physical MySQL setup from the application code. Sessions provide full support of X DevAPI and limited support of SQL.

Isn't SQL Good Enough??

Structured Query Language or SQL has several major benefits as well as several major problems. First among these problems is that SQL is a 'descriptive' language which causes impedance problems with embedded strings of SQL in procedural or object oriented programming language. Besides looking ugly, SQL strings embedded in other languages are often hard to comprehend, harder to modify, and subject to all sorts of programming sins (SQL Injection vulnerabilities et cetera).

Novice programmers often have difficulties modifying existing queries as they do not have enough training with SQL and end up fighting the syntax. Combined with a lack of understanding of relational theory this creates an awful lot of signal to noise on sites like Stackoverflow.com

Some will decide to use an object relational mapper (ORM) which ends up making many poor decisions on the behalf of the programmer, adds more complexity to their stack, and injects a point of failure all while slowing the entire application's performance.

X DevAPI Connection

The code that is needed to connect to a MySQL document store looks a lot like the traditional MySQL connection code, but now applications can establish logical sessions to MySQL server instances running the X Plugin. Sessions are produced by the mysqlx factory, and the returned Sessions can encapsulate access to one or more MySQL server instances running X Plugin. Applications that use Session objects by default can be deployed on both single server setups and database clusters with no code changes. Create a Session using the `mysqlx.getSession(connection)` method. You pass in the connection parameters to

connect to the MySQL server, such as the hostname, user and so on, very much like the code in one of the classic APIs.

The connection parameters can be specified as either a URI type string, for example `user:@localhost:33060`, or as a data dictionary, for example {`user: myuser, password: mypassword, host: example.com, port: 33060`}.

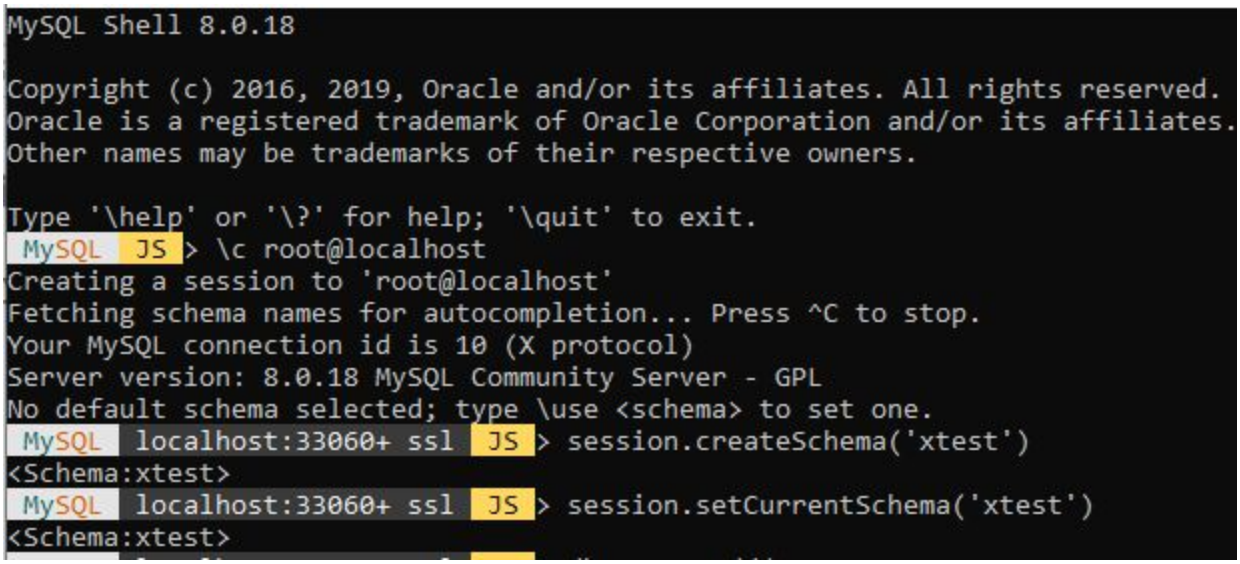
The MySQL user account used for the connection should use either the `mysql_native_password` or `caching_sha2_password` authentication plugin. The server you are connecting to should have encrypted connections enabled, the default in MySQL 8.0. This ensures that the client uses the X Protocol PLAIN password mechanism which works with user accounts that use either of the authentication plugins. If you try to connect to a server instance which does not have encrypted connections enabled, for user accounts that use the `mysql_native_password` plugin authentication is attempted using `MYSQL41` first, and for user accounts that use `caching_sha2_password` authentication falls back to `SHA256_MEMORY`.

The X DevAPI listens on port ***33060*** instead of the older protocols ***3306***. MySQL 8.0 has the X DevAPI enabled by default and it is optional on MySQL 5.7.

Example 3

Creating a schema and a collection from scratch is very simple. Please follow along on your own system if you are able to do so.

The New Shell (mysqlsh) is used to authenticate into the server. A new schema named xtest is created at the session level with the command **session.createSchema('xtest')**



However the **\use xtest** command or **session.getSchema('xtest')** command must be used to point the db object to the schema.

So lets add a new collection named '**a**'. And a new record is added. Unlike the previous example, the data is not all on one line. Note that a value is specified for the *_id* field so that the system will not automatically generate a value for that field.

```
 MySQL  localhost:33060+ ssl  JS > \use xtest
Default schema `xtest` accessible through db.
 MySQL  localhost:33060+ ssl  xtest  JS > db
<Schema:xtest>
 MySQL  localhost:33060+ ssl  xtest  JS > db.createCollection('a')
<Collection:a>
 MySQL  localhost:33060+ ssl  xtest  JS > db.a.add(
                                      -> {
                                      ->  "foo"  : "bar",
                                      ->  _id    : 12345,
                                      ->  a_ray  : [1,3,5,7]
                                      -> }
                                      -> )
                                      ->
Query OK, 1 item affected (0.0078 sec)
```

Next we can select the record just added.

```
 MySQL  localhost:33060+ ssl  xtest  JS > db.a.find("_id = 12345").fields('foo')
{
    "foo": "bar"
}
1 document in set (0.0007 sec)
```

We are able to specify the exact _id value we want and the desired field of data with **db.a.find("_id = 12345").fields('foo')** -- note the stacking of the **fields()** function on the **find()** function. As you will see shortly it is easy to stack other functions like sort() or having() having without having to fight SQL syntax.

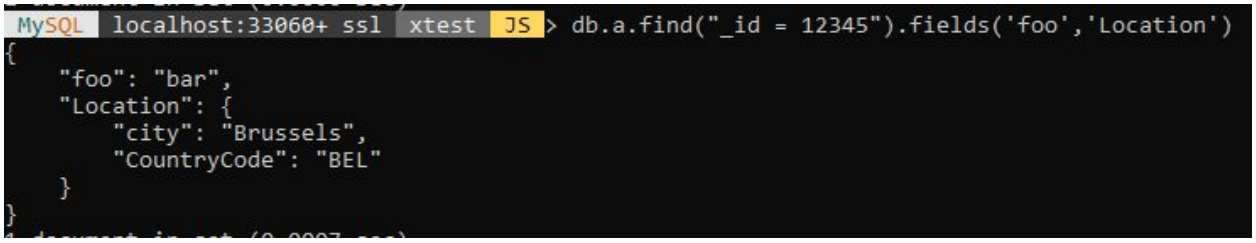
Modify Data

Data is rarely static and the ability to update is a necessity. Adding additional information to a document is straightforward.

```
 MySQL  localhost:33060+ ssl  xtest  JS > db.a.modify("_id = 12345").set('Location:', { city: 'Brussels', CountryCode: 'BEL'})
Query OK, 1 item affected (0.0046 sec)

Rows matched: 1  Changed: 1  Warnings: 0
```

We use **_id = 12345** as an argument to specify the exact record to modify. And just like SQL, if you do not specify a particular record then ALL records will get modified. The command **db.a.modify("_id = 12345").set('Location:', { city: 'Brussels', CountryCode: 'BEL'})** will set up a new key/value pair with the key named *Location* and the value of *{ city: 'Brussels', CountryCode: 'BEL'}.*



The **modify()** arguments are *set, unset, arrayInsert, arrayAppend, arrayDelete,* and *patch*.

Exercise 2

Create a schema named *demox*, create a document collection in that schema named *a*, and create a document with your name and postal code as data. Then modify the document to add another key/value pair with *shoeSize* as the key and your shoe size as the data.

Group Exercise

Does **db.a.find("_id = 12345").fields('foo',"Location")** work with mixed single and double quotes? What if you run **db.a.modify("_id = 12345").set('location:', { city: 'Brussels', CountryCode: 'BEL'})** -- do you get two key/values or is it case insensitive?

CRUD Operations Overview

CRUD operations are available as methods, which operate on Schema objects. The available Schema objects consist of Collection objects containing Documents, or Table objects consisting of rows and Collections containing Documents.

The following table shows the available CRUD operations for both Collection and Table objects.

Operation	Document	Relational
Create	Collection.add()	Table.insert()
Read	Collection.find()	Table.select()
Update	Collection.modify()	Table.update()
Delete	Collection.remove()	Table.delete()

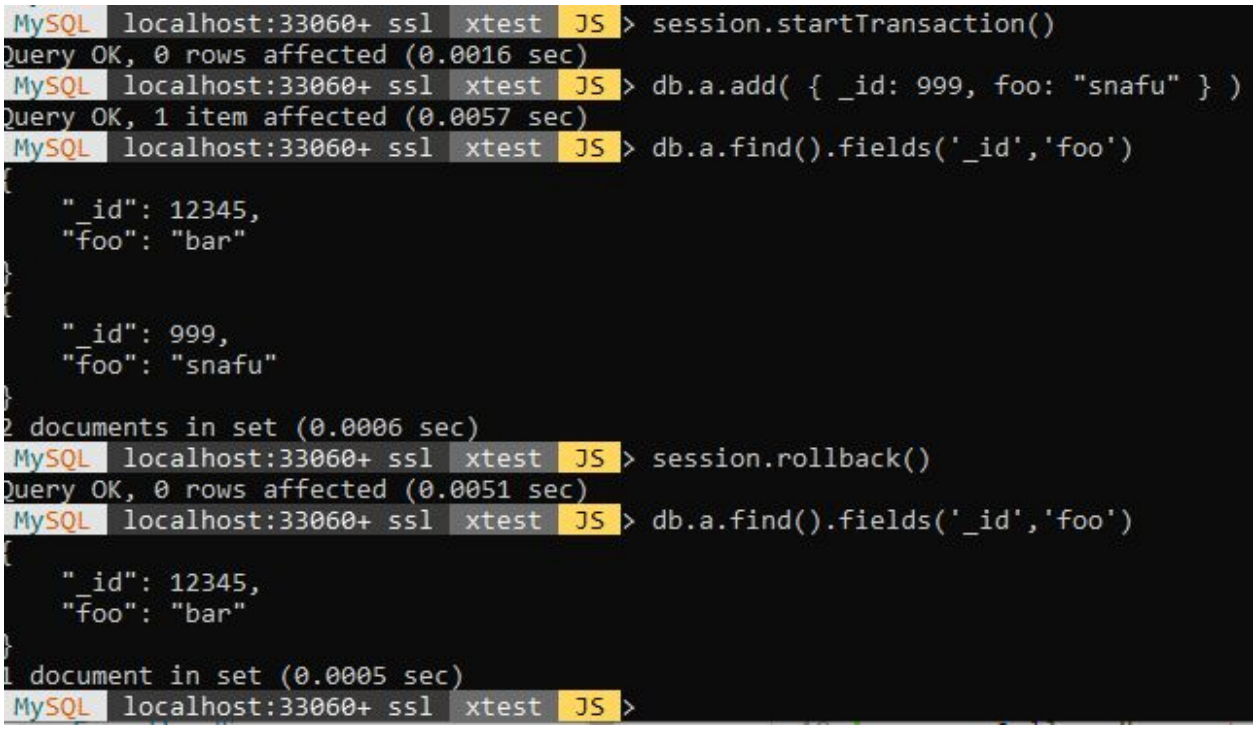
The X DevAPI supports a number of modern practices to make working with CRUD operations easier and to fit naturally into modern development environments. This allows using method chaining instead of working with SQL strings of JSON structures.

Transactions

The MySQL Document Store Supports transactions. The transaction is initiated with a **session,startTransaction()** command. Then a new document is added to the collection ‘a’ with **db.a.add({ _id: 999, foo: "snafu" })**.

A simple find is used to return the records and it is evident that the old document and the new document are there. Finally the new record is expunged by the **session.rollback()** command. Another find() shows that the new record is gone.

Example 4



Finding ‘things’

The ability to locate rows, er, documents by specific criteria is a must for a data store to be useful. In the next few examples the world_x document collection will be used. The find() function can accept a string encapsulated in double quotes (“) to allow picking documents matching the desired parameters. In the following find() is first used with no argument or search string and in the second the desired results are qualified to a certain set of records matching the desired condition of having an *IndepYear* value greater than 1900, **find("IndepYear > 1900")**.

```
MySQL  localhost:33060+ ssl  world_x  JS > db.countryinfo.find().fields(['Name','geography.Region','IndepYear']).limit(3)
{
    "Name": "Aruba",
    "IndepYear": null,
    "geography.Region": "Caribbean"
}
{
    "Name": "Afghanistan",
    "IndepYear": 1919,
    "geography.Region": "Southern and Central Asia"
}
{
    "Name": "Angola",
    "IndepYear": 1975,
    "geography.Region": "Central Africa"
}
3 documents in set (0.0005 sec)
MySQL  localhost:33060+ ssl  world_x  JS > db.countryinfo.find("IndepYear > 1900").fields(['Name','geography.Region','IndepYear']).limit(3)
{
    "Name": "Afghanistan",
    "IndepYear": 1919,
    "geography.Region": "Southern and Central Asia"
}
{
    "Name": "Angola",
    "IndepYear": 1975,
    "geography.Region": "Central Africa"
}
{
    "Name": "Albania",
    "IndepYear": 1912,
    "geography.Region": "Southern Europe"
}
3 documents in set (0.0007 sec)
MySQL  localhost:33060+ ssl  world_x  JS >
```

If you duplicate keys in the fields() do you get multiple values (or same value repeated)? Try **db.countryinfo.find('IndepYear > 1900').fields('Name','IndepYear','geography.Region','IndepYear').limit(3)**

Other Options

```
MySQL  localhost:33060+ ssl  world_x  JS > db.countryinfo.find("geography.Region = 'North America'").fields(['Name','geography.Region','IndepYear']).sort(['Name','IndepYear DESC'])
{
    "Name": "Bermuda",
    "IndepYear": null,
    "geography.Region": "North America"
}
{
    "Name": "Canada",
    "IndepYear": 1867,
    "geography.Region": "North America"
}
{
    "Name": "Greenland",
    "IndepYear": null,
    "geography.Region": "North America"
}
{
    "Name": "Saint Pierre and Miquelon",
    "IndepYear": null,
    "geography.Region": "North America"
}
{
    "Name": "United States",
    "IndepYear": 1776,
    "geography.Region": "North America"
}
5 documents in set (0.0012 sec)
MySQL  localhost:33060+ ssl  world_x  JS >
```

Here **db.countryinfo.find("geography.Region = 'North America'").fields(['Name','geography.Region','IndepYear']).sort(['Name','IndepYear DESC'])** is used or we could have tried something like “GNP > 500000”.

MySQL localhost:33060+ ssl world_x JS > db.countryinfo.find("geography.Region = :x").fields("[Name]").sort(["Name DESC"]).limit(3).bind("x","Western Europe")

```
{
    "[Name]": [
        "Switzerland"
    ]
}
{
    "[Name]": [
        "Netherlands"
    ]
}
{
    "[Name]": [
        "Monaco"
    ]
}
3 documents in set (0.0012 sec)
 MySQL  localhost:33060+ ssl  world_x  JS >
```

Indexes

Indexes are a vital part of databases and the MySQL Document Store supports creating (and deleting) indexes. The createIndex() takes two arguments -- the name of the index and the configuration parameters of that index.

db.b.createIndex("nbr_idx", {fields:[{"field": "$.nbr", "type":"INT", required:true}]});

There is no equivalent of the SQL '**SHOW CREATE TABLE**' and there is a need to sqitch from JavaScript (or Python) mode of the mysqlsh to SQL mode to see how the server understands the underlying structure on the server.

```
CREATE TABLE `b` (
  `doc` json DEFAULT NULL,
  `_id` varbinary(32) GENERATED ALWAYS AS
(json_unquote(json_extract(`doc`,_utf8mb4'$._id'))) STORED NOT NULL,
  `$ix_i_r_35C59DBC0A81294176AE5C80A7DD21079333A37E` int(11) GENERATED
ALWAYS AS (json_extract(`doc`,_utf8mb4'$.nbr')) VIRTUAL NOT NULL,
  PRIMARY KEY (`_id`),
  KEY `nbr_idx` (`$ix_i_r_35C59DBC0A81294176AE5C80A7DD21079333A37E`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
```

You can also use EXPLAIN on a query while in SQL mode to see if the new indexed is being used.

What is the _id?

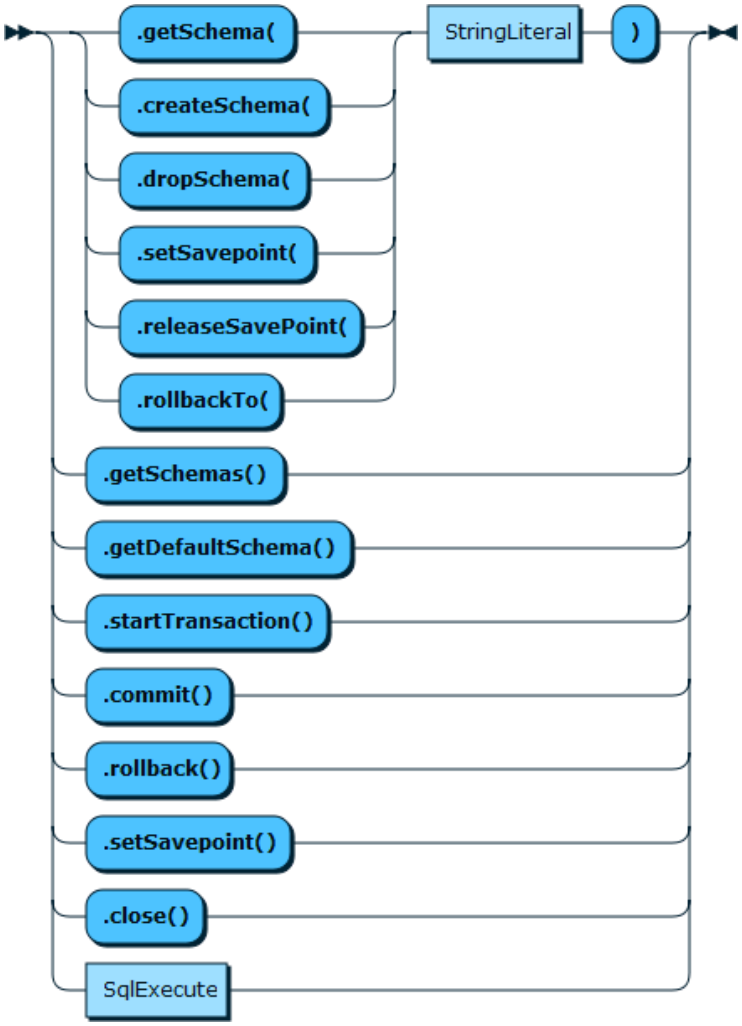
The InnoDB used the _id field as a primary key for efficiency and, unlike other keys in the document, can not be changed. The document store will automatically assign a value but you can assign your own value but you need to make sure you do not duplicate these values or the server will complain. The _id number made of a 4 byte prefix, an 8 byte timestamp of the startup time of the server instance, and a 16 byte a per-instance automatically incremented integer value, which is hex encoded and has a range of 0 to 2**64-1. The initial value of serial is set to the `auto_increment_offset` system variable, and the increment of the value is set by the `auto_increment_increment` system variable.

Users of multi-primary Group Replication or InnoDB cluster can depend on inserts to the same table from different instances do not have conflicting primary key values; assuming that the instances have the `auto_increment_*` system variables configured properly.

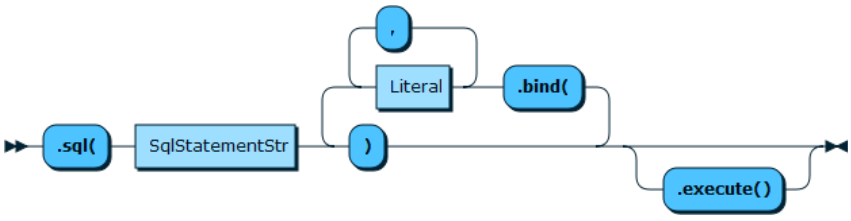
CRUD EBNF Definitions

Chapter 11 of the X DevAPI user guide lists the extended Backus–Naur form notation to describe the functions used with the MySQL Document Store. Use these as a handy reference to the API.

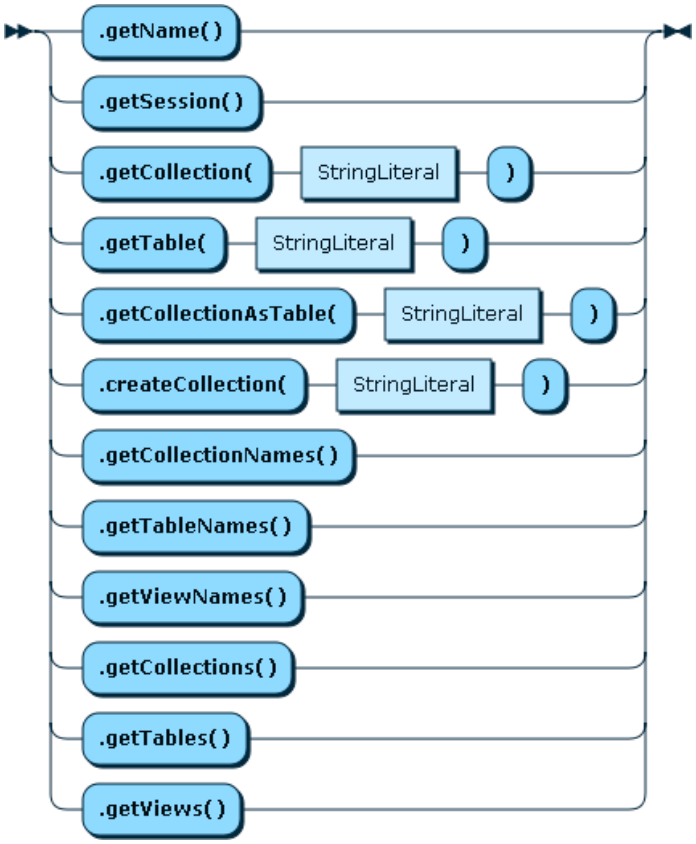
Session



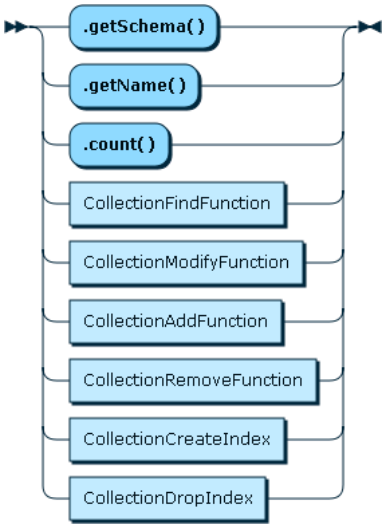
SQL Execute



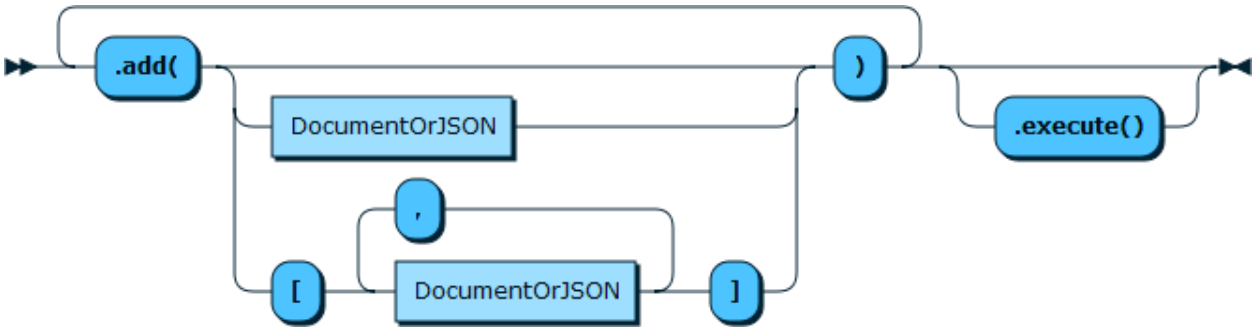
Schema



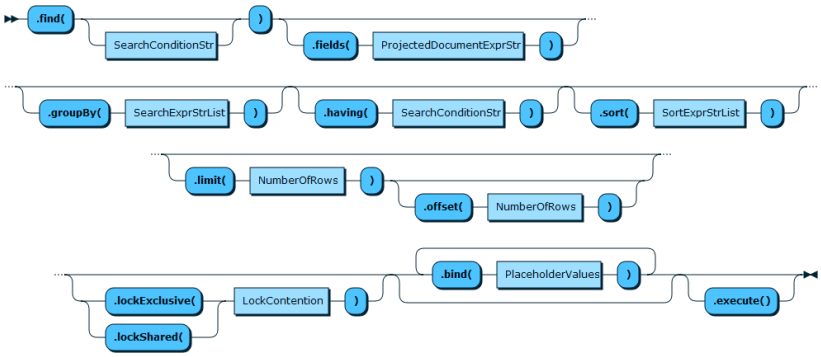
Collections



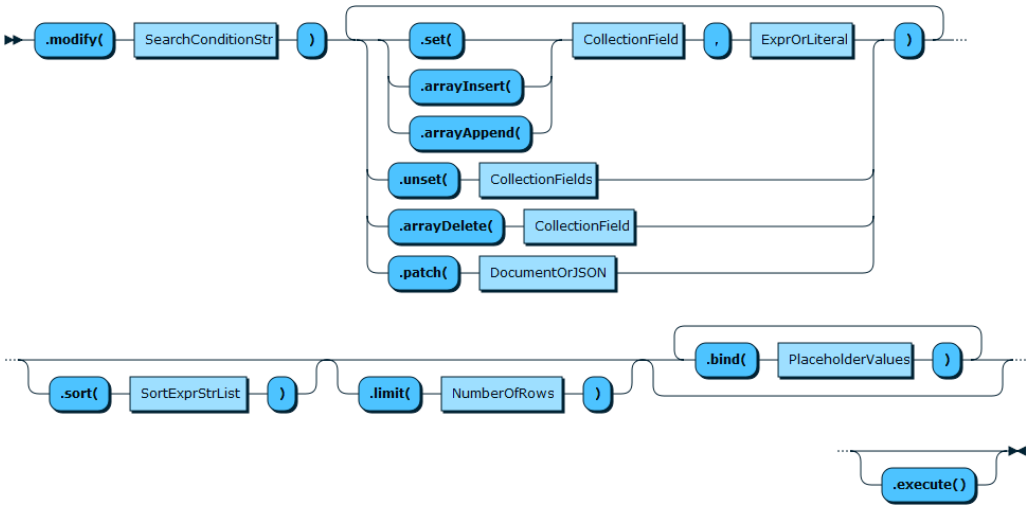
Collection Add



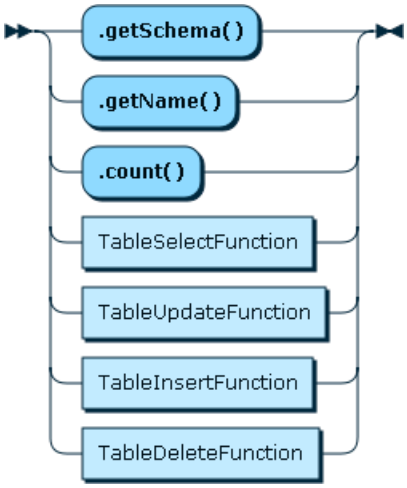
Collection Find



Collection Modify

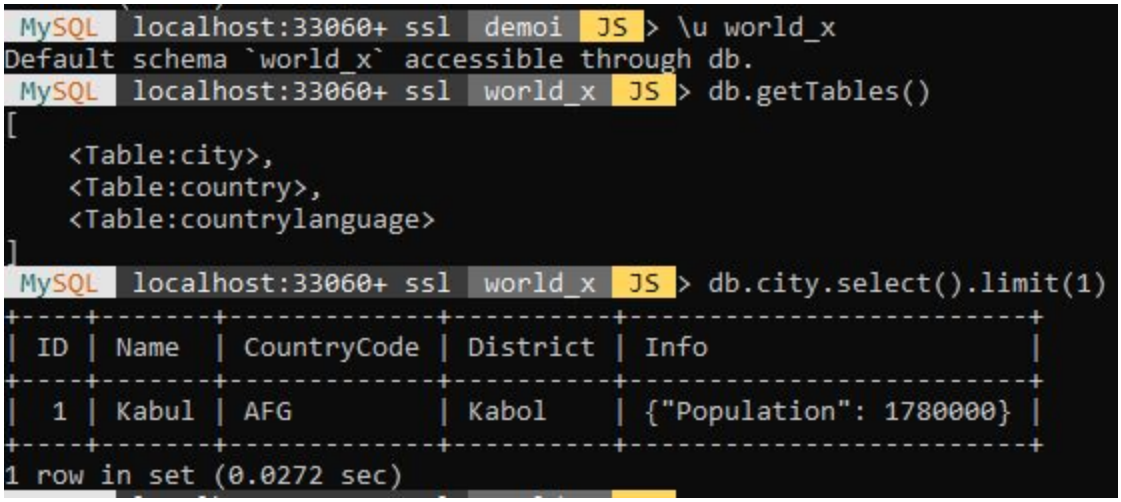


Table


.getSchema()
.getName()
.count()
TableSelectFunction
TableUpdateFunction
TableInsertFunction
TableDeleteFunction

Working With Tables

The MySQL Document Store can work with relational tables or cast collections as a relational table. We will look at dealing with relational tables first. The traditional SQL INSERT,DELETE,UPDATE, & SELECT are now functions and are used with tables (while document collections use add,remove,modify, and find functions) The following Node.JS is provided for illustration.



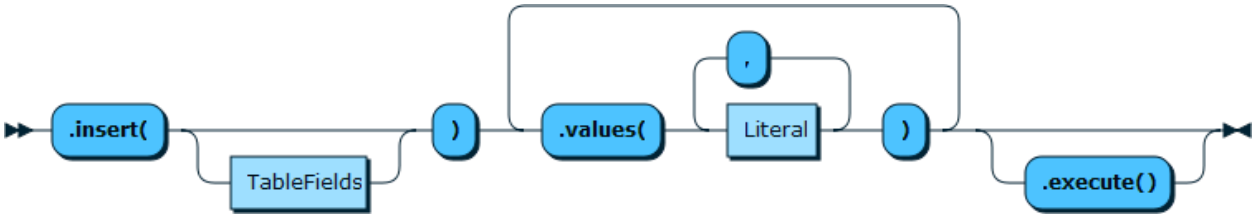
Exercise Final

How many documents are in the world_x schema matching the CountryCode = ‘BEL’?

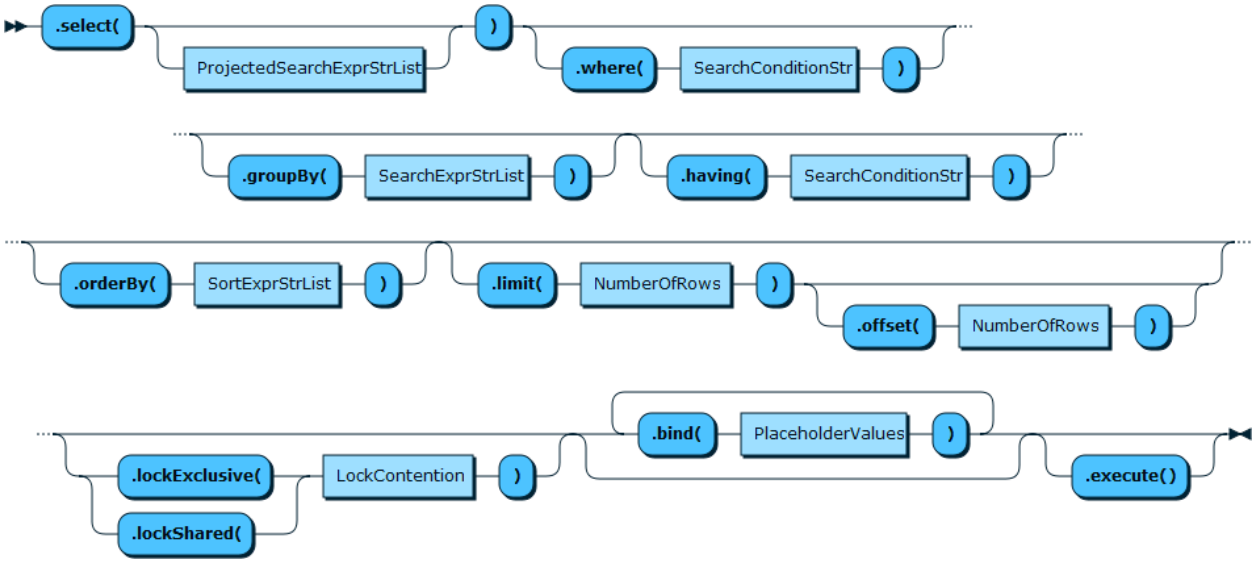
db.city.select().where("CountryCode = 'BEL'")

Table ENBF Diagrams

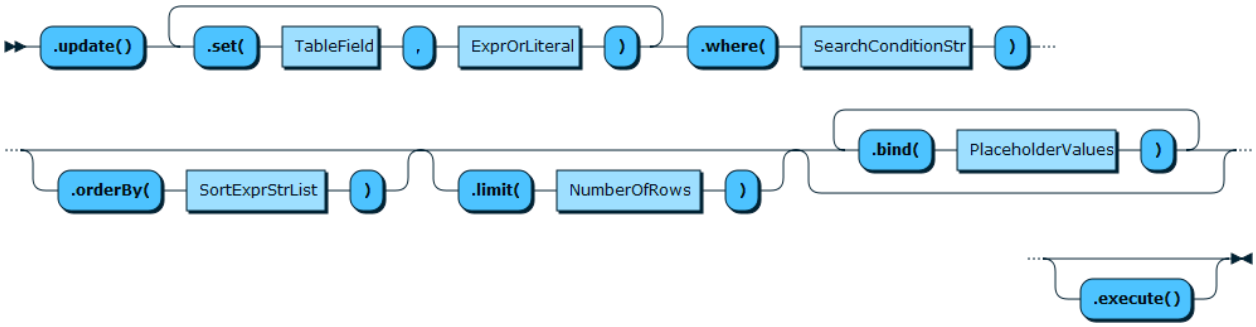
Insert



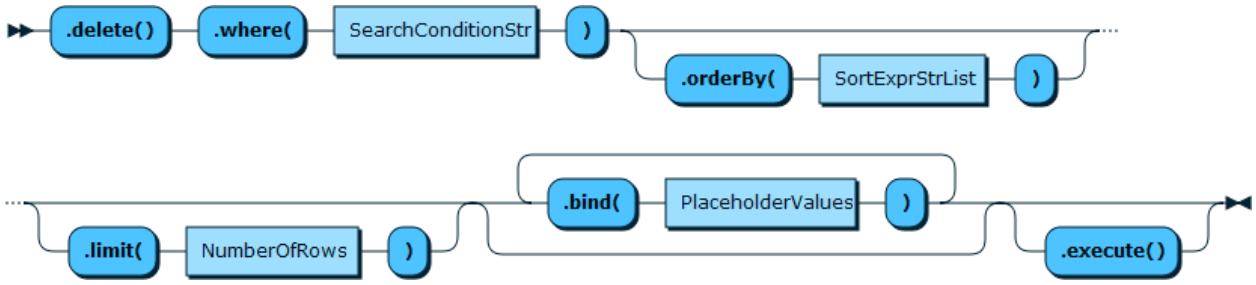
Select



Update



Delete

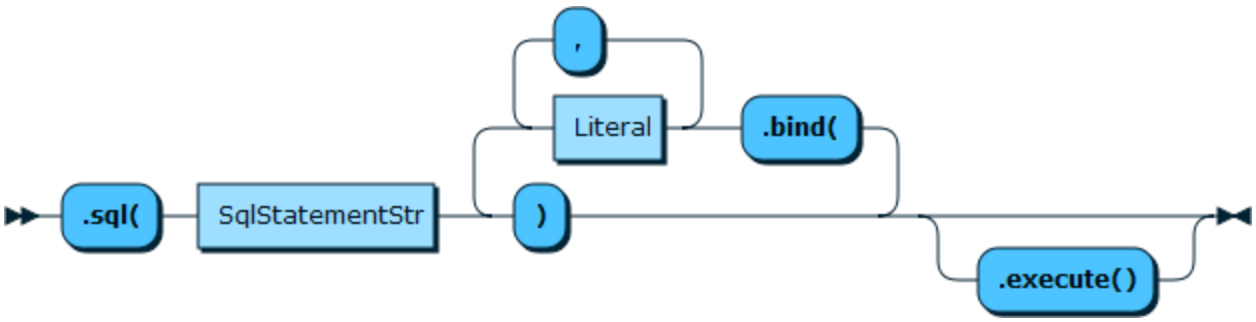


Another Option

What if you have a query you really like or just do not want to use the new API? Then you can use the SQL(). SImply plug the query into the function, such as **session.sql("SELECT * FROM city WHERE Name = 'Dallas'").execute();**

```
MySQL  localhost:33060+ ssl  world_x  JS > session.sql("SELECT * FROM city WHERE Name = 'Dallas'").execute();
+------+--------+-------------+----------+-------------------------+
| ID   | Name   | CountryCode | District | Info                    |
+------+--------+-------------+----------+-------------------------+
| 3800 | Dallas | USA         | Texas    | {"Population": 1188580} |
+------+--------+-------------+----------+-------------------------+
1 row in set (0.0031 sec)
MySQL  localhost:33060+ ssl  world_x  JS >
```

SQL EBNF



Synchronous versus Asynchronous Execution

Many MySQL drivers in the past used a synchronous approach to executing SQL statements. Operations such as opening connections and executing queries were blocked until completion and that could potentially take a long time. If a developer wanted parallel replication they had to write a multithreaded application with minimal help from the driver (or the server).

MySQL clients that support the X Protocol can provide asynchronous execution, either using callbacks, Promises, or by explicitly waiting on a specific result at the moment in time when it is actually needed. The following is an example of using callbacks is a very common way to implement asynchronous operations. When a callback function is specified, the CRUD operation is non-blocking which means that the next statement is called immediately even though the result from the database has not yet been fetched. Only when the result is available is the callback called.

```
var employees = db.getTable('employee');

employees.select('name', 'age')
  .where('name like :name')
  .orderBy('name')
  .bind('name', 'm%')
  .execute(function (row) {
    // do something with a row
  })
  .catch(err) {
    // Handle error
  });
```

Some (general) Definitions:

Session - A connection between a client (say MySQL Shell) and a MySQL server
Schema -- A logical pale to hold related data
Collection, Document -- A single JSON document (think RDMS row)

Resources
Using MySQL as a Document Store
https://dev.mysql.com/doc/refman/8.0/en/document-store.html

Documentation on X Devapi
https://dev.mysql.com/doc/

Connectors – C++, Java, JavaScript, .Net, Node.JS, Python, PHP
https://dev.mysql.com/downloads/

Books
Introducing the MySQL Document Store – Dr. Charles Bell
MySQL & JSON – A Practical Programming Guide – Dave Stokes
MySQL Connector Python – Jesper Wisborg Krogh