

The MySQL Document Store

Email: David.Stokes@Oracle.com
Twitter: @Stoker
Blog: elephantdolphin.blogspot.com

Ground Rules

Hi! This is a three hour tutorial for PHP developers on the MySQL Document Store, also known as the X DevAPI. The Document Store is a big change in the way to use MySQL as it allows its use as a JSON Document data store and comes with many new exciting features. The goal is to have you able to use these new features at the end of this tutorial.

I am assuming or hoping that you are PHP developer with some experience working with a relational database. Please do NOT hesitate to ask questions as you have them. I would rather answer your questions rather than have you anxious about having the right time to ask your question. There is no such thing as a dumb question; I wish I could say the same thing about my answers.

And <https://github.com/davidmstokes/PHP-X-DevAPI> will house all material for this tutorial and the accompanying talk at SunshinePHP 2019.

Feel free to follow along on your own system running MySQL 8, MySQL Shell, and PHP 7.2+ but we do not have time to load everyone's laptop and still cover material. Using the new MySQL Shell with MySQL 8 to test queries is highly recommended.

There are example programs on the Github directory that will be referred to as various points are illustrated.

Data for examples is often very hard to create from scratch. The data sets used in this tutorial will be either created on the fly, take advantage of the well know MySQL World/World_x sets, and/or the MongoDB primer-dataset.json (See appendix for links).

All references unless otherwise noted are for the MySQL Community Edition.

The MySQL Document Store

The MySQL Document Store is a JSON based NoSQL database and does not require the use of Structured Query Language (SQL) to interact with the database. No more embedding ugly strings of SQL in your beautiful PHP code! The new API calls (via the MySQL X Devapi PECL extension) follow modern programming design for all the CRUD functions of the document store. Each document has a payload of 1GB (compared to MongoDB's 16mb) and runs on proven, reliable MySQL technology. And if you have older relational data, the MySQL Document Store lets you access them plus the new document store data at the same time. This is a hands on workshop (please load MySQL 8 on your laptop!) for those wishing to switch over to the MySQL Document Store with plenty of programming examples.

The server-side is implemented through the X plugin (called `mysqlx` in the `information_schema.PLUGINS` view), and was first introduced as a beta release with MySQL Server 5.7.12. The X plugin reached general availability (GA) status with MySQL Server 8.0.11 and is now a built-in plugin and enabled by default. That is, on the server side you do not need to do anything to start using the MySQL Document Store.

There are a few more components to the MySQL Document Store:

- **X Plugin:** This is the server-side plugin that provides support for the X DevAPI.
- **X Protocol:** The protocol used for an application to communicate with the X Plugin.
- **The X DevAPI:** The API used with the X Protocol.

Collectively these components are known as the MySQL Document Store.

The X Plugin

The plugin for the X DevAPI is a shared object. It is optional in 5.7 and installed by default in MySQL 8.

To see if the plugin is loaded, you can use the command **mysql> SHOW PLUGINS;** and you should see *mysqlx* listed.

There is also details on the plugin to be found in the `information_schema` schema.

```
mysql> select * from information_schema.plugins where plugin_name like 'mysqlx%'\G
```

```
***** 1. row *****
  PLUGIN_NAME: mysqlx
  PLUGIN_VERSION: 1.0
  PLUGIN_STATUS: ACTIVE
  PLUGIN_TYPE: DAEMON
  PLUGIN_TYPE_VERSION: 80013.0
  PLUGIN_LIBRARY: NULL
  PLUGIN_LIBRARY_VERSION: NULL
  PLUGIN_AUTHOR: Oracle Corp
  PLUGIN_DESCRIPTION: X Plugin for MySQL
  PLUGIN_LICENSE: GPL
  LOAD_OPTION: ON
***** 2. row *****
  PLUGIN_NAME: mysqlx_cache_cleaner
  PLUGIN_VERSION: 1.0
  PLUGIN_STATUS: ACTIVE
  PLUGIN_TYPE: AUDIT
  PLUGIN_TYPE_VERSION: 4.1
  PLUGIN_LIBRARY: NULL
  PLUGIN_LIBRARY_VERSION: NULL
  PLUGIN_AUTHOR: Oracle Inc
  PLUGIN_DESCRIPTION: Cache cleaner for sha2 authentication in X plugin
  PLUGIN_LICENSE: GPL
  LOAD_OPTION: ON
2 rows in set (0.00 sec)

mysql>
```

The X Protocol

The X Protocol is based on Google Protobufs.

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

<https://developers.google.com/protocol-buffers/>

```
Mysqlx.Crud.Find {
  collection { name: "collection_name", schema: "test" }
  data_model: DOCUMENT
  criteria {
    type: OPERATOR
    operator {
      name: "=="
      param {
        type: IDENT,
        identifier { name: "_id" }
      }
      param {
        type: LITERAL,
        literal {
          type: V_STRING,
          v_string: { value: "some_string" }
        }
      }
    }
  }
}
```

Illustration 1: Google Protobuf Example

The X DevAPI uses wrapped Google Protobuf messages. Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.

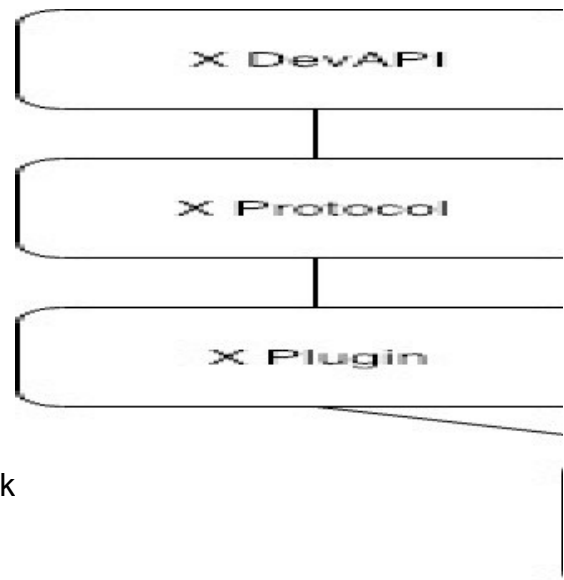
The X DevAPI

From a developer's perspective, the most interesting part of the MySQL Document Store is the X DevAPI. This is the API used to interact with the MySQL Document Store from your programs and from new MySQL Shell.

The X DevAPI is designed from the ground up with modern day usage in mind. It is available for a range of languages, for example: Python (MySQL Connector/Python), JavaScript (MySQL Connector/Node.js), PHP (mysql_xdevapi PECL extension), Java (MySQL Connector/J), C++ (MySQL Connector/C++), DotNet (MySQL Connector/NET). The X DevAPI is uniform across the supported programming languages while still maintaining the characteristics of the language.

The X DevAPI has three different parts. Which part you should use depends on how you want to interact with MySQL:

- **Collections:** The create-read-update-delete (CRUD) methods to work with JSON documents, i.e. using MySQL as a document store. This is a NoSQL API.
- **SQL Tables:** The CRUD methods to work with SQL (relational) tables. This is a NoSQL API.
- **SQL:** This can be used to execute arbitrary SQL statements against both collections and SQL tables.



The easiest way to try the X DevAPI is to use the MySQL Shell. The MySQL Shell is a relatively new command-line tool that not only support SQL statements but also Python and JavaScript. This makes it possible to test code before implementing it in an actual program, or use MySQL Shell to execute scripts that include use of Python or JavaScript routines.

```
dstokes@testbox:~/xdevapi$ mysqlsh root:hidave@localhost/world_x
mysqlsh: [Warning] Using a password on the command line interface can be insecure.
Creating a session to 'root@localhost/world_x'
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 24 (X protocol)
Server version: 8.0.13 MySQL Community Server - GPL
Default schema 'world_x' accessible through db.
MySQL Shell 8.0.13

Copyright (c) 2016, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type '\help' or '\?' for help; '\quit' to exit.

MySQL localhost:33060+ ssl world_x JS >
```

Illustration 2: Screen shot of the new MySQL shell

The MySQL Document Store is a JSON based NoSQL database

The MySQL Document Store provides a way for developers to work with both schema-less JSON document collections and SQL relational tables. MySQL has created the X DevAPI to focus on CRUD to let developers work with JSON document in a highly extensible fashion.

NoSQL + SQL = MySQL

The MySQL Document Store provides maximum flexibility for developing traditional SQL relational applications and NoSQL schema-less document store applications on the same platform. No need for separate databases for SQL and NoSQL. So both data models can be queried in the same application and the results can be in table, tabular, or JSON formats. Plus you get the statistical, logical, and analytical functions traditionally found with relational databases.

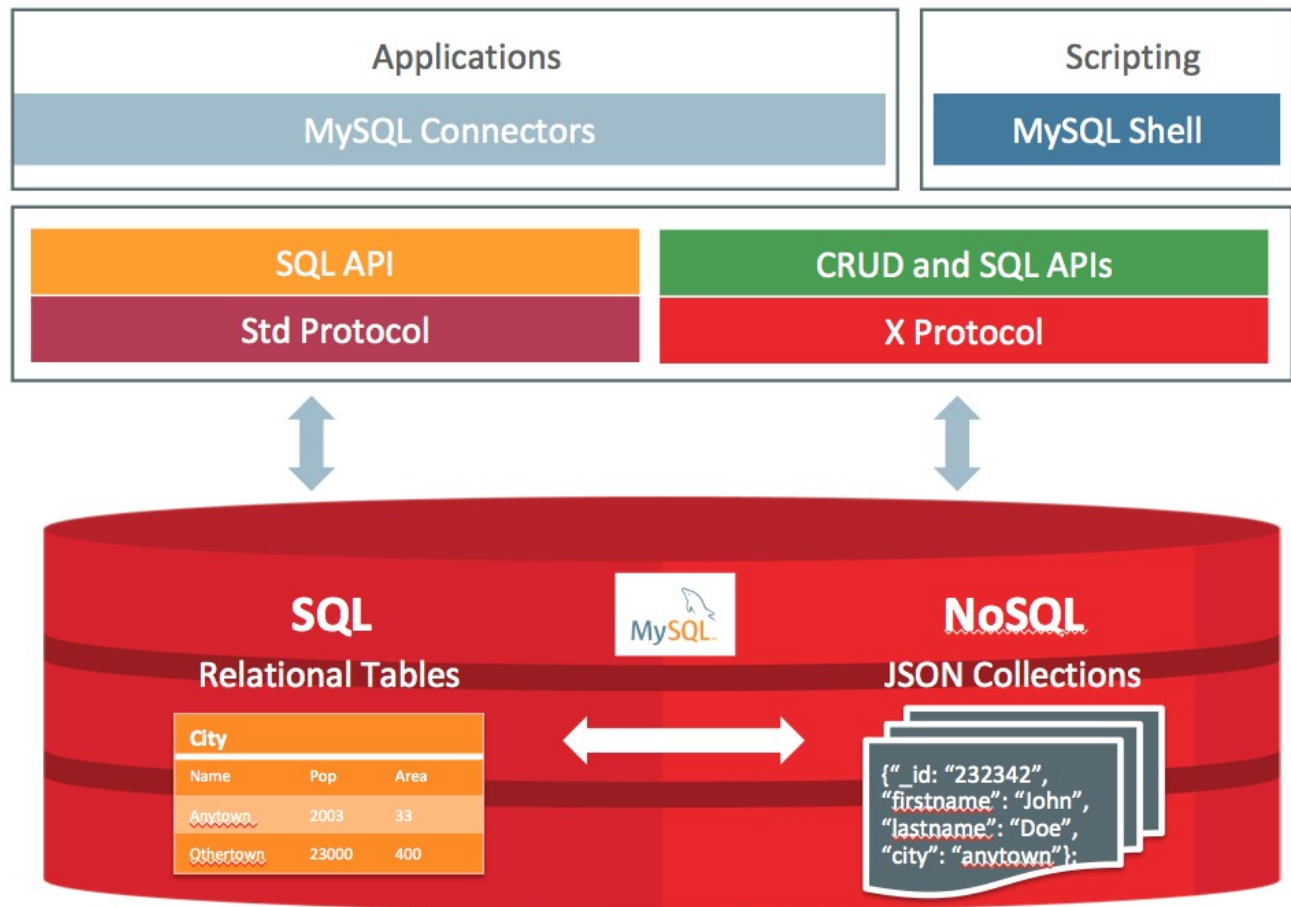


Illustration 3: MySQL Document Store Architecture

Architecture

The MySQL Document Store is a multi-document transaction supporting with full ACID compliance data storage for schema-less JSON documents. It is based on the InnoDB storage engine with the same data guarantees and performance advantages as relational data.

The MySQL Document Store can take advantage of Group Replication and InnoDB cluster to scale out application and achieve High Availability. Documents are replicated across all the members of the HA group with transaction being committed in sync across masters. And masters can take over from another if one fails, without down time.

The MySQL Document Store architecture consists of the following components:

- Native JSON Document Storage** - MySQL provides a native JSON datatype is efficiently stored in binary with the ability to create virtual columns that can be indexed. JSON Documents are automatically validated.
- X Plugin** - The X Plugin enables MySQL to use the X Protocol and uses Connectors and the Shell to act as clients to the server.
- X Protocol** - The X Protocol is a new client protocol based on top of the Protobuf library, and works for both, CRUD and SQL operations.
- X DevAPI** - The X DevAPI is a new, modern, async developer API for CRUD and SQL operations on top of X Protocol. It introduces Collections as new Schema objects. Documents are stored in Collections and have their dedicated CRUD operation set.
- MySQL Shell** - The MySQL Shell is an interactive Javascript, Python, or SQL interface supporting development and administration for the MySQL Server. You can use the MySQL Shell to perform data queries and updates as well as various administration operations.
- MySQL Connectors** - The following MySQL Connectors support the X Protocol and enable you to use X DevAPI in your chosen language.
 - MySQL Connector/Node.js
 - MySQL Connector/PHP
 - MySQL Connector/Python
 - MySQL Connector/J
 - MySQL Connector/NET
 - MySQL Connector/C++

Other connectors are on the road map!

The ‘Under The Hood’ View

Before we dive into the MySQL Document Store, it may help those of you with a RDMS and MySQL background to understand what the server is doing ‘behind the curtain’. The Document Store at a low level is all based on the JSON data type.

The JSON Data Type

MySQL added a native JSON document data type in version 5.7 as defined in RFC7159 and provides automatic validation of JSON documents (non conforming documents are rejected). MySQL uses an optimized binary storage format where the columns are converted to an internal b-tree format (think alphabetize keys) that allows quick reading and parsing plus it allows rapid access to directly to nested values or array indexes without reading all values before or after in the document.

Roughly 1GB payload per column (use multiple columns for more documents/payload). Storage size is roughly the same as a LONGBLOB or LONGTEXT and the size is actually limited by the MAX_ALLOWED_PACKET system variable.

Please that JSON columns can not have a default value or directly indexed.

Sample syntax: **CREATE TABLE foo (a INT default ‘1’, b JSON);**

There is a supporting set of 25 plus SQL functions for handing JSON values plus spatial functions for GeoJSON.

JSON columns, like other binary data types, are not Directly index-able from SQL. But you can use generated columns to extract scalar values from a JSON column to materialize values in its own column that can be indexed. (Note different in Document Store)

Generated columns are easy to create:

```
mysql> CREATE TABLE gentest (a INT, b INT AS (a + 1) STORED,  
INDEX(b) );  
mysql> INSERT INTO gentest (a) VALUES (1), (2), (3), (4);  
mysql> SELECT * FROM gentest;  
+----+----+  
| a  | b  |
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
+---+---+
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |
+---+---+
4 rows in set (0.0007 sec)
```

The MySQL Document Store uses by default a GENERATED column for the column **_id**. More about the **_id** column and its meaning later but for now note that the InnoDB storage engines needs a primary key for each table and **_id** fills that role. Here is what a table behind the Document Store looks like:

```
mysql> DESC countryinfo;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| doc   | json          | YES  |     | NULL    |                |
| _id   | varchar(32)   | NO   | PRI | NULL    | STORED GENERATED |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.0024 sec)
```

There is a little more information found in the SHOW CREATE TABLE results:

```
mysql> SHOW CREATE TABLE countryinfo\G
***** 1. row *****
      Table: countryinfo
Create Table: CREATE TABLE `countryinfo` (
  `doc` json DEFAULT NULL,
  `_id` varchar(32) GENERATED ALWAYS AS
(json_unquote(json_extract(`doc`,_utf8mb3'$._id')))) STORED NOT NULL,
  PRIMARY KEY (`_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.0007 sec)
```

The Document Store is designed to use a key in the JSON data named **_id** as the primary key, lacking one it will create a value. More on how those values are created will be covered later.

JSON Functions

MySQL 8 has many JSON functions for use from SQL and they are listed below.

Name	Description
<u>JSON_ARRAY()</u>	Create JSON array
<u>JSON_ARRAY_APPEND()</u>	Append data to JSON document
<u>JSON_ARRAY_INSERT()</u>	Insert into JSON array
<u>-></u>	Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT() .
<u>JSON_CONTAINS()</u>	Whether JSON document contains specific object at path
<u>JSON_CONTAINS_PATH()</u>	Whether JSON document contains any data at path
<u>JSON_DEPTH()</u>	Maximum depth of JSON document
<u>JSON_EXTRACT()</u>	Return data from JSON document
<u>->></u>	Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()) .
<u>JSON_INSERT()</u>	Insert data into JSON document
<u>JSON_KEYS()</u>	Array of keys from JSON document
<u>JSON_LENGTH()</u>	Number of elements in JSON document
<u>JSON_MERGE()</u> (deprecated 8.0.3)	Merge JSON documents, preserving duplicate keys. Deprecated synonym for JSON_MERGE_PRESERVE()
<u>JSON_MERGE_PATCH()</u>	Merge JSON documents, replacing values of duplicate keys
<u>JSON_MERGE_PRESERVE()</u>	Merge JSON documents, preserving duplicate keys
<u>JSON_OBJECT()</u>	Create JSON object
<u>JSON_PRETTY()</u>	Prints a JSON document in human-readable format, with each array element or object member printed on a new line, indented two spaces with respect to its parent.
<u>JSON_QUOTE()</u>	Quote JSON document
<u>JSON_REMOVE()</u>	Remove data from JSON document
<u>JSON_REPLACE()</u>	Replace values in JSON document
<u>JSON_SEARCH()</u>	Path to value within JSON document
<u>JSON_SET()</u>	Insert data into JSON document
<u>JSON_STORAGE_FREE()</u>	Freed space within binary representation of a JSON column value following a partial update
<u>JSON_STORAGE_SIZE()</u>	Space used for storage of binary representation of a JSON document; for a JSON column, the space used when the document

Name	Description
	was inserted, prior to any partial updates
<u>JSON_TABLE()</u>	Returns data from a JSON expression as a relational table
<u>JSON_TYPE()</u>	Type of JSON value
<u>JSON_UNQUOTE()</u>	Unquote JSON value
<u>JSON_VALID()</u>	Whether JSON value is valid

Partial Updates of JSON Data

The MySQL 8.0 optimizer will perform a partial, in-place update of a JSON column instead of removing the old document and writing the new document in its entirety to the column. This optimization can be performed for an update that meets the following conditions:

- The column being updated was declared as JSON.
- The UPDATE statement uses any of the three functions `JSON_SET()`, `JSON_REPLACE()`, or `JSON_REMOVE()` to update the column. A direct assignment of the column value (for example, `UPDATE mytable SET jcol = '{"a": 10, "b": 25}'`) cannot be performed as a partial update.
- Updates of multiple JSON columns in a single UPDATE statement can be optimized in this fashion; MySQL can perform partial updates of only those columns whose values are updated using the three functions just listed.
- The input column and the target column must be the same column; a statement such as `UPDATE mytable SET jcol1 = JSON_SET(jcol2, '$.a', 100)` cannot be performed as a partial update.
- The update can use nested calls to any of the functions listed in the previous item, in any combination, as long as the input and target columns are the same.
- All changes replace existing array or object values with new ones, and do not add any new elements to the parent object or array.

- The value being replaced must be at least as large as the replacement value. In other words, the new value cannot be any larger than the old one.
- A possible exception to this requirement occurs when a previous partial update has left sufficient space for the larger value. You can use the function `JSON_STORAGE_FREE()` see how much space has been freed by any partial updates of a JSON column.

Such partial updates can be written to the binary log using a compact format that saves space; this can be enabled by setting the `binlog_row_value_options` system variable to `PARTIAL_JSON`. See the description of this variable for more information.

My Book on MySQL's JSON Data type

See **MySQL & JSON a Practical Programming Guide**

A good reference book on using the MySQL JSON Data type is available and be ordered off Amazon

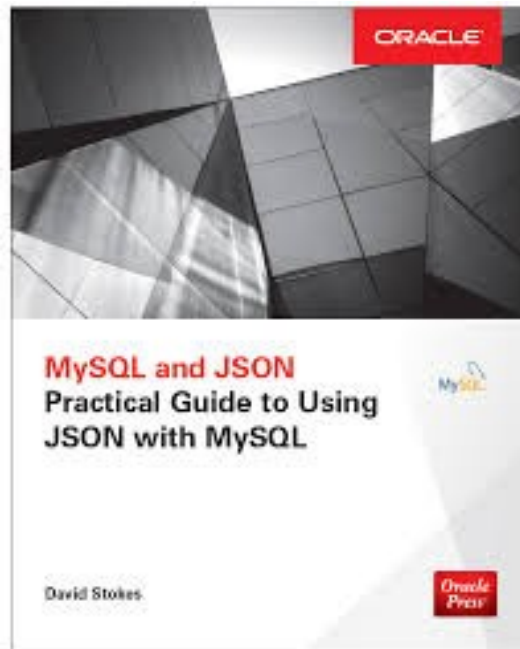


Illustration 4: MySQL and JSON - A Practical Programming Guide

X DevAPI

The X DevAPI wraps powerful concepts in a simple API that brings a new high-level session concept that enables developers to write code that can transparently scale from single MySQL Server to a multiple server environment.

- 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. If you are using MySQL Shell, when a low-level MySQL connection to a single MySQL instance is needed this is still supported by using a ClassicSession, which provides full support of SQL.

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	<u>Collection.modify()</u>	Table.update()

Operation	Document	Relational
Delete	Collection.remove()	Table.delete()

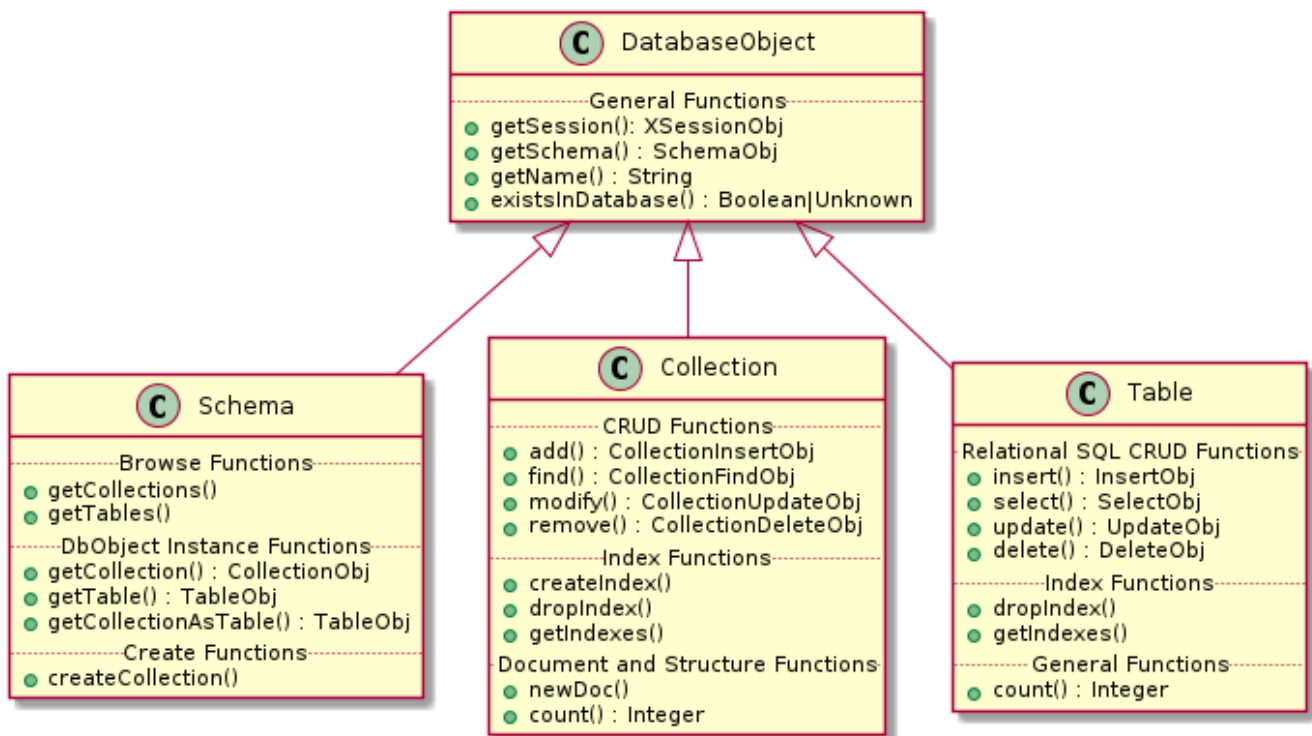


Illustration 5: How the Objects Relate

No longer only SQL based

Traditionally, relational databases such as MySQL have usually required a schema to be defined before documents can be stored. The features described in this section enable you to use MySQL as a document store, which is a schema-less, and therefore schema-flexible, storage system for documents. For example, when 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. The features described in this chapter enable you to choose how you configure MySQL, using only the document store model, or combining the flexibility of the document store model with the power of the relational model.

To use MySQL as a document store, you use the following server features:

- X Plugin enables MySQL Server to communicate with clients using X Protocol, which is a prerequisite for using MySQL as a document store. X Plugin is enabled by default in MySQL Server as of MySQL 8.0. This is a shared object that plugs into the server, optional in MySQL 5.7 and installed by default in MySQL 8.
- X Protocol supports both CRUD and SQL operations, authentication via SASL, allows streaming (pipelining) of commands and is extensible on the protocol and the message layer. Clients compatible with X Protocol include MySQL Shell and MySQL 8.0 Connectors.
- Clients that communicate with a MySQL Server using X Protocol can use X DevAPI to develop applications. X DevAPI offers a modern programming interface with a simple yet powerful design which provides support for established industry standard concepts. This chapter explains how to get started using either the JavaScript or Python implementation of X DevAPI in MySQL Shell as a client. Note that the new protocol ‘listens’ on port 33060 instead of the tradition MySQL port of 3306 (and you may needs to adjust network firewalls to allow packets to pass).

Modern Program Design

Traditionally it has been common to see code such as:

```
$query = “SELECT * FROM secret_table WHERE user=” . $_POST[‘user’] ;
```

This type of query is often hard to read/debug for those not use to Structured Query Language (SQL), can invite SQL injection, and most IDEs can not assist in coding. SQL is a declarative language while PHP is a procedural and/or object oriented language. Embedding SQL in a PHP script is not only ugly but an ‘impedance mismatch’ (mixing two things that are probably better not mixed).

Since the early days of PHP working with MySQL, you either had to embed ugly strings of SQL code or use an Object Relation Mapper like Doctrine. Doctrine is great but it can produce less than optimal queries and frankly there is a big learning curve with any ORM (and it is probably less effort to just learn SQL).

The X DevAPI is designed with modern programming language designs in mind with stack-able calls and works with IDEs so you no longer have to mix SQL with PHP. And as a benefit the new API calls are standardize over many languages.

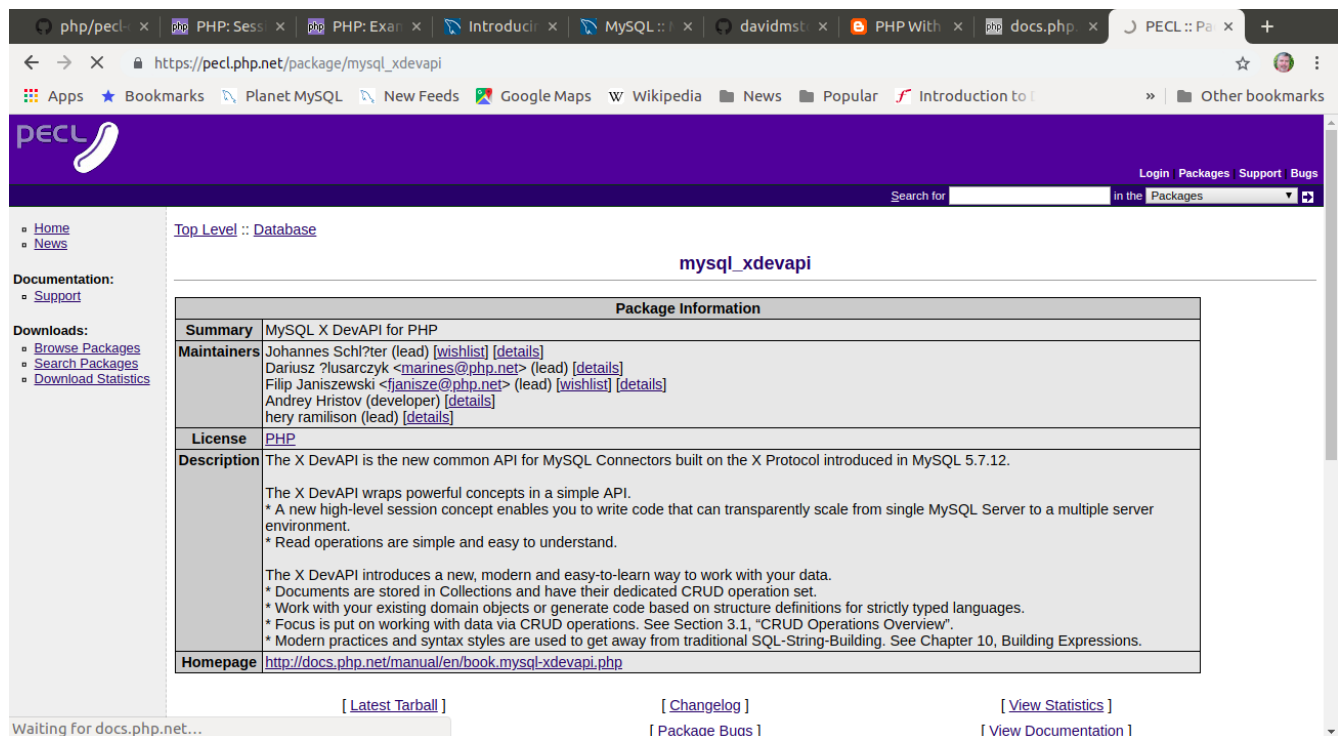
MySQL Document Store Tutorial – Sunshine PHP 2019

The following code snippet shows how the XdevAPI follows modern programming design.

```
$row = $table->select('Name','District')
    ->where('District' like ':district')
    ->bind(['district' => 'Texas'])
    ->limit(25)
    ->execute()->fetchAll();
```

The above code looks much more like PHP than SQL and is easier to decipher for most programmers, especially for those with minimal or no SQL programming experience.

The X DevAPI PECL Extension



The screenshot shows the PECL website interface. The browser address bar displays https://pecl.php.net/package/mysql_xdevapi. The page title is "mysql_xdevapi". The left sidebar contains navigation links: Home, News, Documentation (Support), and Downloads (Browse Packages, Search Packages, Download Statistics). The main content area is titled "Package Information" and contains the following details:

Package Information	
Summary	MySQL X DevAPI for PHP
Maintainers	Johannes Schlöter (lead) [wishlist] [details] Dariusz Żylusarczyk <marines@php.net> (lead) [details] Filip Janiszewski <fjanisze@php.net> (lead) [wishlist] [details] Andrey Hristov (developer) [details] hery ramilison (lead) [details]
License	PHP
Description	<p>The X DevAPI is the new common API for MySQL Connectors built on the X Protocol introduced in MySQL 5.7.12.</p> <p>The X DevAPI wraps powerful concepts in a simple API.</p> <ul style="list-style-type: none">* A new high-level session concept enables you to write code that can transparently scale from single MySQL Server to a multiple server environment.* Read operations are simple and easy to understand. <p>The X DevAPI introduces a new, modern and easy-to-learn way to work with your data.</p> <ul style="list-style-type: none">* 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. See Section 3.1, "CRUD Operations Overview".* Modern practices and syntax styles are used to get away from traditional SQL-String-Building. See Chapter 10, Building Expressions.
Homepage	http://docs.php.net/manual/en/book.mysql-xdevapi.php

At the bottom of the package information section, there are links: [Latest Tarball], [Changelog], [View Statistics], [Package Bugs], and [View Documentation].

You will find the MySQL X DevAPI extension at https://github.com/php/pecl-database-mysql_xdevapi or https://pecl.php.net/package/mysql_xdevapi and you will need to build the pieces as it is not part of the 'mainstream' MySQLnd or MySQLi connector. This extension requires a MySQL 8+ server with the X plugin enabled (default).

Prerequisite libraries for compiling this extension are: Boost (1.53.0 or higher), OpenSSL, and Protobuf. And, of course, PHP!

How to build

An example installation procedure on Ubuntu 18.04 with PHP 7.2:

```
// Dependencies
```

```
$ apt install build-essential libprotobuf-dev libboost-dev openssl protobuf-compiler
```

```
// PHP with the desired extensions; php7.2-dev is required to compile
```

```
$ apt install php7.2-cli php7.2-dev php7.2-mysql php7.2-pdo php7.2-xml
```

```
// Compile the extension
```

```
$ pecl install mysql_xdevapi
```

The pecl install command does not enable PHP extensions (by default) and enabling PHP extensions can be done in several ways. Another PHP 7.2 on Ubuntu 18.04 example:

```
// Create its own ini file
```

```
$ echo "extension=mysql_xdevapi.so" > /etc/php/7.2/mods-available/mysql_xdevapi.ini
```

```
// Use the 'phpenmod' command (note: it's Debian/Ubuntu specific)
```

```
$ phpenmod -v 7.2 -s ALL mysql_xdevapi
```

```
// A 'phpenmod' alternative is to manually symlink it
```

```
// $ ln -s /etc/php/7.2/mods-available/mysql_xdevapi.ini /etc/php/7.2/cli/conf.d/20-mysql_xdevapi.ini
```

```
// Let's see which MySQL extensions are enabled now
```

```
$ php -m |grep mysql
```

```
mysql_xdevapi
```

```
mysqli
```

```
mysqlnd
```

```
pdo_mysql
```

Why not main stream?

Simple test programming

```
<?php
```

```
$session =
```

```
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost:33060");
```

```
if ($session === NULL ) {
```

```
    die("Connection not established!\n");  
}
```

```
echo "Connection established!\n");  
?>
```

The basics of the MySQL Document Store

Note: The railroad diagrams or EBNF diagrams are including later in this document.

Since the Document Store and X DevAPI are pretty much the same regardless of language, the new MySQL Shell (mysqlsh) can be used to prototype queries. This tutorial will make extensive use of the new shell for prototyping and demonstration.

Program flow

This a basic flow to programs when using a API to a database and it is:

1. Connect to server
2. Send Query
3. Process returned data, if any
4. Disconnect

Each one of those steps have many steps involved that need to be completed successfully. For instance connection requires working network (or loop back) connection(s), a valid username, a valid authentication string, or password proper network white-listing, proper resources for the account, access grants allowing access to database/schema, and that the server is allowing additional connections. The query itself has to have valid syntax, the columns used must be valid, and the connecting user must have the required access to those columns, before the optimizer will build a query plan to get the data. Then the results of the query are returned which may include a return code, various statuses, and data. Finally the program should disconnect, however most programming languages and/or operating system will tear down connections when the PHP process terminates.

Connector URI

Example X DevAPI URIs

```
mysqlx://foobar  
mysqlx://root@localhost?socket=%2Ftmp%2Fmysqld.sock%2F
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
mysqlx://foo:bar@localhost:33060
mysqlx://foo:bar@localhost:33160?ssl-mode=disabled
mysqlx://foo:bar@localhost:33260?ssl-mode=required
mysqlx://foo:bar@localhost:33360?ssl-mode=required&auth=mysql41
mysqlx://foo:bar@(/path/to/socket)
mysqlx://foo:bar@(/path/to/socket)?auth=sha256_mem
mysqlx://foo:bar@[localhost:33060, 127.0.0.1:33061]
mysqlx://foobar?ssl-ca=(/path/to/ca.pem)&ssl-crl=(/path/to/crl.pem)
mysqlx://foo:bar@[localhost:33060, 127.0.0.1:33061]?ssl-mode=disabled
mysqlx://foo:bar@localhost:33160/?connect-timeout=0
mysqlx://foo:bar@localhost:33160/?connect-timeout=10
```

An Example of URI Usage

The listschemas.php program shows a common use of the URI for X DevAPI use.

```
<?php

$session = mysql_xdevapi\
getSession("mysqlx://myuser:mypass@localhost");

$schemas = $session->getSchemas();
echo "Available Schemas:\n";
print_r($schemas);
?>
```

Now that you can get connected to the server via the X DevAPI, it is time to start looking at example code.

A Quick Example – sunshine01.php

The following is a very simple example of using the X DevAPI with PHP.

Example: sunshine01.php

```
#!/bin/php
<?php

$session =
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost:33060");
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
if ($session === NULL) {
    die("Connection could not be established");
}

$marco = [
    "name" => "Marco",
    "age"   => 19,
    "job"   => "Programmer"
];
$mike = [
    "name" => "Mike",
    "age"   => 39,
    "job"   => "Manager"
];

$schema = $session->getSchema("testxx");
$collection = $schema->createCollection("example1");
$collection = $schema->getCollection("example1");

$collection->add($marco, $mike)->execute();

var_dump($collection->find("name = 'Mike'")->execute()->fetchOne());
?>
```

Note that it helps to wrap calls with try/catch to handle any errors properly.

There is a lot missing from the following program (error and return code checking chief among them)

The program execution:

```
$ php sunshine01.php
array(4) {
    ["_id"]=>
    string(28) "00005c07ea8100000000000000002"
    ["age"]=>
    int(39)
    ["job"]=>
    string(7) "Manager"
    ["name"]=>
```

```
    string(4) "Mike"  
}
```

A Second Example

The file `simpleopensesession.php` is an example of dealing with session data. Your program must connect to the server to establish a session.

```
<?php  
  
/**  
 * Simple MySQL X DevAPI example program  
 *  
 * This program opens a session to the server,  
 * gets the clientid information,  
 * gets the MySQL Server Version,  
 * gets the client information,  
 * and finally gets the schemas available to client user  
 * before closing the connection.  
 */  
  
# Open session to MySQL Server  
$session =  
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost");  
  
# Client ID information  
$clientid = $session->getClientId();  
echo "Client id: " . $clientid . PHP_EOL;  
  
# Server information  
$serverVersion = $session->getServerVersion();  
echo "Server version: " . $serverVersion . PHP_EOL;  
  
# Client Information  
$ids = $session->listClients();  
echo "Current clients: ". json_encode($ids) . PHP_EOL;  
  
$schemas = $session->getSchemas();  
var_dump($schemas);
```

```
# Close connection
$session->close();
?>
```

Once the session is established then you can use a document collection (JSON) or a relational table (SQL).

A Quick Diversion on Connections

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 the `caching_sha2_password` authentication plugin for 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.

Third Example

This example is a little different in that besides the PHP code and the X DevAPI there is the database to deal with. Lets look at the problems that can arise around a simple program such as the following:

```
<?php
$uri  = 'mysqlx://myuser:mypass@127.0.0.1:33060/';
$sess = mysql_xdevapi\getSession($uri);
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
try {
    if ($schema = $sess->createSchema('fruit')) {
        echo "Info: I created a schema named 'fruit'\n";
    }

} catch (Exception $e) {
    echo $e->getMessage();
}
?>
```

The code is good but the first time you try to run it the results are not what is desired.

```
$php createschema.php
[42000] Access denied for user 'myuser'@'localhost' to database
'fruit'@testbox
```

In this case the code assumes that there is a database named 'fruit' and that our login/password is valid. The above error message is saying that access was denied. The problem is 'myuser' can not access the schema 'fruit' because it does not exist and we do not have the rights to access it.

```
$mysql -u root -p
```

Enter password:

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 9

Server version: 8.0.13 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

```
mysql> grant all on fruit.* to 'myuser'@'localhost';
```


MySQL Document Store Tutorial – Sunshine PHP 2019

```
Query OK, 0 rows affected (0.29 sec)
mysql> flush privileges;
Query OK, 0 rows affected (0.05 sec)
mysql> \q
Bye
```

Now when the program runs it does what is expected.

```
$ php createschema.php
Info: I created a schema named 'fruit'
```

But if we try to run it again, we get the following message.

```
$ php createschema.php
[HY000] Can't create database 'fruit'; database exists
```

So the fruit database can not be recreated. In this case the program is absolutely correct but the underlying database hygiene was neglected. Please note that a better choice would to refactor the code to check to see if the database existed and skipping creation if it does exist.

Collections

Collections hold JSON documents. These are considered unstructured NoSQL JSON documents but please note that the MySQL server puts the keys into a B-Tree for rapid access which also sorts the keys. Tables are for relational data and will be covered later.

Creating (and deleting collections)

file: collectionexist.php

```
<?php
$session =
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost");
$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();
$session->sql("CREATE DATABASE addressbook")->execute();

$schema = $session->getSchema("addressbook");
$create = $schema->createCollection("people");

echo "Collection 'people' created\n And deleting\n";
$schema->dropCollection("people");
// ...

$collection = $schema->getCollection("people");

// ...

if (!$collection->existsInDatabase()) {
    echo "The collection no longer exists in the database named
addressbook. What happened?";
}
?>
```

Adding documents

file: collectionadd.php

```
<?php
$session =
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost");
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();
$session->sql("CREATE DATABASE addressbook")->execute();

$schema = $session->getSchema("addressbook");
$create = $schema->createCollection("people");

$collection = $schema->getCollection("people");

// Add two documents
$collection->add('{"name": "Fred", "age": 21, "job":
"Construction"}')->execute();
$collection->add('{"name": "Wilma", "age": 23, "job": "Teacher"}')-
>execute();

// Add two documents using a single JSON object
$result = $collection->add(
    '{"name": "Bernie",
      "jobs": [{"title": "Cat Herder", "Salary": 42000},
{"title": "Father", "Salary": 0}],
      "hobbies": ["Sports", "Making cupcakes"]}',
    '{"name": "Jane",
      "jobs": [{"title": "Scientist", "Salary": 18000},
{"title": "Mother", "Salary": 0}],
      "hobbies": ["Walking", "Making pies"]}')->execute();

// Fetch a list of generated ID's from the last add()
$ids = $result->getGeneratedIds();
print_r($ids);
?>
```

```
$ php collectionadd.php
Array
(
    [0] => 00005c34d4d8000000000000000003
    [1] => 00005c34d4d8000000000000000004
)
```

You might be asking ‘we added FOUR records in total but getGeneratedIDs() only returned TWO generated IDs?’. That function returns the IDs for the last insert operation only. But as can be seen below are the four records that were inserted in the above example.

MySQL Document Store Tutorial – Sunshine PHP 2019

```
mysql> SELECT * FROM people\G
***** 1. row *****
doc: {"_id": "00005c34d4d80000000000000001", "age": 21, "job":
"Construction", "name": "Fred"}
_id: 00005c34d4d8000000000000000001
***** 2. row *****
doc: {"_id": "00005c34d4d8000000000000000002", "age": 23, "job":
"Teacher", "name": "Wilma"}
_id: 00005c34d4d8000000000000000002
***** 3. row *****
doc: {"_id": "00005c34d4d8000000000000000003", "jobs": [{"title": "Cat
Herder", "Salary": 42000}, {"title": "Father", "Salary": 0}], "name":
"Bernie", "hobbies": ["Sports", "Making cupcakes"]}
_id: 00005c34d4d8000000000000000003
***** 4. row *****
doc: {"_id": "00005c34d4d8000000000000000004", "jobs": [{"title":
"Scientist", "Salary": 18000}, {"title": "Mother", "Salary": 0}],
"name": "Jane", "hobbies": ["Walking", "Making pies"]}
_id: 00005c34d4d8000000000000000004
4 rows in set (0.01 sec)collectioncount.php
```

Counting records in collections

Sometimes you want to know how many JSON documents are in a collection and for that we have a `count()` function.

```
<?php
$session =
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost");
$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();
$session->sql("CREATE DATABASE addressbook")->execute();

$schema = $session->getSchema("addressbook");
$create = $schema->createCollection("people");

$collection = $schema->getCollection("people");

$result = $collection
    ->add(
        '{"name": "Bernie",
```

```
"jobs": [
    {"title":"Cat Herder","Salary":42000},
    {"title":"Father","Salary":0}
],
"hobbies": ["Sports","Making cupcakes"]}',
'{"name": "Jane",
  "jobs": [
    {"title":"Scientist","Salary":18000},
    {"title":"Mother","Salary":0}
  ],
  "hobbies": ["Walking","Making pies"]}'
->execute();
```

```
echo $collection->count() . " Records in collection\n";
?>
```

And the results

```
$php collectioncount.php
2 Records in collection
```

Finding records in a collections

It is very important to be able to retrieve records by either a specific qualifier. The find() function has many options to help you get the data desired. In the example below we pass two qualifiers to find() with a logical AND operator in between the qualifier. Note the use of the colon (:) to specify a binding to a variable and the supporting bind() post pended to the find().

The query could be rewritten to not use binding. Please see collectionfind2.php

```
$find = $collection->find('job = "Teacher" AND age > 20');
```

An example with bindings

```
<?php
$session =
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost");

$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();
$session->sql("CREATE DATABASE addressbook")->execute();
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
$schema      = $session->getSchema("addressbook");
$collection  = $schema->createCollection("people");

$collection->add(['name': "Alfred",      "age": 18, "job":
"Butler"]);
$collection->add(['name': "Bob",        "age": 19, "job":
"Swimmer"]);
$collection->add(['name': "Fred",       "age": 20, "job":
"Construction"]);
$collection->add(['name': "Wilma",      "age": 21, "job":
"Teacher"]);
$collection->add(['name': "Suki",       "age": 22, "job":
"Teacher"]);

$find  = $collection->find('job LIKE :job AND age > :age');
$result = $find
    ->bind(['job' => 'Teacher', 'age' => 20])
    ->sort('age DESC')
    ->limit(2)
    ->execute();

print_r($result->fetchAll());
?>
```

Output:

\$php collectionfind.php

Array

```
(
    [0] => Array
        (
            [_id] => 00005c34d4d8000000000000000016
            [age] => 22
            [job] => Teacher
            [name] => Suki
        )

    [1] => Array
        (
            [_id] => 00005c34d4d8000000000000000015
```

```
        [age] => 21
        [job] => Teacher
        [name] => Wilma
    )
)
```

And as a personal note, I do not like to break up the \$find and \$result as seen in the above. I prefer not to break them up as a rule. But there are some interesting use cases when they are broken up as when you need basically the same query sorted differently. See collectionfind3.php

```
$result = $collection->find('job like :job AND age > :age')
    ->bind(['job' => 'Teacher', 'age' => 20])
    ->sort('age DESC')
    ->limit(2)
    ->execute();
```

Collection: modify a documents

```
$cat collectionmodify.php
<?php
$session =
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost");

$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();
$session->sql("CREATE DATABASE addressbook")->execute();

$schema      = $session->getSchema("addressbook");
$collection = $schema->createCollection("people");

$collection->add('{"name": "Alfred", "age": 18, "job": "Butler"}')-
>execute();
$collection->add('{"name": "Bob",      "age": 19, "job": "Painter"}')-
>execute();
$result1 = $collection->find()->execute();
print_r($result1->fetchALL());
echo "^ before, after below\n";

// Add two new jobs for all Painters: Artist and Crafter
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
$collection
->modify("job in ('Butler', 'Painter')")
->arrayAppend('job', 'Artist')
->arrayAppend('job', 'Crafter')
->execute();

// Remove the 'beer' field from all documents with the age 21
$collection
->modify('age < 21')
->unset(['beer'])
->execute();

// And peek at the results
$result2 = $collection->find()->execute();
print_r($result2->fetchAll());
?>
```

Output:

```
Array
(
    [0] => Array
        (
            [_id] => 00005c34d4d8000000000000000027
            [age] => 18
            [job] => Butler
            [name] => Alfred
        )

    [1] => Array
        (
            [_id] => 00005c34d4d8000000000000000028
            [age] => 19
            [job] => Painter
            [name] => Bob
        )

)
^ before, after below
Array
```



```
(
  [0] => Array
    (
      [_id] => 00005c34d4d800000000000000027
      [age] => 18
      [job] => Array
        (
          [0] => Butler
          [1] => Artist
          [2] => Crafter
        )
      [name] => Alfred
    )
  [1] => Array
    (
      [_id] => 00005c34d4d800000000000000028
      [age] => 19
      [job] => Array
        (
          [0] => Painter
          [1] => Artist
          [2] => Crafter
        )
      [name] => Bob
    )
)
```

Remove documents from a collection

```
$ cat collectionremove.php
<?php
$session =
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost");

$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();
$session->sql("CREATE DATABASE addressbook")->execute();
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
$schema      = $session->getSchema("addressbook");
$collection = $schema->createCollection("people");

$collection->add('{"name": "Alfred", "age": 18, "job": "Butler"}')-
>execute();
$collection->add('{"name": "Bob",      "age": 19, "job": "Painter"}')-
>execute();
$collection->add('{"name": "Cal",      "age": 99, "job":
"Presenter"}')->execute();
$collection->add('{"name": "Adam",     "age": 89, "job":
"Organizer"}')->execute();
$collection->add('{"name": "Ada",      "age": 22, "job":
"Programmer"}')->execute();

echo "All records\n";
$result = $collection->find()->execute();
print_r($result->fetchAll());

// Remove all painters
$result = $collection->remove("job in ('Painter')")->execute();

// Remove the oldest butler
$collection
    ->remove("job in ('Butler')")
    ->sort('age desc')
    ->limit(1)
    ->execute();

// Remove record with lowest age
$collection
    ->remove('true')
    ->sort('age desc')
    ->limit(1)
    ->execute();

echo "After we have removed painter, Butler, and youngest\n";
$result = $collection->find()->execute();
print_r($result->fetchAll());
?>
```

Output:

All records

Array

```
(
  [0] => Array
    (
      [_id] => 00005c34d4d80000000000000004d
      [age] => 18
      [job] => Butler
      [name] => Alfred
    )

  [1] => Array
    (
      [_id] => 00005c34d4d80000000000000004e
      [age] => 19
      [job] => Painter
      [name] => Bob
    )

  [2] => Array
    (
      [_id] => 00005c34d4d80000000000000004f
      [age] => 99
      [job] => Presenter
      [name] => Cal
    )

  [3] => Array
    (
      [_id] => 00005c34d4d800000000000000050
      [age] => 89
      [job] => Organizer
      [name] => Adam
    )

  [4] => Array
    (
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
        [_id] => 00005c34d4d800000000000000051
        [age] => 22
        [job] => Programmer
    )
```

```
)
```

After we have removed painter, Butler, and youngest
Array

```
(
    [0] => Array
        (
            [_id] => 00005c34d4d800000000000000050
            [age] => 89
            [job] => Organizer
            [name] => Adam
        )

    [1] => Array
        (
            [_id] => 00005c34d4d800000000000000051
            [age] => 22
            [job] => Programmer
            [name] => Ada
        )

)
```

Intermediate queries

Modifiers

Part of the design of the X DevAPI is the ability to ‘stack’ calls. You are probably used to seeing this type of programming style from Object Orientated Programming.

The following program uses two separate lines to get the schema and then get the collection. Those lines are highlighted below.

```
#!/bin/php
<?php
```

MySQL Document Store Tutorial – Sunshine PHP 2019

```
$session =
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost:33060");
if ($session === NULL) {
    die("Connection could not be established");
}

$schema = $session->getSchema("testxx");
$collection = $schema->getCollection("example1");

var_dump($collection->find("name = 'Mike'")->execute()->fetchOne());

$result = $collection->find()->execute();
foreach ($result as $doc) {
    echo "${doc["name"]} is a ${doc["job"]}.\n";
}
?>
```

The next example combines those two lines into one.

```
#!/bin/php
<?php

$session =
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost:33060");
if ($session === NULL) {
    die("Connection could not be established");
}

$collection = $session->getSchema("testxx")-
>getCollection("example1");

var_dump($collection->find("name = 'Mike'")->execute()->fetchOne());

$result = $collection->find()->execute();
foreach ($result as $doc) {
    echo "${doc["name"]} is a ${doc["job"]}.\n";
}
?>
```

This method of writing calls more closely tracks what modern programming design practices are doing and this chaining of calls is often easier to read. The choice is yours (or your employers programming practice standards choice) but please err on the side of easy to read and comprehend. The rest of the examples in this tutorial will try to follow this design.

Advanced queries

Binding variables

There really are no advanced queries but there are somethings to make life easier. As can be seen in the following example, a variable is assigned a name and preceded with a colon (:) in the find() function and then bind() is used to assign values to those variables. You can use PHP variables within the bind() which helps if you are iterating an array or something similar.

```
$result = $collection
->find('job like :job and age > :age')
->bind(['job' => 'Butler', 'age' => 16])
->execute();
```

Indexing

Indexing allows direct access the records or records desired. Otherwise MySQL will read ALL the rows looking for matches.

```
// Creating a text index
$collection->createIndex(
    'myindex1',
    '{"fields": [{
        "field": "$.name",
        "type": "TEXT(25)",
        "required": true}],
    "unique": false}'
);
```

Each index needs a name so that it can be referenced later. The optimizer will consult the list of available indexes when creating a query plan.

The definition of the index to be created contains several fields contained in an array of IndexField objects, and each object describes a single document member to include in the index, and an optional string for the type of index that might be INDEX (default) or SPATIAL. A single IndexField description consists of the following fields:

- field: string, the full document path to the document member or field to be indexed.
- type: string, definition of the index to create. It contains an array of IndexField objects, and each object describes a single document member to include in the index, and an optional string for the type of index that might be INDEX (default) or SPATIAL, one of the supported SQL column types to map the field into. For numeric types, the optional UNSIGNED keyword may follow. For the TEXT type, the length to consider for indexing may be added.
- required: bool, (optional) true if the field is required to exist in the document. Defaults to FALSE, except for GEOJSON where it defaults to TRUE. If this is a required field the server will reject document without this key.
- options: integer, (optional) special option flags for use when decoding GEOJSON data.
- srid: integer, (optional) srid value for use when decoding GEOJSON data.

Indexes will speed up searching for records however there is overhead for inserting, modifying, and deleting records so index only what you need!

Working with Tables

Traditionally MySQL is a relation database management system which means schemas with relation tables. The X DevAPI works with both tables or collections, and it will be seen later with both at the same time. The biggest difference to remember is that the CRUD function names have analogs with the traditional SQL names – create, select, update, and delete.

Tables

Relational databases have tables while JSON Document Stores have collections. Relational tables are not built around the JSON data type. For those used to traditional Structured Query Language (SQL) database queries, you will find much of this familiar and the X DevAPI function calls for queries will look very familiar.

The query in the program below is based on the SQL query **SELECT Name, District FROM city WHERE District like 'Texas' LIMIT 25;**

\$ cat texas.php

```
#!/bin/php
```

```
<?php
```

```
$session =  
mysql_xdevapi\getSession("mysqlx://root:hidave@localhost:33060");  
if ($session === NULL) {  
    die("Connection could not be established");  
}
```

```
$schema = $session->getSchema("world");  
$table = $schema->getTable("city");
```

```
$row = $table->select('Name','District')  
    ->where('District like :district')  
    ->bind(['district' => 'Texas'])  
    ->limit(25)  
    ->execute()->fetchAll();
```

```
##$row = $result->fetchAll();  
print_r($row);
```

In the below note the use of the **sql()** function.

```
<?php
```

```
$session =
```

```
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost");
```

```
$session->sql("DROP DATABASE IF EXISTS addressbook")->execute();  
$session->sql("CREATE DATABASE addressbook")->execute();  
$session->sql("CREATE TABLE addressbook.names(name text, age int)")-  
>execute();  
$session->sql("INSERT INTO addressbook.names values ('John', 42),  
('Sam', 33)")->execute();
```

```
$schema = $session->getSchema("addressbook");
```


MySQL Document Store Tutorial – Sunshine PHP 2019

```
$table = $schema->getTable("names");

$row = $table->select('name', 'age')
        ->execute()
        ->fetchAll();
print_r($row);

echo "Deleting\n";

$table->delete()
    ->where("age = :id")
    ->bind(['id' => 42])
    ->limit(1)
    ->execute();
echo "After delete\n";
$row = $table->select('name', 'age')->execute()->fetchAll();
print_r($row);
?>
```

There are more examples in the Github repository of using tables with X DevAPI

Working with tables and collections

It is also possible to work with tables and collections at the same time. Working with the different objects in the same code should not be difficult but there may be times when you want to use a collection as a table.

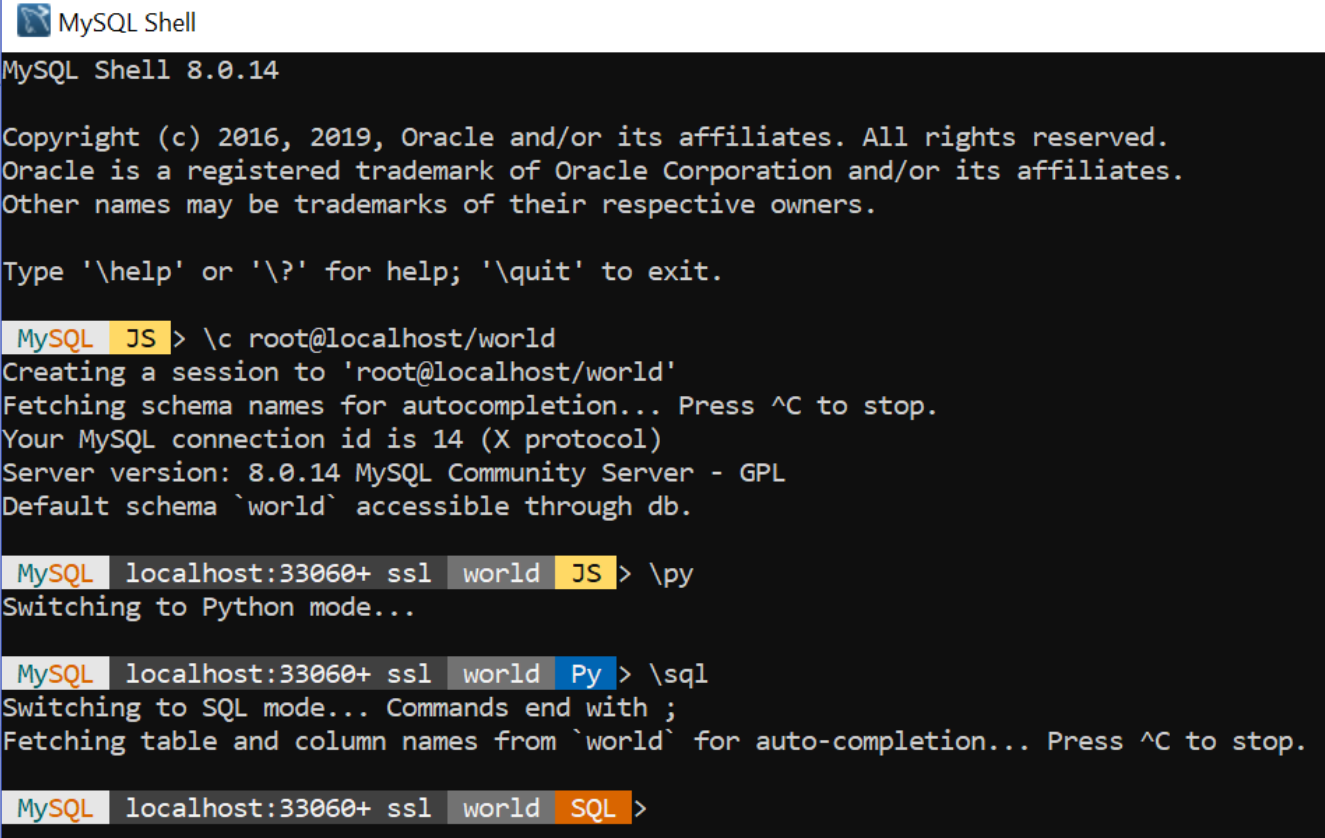
```
$table = $schema->getCollectionAsTable("people");
```

DevAPI versus PDO and MySQLi

With the deprecation of the old *mysql* connector and the introduction of the new X DevAPI connector, there are three connectors for MySQL available from PHP. The PDO connector is open source, and supports many other databases besides MySQL. The MySQLi connector replaced the older MySQL connector, support more MySQL features than PDO, and is supported by Oracle. Finally X DevAPI is the only connector that supports both the old MySQL and the new X DevAPI protocol and is also supported by Oracle.

MySQL Shell to prototype queries

The new MySQL Shell supports both the old MySQL and new MySQL protocols, had many administrative functions (including a server 5.7 → 8.0 upgrade checker, and a JSON document bulk loader), and can also be used for queries. It has three modes – Python, JavaScript, and SQL.



```
MySQL Shell 8.0.14

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.

MySQL JS > \c root@localhost/world
Creating a session to 'root@localhost/world'
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 14 (X protocol)
Server version: 8.0.14 MySQL Community Server - GPL
Default schema `world` accessible through db.

MySQL localhost:33060+ ssl world JS > \py
Switching to Python mode...

MySQL localhost:33060+ ssl world Py > \sql
Switching to SQL mode... Commands end with ;
Fetching table and column names from `world` for auto-completion... Press ^C to stop.

MySQL localhost:33060+ ssl world SQL >
```

The shell can be used to prototype queries, set up schemas & collectors, or manage data. It is also great for ad hoc queries and I highly recommend using it to work out queries before putting them into your PHP code.

You can do a lot of processing in any of the three modes and it is easy to take queries from the shell to PHP. However you have to remember that while the X DevAPI calls are the same between any of the available programming languages that PHP users → instead of a . Instead of a period, like Python and JavaScript.

```
MySQL Shell
MySQL localhost:33060+ ssl so SQL > create database miami;
Query OK, 1 row affected (0.0091 sec)

MySQL localhost:33060+ ssl so SQL > \js
Switching to JavaScript mode...

MySQL localhost:33060+ ssl so JS > \use miami
Default schema `miami` accessible through db.

MySQL localhost:33060+ ssl miami JS > db
<Schema:miami>

MySQL localhost:33060+ ssl miami JS > db.createCollection("beach")
<Collection:beach>

MySQL localhost:33060+ ssl miami JS > db.beach.add(
    -> {
    ->   "name" : "Adam",
    ->   "pasttime" : "running"
    -> }
    -> )
    ->
Query OK, 1 item affected (0.0095 sec)

MySQL localhost:33060+ ssl miami JS > db.beach.find()
[
  {
    "_id": "00005c517afd0000000000000001",
    "name": "Adam",
    "pasttime": "running"
  }
]
1 document in set (0.0008 sec)

MySQL localhost:33060+ ssl miami JS >
```

Both SQL and NoSQL with Analytics

It is possible to deal with NoSQL data as SQL data using the JSON_TABLE function to temporarily convert JSON documents to a relational table. Once the data is in a relational table it is possible to perform advanced analytic analysis on that table.

```
#!/bin/php
<?php

$session =
mysql_xdevapi\getSession("mysqlx://myuser:mypass@localhost:33060");

if ($session === NULL) {
    die("Connection could not be established");
}

$schema = $session->getSchema("nyeats");
$table   = $schema->getTable("restaurants");

$sqlx =
` WITH cte1 AS (SELECT doc->>"$.name" AS name,
doc->>"$.cuisine" AS cuisine,          (SELECT AVG(score) FROM
JSON_TABLE(doc, "$.grades[*]" COLUMNS (score INT PATH "$.score")) AS
r) AS avg_score FROM restaurants) SELECT *, RANK() OVER (PARTITION
BY cuisine ORDER BY avg_score DESC) AS `rank` FROM cte1 ORDER BY
`rank`, avg_score DESC LIMIT 10;`

$row->sql($sqlx)->execute()->fetchAll();

print_r($row);
```

In the above example the restaurants collection is converted to a temporary relational table with the JSON_TABLE function, that relational table is queried with a CTE compute the average rating, and a Window Function is used to rank the data.

Result Grid Filter Rows: Export: Wrap Cell Content:				
	name	cuisine	avg_score	rank
▶	Juice It Health Bar	Juice, Smoothies, Fruit Salads	75.0000	1
	Golden Dragon Cuisine	Chinese	73.0000	1
	Palombo Pastry Shop	Bakery	69.0000	1
	Go Go Curry	Japanese	65.0000	1
	K & D Internet Inc	Café/Coffee/Tea	61.0000	1
	Koyla	Middle Eastern	61.0000	1
	Ivory D O S Inc	Other	60.0000	1
	Espace	American	56.0000	1
	Rose Pizza	Pizza	52.0000	1
	Tacos Al Suadero	Mexican	52.0000	1

Wrap-up

This workshop is a work in progress and I will update so check back on the

Github from time to time for changes. And feel free to make suggestions to make this a better tutorial.

If you have questions do not hesitate to contact me or the X DevAPI forum on

<https://forums.mysql.com/list.php?176>

And what if you want to use the new protocol but do not want to rewrite queries? Encapsulate your old queries in a SQL() function.

```
$row->sql("SELECT Name FROM city WHERE CountryCode = 'USA'")
    ->execute()
    ->fetchAll();
```

Appendix

Using MySQL 8 with Doctrine

It is possible to use MySQL 8 with the Doctrine ORM and the tl;dr is you need to configure Doctrine to use DATABASE_URL=mysql://account:[password@localhost](#):3306/databasename and not use 127.0.0.1 ! See more at <https://elephantdolphin.blogspot.com/2018/11/doctrine-and-mysql-8-odd-connection.html>

JSON Format Specification

The JavaScript Object Notation (JSON) Data Interchange Format <https://tools.ietf.org/html/rfc7159>

The PHP PECL MySQL X DevAPI package

https://pecl.php.net/package/mysql_xdevapi

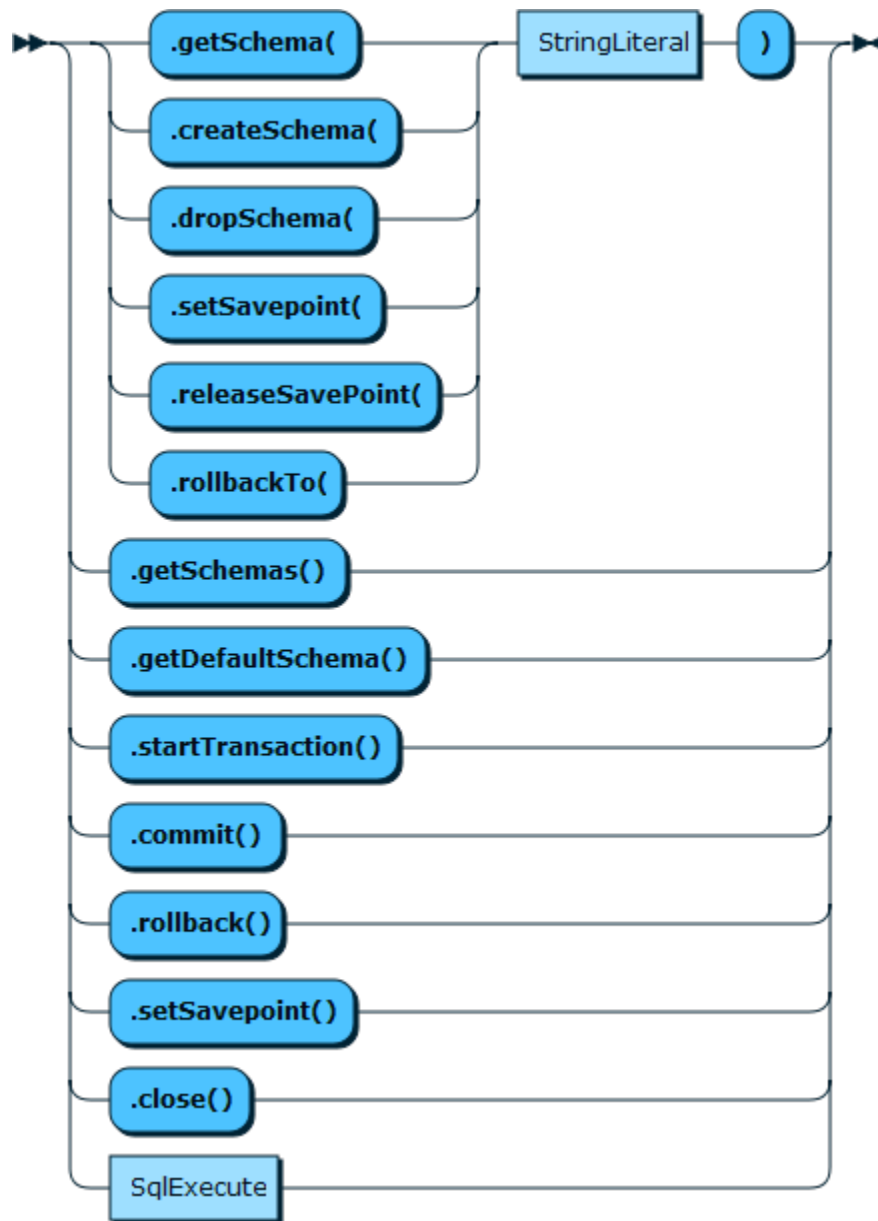
The PHP MySQL X DevAPI Manual

<http://php.net/manual/en/book.mysql-xdevapi.php>

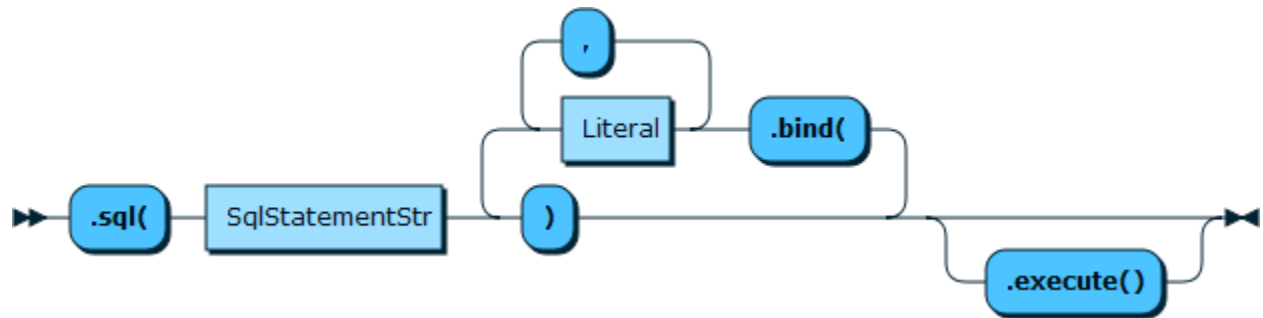
Railroad Diagrams

Railroad Diagrams are an easy way to explore a given function and a great reference.

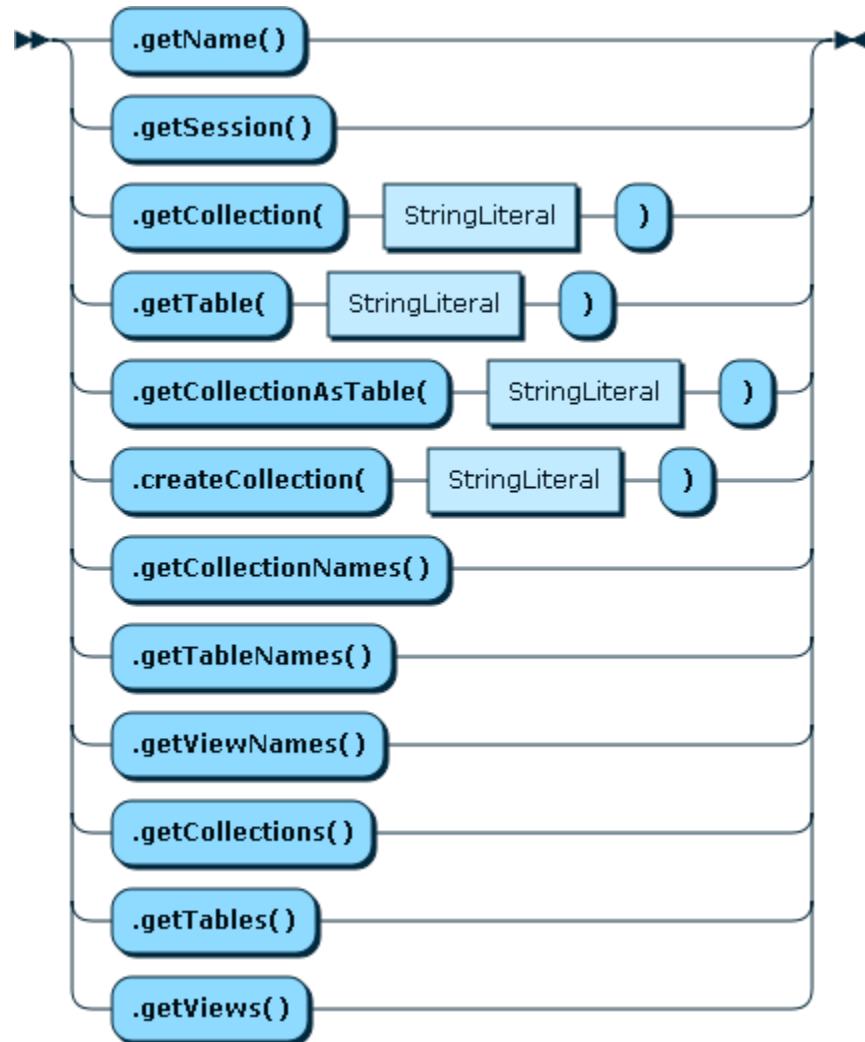
Sessions



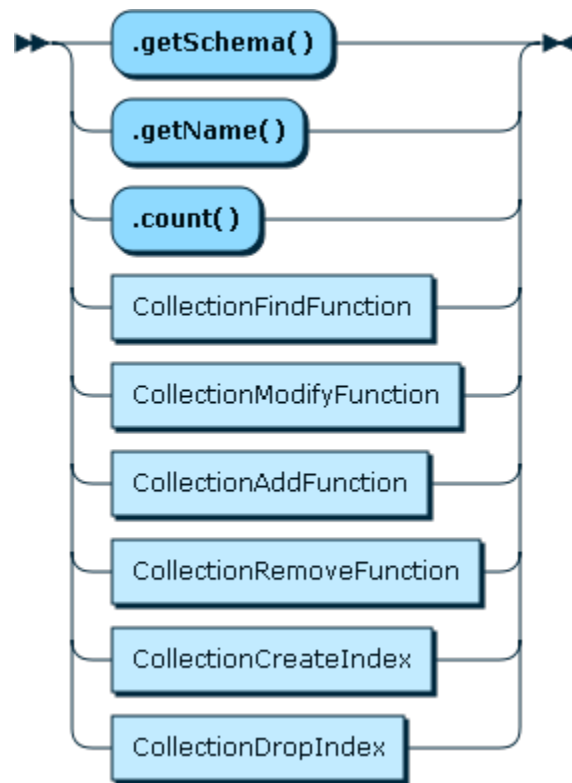
SQL Execute



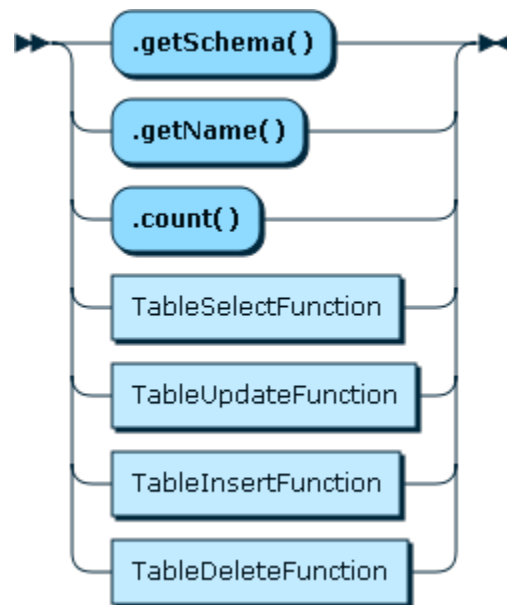
Schema



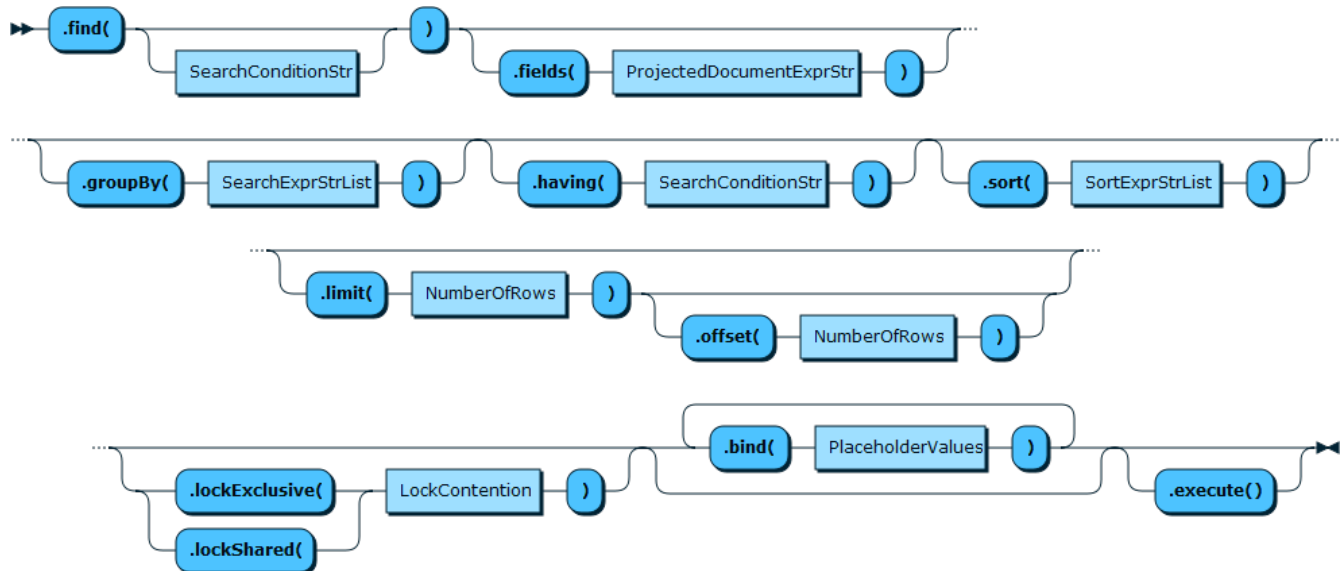
Collection



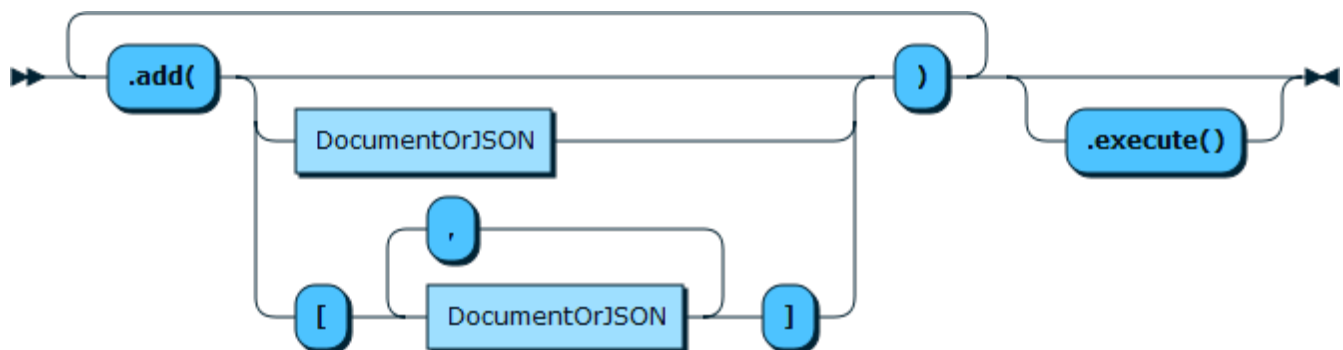
Table



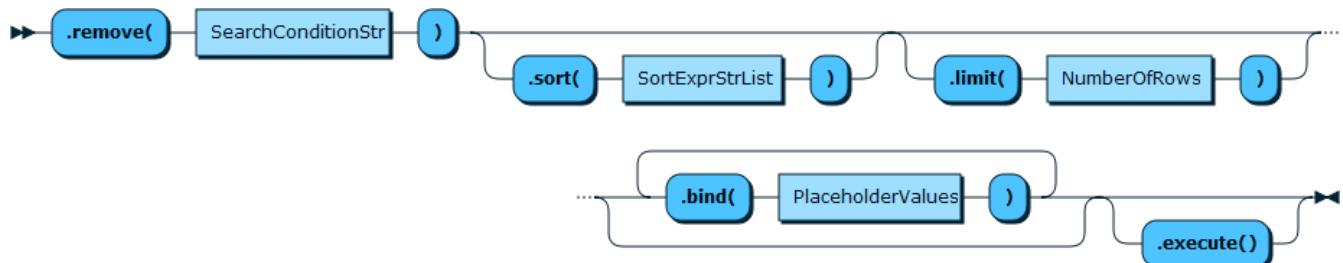
Find collection



Add collection



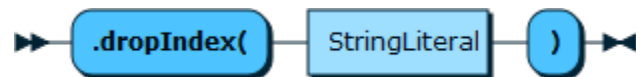
Remove collection



Create Index Collection

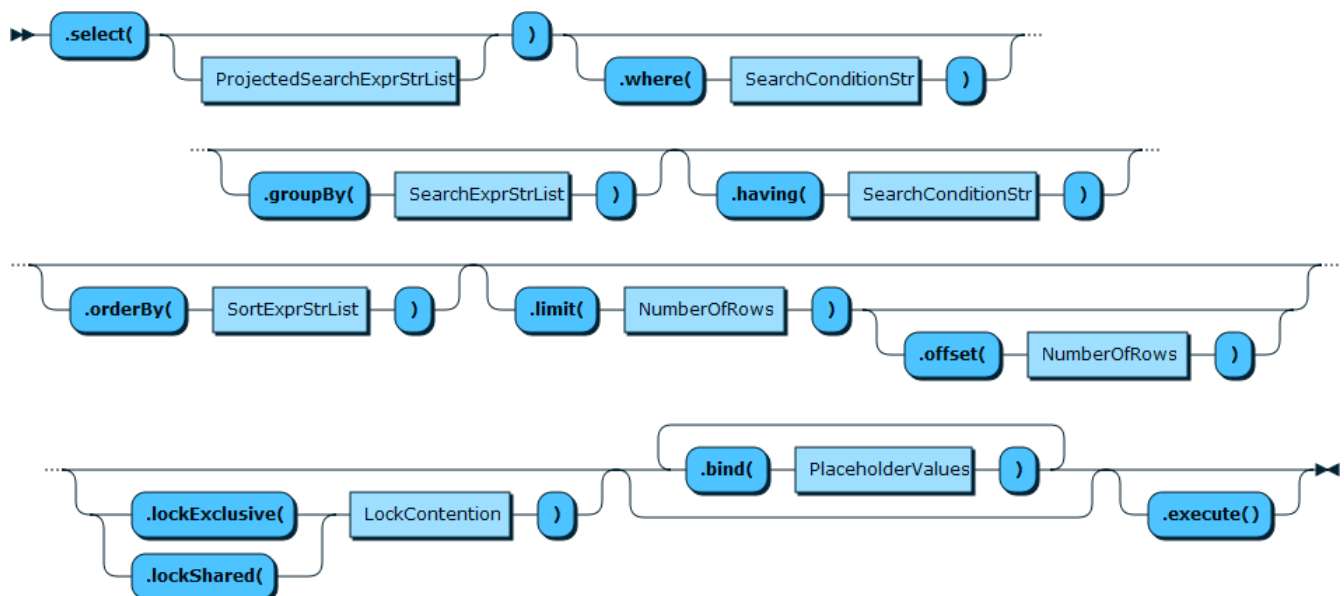


Drop Index Collection

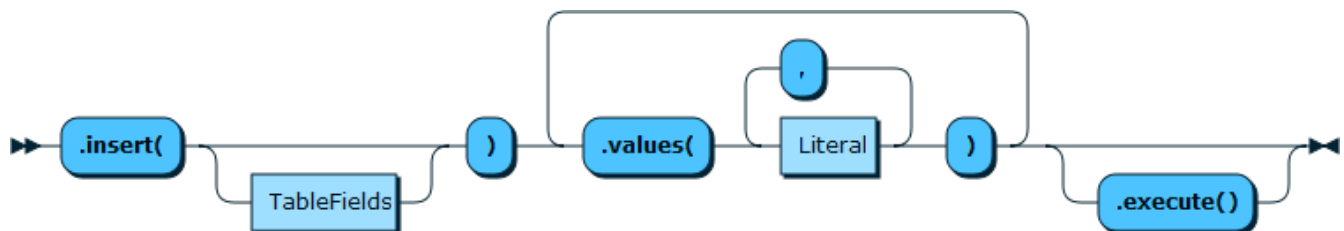


Tables:

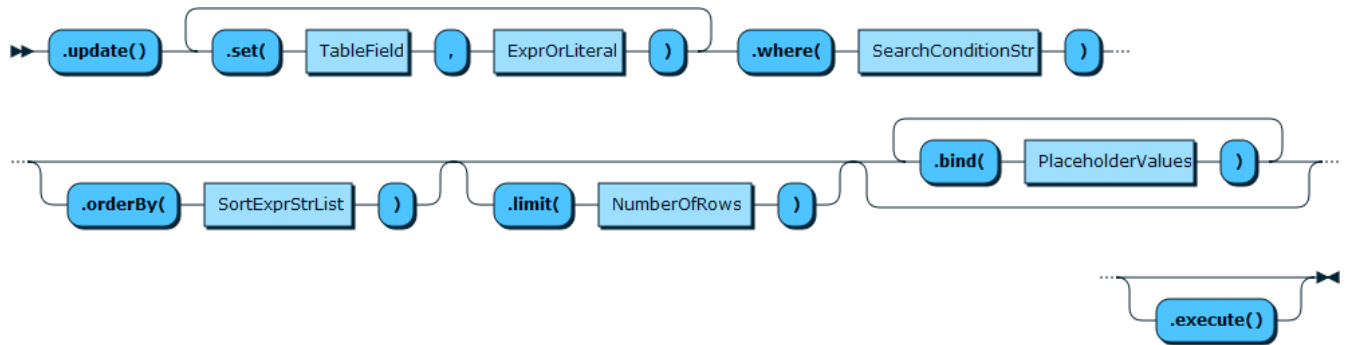
Select table



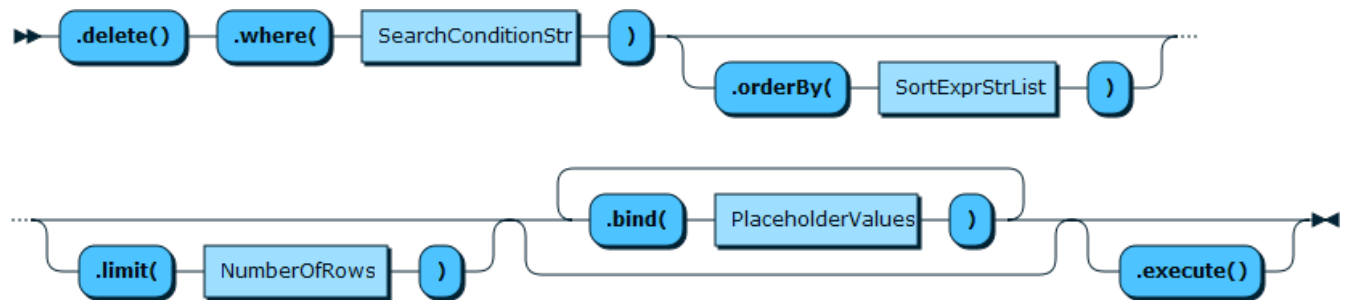
Insert table



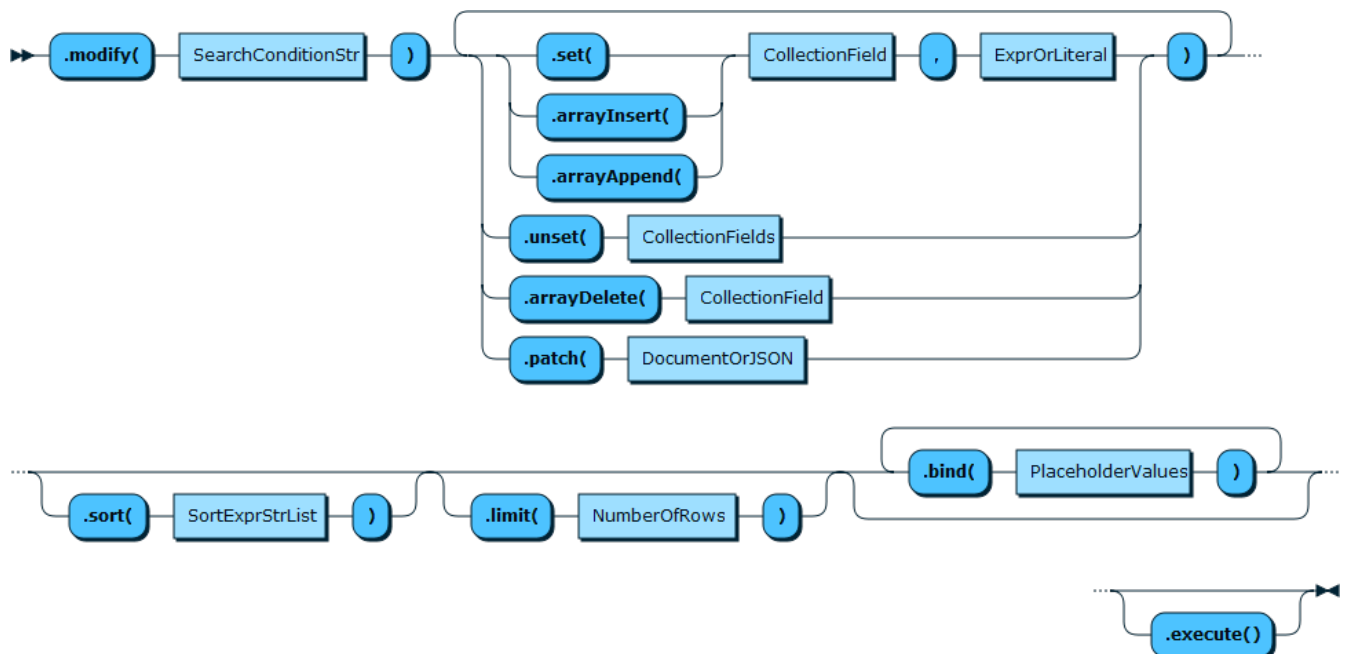
Update table



Delete Table



Modify collections



Sample Database Data Sets

MySQL Example Datasets -

<https://dev.mysql.com/doc/index-other.html>

MongoDB Primer Dataset -

<https://github.com/OpenKitten/Mongo-Assets/blob/master/primer-dataset.json>

Quickly Load JSON Data into The MySQL Document Store with `util.importJson`

With new MySQL Shell 8.0.13 comes a new way to quickly load JSON data sets very quickly. In a past blog and in several talks I have shown how to use the shell with the Python mode to pull in the data. But now there is a much faster way to load JSON

Load JSON Quickly

Start a copy of the new shell with `mysqlsh`. Connect to your favorite server `\c dave@localhost` and then create a new schema `session.createSchema('bulk')`. Then point you session to the schema just created with `\use bulk`. Version 8.0.13 has a new utility function named `importJson` that does the work. The first argument is the path to the data set (here the MongoDB restaurant collection) and the second allows you to designate the schema and collection where you wish to have the data stored. In this example the data set was in the downloads directory of my laptop and I wanted to put it in the newly created 'bulk' schema in a collection named 'restaurants'

```
MySQL localhost:33060+ ssl bulk JS > util.importJson("C:/Users/dstokes/Downloads/primer-dataset.json",{schema: "bulk", collection: "restaurants"})
Importing from file "C:/Users/dstokes/Downloads/primer-dataset.json" to collection `bulk`.`restaurants` in MySQL Server at localhost:33060
.. 25359
Processed 11.85 MB in 25359 documents in 2.2091 sec (11.48K documents/s)
Total successfully imported documents 25359 (11.48K documents/s)
```

An Example of using `util.importJson` to quickly load JSON data into the MySQL Document Store

It took just over 2 seconds to read in over 25,000 records, not bad.

```
MySQL localhost:33060+ ssl bulk JS > db.restaurants.find("cuisine = 'Italian'").fields(["name"]).limit(3)
[
  {
    "name": "Philadelphia Grille Express"
  },
  {
    "name": "Isle Of Capri Resturant"
  },
  {
    "name": "Marchis Restaurant"
  }
]
3 documents in set (0.0019 sec)
```

And a quick check of the data shows that is loaded successfully!

The Document `_id`

Every document needs a unique identifier called the document ID. The document ID value is usually automatically generated by the server when the document is created however can also be manually assigned. The assigned document ID is returned in the result of the `collection.add()` operation.

The X DevAPI relies on server based document ID generation, added in MySQL version 8.0.11, which results in *sequentially* increasing document IDs across all clients which is vital for InnoDB Cluster operations. InnoDB uses this document ID as a primary key and provides sequential primary keys for all clients resulting in efficient page splits and tree reorganizations.

The `_id` field of a document behaves in the same way as any other field of the document during queries, except that its value cannot change once inserted to the collection. The `_id` field is used as the primary key of the collection (using stored generated columns). It is possible to override the automatic generation of document IDs by manually including an ID in an inserted document.

If you are using manual document IDs, it is vital that IDs from the server's automatically generated document ID sequence are never used twice. The X Plugin is not aware of the data inserted into the collection, including any IDs you use. If the document ID which you assigned manually when inserting a document uses an ID which the server was going to use, the insert operation fails with an error due to primary key duplication.

If an `_id` field value is not present in an inserted document, the server generates an `_id` value. The generated `_id` value used for a document is returned to the client as part of the document insert Result message. If you are using X DevAPI on an InnoDB cluster, the automatically generated `_id` must be unique within the a replica set of the cluster. Use the `mysqlx_document_id_unique_prefix` option to ensure that the `unique_prefix` part of the document ID is unique to the replicaset or group.

The `_id` field must be sequential (always incrementing) for optimal InnoDB insertion performance (at least within a single server). The sequential nature of `_id` values is maintained across server restarts.

In a multi-primary Group Replication or InnoDB cluster environment, the generated `_id` values of a table are unique across instances to avoid primary key conflicts and minimize transaction certification.

Document ID Generation

The general structure of the collection table remains unchanged, except for the type of the generated `_id` column, which changes from `VARCHAR(32)` to `VARBINARY(32)`.

The format of automatically generated document ID is:

unique_prefix	start_timestamp	serial
4 bytes	8 bytes	16 bytes

Where:

- serial is a per-instance automatically incremented integer serial number 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.
- start_timestamp is the time stamp of the startup time of the server instance, which is hex encoded. In the unlikely event that the value of serial overflows, the start_timestamp is incremented by 1 and the serial value then restarts at 0.
- unique_prefix is a value assigned by InnoDB cluster to the instance, which is used to make the document ID unique across all replicaset from the same cluster. The range of unique_prefix is from 0 to $2^{16}-1$, which is hex encoded, and defaults to 0 if not set by InnoDB cluster or the `mysqlx_document_id_unique_prefix` system variable has not been configured.

This document ID format ensures that:

- The primary key value monotonically increments for inserts originating from a single server instance, although the interval between values is not uniform within a table.
- When using multi-primary Group Replication or InnoDB cluster, 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.

Creating MySQL Accounts

Account Setup is very important and the granting of privileges must be done before attempting to access data. The following is a quick example of 1) creating an account, 2) setting the password, and 3) granting permissions to use a schema.

- `mysql> CREATE USER userx@localhost`
 `-> IDENTIFIED WITH 'mysql_native_password';`
 Query OK, 0 rows affected (0.28 sec)
- `mysql> SET PASSWORD for 'userx'@'localhost' ='Userx123!';`
 Query OK, 0 rows affected (0.41 sec)
- `mysql> GRANT ALL ON addressbook.* TO 'userx'@'localhost';`
 Query OK, 0 rows affected (0.19 sec)

Example Files on Github

address.php
collectionadd.php
collectionastable.php
collectioncount.php
collectionexist.php
collectionfind.php
collectionfind2.php
collectionfind3.php
collectionmodify.php
collectionremove.php
collectionaddorreplaced.php
createschema.php
createschema2.php
exampledoc.php
getsession.php
isaview.php
listschemas.php
nyeats.php
schema001.php
simpleopensession.php
sunshine01.php
tablecolumncount.php
tableinsertwithsql.php
tablenames.php
tableselect.php
texas.php
warnings.php

Table of Contents

Ground Rules.....	1
The X Plugin.....	2
The X Protocol.....	3
The X DevAPI.....	4
The MySQL Document Store is a JSON based NoSQL database.....	6
Architecture.....	7
The ‘Under The Hood’ View.....	9
The JSON Data Type.....	9
JSON Functions.....	11
Partial Updates of JSON Data.....	12
My Book on MySQL’s JSON Data type.....	13
X DevAPI.....	14
CRUD Operations Overview.....	14
No longer only SQL based.....	15
Modern Program Design.....	16
The X DevAPI PECL Extension.....	17
How to build.....	18
The basics of the MySQL Document Store.....	19
Program flow.....	19
Connector URI.....	19
Example X DevAPI URIs.....	19
An Example of URI Usage.....	20
A Quick Example – sunshine01.php.....	20
A Second Example.....	22
A Quick Diversion on Connections.....	23
Third Example.....	23
Collections.....	26
Creating (and deleting collections).....	26
Adding documents.....	26
Counting records in collections.....	28
Finding records in a collections.....	29
Collection: modify a documents.....	31
Remove documents from a collection.....	33
Intermediate queries.....	36
Modifiers.....	36
Advanced queries.....	38
Binding variables.....	38
Indexing.....	38
Working with Tables.....	39
Tables.....	39
Working with tables and collections.....	41
DevAPI versus PDO and MySQLi.....	41

MySQL Shell to prototype queries.....	42
Both SQL and NoSQL with Analytics.....	44
Wrap-up.....	45
Appendix.....	46
Using MySQL 8 with Doctrine.....	46
JSON Format Specification.....	46
The PHP PECL MySQL X DevAPI package.....	46
The PHP MySQL X DevAPI Manual.....	46
Railroad Diagrams.....	46
Sessions.....	47
SQL Execute.....	48
Schema.....	49
Collection.....	50
Table.....	51
Find collection.....	52
Add collection.....	52
Remove collection.....	52
Create Index Collection.....	53
Drop Index Collection.....	53
Tables:.....	53
Select table.....	53
Insert table.....	53
Update table.....	54
Delete Table.....	54
Modify collections.....	54
Sample Database Data Sets.....	55
MySQL Example Datasets -.....	55
https://dev.mysql.com/doc/index-other.html	55
MongoDB Primer Dataset -.....	55
https://github.com/OpenKitten/Mongo-Assets/blob/master/primer-dataset.json	55
Quickly Load JSON Data into The MySQL Document Store with util.importJson.....	56
Load JSON Quickly.....	56
The Document _id.....	57
Document ID Generation.....	57
Creating MySQL Accounts.....	59
Example Files on Github.....	59