X DevAPI User Guide

Abstract

User documentation for developers using X DevAPI.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Document generated on: 2020-03-17 (revision: 65332)

Table of Contents

Preface and Legal Notices

This is the X DevAPI User Guide.

Legal Notices

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Chapter 1 Overview

This guide explains how to use the X DevAPI and provides examples of its functionality. The X DevAPI is implemented by MySQL Shell and MySQL Connectors that support X Protocol. For more background information and instructions on how to install and get started using X DevAPI, see Using MySQL as a Document Store. For quick-start tutorials introducing you to X DevAPI, see JavaScript Quick-Start Guide: MySQL Shell for Document Store and Python Quick-Start Guide: MySQL Shell for Document Store.

This section introduces the X DevAPI and provides an overview of the features available when using it to develop applications.

Important

The X DevAPI implementation in MySQL Shell can differ from the implementation in the Connector products. This guide provides an overview of using the concepts in all X DevAPI implementations.

The X DevAPI wraps 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. See Chapter 2, *Connection and Session Concepts*.
- 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. See Chapter 4, *Working with Collections* and Chapter 5, *Working with Documents*.
- Work with your existing domain objects or generate code based on structure definitions for strictly typed languages. See Chapter 5, *Working with Documents*.
- 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* for details.

Chapter 2 Connection and Session Concepts

Table of Contents

This section explains the concepts of connections and sessions as used by the X DevAPI. Code examples for connecting to a MySQL Document Store (see Using MySQL as a Document Store) and using sessions are provided.

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.

For 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.

Before looking at the concepts in more detail, the following examples show how to connect using a session.

2.1 Database Connection Example

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 an X DevAPI session using the `mysqlx.getSession(connection)` method. You pass in the connection parameters to connect to the MySQL server, such as the hostname and user, 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}`. See Connecting to the Server Using URI-Like Strings or Key-Value Pairs for more information.

The MySQL user account used for the connection should use either the `mysql_native_password` or `caching_sha2_password` authentication plugin, see Pluggable Authentication. 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 following example code shows how to connect to a MySQL server and get a document from the `my_collection` collection that has the field `name` starting with `L`. The example assumes that a schema called `test` exists, and the `my_collection` collection exists. To make the example work, replace `user` with your username, and *`password`* with your password. If you are connecting to a different host or through a different port, change the host from `localhost` and the port from `33060`.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

// Connect to server on localhost
var mySession = mysqlx.getSession( {
                host: 'localhost', port: 33060,
                user: 'user', password: 'password' } );

var myDb = mySession.getSchema('test');

// Use the collection 'my_collection'
var myColl = myDb.getCollection('my_collection');

// Specify which document to find with Collection.find() and
// fetch it from the database with .execute()
var myDocs = myColl.find('name like :param').limit(1).
        bind('param', 'L%').execute();

// Print document
print(myDocs.fetchOne());

mySession.close();
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

# Connect to server on localhost
mySession = mysqlx.get_session( {
        'host': 'localhost', 'port': 33060,
        'user': 'user', 'password': 'password' } )

myDb = mySession.get_schema('test')

# Use the collection 'my_collection'
myColl = myDb.get_collection('my_collection')

# Specify which document to find with Collection.find() and
# fetch it from the database with .execute()
myDocs = myColl.find('name like :param').limit(1).bind('param', 'L%').execute()

# Print document
document = myDocs.fetch_one()
print(document)

mySession.close()
```

Node.js JavaScript Code

```
var mysqlx = require('@mysql/xdevapi');

// Connect to server on localhost
mysqlx
  .getSession({
    user: 'user',
    password: 'password',
    host: 'localhost',
    port: '33060'
  })
  .then(function (session) {
    var db = session.getSchema('test');
    // Use the collection 'my_collection'
    var myColl = db.getCollection('my_collection');
    // Specify which document to find with Collection.find() and
    // fetch it from the database with .execute()
    return myColl
      .find('name like :param')
      .limit(1)
      .bind('param', 'L%')
      .execute(function (doc) {
        console.log(doc);
      });
  })
  .catch(function (err) {
    // Handle error
  });
```

C# Code

```
// Connect to server on localhost
var mySession = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");

var myDb = mySession.GetSchema("test");

// Use the collection "my_collection"
var myColl = myDb.GetCollection("my_collection");

// Specify which document to find with Collection.Find() and
// fetch it from the database with .Execute()
var myDocs = myColl.Find("name like :param").Limit(1)
 .Bind("param", "L%").Execute();

// Print document
Console.WriteLine(myDocs.FetchOne());

mySession.Close();
```

Python Code

```
import mysqlx

# Connect to server on localhost
my_session = mysqlx.get_session({
     'host': 'localhost', 'port': 33060,
     'user': 'user', 'password': 'password'
 })

my_schema = my_session.get_schema('test')

# Use the collection 'my_collection'
my_coll = my_schema.get_collection('my_collection')

# Specify which document to find with Collection.find() and
# fetch it from the database with .execute()
docs = my_coll.find('name like :param').limit(1).bind('param', 'L%').execute()

# Print document
doc = docs.fetch_one()
print(doc)

my_session.close()
```

Java Code

```
import com.mysql.cj.xdevapi.*;

// Connect to server on localhost
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=p

Schema myDb = mySession.getSchema("test");

// Use the collection 'my_collection'
Collection myColl = myDb.getCollection("my_collection");

// Specify which document to find with Collection.find() and
// fetch it from the database with .execute()
DocResult myDocs = myColl.find("name like :param").limit(1).bind("param", "L%").execute();

// Print document
System.out.println(myDocs.fetchOne());

mySession.close();
```

C++ Code

```
#include <mysqlx/xdevapi.h>

// Scope controls life-time of objects such as session or schema

{
  Session sess("localhost", 33060, "user", "password");
  Schema db= sess.getSchema("test");
  // or Schema db(sess, "test");

  Collection myColl = db.getCollection("my_collection");
  // or Collection myColl(db, "my_collection");

  DocResult myDocs = myColl.find("name like :param")
                           .limit(1)
                           .bind("param","L%").execute();

  cout << myDocs.fetchOne();
}
```

2.2 Connecting to a Session

There are several ways of using a session to connect to MySQL depending on the specific setup in use. This section explains the different methods available.

2.2.1 Connecting to a Single MySQL Server

In this example a connection to a local MySQL Server instance running X Plugin on the default TCP/IP port 33060 is established using the MySQL user account *user* with its password. As no other parameters are set, default values are used.

MySQL Shell JavaScript Code

```
// Passing the parameters in the { param: value } format
var dictSession = mysqlx.getSession( {
        host: 'localhost', 'port': 33060,
        user: 'user', password: 'password' } )

var db1 = dictSession.getSchema('test')

// Passing the parameters in the URI format
var uriSession = mysqlx.getSession('user:password@localhost:33060')

var db2 = uriSession.getSchema('test')
```

MySQL Shell Python Code

```
# Passing the parameters in the { param: value } format
dictSession = mysqlx.get_session( {
        'host': 'localhost', 'port': 33060,
        'user': 'user', 'password': 'password' } )

db1 = dictSession.get_schema('test')

# Passing the parameters in the URI format
uriSession = mysqlx.get_session('user:password@localhost:33060')

db2 = uriSession.get_schema('test')
```

The following example shows how to connect to a single MySQL Server instance by providing a TCP/IP address “localhost” and the same user account as before. You are prompted to enter the username and password in this case.

MySQL Shell JavaScript Code

```
// Passing the parameters in the { param: value } format
// Query the user for the account information
print("Please enter the database user information.");
var usr = shell.prompt("Username: ", {defaultValue: "user"});
var pwd = shell.prompt("Password: ", {type: "password"});

// Connect to MySQL Server on a network machine
mySession = mysqlx.getSession( {
        host: 'localhost', 'port': 33060,
        user: usr, password: pwd} );

myDb = mySession.getSchema('test');
```

MySQL Shell Python Code

```
# Passing the parameters in the { param: value } format
# Query the user for the account information
print("Please enter the database user information.")
usr = shell.prompt("Username: ", {'defaultValue': "user"})
pwd = shell.prompt("Password: ", {'type': "password"})

# Connect to MySQL Server on a network machine
mySession = mysqlx.get_session( {
        'host': 'localhost', 'port': 33060,
        'user': usr, 'password': pwd} )

myDb = mySession.get_schema('test')
```

Node.js JavaScript Code

```
// Passing the parameters in the { param: value } format
mysqlx.getSession({ host: 'localhost', port: 33060, user: 'user', password: 'password' })
  .then(function (dictSession) {
    var db1 = dictSession.getSchema('test')
  })
// Passing the parameters in the URI format
mysqlx.getSession('user:password@localhost:33060')
  .then(function (uriSession) {
    var db2 = uriSession.getSchema('test')
  })
```

C# Code

```
// Query the user for the user information
Console.WriteLine("Please enter the database user information.");
Console.Write("Username: ");
var usr = Console.ReadLine();
Console.Write("Password: ");
var pwd = Console.ReadLine();

// Connect to server on localhost using a connection URI
var mySession = MySQLX.GetSession(string.Format("mysqlx://localhost:33060/test?user={0}&password={1}",

var myDb = mySession.GetSchema("test");
```

Python Code

```
# Passing the parameters in the { param: value } format
dict_session = mysqlx.get_session({
    'host': 'localhost', 'port': 33060,
    'user': 'user', 'password': 'password'
})

my_schema_1 = dict_session.get_schema('test')
```

```
# Passing the parameters in the URI format
uri_session = mysqlx.get_session('user:password@localhost:33060')

my_schema_2 = uri_session.get_schema('test')
```

Java Code

```
import com.mysql.cj.xdevapi.*;

// Connect to server on localhost using a connection URI
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=passw

Schema myDb = mySession.getSchema("test");
```

C++ Code

```
// This code sample assumes that we have function prompt() defined somewhere.

string usr = prompt("Username:");
string pwd = prompt("Password:");

// Connect to MySQL Server on a network machine
Session mySession(SessionOption::HOST, "localhost",
                  SessionOption::PORT, 33060,
                  SessionOption::USER, usr,
                  SessionOption::PWD, pwd);

// An alternative way of defining session settings.

SessionSettings settings(SessionOption::HOST,"localhost",
                         SessionOption::PORT, 33060);

settings.set(SessionOption::USER, usr);
settings.set(SessionOption::PWD, pwd);

Session mySession(settings);

Schema myDb= mySession.getSchema("test");
```

2.2.2 Connecting to a Single MySQL Server Using Connection Pooling

X DevAPI supports connection pooling, which can reduce overhead for applications that open many connections to a MySQL Server. Connections are managed as a pool by a Client object. When opening a new Session with a Client, before a new network connection is opened, an attempt is made to retrieve from the pool an existing and currently unused connection, which is then reset and reused.

A connection pool is configured using a single key-value pair (see Connecting Using Key-Value Pairs) that contains a single key named `pooling`. The value for the `pooling` key is another set of key-value pairs containing any combination of the keys described in the following table:

Table 2.1 Options for Configuring a Connection Pool

Option	Meaning	Default
enabled	Connection pooling enabled. When the option is set to `false`, a regular, non-pooled connection is returned, and the other connection pool options listed below are ignored.	true
maxSize	The maximum number of connections allowed in the pool	25
maxIdleTime	The maximum number of milliseconds a connection is allowed to idle in the queue before being closed. A zero value means infinite.	0

Option	Meaning	Default
queueTimeout	The maximum number of milliseconds a request is allowed to wait for a connection to become available. A zero value means infinite	0

Closing the Session marks the underlying connection as unused and returns it to the Client object's connection pool.

Closing the Client object closes all connections it manages, invalidates all Sessions the Client has created, and destroys the managed pool.

Note

Connection pooling is not supported by MySQL Shell.

Node.js JavaScript Code

```
var mysqlx = require('@mysql/xdevapi');
var client = mysqlx.getClient(
  { user: 'root', host: 'localhost', port: 33060 },
  { pooling: { enabled: true, maxIdleTime: 30000, maxSize: 25, queueTimeout: 10000 } }
);
client.getSession()
  .then(session => {
    console.log(session.inspect())
    return session.close() // the connection becomes idle in the client pool
  })
  .then(() => {
    return client.getSession()
  })
  .then(session => {
    console.log(session.inspect())
    return client.close() // closes all connections and destroys the pool
  })
```

C# Code

```
using (Client client = MySQLX.GetClient("server=localhost;user=root:port=33060;",
  new { pooling = new { Enabled = true, MaxSize = 100, MaxIdleTime=30000, QueueTimeout = 10000 } }))
   {
      using (Session session = client.GetSession())
      {
         foreach (Collection coll in session.Schema.GetCollections())
         {
            Console.WriteLine(coll.Name);
         }
      } // session.Dispose() is called and the session becomes idle in the pool
   } // client.Dispose() is called then all sessions are closed and pool is destroyed
```

Python Code

```
connection_string = {
    'host': 'localhost',
    'port': 37210,
    'user': 'user',
    'password': 'password'
}
client_options = {
    'pooling': {
        "max_size": 10,
        "max_idle_time": 30000
    }
}
client = mysqlx.get_client(connection_string, client_options)
session1 = client.get_session()
session2 = client.get_session()

# closing all the sessions
```

```
client.close()
```

Java Code

```
//Obtain new ClientFactory
ClientFactory cf = new ClientFactory();

//Obtain Client from ClientFactory
Client cli = cf.getClient(this.baseUrl, "{\"pooling\":{\"enabled\":true, \"maxSize\":8,
  \"maxIdleTime\":30000, \"queueTimeout\":10000} }");
Session sess = cli.getSession();

//Use Session as usual

//Close Client after use
cli.close();
```

C++ Code

```
using namespace mysqlx;

Client cli("user:password@host_name/db_name", ClientOption::POOL_MAX_SIZE, 7);
Session sess = cli.getSession();

// use Session sess as usual

cli.close();  // close all Sessions
```

Connector/C++ Code using X DevAPI for C

```
char error_buf[255];
int  error_code;

mysqlx_client_t *cli
 = mysqlx_get_client_from_url(
     "user:password@host_name/db_name", "{ \"maxSize\": 7 }", error_buf, &error_code
   );
mysqlx_session_t *sess = mysqlx_get_session_from_client(cli);

// use sess as before

mysqlx_close_client(cli);  // close session sess
```

2.2.3 Connections Using DNS SRV Records

X DevAPI supports the use of DNS SRV records for connections. In the Domain Name System (DNS), SRV records (service location records) are a type of resource record that enables a client to specify a protocol and domain for a connection, and get a reply with the names of multiple available servers in the domain that provide the required service. The client then attempts to connect to the listed hosts in order of preference, based on the priority and weighting that was assigned to each of them by the DNS administrator in the SRV records.

Where multiple MySQL instances, such as a cluster of servers, can provide the same service for your applications, DNS SRV records can be used to assist with failover, load balancing, and replication services. They remove the need for clients to identify each possible host in the connection string, or for connections to be handled by an additional software component. They can also be updated centrally by administrators when servers are added or removed from the configuration or when their host names are changed. DNS SRV records can be used in combination with connection pooling, in which case connections to hosts that are no longer in the current list of SRV records are removed from the pool when they become idle.

MySQL Connectors which implement X DevAPI can request SRV record lookup by specifying `mysqlx+srv` as the scheme element of the URI-like connection string, for example:

```
mysqlx+srv://example.com/db?options
```

X DevAPI's `mysqlx.getSession()` method, and the `mysqlx.getClient()` method for connection pooling, validate connection information with this protocol scheme extension, and handle the resulting SRV records as a list of hosts for the purposes of failover behavior and connection pooling. The priority and weighting specified in the SRV records are respected.

MySQL Connectors also have connector-specific options to request SRV record lookup both for X Protocol connections and for classic MySQL protocol connections. For details, see the documentation for the individual MySQL Connector.

Note

DNS SRV records are not currently supported by MySQL Shell.

When SRV record lookup is used, clients must apply these rules for the connection:

1. The client must specify the full name for the service as given in the DNS SRV records. In SRV records, the name of the service and of the protocol are prepended with underscores. For example, these SRV records relate to a service using X Protocol that can be provided by four servers in the installation:

   ```
   Record                    TTL   Class     Priority Weight Port  Target
   _mysqlx._tcp.example.com. 86400 IN SRV    0        5      33060 server1.example.com.
   _mysqlx._tcp.example.com. 86400 IN SRV    0        10     33060 server2.example.com.
   _mysqlx._tcp.example.com. 86400 IN SRV    10       5      33060 server3.example.com.
   _mysqlx._tcp.example.com. 86400 IN SRV    20       5      33060 server4.example.com.
   ```

   A client can request this service using syntax like this:

   ```
   var client = mysqlx.getClient("mysqlx+srv://_mysqlx._tcp.example.com")
   ```

2. The client must not specify a port number in the URI-like connection string or in the connection options.

3. Unix socket files and Windows named pipes cannot be used.

4. Multiple host names cannot be specified in the connection request (with some connector-specific exceptions).

Java Code

```
Session mySession = new
SessionFactory().getSession("mysqlx+srv://user:password@_mysql._tcp.example.com/db");
```

Node.js JavaScript Code

```
mysqlx.getSession({ host: '_mysqlx._tcp.example.com', resolveSrv: true })
```

C# Code

```
var session = MySQLX.GetSession("mysqlx+srv://user:password@_mysqlx._tcp.example.com.");
```

Connector/C++ Code using X DevAPI for C

```
mysqlx::Session sess(
    SessionOption::HOST, "tcp.example.com",
    SessionOption::DNS_SRV, true,
    SessionOption::USER, "user",
    SessionOption::PWD, "password");
```

Python Code

```
session = mysqlx.get_session(host="tcp.example.com", dns_srv=True)
```

2.2.4 Connection Option Summary

When using an X DevAPI session the following options are available to configure the connection.

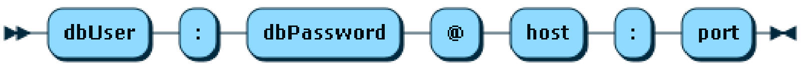
Option	Name	Optional	Default	Notes
TCP/IP Host	host	-		localhost, IPv4 host name, no IP-range
TCP/IP Port	port	Yes	33060	Standard X Plugin port is 33060
MySQL user	dbUser	-		MySQL database user
MySQL password	dbPassword	-		The MySQL user's password

Supported authentication methods are:

- PLAIN
- MYSQL 4.1

URI elements and format.

Figure 2.1 Connection URI



ConnectURI1::= 'dbUser' ':' 'dbPassword' '@' 'host' ':' 'port'

2.3 Working with a Session Object

All previous examples used the `getSchema()` or `getDefaultSchema()` methods of the Session object, which return a Schema object. You use this Schema object to access Collections and Tables. Most examples make use of the X DevAPI ability to chain all object constructions, enabling you to get to the Schema object in one line. For example:

```
schema = mysqlx.getSession(...).getSchema();
```

This object chain is equivalent to the following, with the difference that the intermediate step is omitted:

```
session = mysqlx.getSession();
schema = session.getSchema().
```

There is no requirement to always chain calls until you get a Schema object, neither is it always what you want. If you want to work with the Session object, for example, to call the Session object method `getSchemas()`, there is no need to navigate down to the Schema. For example:

```
session = mysqlx.getSession(); session.getSchemas().
```

In this example the `mysqlx.getSession()` function is used to open a Session. Then the `Session.getSchemas()` function is used to get a list of all available schemas and print them to the console.

MySQL Shell JavaScript Code

```
// Connecting to MySQL and working with a Session
var mysqlx = require('mysqlx');

// Connect to a dedicated MySQL server using a connection URI
var mySession = mysqlx.getSession('user:password@localhost');

// Get a list of all available schemas
var schemaList = mySession.getSchemas();

print('Available schemas in this session:\n');

// Loop over all available schemas and print their name
for (index in schemaList) {
  print(schemaList[index].name + '\n');
}
```

```
mySession.close();
```

MySQL Shell Python Code

```
# Connecting to MySQL and working with a Session
from mysqlsh import mysqlx

# Connect to a dedicated MySQL server using a connection URI
mySession = mysqlx.get_session('user:password@localhost')

# Get a list of all available schemas
schemaList = mySession.get_schemas()

print('Available schemas in this session:\n')

# Loop over all available schemas and print their name
for schema in schemaList:
        print('%s\n' % schema.name)

mySession.close()
```

Node.js JavaScript Code

```
// Connecting to MySQL and working with a Session
var mysqlx = require('@mysql/xdevapi');

// Connect to a dedicated MySQL server using a connection URI
mysqlx
  .getSession('user:password@localhost')
  .then(function (mySession) {
    // Get a list of all available schemas
    return mySession.getSchemas();
  })
  .then(function (schemaList) {
    console.log('Available schemas in this session:\n');

    // Loop over all available schemas and print their name
    schemaList.forEach(function (schema) {
      console.log(schema.getName() + '\n');
    });
  });
```

C# Code

```
// Connect to a dedicated MySQL server node using a connection URI
var mySession = MySQLX.GetSession("mysqlx://user:password@localhost:33060");

// Get a list of all available schemas
var schemaList = mySession.GetSchemas();

Console.WriteLine("Available schemas in this session:");

// Loop over all available schemas and print their name
foreach (var schema in schemaList)
{
      Console.WriteLine(schema.Name);
}

mySession.Close();
```

Python Code

```
# Connector/Python
# Connecting to MySQL and working with a Session
from mysqlsh import mysqlx

# Connect to a dedicated MySQL server using a connection URI
mySession = mysqlx.get_session('user:password@localhost')

# Get a list of all available schemas
```

```
schemaList = mySession.get_schemas()

print('Available schemas in this session:\n')

# Loop over all available schemas and print their name
for schema in schemaList:
        print('%s\n' % schema.name)

mySession.close()
```

Java Code

```
import java.util.List;
import com.mysql.cj.api.xdevapi.*;
import com.mysql.cj.xdevapi.*;

// Connecting to MySQL and working with a Session
// Connect to a dedicated MySQL server using a connection URI
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=passw

// Get a list of all available schemas
List<Schema> schemaList = mySession.getSchemas();

System.out.println("Available schemas in this session:");

// Loop over all available schemas and print their name
for (Schema schema : schemaList) {
System.out.println(schema.getName());
}

mySession.close();
```

C++ Code

```
#include <mysqlx/xdevapi.h>

// Connecting to MySQL and working with a Session

// Connect to a dedicated MySQL server using a connection URI
string url = "mysqlx://localhost:33060/test?user=user&password=password";
{
  Session mySession(url);

  // Get a list of all available schemas
  std::list<Schema> schemaList = mySession.getSchemas();

  cout << "Available schemas in this session:" << endl;

  // Loop over all available schemas and print their name
  for (Schema schema : schemaList) {
    cout << schema.getName() << endl;
  }
}
```

2.4 Using SQL with Session

In addition to the simplified X DevAPI syntax of the Session object, the Session object has a `sql()` function that takes any SQL statement as a string.

The following example uses a Session to call an SQL Stored Procedure on the specific node.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

// Connect to server using a Session
var mySession = mysqlx.getSession('user:password@localhost');

// Switch to use schema 'test'
mySession.sql("USE test").execute();
```

```
// In a Session context the full SQL language can be used
mySession.sql("CREATE PROCEDURE my_add_one_procedure " +
  " (INOUT incr_param INT) " +
  "BEGIN " +
  "  SET incr_param = incr_param + 1;" +
  "END;").execute();
mySession.sql("SET @my_var = ?;").bind(10).execute();
mySession.sql("CALL my_add_one_procedure(@my_var);").execute();
mySession.sql("DROP PROCEDURE my_add_one_procedure;").execute();

// Use an SQL query to get the result
var myResult = mySession.sql("SELECT @my_var").execute();

// Gets the row and prints the first column
var row = myResult.fetchOne();
print(row[0]);

mySession.close();
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

# Connect to server using a Session
mySession = mysqlx.get_session('user:password@localhost')

# Switch to use schema 'test'
mySession.sql("USE test").execute()

# In a Session context the full SQL language can be used
sql = """CREATE PROCEDURE my_add_one_procedure
                                 (INOUT incr_param INT)
                                 BEGIN
                                         SET incr_param = incr_param + 1;
                                 END
                        """

mySession.sql(sql).execute()
mySession.sql("SET @my_var = ?").bind(10).execute()
mySession.sql("CALL my_add_one_procedure(@my_var)").execute()
mySession.sql("DROP PROCEDURE my_add_one_procedure").execute()

# Use an SQL query to get the result
myResult = mySession.sql("SELECT @my_var").execute()

# Gets the row and prints the first column
row = myResult.fetch_one()
print(row[0])

mySession.close()
```

Node.js JavaScript Code

```
var mysqlx = require('@mysql/xdevapi');
var session;

// Connect to server using a Low-Level Session
mysqlx
  .getSession('root:password@localhost')
  .then(function (s) {
    session = s;

    return session.getSchema('test');
  })
  .then(function () {
    return Promise.all([
      // Switch to use schema 'test'
      session.sql('USE test').execute(),
      // In a Session context the full SQL language can be used
      session.sql('CREATE PROCEDURE my_add_one_procedure' +
        ' (INOUT incr_param INT) ' +
        'BEGIN ' +
        '  SET incr_param = incr_param + 1;' +
        'END;').execute(),
      session.executeSql('SET @my_var = ?;', 10).execute(),
      session.sql('CALL my_add_one_procedure(@my_var);').execute(),
      session.sql('DROP PROCEDURE my_add_one_procedure;').execute()
    ])
  })
  .then(function() {
    // Use an SQL query to get the result
    return session.sql('SELECT @my_var').execute(function (row) {
      // Print result
      console.log(row);
    });
  });
```

C# Code

```
// Connect to server using a Session
var mySession = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");

// Switch to use schema "test"
mySession.SQL("USE test").Execute();

// In a Session context the full SQL language can be used
mySession.SQL("CREATE PROCEDURE my_add_one_procedure " +
      " (INOUT incr_param INT) " +
      "BEGIN " +
      "  SET incr_param = incr_param + 1;" +
      "END;").Execute();
mySession.SQL("SET @my_var = 10;").Execute();
mySession.SQL("CALL my_add_one_procedure(@my_var);").Execute();
mySession.SQL("DROP PROCEDURE my_add_one_procedure;").Execute();

// Use an SQL query to get the result
var myResult = mySession.SQL("SELECT @my_var").Execute();

// Gets the row and prints the first column
var row = myResult.FetchOne();
Console.WriteLine(row[0]);

mySession.Close();
```

Python Code

```
# Connector/Python
from mysqlsh import mysqlx

# Connect to server using a Session
mySession = mysqlx.get_session('user:password@localhost')

# Switch to use schema 'test'
mySession.sql("USE test").execute()

# In a Session context the full SQL language can be used
sql = """CREATE PROCEDURE my_add_one_procedure
                                 (INOUT incr_param INT)
                                 BEGIN
                                         SET incr_param = incr_param + 1;
                                 END
                        """

mySession.sql(sql).execute()
mySession.sql("SET @my_var = ?").bind(10).execute()
mySession.sql("CALL my_add_one_procedure(@my_var)").execute()
mySession.sql("DROP PROCEDURE my_add_one_procedure").execute()

# Use an SQL query to get the result
myResult = mySession.sql("SELECT @my_var").execute()

# Gets the row and prints the first column
row = myResult.fetch_one()
```

```
print(row[0])

mySession.close()
```

Java Code

```
import com.mysql.cj.xdevapi.*;

// Connect to server on localhost
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=p

// Switch to use schema 'test'
mySession.sql("USE test").execute();

// In a Session context the full SQL language can be used
mySession.sql("CREATE PROCEDURE my_add_one_procedure " + " (INOUT incr_param INT) " + "BEGIN " + "  SET
        .execute();
mySession.sql("SET @my_var = ?").bind(10).execute();
mySession.sql("CALL my_add_one_procedure(@my_var)").execute();
mySession.sql("DROP PROCEDURE my_add_one_procedure").execute();

// Use an SQL query to get the result
SqlResult myResult = mySession.sql("SELECT @my_var").execute();

// Gets the row and prints the first column
Row row = myResult.fetchOne();
System.out.println(row.getInt(0));

mySession.close();
```

C++ Code

```
#include <mysqlx/xdevapi.h>

// Connect to server on localhost
string url = "mysqlx://localhost:33060/test?user=user&password=password";
Session mySession(url);

// Switch to use schema 'test'
mySession.sql("USE test").execute();

// In a Session context the full SQL language can be used
mySession.sql("CREATE PROCEDURE my_add_one_procedure "
              " (INOUT incr_param INT) "
              "BEGIN "
              "  SET incr_param = incr_param + 1;"
              "END;")
         .execute();
mySession.sql("SET @my_var = ?;").bind(10).execute();
mySession.sql("CALL my_add_one_procedure(@my_var);").execute();
mySession.sql("DROP PROCEDURE my_add_one_procedure;").execute();

// Use an SQL query to get the result
auto myResult = mySession.sql("SELECT @my_var").execute();

// Gets the row and prints the first column
Row row = myResult.fetchOne();
cout << row[0] << endl;
```

When using literal/verbatim SQL the common API patterns are mostly the same compared to using DML and CRUD operations on Tables and Collections. Two differences exist: setting the current schema and escaping names.

2.5 Setting the Current Schema

A default schema for a session can be specified using the `schema` attribute in the URI-like connection string or key-value pairs when opening a connection session. The `Session` class `getDefaultSchema()` method returns the default schema for the `Session`.

If no default schema has been selected at connection, the `Session` class `setCurrentSchema()` function can be used to set a current schema.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

// Direct connect with no client-side default schema specified
var mySession = mysqlx.getSession('user:password@localhost');
mySession.setCurrentSchema("test");
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

# Direct connect with no client-side default schema specified
mySession = mysqlx.get_session('user:password@localhost')
mySession.set_current_schema("test")
```

Node.js JavaScript Code

```
/*
  Connector/Node.js does not support the setCurrentSchema() method.
  One can specify the default schema in the connection string.
*/
```

C# Code

```
// Direct connect with no client-side default schema specified
var mySession = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");
mySession.SetCurrentSchema("test");
```

Python Code

```
# Connector/Python
from mysqlsh import mysqlx

# Direct connect with no client-side default schema specified
mySession = mysqlx.get_session('user:password@localhost')
mySession.set_current_schema("test")
```

Java Code

```
/*
  Connector/J does not support the setCurrentSchema() method.
  One can specify the default schema in the connection string.
*/
```

C++ Code

```
/*
  Connector/C++ does not support the setCurrentSchema() method.
  One can specify the default schema in the connection string.
*/
```

Notice that `setCurrentSchema()` does not change the session's default schema, which remains unchanged throughout the session, or remains `null` if not set at connection. The schema set by `setCurrentSchema()` can be returned by the `getCurrentSchema()` method.

An alternative way to set the current schema is to use the `Session` class sql() method and the USE *db_name* statement.

2.6 Dynamic SQL

A quoting function exists to escape SQL names and identifiers. `Session.quoteName()` escapes the identifier given in accordance to the settings of the current connection.

Note

The quoting function must not be used to escape values. Use the value binding syntax of `Session.sql()` instead; see Section 2.4, “Using SQL with Session” for some examples.

MySQL Shell JavaScript Code

```
function createTestTable(session, name) {

  // use escape function to quote names/identifier
  quoted_name = session.quoteName(name);

  session.sql("DROP TABLE IF EXISTS " + quoted_name).execute();

  var create = "CREATE TABLE ";
  create += quoted_name;
  create += " (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT)";

  session.sql(create).execute();

  return session.getCurrentSchema().getTable(name);
}

var mysqlx = require('mysqlx');

var session = mysqlx.getSession('user:password@localhost:33060/test');

var default_schema = session.getDefaultSchema().name;
session.setCurrentSchema(default_schema);

// Creates some tables
var table1 = createTestTable(session, 'test1');
var table2 = createTestTable(session, 'test2');
```

MySQL Shell Python Code

```
def createTestTable(session, name):

    # use escape function to quote names/identifier
    quoted_name = session.quote_name(name)
    session.sql("DROP TABLE IF EXISTS " + quoted_name).execute()
    create = "CREATE TABLE "
    create += quoted_name
    create += " (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT)"
    session.sql(create).execute()
    return session.get_current_schema().get_table(name)

from mysqlsh import mysqlx

session = mysqlx.get_session('user:password@localhost:33060/test')

default_schema = session.get_default_schema().name
session.set_current_schema(default_schema)

# Creates some tables
table1 = createTestTable(session, 'test1')
table2 = createTestTable(session, 'test2')
```

Node.js JavaScript Code

```
var mysqlx = require('mysqlx');

function createTestTable(session, name) {
  var create = 'CREATE TABLE ';
  create += name;
  create += ' (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT)';

  return session
```

```
    .sql('DROP TABLE IF EXISTS ' + name)
    .execute()
    .then(function () {
      return session.sql(create).execute();
    });
}

var session;

mysqlx
  .getSession({
    user: 'user',
    password: 'password'
  })
  .then(function (s) {
    session = s;

    return session
      .sql('use myschema')
      .execute()
  })
  .then(function () {
    // Creates some tables
    return Promise.map([
      createTestTable(session, 'test1'),
      createTestTable(session, 'test2')
    ])
  })
  .then(function () {
    session.close();
  })
});
```

C# Code

```
var session = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");

session.SQL("use test;").Execute();
session.GetSchema("test");

// Creates some tables
var table1 = CreateTestTable(session, "test1");
var table2 = CreateTestTable(session, "test2");
```

```
private Table CreateTestTable(Session session, string name)
{
  // use escape function to quote names/identifier
  string quoted_name = "`" + name + "`";

  session.SQL("DROP TABLE IF EXISTS " + quoted_name).Execute();

  var create = "CREATE TABLE ";
  create += quoted_name;
  create += " (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT)";

  session.SQL(create).Execute();

  return session.Schema.GetTable(name);
}
```

Python Code

```
# Connector/Python
def createTestTable(session, name):

    # use escape function to quote names/identifier
    quoted_name = session.quote_name(name)
    session.sql("DROP TABLE IF EXISTS " + quoted_name).execute()
    create = "CREATE TABLE "
    create += quoted_name
    create += " (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT)"
    session.sql(create).execute()
    return session.get_current_schema().get_table(name)

from mysqlsh import mysqlx

session = mysqlx.get_session('user:password@localhost:33060/test')

default_schema = session.get_default_schema().name
session.set_current_schema(default_schema)

# Creates some tables
table1 = createTestTable(session, 'test1')
table2 = createTestTable(session, 'test2')
```

Java Code

```
Java does not currently support the quoteName() method.
```

C++ Code

```
#include <mysqlx/xdevapi.h>

// Note: The following features are not yet implemented by
// Connector/C++:
// - DataSoure configuration files,
// - quoteName() method.

Table createTestTable(Session &session, const string &name)
{
  string quoted_name = string("`")
                     + session.getDefaultSchemaName()
                     + L"`.`" + name + L"`";
  session.sql(string("DROP TABLE IF EXISTS") + quoted_name).execute();

  string create = "CREATE TABLE ";
  create += quoted_name;
  create += L"(id INT NOT NULL PRIMARY KEY AUTO_INCREMENT)";

  session.sql(create).execute();
  return session.getDefaultSchema().getTable(name);
}

Session session(33060, "user", "password");

Table table1 = createTestTable(session, "test1");
Table table2 = createTestTable(session, "test2");
```

Code that uses X DevAPI does not need to escape identifiers. This is true for working with collections and for working with relational tables.

Chapter 3 CRUD Operations

Table of Contents

This section explains how to use the X DevAPI for Create Read, Update, and Delete (CRUD) operations.

MySQL's core domain has always been working with relational tables. X DevAPI extends this domain by adding support for CRUD operations that can be run against collections of documents. This section explains how to use these.

3.1 CRUD Operations Overview

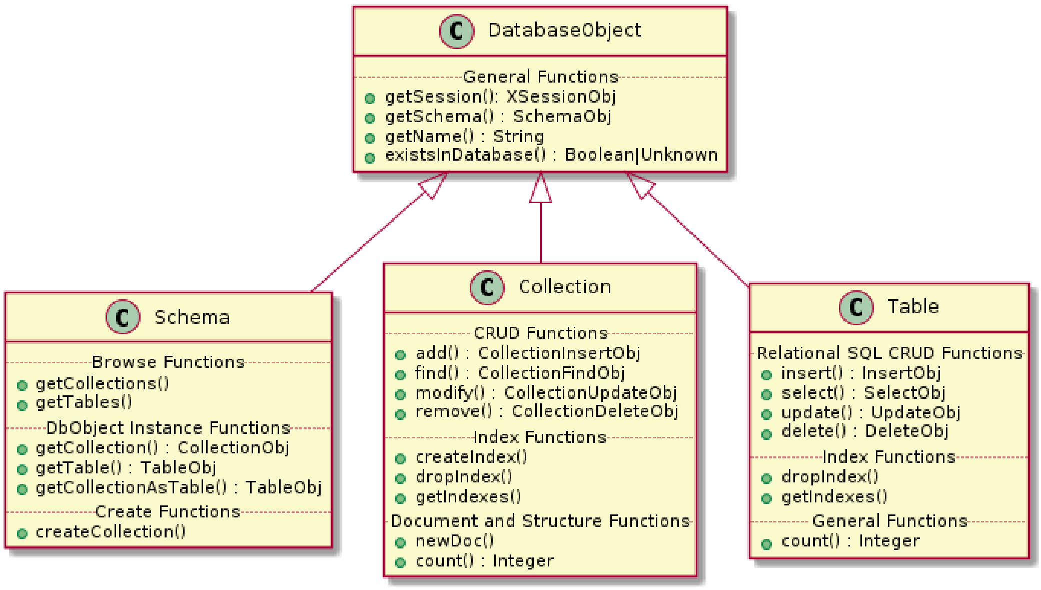
CRUD operations are available as methods, which operate on Schema objects. The available Schema objects consist of Collection objects containing Documents, or Table objects consisting of rows and Collections containing Documents.

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()

Database Object Classes

Figure 3.1 Database Object - Class Diagram



3.2 Method Chaining

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

The following example shows how method chaining is used instead of an SQL string when working with Session objects. The example assumes that the `test` schema exists and an `employee` table exists.

MySQL Shell JavaScript Code

```
// New method chaining used for executing an SQL SELECT statement
// Recommended way for executing queries
var employees = db.getTable('employee');

var res = employees.select(['name', 'age']).
        where('name like :param').
        orderBy(['name']).
        bind('param', 'm%').execute();

// Traditional SQL execution by passing an SQL string
// It should only be used when absolutely necessary
var result = session.sql('SELECT name, age ' +
  'FROM employee ' +
  'WHERE name like ? ' +
  'ORDER BY name').bind('m%').execute();
```

MySQL Shell Python Code

```
# New method chaining used for executing an SQL SELECT statement
# Recommended way for executing queries
employees = db.get_table('employee')

res = employees.select(['name', 'age']) \
        .where('name like :param') \
        .order_by(['name']) \
        .bind('param', 'm%').execute()

# Traditional SQL execution by passing an SQL string
# It should only be used when absolutely necessary
result = session.sql('SELECT name, age ' +
                'FROM employee ' +
                'WHERE name like ? ' +
                'ORDER BY name').bind('m%').execute()
```

Node.js JavaScript Code

```
// New method chaining used for executing an SQL SELECT statement
// Recommended way for executing queries
var employees = db.getTable('employee');
var promise = employees.select('name', 'age')
    .where('name like :name')
    .orderBy('name')
    .bind('m%')
    .execute();

// Traditional SQL execution by passing an SQL string
var sqlString = 'SELECT name, age ' +
  'FROM employee ' +
  'WHERE name like ? ' +
  'ORDER BY name';
var promise = db.executeSql(sqlString, 'm%').execute();
```

C# Code

```
// New method chaining used for executing an SQL SELECT statement
// Recommended way for executing queries
var employees = db.GetTable("employee");

var res = employees.Select("name", "age")
```

```
.Where("name like :param")
.OrderBy("name")
.Bind("param", "m%").Execute();

// Traditional SQL execution by passing an SQL string
// It should only be used when absolutely necessary
var result = session.SQL("SELECT name, age " +
"FROM employee " +
"WHERE name like ? " +
"ORDER BY name").Bind("m%").Execute();
```

Python Code

```
# Connector/Python
# New method chaining used for executing an SQL SELECT statement
# Recommended way for executing queries
employees = db.get_table('employee')

res = employees.select(['name', 'age']) \
        .where('name like :param') \
        .order_by(['name']) \
        .bind('param', 'm%').execute()

# Traditional SQL execution by passing an SQL string
# It should only be used when absolutely necessary
result = session.sql('SELECT name, age ' +
                'FROM employee ' +
                'WHERE name like ? ' +
                'ORDER BY name').bind('m%').execute()
```

Java Code

```
// New method chaining used for executing an SQL SELECT statement
// Recommended way for executing queries
Table employees = db.getTable("employee");

RowResult res = employees.select("name, age")
  .where("name like :param")
  .orderBy("name")
  .bind("param", "m%").execute();

// Traditional SQL execution by passing an SQL string
// It should only be used when absolutely necessary
SqlResult result = session.sql("SELECT name, age " +
  "FROM employee " +
  "WHERE name like ? " +
  "ORDER BY name").bind("m%").execute();
```

C++ Code

```
// New method chaining used for executing an SQL SELECT statement
// Recommended way for executing queries
Table employees = db.getTable("employee");

RowResult res = employees.select("name", "age")
  .where("name like :param")
  .orderBy("name")
  .bind("param", "m%").execute();

// Traditional SQL execution by passing an SQL string
// It should only be used when absolutely necessary
RowResult result = session.sql("SELECT name, age "
  "FROM employee "
  "WHERE name like ? "
  "ORDER BY name").bind("m%").execute();
```

3.3 Synchronous versus Asynchronous Execution

Traditionally, many MySQL drivers used a synchronous approach when executing SQL statements. This meant that operations such as opening connections and executing queries were blocked until completion, which could take a long time. To allow for parallel execution, a developer had to write a multithreaded application.

Any MySQL client that supports 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.

Note

MySQL Shell does not support asynchronous operations.

Asynchronous Operations

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.

Node.js JavaScript Code

```
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
  });
```

C# Code

```
var employees = db.GetTable("employee");

var select = employees.Select("name", "age")
  .Where("name like :name")
  .OrderBy("name")
  .Bind("name", "m%")
  .ExecuteAsync();

select.ContinueWith(t =>
{
  if (t.Exception != null)
  {
    // Handle error
  }
  // Do something with the resultset
});
```

Java Code

```
Table employees = db.getTable("employee");

// execute the query asynchronously, obtain a future
CompletableFuture<RowResult> rowsFuture = employees.select("name", "age")
  .where("name like :name")
  .orderBy("name")
  .bind("name", "m%").executeAsync();

// dependent functions can be attached to the CompletableFuture
```

C++ Code

```
// Asynchronous execution is not yet implemented in Connector/C++
```

Asynchronous Operations using Awaits

Some languages can use an async/await pattern.

C# Code

```
Task<RowResult> getEmployeesTask = employees.Select("name", "age")
  .Where("name like :name").OrderBy("name")
  .Bind("name", "m%").ExecuteAsync();

// Do something else while the getEmployeesTask is executing in the background

// at this point we are ready to get our results back. If it is not done,
// this will block until done
RowResult res = await getEmployeesTask;

foreach (var row in res.FetchAll())
{
  // use row object
}
```

Connector/Node.js uses asynchronous operations using Promises for all network operations. See other examples.

Java Code

```
Table employees = db.getTable("employee");

// execute the query asynchronously, obtain a future
CompletableFuture<RowResult> rowsFuture = employees.select("name", "age")
  .where("name like :name")
  .orderBy("name")
  .bind("name", "m%").executeAsync();

// wait until it's ready
RowResult rows = rowsFuture.get();
```

C++ Code

```
// Asynchronous execution is not yet implemented in Connector/C++
```

Syntax Differences

Depending on which language you are using, the X DevAPI may implement a function such as `executeAsync()` in exchange for `execute([mysqlx.Async])` or in addition to `execute([mysqlx.Async])`.

For example, in a Node.js context all executions are asynchronous. Therefore, Connector/Node.js does not need to distinguish between `execute()` and `executeAsync()`. To denote the asynchronous default execution, Connector/Node.js only implements `execute()` which returns JavaScript Promise objects.

Strongly typed programming languages, such as Java or C#, can take advantage of having two distinctly named API calls for synchronous and asynchronous executions. The two calls can have different return types. For example, Connector/J can use `execute()` to return a `RowResult` or `DocResult` and `executeAsync()` to return a `CompletableFuture<T>` where the type parameter is one of the result types.

3.4 Parameter Binding

Instead of using values directly in an expression string it is good practice to separate values from the expression string. This is done using parameters in the expression string and the `bind()` function to bind values to the parameters.

Parameters can be specified in the following ways: anonymous and named.

Parameter Type	Syntax	Example	Allowed in CRUD operations	Allowed in SQL strings
Anonymous	?	'age > ?'	no	yes
Named	:<name>	'age > :age'	yes	no

The following example shows how to use the `bind()` function before an `execute()` function. For each named parameter, provide an argument to `bind()` that contains the parameter name and its value. The order in which the parameter value pairs are passed to `bind()` is of no importance. The example assumes that the `test` schema has been assigned to the variable `db` and that the collection `my_collection` exists.

MySQL Shell and Node.js JavaScript Code

```
// Collection.find() function with fixed values
var myColl = db.getCollection('my_collection');

var myRes1 = myColl.find('age = 18').execute();

// Using the .bind() function to bind parameters
var myRes2 = myColl.find('name = :param1 AND age = :param2').bind('param1','Rohit').bind('param2', 18).exec

// Using named parameters
myColl.modify('name = :param').set('age', 55).
        bind('param', 'Nadya').execute();

// Binding works for all CRUD statements except add()
var myRes3 = myColl.find('name like :param').
        bind('param', 'R%').execute();
```

When running this with Connector/Node.js be aware that `execute()` returns a Promise. You might want to check the results to avoid errors being lost.

MySQL Shell Python Code

```
# Collection.find() function with hardcoded values
myColl = db.get_collection('my_collection')

myRes1 = myColl.find('age = 18').execute()

# Using the .bind() function to bind parameters
myRes2 = myColl.find('name = :param1 AND age = :param2').bind('param1','Rohit').bind('param2', 18).execute(

# Using named parameters
myColl.modify('name = :param').set('age', 55).bind('param', 'Nadya').execute()

# Binding works for all CRUD statements except add()
myRes3 = myColl.find('name like :param').bind('param', 'R%').execute()
```

C# Code

```
// Collection.Find() function with fixed values
var myColl = db.GetCollection("my_collection");

var myRes1 = myColl.Find("age = 18").Execute();

// Using the .Bind() function to bind parameters
var myRes2 = myColl.Find("name = :param1 AND age = :param2").Bind("param1", "Rohit").Bind("param2", 18).Exe

// Using named parameters
myColl.Modify("name = :param").Set("age", 55)
  .Bind("param", "Nadya").Execute();

// Binding works for all CRUD statements except Add()
var myRes3 = myColl.Find("name like :param")
```

```
  .Bind("param", "R%").Execute();
```

Python Code

```
# Collection.find() function with hardcoded values
my_coll = my_schema.get_collection('my_collection')

my_res_1 = my_coll.find('age = 18').execute()

# Using the .bind() function to bind parameters
my_res_2 = my_coll.find('name = :param1 AND age = :param2').bind('param1', 'Rohit').bind('param2', 18).

# Using named parameters
my_coll.modify('name = :param').set('age', 55).bind('param', 'Nadya').execute()

# Binding works for all CRUD statements except add()
my_res_3 = my_coll.find('name like :param').bind('param', 'R%').execute()
```

Java Code

```
// Collection.find() function with fixed values
Collection myColl = db.getCollection("my_collection");

DocResult myRes1 = myColl.find("age = 18").execute();

// Using the .bind() function to bind parameters
DocResult myRes2 = myColl.find("name = :param1 AND age = :param2").bind("param1", "Rohit").bind("param2

// Using named parameters
myColl.modify("name = :param").set("age", 55)
  .bind("param", "Nadya").execute();

// Using named parameters with a Map
Map<String, Object> params = new HashMap<>();
params.put("name", "Nadya");
myColl.modify("name = :name").set(".age", 55).bind(params).execute();

// Binding works for all CRUD statements except add()
DocResult myRes3 = myColl.find("name like :param")
  .bind("param", "R%").execute();    }
```

C++ Code

```
/// Collection.find() function with fixed values
Collection myColl = db.getCollection("my_collection");

auto myRes1 = myColl.find("age = 18").execute();

// Using the .bind() function to bind parameters
auto myRes2 = myColl.find("name = :param1 AND age = :param2")
                    .bind("param1","Rohit").bind("param2", 18)
                    .execute();

// Using named parameters
myColl.modify("name = :param").set("age", 55)
      .bind("param", "Nadya").execute();

// Binding works for all CRUD statements except add()
auto myRes3 = myColl.find("name like :param")
                    .bind("param", "R%").execute();
```

Anonymous placeholders are not supported in X DevAPI. This restriction improves code clarity in CRUD command chains with multiple methods using placeholders. Regardless of the `bind()` syntax variant used there is always a clear association between parameters and placeholders based on the parameter name.

All methods of a CRUD command chain form one namespace for placeholders. In the following example `find()` and `fields()` are chained. Both methods take an expression with placeholders. The placeholders refer to one combined namespace. Both use one placeholder called `:param`. A single call to `bind()` with one name value parameter for `:param` is used to assign a placeholder value to both occurrences of `:param` in `find()` and `fields()`.

MySQL Shell JavaScript Code

```
// one bind() per parameter
var myColl = db.getCollection('relatives');
var juniors = myColl.find('alias = "jr"').execute().fetchAll();

for (var index in juniors){
  myColl.modify('name = :param').
    set('parent_name',mysqlx.expr(':param')).
    bind('param', juniors[index].name).execute();
}
```

MySQL Shell Python Code

```
# one bind() per parameter
myColl = db.get_collection('relatives')
juniors = myColl.find('alias = "jr"').execute().fetch_all()

for junior in juniors:
  myColl.modify('name = :param'). \
    set('parent_name',mysqlx.expr(':param')). \
    bind('param', junior.name).execute()
```

Node.js JavaScript Code

```
// one bind() per parameter
db
  .getCollection('relatives');
  .find('alias = "jr"')
  .execute(function (junior) {
    return myColl
      .modify('name = :param')
      .set('parent_name', mysqlx.expr(':param'))
      .bind('param', junior.name)
      .execute();
  });
```

C# Code

```
// one bind() per parameter
myColl.Find("a = :param").Fields(":param as b")
  .Bind(new { param = "c"}).Execute();
```

Python Code

```
# one bind() per parameter
my_coll = my_schema.get_collection('relatives')
juniors = my_coll.find('alias = "jr"').execute().fetch_all()

for junior in juniors:
    my_coll.modify('name = :param') \
        .set('parent_name', mysqlx.expr(':param')) \
        .bind('param', junior.name).execute()
```

Java Code

```
// one bind() per parameter
myColl.find("a = :param").fields(":param as b")
  .bind("param", "c").execute();
```

C++ Code

```
// one bind() per parameter
Collection myColl = db.getCollection("relatives");
DocResult  juniors = myColl.find("alias = 'jr'").execute();

DbDoc junior;
```

```
while ((junior = juniors.fetchOne()))
{
  myColl.modify("name = :param")
        .set("parent_name", expr(":param"))
        .bind("param", junior["name"]).execute();
}
```

It is not permitted for a named parameter to use a name that starts with a digit. For example, `:1one` and `:1` are not allowed.

Preparing CRUD Statements

Instead of directly binding and executing CRUD operations with `bind()` and `execute()` or `execute()` it is also possible to store the CRUD operation object in a variable for later execution.

The advantage of doing so is to be able to bind several sets of variables to the parameters defined in the expression strings and therefore get better performance when executing a large number of similar operations. The example assumes that the `test` schema has been assigned to the variable `db` and that the collection `my_collection` exists.

MySQL Shell JavaScript Code

```
var myColl = db.getCollection('my_collection');

// Only prepare a Collection.remove() operation, but do not run it yet
var myRemove = myColl.remove('name = :param1 AND age = :param2');

// Binding parameters to the prepared function and .execute()
myRemove.bind('param1', 'Leon').bind('param2', 39).execute();
myRemove.bind('param1', 'Johannes').bind('param2', 28).execute();

// Binding works for all CRUD statements but add()
var myFind = myColl.find('name like :param1 AND age > :param2');

var myDocs = myFind.bind('param1', 'L%').bind('param2', 20).execute();
var MyOtherDocs = myFind.bind('param1', 'J%').bind('param2', 25).execute();
```

MySQL Shell Python Code

```
myColl = db.get_collection('my_collection')

# Only prepare a Collection.remove() operation, but do not run it yet
myRemove = myColl.remove('name = :param1 AND age = :param2')

# Binding parameters to the prepared function and .execute()
myRemove.bind('param1', 'Leon').bind('param2', 39).execute()
myRemove.bind('param1', 'Johannes').bind('param2', 28).execute()

# Binding works for all CRUD statements but add()
myFind = myColl.find('name like :param1 AND age > :param2')

myDocs = myFind.bind('param1', 'L%').bind('param2', 20).execute()
MyOtherDocs = myFind.bind('param1', 'J%').bind('param2', 25).execute()
```

Node.js JavaScript Code

```
var myColl = db.getCollection('my_collection');

// Only prepare a Collection.remove() operation, but do not run it yet
var myRemove = myColl.remove('name = :param1 AND age = :param2');

// Binding parameters to the prepared function and .execute()
myRemove.bind('param1', 'Leon').bind('param2', 39).execute();
myRemove.bind('param1', 'Johannes').bind('param2', 28).execute();

// Binding works for all CRUD statements but add()
var myFind = myColl.find('name like :param1 AND age > :param2');

var myDocs = myFind.bind('param1', 'L%').bind('param2', 20).execute();
var MyOtherDocs = myFind.bind('param1', 'J%').bind('param2', 25).execute();
```

C# Code

```
var myColl = db.GetCollection("my_collection");

// Only prepare a Collection.Remove() operation, but do not run it yet
var myRemove = myColl.Remove("name = :param1 AND age = :param2");

// Binding parameters to the prepared function and .Execute()
myRemove.Bind("param1", "Leon").Bind("param2", 39).Execute();
myRemove.Bind("param1", "Johannes").Bind("param2", 28).Execute();

// Binding works for all CRUD statements but Add()
var myFind = myColl.Find("name like :param1 AND age > :param2");

var myDocs = myFind.Bind("param1", "L%").Bind("param2", 20).Execute();
var MyOtherDocs = myFind.Bind("param1", "J%").Bind("param2", 25).Execute();
```

Python Code

```
my_coll = my_schema.get_collection('my_collection')

# Only prepare a Collection.remove() operation, but do not run it yet
my_remove = my_coll.remove('name = :param1 AND age = :param2')

# Binding parameters to the prepared function and .execute()
my_remove.bind('param1', 'Leon').bind('param2', 39).execute()
my_remove.bind('param1', 'Johannes').bind('param2', 28).execute()

# Binding works for all CRUD statements but add()
my_find = my_coll.find('name like :param1 AND age > :param2')

my_docs = my_find.bind('param1', 'L%').bind('param2', 20).execute()
my_other_docs = my_find.bind('param1', 'J%').bind('param2', 25).execute()
```

Java Code

```
Collection myColl = db.getCollection("my_collection");

// Create Collection.remove() operation, but do not run it yet
RemoveStatement myRemove = myColl.remove("name = :param1 AND age = :param2");

// Binding parameters to the prepared function and .execute()
myRemove.bind("param1", "Leon").bind("param2", 39).execute();
myRemove.bind("param1", "Johannes").bind("param2", 28).execute();

// Binding works for all CRUD statements but add()
FindStatement myFind = myColl.find("name LIKE :name AND age > :age");

Map<String, Object> params = new HashMap<>();
params.put("name", "L%");
params.put("age", 20);
DocResult myDocs = myFind.bind(params).execute();
params.put("name", "J%");
params.put("age", 25);
DocResult myOtherDocs = myFind.bind(params).execute();
```

C++ Code

```
Collection myColl = db.getCollection("my_collection");

// Create Collection.remove() operation, but do not run it yet
auto myRemove = myColl.remove("name = :param1 AND age = :param2");

// Binding parameters to the prepared function and .execute()
myRemove.bind("param1", "Leon").bind("param2", 39).execute();
myRemove.bind("param1", "Johannes").bind("param2", 28).execute();

// Binding works for all CRUD statements but Add()
auto myFind = myColl.find("name like :param1 AND age > :param2");
auto myDocs = myFind.bind("param1", "L%").bind("param2", 20).execute();
auto MyOtherDocs = myFind.bind("param1", "J%").bind("param2", 25).execute();
```

3.5 MySQL Shell Automatic Code Execution

When you use X DevAPI in a programming language that fully specifies the syntax to be used, for example, when executing SQL statements through an X DevAPI session or working with any of the CRUD operations, the actual operation is performed only when the `execute()` function is called. For example:

```
var result = mySession.sql('show databases').execute()
var result2 = myColl.find().execute()
```

The call of the `execute()` function above causes the operation to be executed and returns a Result object. The returned Result object is then assigned to a variable, and the assignment is the last operation executed, which returns no data. Such operations can also return a Result object, which is used to process the information returned from the operation.

Alternatively, MySQL Shell provides the following usability features that make it easier to work with X DevAPI interactively:

- Automatic execution of CRUD and SQL operations.
- Automatic processing of results.

To achieve this functionality MySQL Shell monitors the result of the last operation executed every time you enter a statement. The combination of these features makes using the MySQL Shell interactive mode ideal for prototyping code, as operations are executed immediately and their results are displayed without requiring any additional coding. For more information see MySQL Shell 8.0 (part of MySQL 8.0).

Automatic Code Execution

If MySQL Shell detects that a CRUD operation ready to execute has been returned, it automatically calls the `execute()` function. Repeating the example above in MySQL Shell and removing the assignment operation shows the operation is automatically executed.

```
mysql-js> mySession.sql('show databases')
mysql-js> myColl.find()
```

MySQL Shell executes the SQL operation, and as mentioned above, once this operation is executed a Result object is returned.

Automatic Result Processing

If MySQL Shell detects that a Result object is going to be returned, it automatically processes it, printing the result data in the best format possible. There are different types of Result objects and the format changes across them.

```
mysql-js> db.countryInfo.find().limit(1)
[

    {

        "GNP": 828,

        "IndepYear": null,

        "Name": "Aruba",

        "_id": "ABW",
        "demographics": {
            "LifeExpectancy": 78.4000015258789,
            "Population": 103000
        },
        "geography": {
            "Continent": "North America",
            "Region": "Caribbean",
            "SurfaceArea": 193
        },
        "government": {
            "GovernmentForm": "Nonmetropolitan Territory of The Netherlands",
            "HeadOfState": "Beatrix"
        }
    }
]
1 document in set (0.00 sec)
```

Chapter 4 Working with Collections

Table of Contents

The following section explains how to work with Collections, how to use CRUD operations on Collections and return Documents.

4.1 Basic CRUD Operations on Collections

Working with collections of documents is straightforward when using X DevAPI. The following examples show the basic usage of CRUD operations when working with documents.

After establishing a connection to a MySQL Server instance, a new collection that can hold JSON documents is created and several documents are inserted. Then, a find operation is executed to search for a specific document from the collection. Finally, the collection is dropped again from the database. The example assumes that the `test` schema exists and that the collection `my_collection` does not exist.

MySQL Shell JavaScript Code

```
// Connecting to MySQL Server and working with a Collection

var mysqlx = require('mysqlx');

// Connect to server
var mySession = mysqlx.getSession( {
host: 'localhost', port: 33060,
user: 'user', password: 'password'} );

var myDb = mySession.getSchema('test');

// Create a new collection 'my_collection'
var myColl = myDb.createCollection('my_collection');

// Insert documents
myColl.add({_id: '1', name: 'Laurie', age: 19}).execute();
myColl.add({_id: '2', name: 'Nadya', age: 54}).execute();
myColl.add({_id: '3', name: 'Lukas', age: 32}).execute();

// Find a document
var docs = myColl.find('name like :param1 AND age < :param2').limit(1).
        bind('param1','L%').bind('param2',20).execute();

// Print document
print(docs.fetchOne());

// Drop the collection
myDb.dropCollection('my_collection');
```

MySQL Shell Python Code

```
# Connecting to MySQL Server and working with a Collection
from mysqlsh import mysqlx
```

```
# Connect to server
mySession = mysqlx.get_session( {
'host': 'localhost', 'port': 33060,
'user': 'user', 'password': 'password'} )

myDb = mySession.get_schema('test')

# Create a new collection 'my_collection'
myColl = myDb.create_collection('my_collection')

# Insert documents
myColl.add({'_id': '1', 'name': 'Laurie', 'age': 19}).execute()
myColl.add({'_id': '2', 'name': 'Nadya', 'age': 54}).execute()
myColl.add({'_id': '3', 'name': 'Lukas', 'age': 32}).execute()

# Find a document
docs = myColl.find('name like :param1 AND age < :param2') \
          .limit(1) \
          .bind('param1','L%') \
          .bind('param2',20) \
          .execute()

# Print document
doc = docs.fetch_one()
print(doc)

# Drop the collection
myDb.drop_collection('my_collection')
```

Node.js JavaScript Code

```
// --------------------------------------------------
// Connecting to MySQL Server and working with a Collection
var mysqlx = require('@mysql/xdevapi');
var db;

// Connect to server
mysqlx
  .getSession({
    user: 'user',
    password: 'password',
    host: 'localhost',
    port: '33060',
  })
  .then(function (session) {
    db = session.getSchema('test');
    // Create a new collection 'my_collection'
    return db.createCollection('my_collection');
  })
  .then(function (myColl) {
    // Insert documents
    return Promise
      .all([
        myColl.add({ name: 'Laurie', age: 19 }).execute(),
        myColl.add({ name: 'Nadya', age: 54 }).execute(),
        myColl.add({ name: 'Lukas', age: 32 }).execute()
      ])
      .then(function () {
        // Find a document
        return myColl
            .find('name like :name && age < :age')
            .bind({ name: 'L%', age: 20 })
            .limit(1)
            .execute(function (doc) {
              // Print document
              console.log(doc);
            });
      });
  })
  .then(function(docs) {
    // Drop the collection
    return db.dropCollection('my_collection');
  })
  .catch(function(err) {
    // Handle error
  });
```

C# Code

```
// Connecting to MySQL Server and working with a Collection

// Connect to server
var mySession = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");

var myDb = mySession.GetSchema("test");

// Create a new collection "my_collection"
var myColl = myDb.CreateCollection("my_collection");

// Insert documents
myColl.Add(new { name = "Laurie", age = 19}).Execute();
myColl.Add(new { name = "Nadya", age = 54}).Execute();
myColl.Add(new { name = "Lukas", age = 32}).Execute();

// Find a document
var docs = myColl.Find("name like :param1 AND age < :param2").Limit(1)
.Bind("param1", "L%").Bind("param2", 20).Execute();

// Print document
Console.WriteLine(docs.FetchOne());

// Drop the collection
myDb.DropCollection("my_collection");
```

Python Code

```
# Connecting to MySQL Server and working with a Collection

import mysqlx

# Connect to server
my_session = mysqlx.get_session({
    'host': 'localhost', 'port': 33060,
    'user': 'user', 'password': 'password'
})

my_schema = my_session.get_schema('test')

# Create a new collection 'my_collection'
my_coll = my_schema.create_collection('my_collection')

# Insert documents
my_coll.add({'name': 'Laurie', 'age': 19}).execute()
my_coll.add({'name': 'Nadya', 'age': 54}).execute()
my_coll.add({'name': 'Lukas', 'age': 32}).execute()

# Find a document
docs = my_coll.find('name like :param1 AND age < :param2') \
    .limit(1) \
    .bind('param1', 'L%') \
    .bind('param2', 20) \
    .execute()

# Print document
doc = docs.fetch_one()
print("Name: {0}".format(doc['name']))
print("Age: {0}".format(doc['age]))

# Drop the collection
my_session.drop_collection('test', 'my_collection')
```

Java Code

```
// Connect to server
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=passw

Schema myDb = mySession.getSchema("test");

// Create a new collection 'my_collection'
Collection myColl = myDb.createCollection("my_collection");

// Insert documents
myColl.add("{\"name\":\"Laurie\", \"age\":19}").execute();
myColl.add("{\"name\":\"Nadya\", \"age\":54}").execute();
myColl.add("{\"name\":\"Lukas\", \"age\":32}").execute();

// Find a document
DocResult docs = myColl.find("name like :name AND age < :age")
        .bind("name", "L%").bind("age", 20).execute();

// Print document
DbDoc doc = docs.fetchOne();
System.out.println(doc);

// Drop the collection
myDB.dropCollection("my_collection");
```

C++ Code

```
// Connecting to MySQL Server and working with a Collection

#include <mysqlx/xdevapi.h>

// Connect to server
Session session(33060, "user", "password");
Schema db = session.getSchema("test");

// Create a new collection 'my_collection'
Collection myColl = db.createCollection("my_collection");

// Insert documents
myColl.add(R"({"name": "Laurie", "age": 19})").execute();
myColl.add(R"({"name": "Nadya", "age": 54})").execute();
myColl.add(R"({"name": "Lukas", "age": 32})").execute();

// Find a document
DocResult docs = myColl.find("name like :param1 AND age < :param2").limit(1)
                        .bind("param1","L%").bind("param2",20).execute();

// Print document
cout << docs.fetchOne();

// Drop the collection
db.dropCollection("my_collection");
```

4.2 Collection Objects

Documents of the same type (for example users, products) are grouped together and stored in the database as collections. X DevAPI uses Collection objects to store and retrieve documents.

4.2.1 Creating a Collection

In order to create a new collection call the `createCollection()` function from a Schema object. It returns a Collection object that can be used right away, for example to insert documents into the collection.

Optionally, the field `reuseExistingObject` can be set to true and passed as second parameter to prevent an error being generated if a collection with the same name already exists.

MySQL Shell JavaScript Code

```
// Create a new collection called 'my_collection'
var myColl = db.createCollection('my_collection');
```

MySQL Shell Python Code

```
# Create a new collection called 'my_collection'
myColl = db.create_collection('my_collection')
```

Node.js JavaScript Code

```
// Create a new collection called 'my_collection'
var promise = db.createCollection('my_collection');

// Create a new collection or reuse existing one
var promise = db.createCollection('my_collection', { ReuseExistingObject: true } );
```

C# Code

```
// Create a new collection called "my_collection"
var myColl = db.CreateCollection("my_collection");

// Create a new collection or reuse existing one
var myExistingColl = db.CreateCollection("my_collection", ReuseExistingObject: true);
```

Python Code

```
# Create a new collection called 'my_collection'
my_coll = my_schema.create_collection('my_collection')
```

Java Code

```
// Create a new collection called 'my_collection'
Collection myColl = db.createCollection("my_collection");

// Create a new collection or reuse existing one
// second parameter is: boolean reuseExistingObject
Collection myExistingColl = db.createCollection("my_collection", true);
```

C++ Code

```
// Create a new collection called 'my_collection'
Collection myColl = db.createCollection("my_collection");

// Create a new collection or reuse existing one
Collection myExistingColl = db.createCollection("my_collection", true);
```

4.2.2 Working with Existing Collections

In order to retrieve a Collection object for an existing collection stored in the database call the `getCollection()` function from a Schema object.

MySQL Shell JavaScript Code

```
// Get a collection object for 'my_collection'
var myColl = db.getCollection('my_collection');
```

MySQL Shell Python Code

```
# Get a collection object for 'my_collection'
myColl = db.get_collection('my_collection')
```

Node.js JavaScript Code

```
// Get a collection object for 'my_collection'
var collection = db.getCollection('my_collection');
```

C# Code

```
// Get a collection object for "my_collection"
var myColl = db.GetCollection("my_collection");

// Get a collection object but also ensure it exists in the database
var myColl2 = db.GetCollection("my_collection", ValidateExistence: true);
```

Python Code

```
# Get a collection object for 'my_collection'
my_coll = my_schema.get_collection('my_collection')
```

Java Code

```
// Get a collection object for 'my_collection'
Collection myColl = db.getCollection("my_collection");

// Get a collection object but also ensure it exists in the database
// second parameter is: boolean requireExists
Collection myColl = db.getCollection("my_collection", true);
```

C++ Code

```
// Get a collection object for 'my_collection'
Collection myColl = db.getCollection("my_collection");

// Get a collection object but also ensure it exists in the database
Collection myColl = db.getCollection("my_collection", true);
```

If the collection does not yet exist in the database any subsequent call of a Collection object function throws an error. To prevent this and catch the error right away set the `validateExistence` field to true and pass it as second parameter to `db.getCollection()`.

The `createCollection()` together with the `ReuseExistingObject` field set to true can be used to create a new collection or reuse an existing collection with the given name.

Note

In most cases it is good practice to create database objects during development time and refrain from creating them on the fly during the production phase of a database project. Therefore it is best to separate the code that creates the collections in the database from the actual user application code.

4.2.3 Indexing Collections

To make large collections of documents more efficient to navigate you can create an index based on one or more fields found in the documents in the collection. This section describes how to index a collection.

Creating an Index

Collection indexes are ordinary MySQL indexes on virtual columns that extract data from the documents in the collection. Currently MySQL cannot index JSON values directly, therefore to enable indexing of a collection you provide a JSON document which specifies the document's fields used by the index. You pass the JSON document defining the index as the IndexDefinition parameter to the `Collection.createIndex(`*`name, IndexDefinition`*`)` method. This example shows how to create a mandatory integer type index based on the field `count`:

```
myCollection.createIndex("count", {fields:[{"field": "$.count", "type":"INT", required:true}]});
```

This example shows how to create an index based on a text field, in this case, a zip code. For a text field, you must specify a prefix length for the index, as required by MySQL Server:

```
myCollection.createIndex("zip", {fields: [{field: "$.zip", type: "TEXT(10)"}]})
```

See Defining an Index for information about the format of the JSON document used to define fields as MySQL types, and the currently supported MySQL types.

The `Collection.createIndex()` method fails with an error if an index with the same name already exists or if the index definition is not correctly formed. The name parameter is required and must be a valid index name as accepted by the SQL statement `CREATE INDEX`.

To remove an existing index use the `collection.dropIndex(string name)` method. This would delete the index with the name passed in, and the operation silently succeeds if the named index does not exist.

As the indexes of a collection are stored as virtual columns, to verify a created index use the `SHOW INDEX` statement. For example to use this SQL from MySQL Shell:

```
session.runSql('SHOW INDEX FROM mySchema.myCollection');
```

Defining an Index

To create an index based on the documents in a collection you need to create an `IndexDefinition` JSON document. This section explains the valid fields you can use in such a JSON document to define an index.

To define a document field to index a collection on, the type of that field must be uniform across the whole collection. In other words the type must be consistent. The JSON document used for defining an index, such as `{fields: [{field: '$.username', type: 'TEXT'}]}`, can contain the following:

- `fields`: an array of at least one `IndexField` object, each of which describes a JSON document field to be included in the index.

  A single `IndexField` description consists of the following fields:

  - `field`: a string with the full document path to the document member or field to be indexed
  - `type`: a string with one of the supported column types to map the field to (see Field Data Types). For numeric types, the optional `UNSIGNED` keyword can follow. For the `TEXT` type you must define the length to consider for indexing (the prefix length).
  - `required`: an optional boolean, set to `true` if the field is required to exist in the document. Defaults to `false` for all types except `GEOJSON`, which defaults to `true`.
  - `options`: an optional integer, used as special option flags to use when decoding `GEOJSON` data.
  - `srid`: an optional integer, srid value to use when decoding `GEOJSON` data.
  - `array`: (for MySQL 8.0.17 and later) an optional boolean, set to `true` if the field contains arrays. The default value is `false`. See Indexing Array Fields for details.

  Important

  For MySQL 8.0.16 and earlier, fields that are JSON arrays are not supported in the index; specifying a field that contains array data does not generate an error from the server, but the index does not function correctly.

- `type`: an optional string which defines the type of index. One of `INDEX` or `SPATIAL`. The default is `INDEX` and can be omitted.

For example, to create an index based on multiple fields, issue:

```
myCollection.createIndex('myIndex', {fields: [{field: '$.myField', type: 'TEXT'}, //
{field: '$.myField2', type: 'TEXT(10)'}, {field: '$.myField3', type: 'INT'}]})
```

Including any other fields in an `IndexDefinition` or `IndexField` JSON document which is not described here causes `collection.createIndex()` to fail with an error.

If index type is not specified, or is set to `INDEX` then the resulting index is created in the same way as would be created after issuing `CREATE INDEX`. If index type is set to `SPATIAL` then the created index is the same as would be created after issuing `CREATE INDEX` with the `SPATIAL` keyword, see SPATIAL Index Optimization and Creating Spatial Indexes. For example

```
myCollection.createIndex('myIndex', //
{fields: [{field: '$.myGeoJsonField', type: 'GEOJSON', required: true}], type:'SPATIAL'})
```

Important

When using the `SPATIAL` type of index the `required` field cannot be set to `false` in `IndexField` entries.

The values of indexed fields are converted from JSON to the type specified in the `IndexField` description using standard MySQL type conversions (see Type Conversion in Expression Evaluation), except for the `GEOJSON` type which uses the `ST_GeomFromGeoJSON()` function for conversion. This means that when using a numeric type in an `IndexField` description and the actual field value is non-numeric, it is converted to 0.

The `options` and `srid` fields in `IndexField` can only be present if `type` is set to `GEOJSON`. If present, they are used as parameters for `ST_GeomFromGeoJSON()` when converting `GEOJSON` data into MySQL native `GEOMETRY` values.

Field Data Types

The following data types are supported for document fields. Type descriptions are case insensitive.

- `INT` [UNSIGNED]
- `TINYINT` [UNSIGNED]
- `SMALLINT` [UNSIGNED]
- `MEDIUMINT` [UNSIGNED]
- `INTEGER` [UNSIGNED]
- `BIGINT` [UNSIGNED]
- `REAL` [UNSIGNED]
- `FLOAT` [UNSIGNED]
- `DOUBLE` [UNSIGNED]
- `DECIMAL` [UNSIGNED]
- `NUMERIC` [UNSIGNED]
- `DATE`
- `TIME`
- `TIMESTAMP`
- `DATETIME`
- `TEXT(length)`

- `GEOJSON` (extra options: options, srid)

Indexing Array Fields

For MySQL 8.0.17 and later, X DevAPI supports creating indexes based on array fields by setting the Boolean `array` field in the `IndexField` description to `true`. For example, to create an index on the `emails` array field:

```
collection.createIndex("emails_idx", //
    {fields: [{"field": "$.emails", "type":"CHAR(128)", "array": true}]});
```

The following restrictions apply to creating indexes based on arrays:

- For each index, only one indexed field can be an `array`
- Data types for which index on arrays can be created:
  - Numeric types: `INTEGER [UNSIGNED]` (`INT` is NOT supported)
  - Fixed-point types: `DECIMAL(m, n)` (the precision and scale values are mandatory)
  - Date and time types: `DATE`, `TIME`, and `DATETIME`
  - String types: `CHAR(n)` and `BINARY(n)`; the character or byte length `n` is mandatory (`TEXT` is NOT supported)

4.3 Collection CRUD Function Overview

The following section explains the individual functions of the Collection object.

The most common operations to be performed on a Collection are the Create Read Update Delete (CRUD) operations. In order to speed up find operations it is recommended to make proper use of indexes.

Collection.add()

The `Collection.add()` function is used to store documents in the database. It takes a single document or a list of documents and is executed by the `run()` function.

The collection needs to be created with the `Schema.createCollection()` function before documents can be inserted. To insert documents into an existing collection use the `Schema.getCollection()` function.

The following example shows how to use the `Collection.add()` function. The example assumes that the test schema exists and that the collection `my_collection` does not exist.

MySQL Shell JavaScript Code

```
// Create a new collection
var myColl = db.createCollection('my_collection');

// Insert a document
myColl.add( {_id: '1', name: 'Laurie', age: 19 } ).execute();

// Insert several documents at once
myColl.add( [
{_id: '5', name: 'Nadya', age: 54 },
{_id: '6', name: 'Lukas', age: 32 } ] ).execute();
```

MySQL Shell Python Code

```
# Create a new collection
```

```
myColl = db.create_collection('my_collection')

# Insert a document
myColl.add( {'_id': '1', 'name': 'Laurie', 'age': 19 } ).execute()

# Insert several documents at once
myColl.add( [
{'_id': '5', 'name': 'Nadya', 'age': 54 },
{'_id': '6', 'name': 'Lukas', 'age': 32 } ] ).execute()
```

Node.js JavaScript Code

```
// Create a new collection
db.createCollection('myCollection').then(function (myColl) {
  return Promise.all([
    // Insert a document
    myColl
      .add({ name: 'Laurie', age: 19 })
      .execute(),
    // Insert several documents at once
    myColl
      .add([
        { name: 'Nadya', age: 54 },
        { name: 'Lukas', age: 32 }
      ])
      .execute()
  ])
});
```

C# Code

```
// Assumptions: test schema assigned to db, my_collection collection not exists

// Create a new collection
var myColl = db.CreateCollection("my_collection");

// Insert a document
myColl.Add(new { name = "Laurie", age = 19 }).Execute();

// Insert several documents at once
myColl.Add(new[] {
new { name = "Nadya", age = 54 },
new { name = "Lukas", age = 32 } }).Execute();
```

Python Code

```
# Create a new collection
my_coll = my_schema.create_collection('my_collection')

# Insert a document
my_coll.add({'name': 'Laurie', 'age': 19}).execute()

# Insert several documents at once
my_coll.add([
    {'name': 'Nadya', 'age': 54},
    {'name': 'Lukas', 'age': 32}
]).execute()
```

Java Code

```
// Create a new collection
Collection coll = db.createCollection("payments");

// Insert a document
coll.add("{\"name\":\"Laurie\", \"age\":19}");

// Insert several documents at once
coll.add("{\"name\":\"Nadya\", \"age\":54}",
        "{\"name\":\"Lukas\", \"age\":32}");
```

C++ Code

```
// Create a new collection
Collection coll = db.createCollection("payments");

// Insert a document
coll.add(R"({"name":"Laurie", "age":19})").execute();

// Insert several documents at once
std::list<DbDoc> docs = {
  DbDoc(R"({"name":"Nadya", "age":54})"),
  DbDoc(R"({"name":"Lukas", "age":32})")
};
coll.add(docs).execute();
```

For more information, see CollectionAddFunction.

Document Identity

Every document has a unique identifier called the document ID. The document ID is stored in the `_id` field of a document. The document ID is a `VARBINARY()` with a maximum length of 32 characters. This section describes how document IDs can be used, see Section 5.1, "Working with Document IDs" for more information.

The following example assumes that the `test` schema exists and is assigned to the variable `db`, that the collection `my_collection` exists and that `custom_id` is unique.

MySQL Shell JavaScript Code

```
// If the _id is provided, it will be honored
var result = myColl.add( { _id: 'custom_id', a : 1 } ).execute();
var document = myColl.find("a = 1").execute().fetchOne();
print("User Provided Id:", document._id);

// If the _id is not provided, one will be automatically assigned
result = myColl.add( { b: 2 } ).execute();
print("Autogenerated Id:", result.getGeneratedIds()[0]);
```

MySQL Shell Python Code

```
# If the _id is provided, it will be honored
result = myColl.add( { '_id': 'custom_id', 'a' : 1 } ).execute()
document = myColl.find('a = 1').execute().fetch_one()
print("User Provided Id: %s" % document._id)

# If the _id is not provided, one will be automatically assigned
result = myColl.add( { 'b': 2 } ).execute()
print("Autogenerated Id: %s" % result.get_generated_ids()[0])
```

Node.js JavaScript Code

```
// If the _id is provided, it will be honored
myColl.add({ _id: 'custom_id', a : 1 }).execute().then(function () {
  myColl.getOne('custom_id').then(function (doc) {
    console.log('User Provided Id:', doc._id);
  });
});

// If the _id is not provided, one will be automatically assigned
myColl.add({ b: 2 }).execute().then(function (result) {
  console.log('Autogenerated Id:', result.getGeneratedIds()[0]);
});
```

C# Code

```
// If the _id is provided, it will be honored
var result = myColl.Add(new { _id = "custom_id", a = 1 }).Execute();
```

```
Console.WriteLine("User Provided Id:", result.AutoIncrementValue);

// If the _id is not provided, one will be automatically assigned
result = myColl.Add(new { b = 2 }).Execute();
Console.WriteLine("Autogenerated Id:", result.AutoIncrementValue);
```

Python Code

```
# If the _id is provided, it will be honored
result = my_coll.add({'_id': 'custom_id', 'a': 1}).execute()
print("User Provided Id: {0}".format(result.get_last_document_id()))

# If the _id is not provided, one will be automatically assigned
result = my_coll.add({'b': 2}).execute()
print("Autogenerated Id: {0}".format(result.get_last_document_id()))
```

Java Code

```
// If the _id is provided, it will be honored
AddResult result = coll.add("{\"_id\":\"custom_id\",\"a\":1}").execute();
System.out.println("User Provided Id:" + ((JsonString) coll.getOne("custom_id").get("_id")).getString());

// If the _id is not provided, one will be automatically assigned
result = coll.add("{\"b\":2}").execute();
System.out.println("Autogenerated Id:" + result.getGeneratedIds().get(0));
```

C++ Code

```
// If the _id is provided, it will be honored
Result result = myColl.add(R"({ "_id": "custom_id", "a" : 1 })").execute();
std::vector<string> ids = result.getGeneratedIds();
if (ids.empty())
  cout << "No Autogenerated Ids" << endl;

// If the _id is not provided, one will be automatically assigned
result = myColl.add(R"({ "b": 2 })").execute();
ids = result.getGeneratedIds();
cout << "Autogenerated Id:" << ids[0] << endl;
```

Some documents have a natural unique key. For example, a collection that holds a list of books is likely to include the International Standard Book Number (ISBN) for each document that represents a book. The ISBN is a string with a length of 13 characters which is well within the length limit of the `_id` field.

```
// using a book's unique ISBN as the object ID
myColl.add( {
_id: "978-1449374020",
title: "MySQL Cookbook: Solutions for Database Developers and Administrators"
}).execute();
```

Use `find()` to fetch the newly inserted book from the collection by its document ID.

```
var book = myColl.find('_id = "978-1449374020"').execute();
```

Currently, X DevAPI does not support using any document field other than the implicit `_id` as the document ID. There is no way of defining a different document ID (primary key).

Collection.find()

The `find()` function is used to get documents from the database. It takes a SearchConditionStr as a parameter to specify the documents that should be returned from the database. Several methods such as `fields()`, `sort()`, `skip()` and `limit()` can be chained to the `find()` function to further refine the result.

The `fetch()` function actually triggers the execution of the operation. The example assumes that the test schema exists and that the collection `my_collection` exists.

MySQL Shell JavaScript Code

```
// Use the collection 'my_collection'
var myColl = db.getCollection('my_collection');

// Find a single document that has a field 'name' that starts with 'L'
var docs = myColl.find('name like :param').
            limit(1).bind('param', 'L%').execute();

print(docs.fetchOne());

// Get all documents with a field 'name' that starts with 'L'
docs = myColl.find('name like :param').
        bind('param','L%').execute();

var myDoc;
while (myDoc = docs.fetchOne()) {
  print(myDoc);
}
```

MySQL Shell Python Code

```
# Use the collection 'my_collection'
myColl = db.get_collection('my_collection')

# Find a single document that has a field 'name' that starts with 'L'
docs = myColl.find('name like :param').limit(1).bind('param', 'L%').execute()

print(docs.fetch_one())

# Get all documents with a field 'name' that starts with 'L'
docs = myColl.find('name like :param').bind('param','L%').execute()

myDoc = docs.fetch_one()
while myDoc:
  print(myDoc)
  myDoc = docs.fetch_one()
```

Node.js JavaScript Code

```
// Use the collection 'my_collection'
var myColl = db.getCollection('my_collection');

// Find a single document that has a field 'name' that starts with 'L'
myColl
  .find('name like :name')
  .bind('name', 'L%')
  .limit(1)
  .execute(function (doc) {
    console.log(doc);
  })
  .then(function () {
    // handle details
  });

// Get all documents with a field 'name' that starts with 'L'
myColl
  .find('name like :name')
  .bind('name', 'L%')
  .execute(function (doc) {
    console.log(doc);
  })
  .then(function () {
    // handle details
  });
```

C# Code

```
// Use the collection "my_collection"
var myColl = db.GetCollection("my_collection");
```

```
// Find a single document that has a field "name" that starts with "L"
var docs = myColl.Find("name like :param")
.Limit(1).Bind("param", "L%").Execute();

Console.WriteLine(docs.FetchOne());

// Get all documents with a field "name" that starts with "L"
docs = myColl.Find("name like :param")
.Bind("param", "L%").Execute();

while (docs.Next())
{
    Console.WriteLine(docs.Current);
}
```

Python Code

```
# Use the collection 'my_collection'
my_coll = my_schema.get_collection('my_collection')

# Find a single document that has a field 'name' that starts with 'L'
docs = my_coll.find('name like :param').limit(1).bind('param', 'L%').execute()

print(docs.fetch_one())

# Get all documents with a field 'name' that starts with 'L'
docs = my_coll.find('name like :param').bind('param', 'L%').execute()

doc = docs.fetch_one()
print(doc)
```

Java Code

```
// Use the collection 'my_collection'
Collection myColl = db.getCollection("my_collection");

// Find a single document that has a field 'name' that starts with 'L'
DocResult docs = myColl.find("name like :name").bind("name", "L%").execute();

System.out.println(docs.fetchOne());

// Get all documents with a field 'name' that starts with 'L'
docs = myColl.find("name like :name").bind("name", "L%").execute();

while (docs.hasNext()) {
    DbDoc myDoc = docs.next();
    System.out.println(myDoc);
}
```

C++ Code

```
// Use the collection 'my_collection'
Collection myColl = db.getCollection("my_collection");

// Find a single document that has a field 'name' that starts with 'L'
DocResult docs = myColl.find("name like :param")
                       .limit(1).bind("param", "L%").execute();

cout << docs.fetchOne() << endl;

// Get all documents with a field 'name' that starts with 'L'
docs = myColl.find("name like :param")
             .bind("param","L%").execute();

DbDoc myDoc;
while ((myDoc = docs.fetchOne()))
{
  cout << myDoc << endl;
}
```

For more information, see CollectionFindFunction.

Collection.modify()

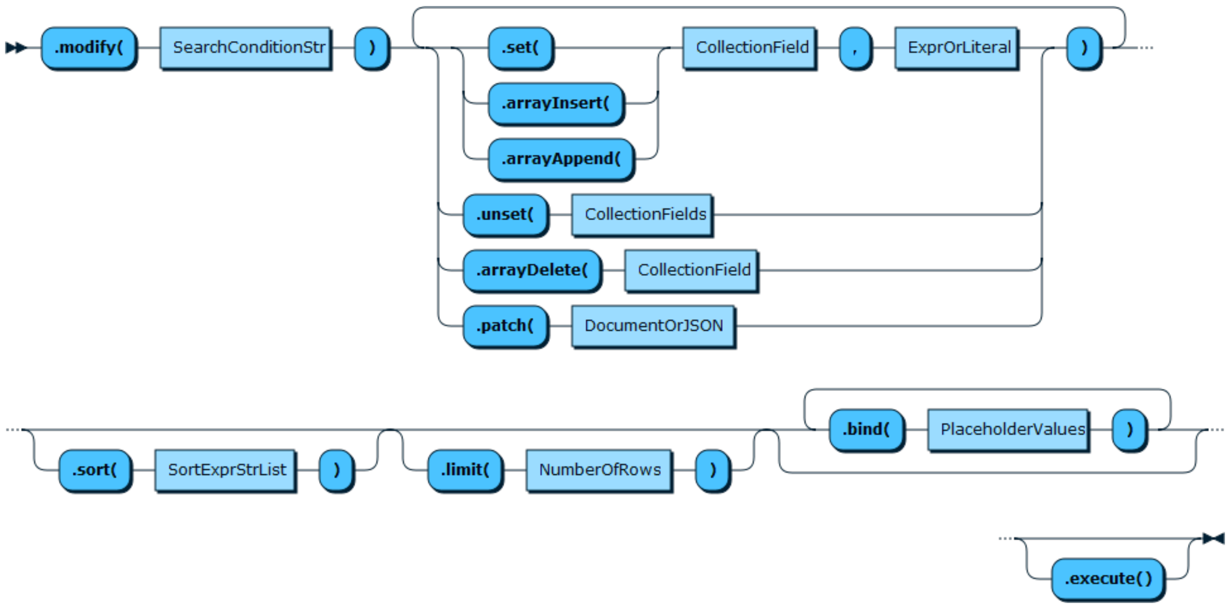


Figure 4.1 Collection.modify() Syntax Diagram

Collection.remove()

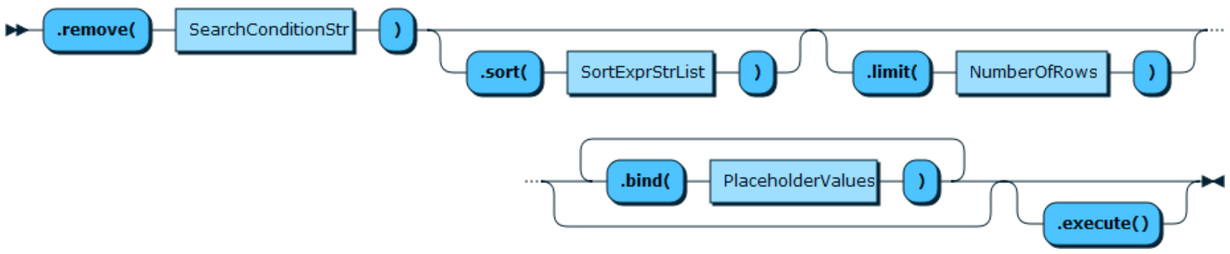


Figure 4.2 Collection.remove() Syntax Diagram

4.4 Single Document Operations

The CRUD commands described at Section 4.3, "Collection CRUD Function Overview" all work on a group of documents in a collection which match a filter. X DevAPI also provides the following operations which work on single documents that are identified by their document ID:

- `Collection.replaceOne(string id, Document doc)` updates or replaces the document identified by id with the provided one, if it exists.
- `Collection.addOrReplaceOne(string id, Document doc)` add the given document. If the id or any other field that has a unique index on it already exists in the collection, the operation updates the matching document instead.
- `Collection.getOne(string id)` returns the document with the given id. This is a shortcut for `Collection.find("_id = :id").bind("id", id).execute().fetchOne()`.
- `Collection.removeOne(string id)` removes the document with the given id. This is a shortcut for `Collection.remove("_id = :id").bind("id", id).execute()`.

Using these operations you can reference documents by ID, making operations on single documents simpler by following a load, modify and save pattern such as the following:

```
doc = collection.getOne(id);
```

```
doc["address"] = "123 Long Street";
collection.replaceOne(id, doc);
```

For more information on document IDs see Section 5.1, “Working with Document IDs”.

Syntax of Single Document Operations

The syntax of the single document operations is as follows:

- `Document getOne(string id)`, where `id` is the document ID of the document to be retrieved. This operation returns the document, or `NULL` if no match is found. Searches for the document that has the given *`id`* and returns it.
- `Result removeOne(string id)`, where *`id`* is the document ID of the document to be removed. This operation returns a `Result` object, which indicates the number of removed documents (1 or 0, if none).
- `Result replaceOne(string id, Document doc)`, where *`id`* is the document ID of the document to be replaced and *`doc`* is the new document, which can contain expressions. If *`doc`* contains an `_id` value, it is ignored. The operation returns a `Result` object, which indicates the number of affected documents (1 or 0, if none). Takes in a document object which replaces the matching document. If no matches are found, the function returns normally with no changes being made.
- `Result addOrReplaceOne(string id, Document doc)`, where `id` is the document ID of the document to be replaced and *`doc`* is the new document, which can contain expressions. If *`doc`* contains an `_id` value, it is ignored. This operation returns a `Result` object, which indicates the number of affected documents (1 or 0, if none). Adds the document to the collection.

Important

These operations accept an explicit `id` to ensure that any changes being made by the operation are applied to the intended document. In many scenarios the document *`doc`* could come from an untrusted user, who could potentially change the `id` in the document and thus replace other documents than the application expects. To mitigate this risk, you should transfer the explicit document `id` through a secure channel.

If *`doc`* has a value for `_id` and it does not match the given *`id`* then an error is generated. If the collection has a document with the given document ID then the collection is checked for any document that conflicts with unique keys from *`doc`* and where the document ID of conflicting documents is not *`id`* then an error is generated. Otherwise the existing document in the collection is replaced by *`doc`*. If the collection has any document that conflicts with unique keys from *`doc`* then an error is generated. Otherwise *`doc`* is added to collection.

`Document getOne(string id)`, where `id` is the document ID of the document to be retrieved. This operation returns the document, or `NULL` if no match is found. Searches for the document that has the given *`id`* and returns it.

`Result removeOne(string id)`, where *`id`* is the document ID of the document to be removed. This operation returns a `Result` object, which indicates the number of removed documents (1 or 0, if none).

Chapter 5 Working with Documents

Table of Contents

Once a collection has been created, it can store JSON documents. You store documents by passing a JSON data structure to the `Collection.add()` function. Some languages have direct support for JSON data, others have an equivalent syntax to represent that data. MySQL Connectors which implement X DevAPI aim to implement support for all methods that are native to the specific language.

In addition, in some MySQL Connectors the generic `DbDoc` objects can be used. The most convenient way to create them is by calling the `Collection.newDoc()`. `DbDoc` is a data type to represent JSON documents and how it is implemented is not defined. Languages implementing X DevAPI are free to follow an object oriented approach with getter and setter methods, or use a C struct style with public members.

For strictly typed languages it is possible to create class files based on the document structure definition of collections. MySQL Shell can be used to create those files.

Document Objects	Supported languages	Advantages
Native JSON	Scripting (JavaScript, Python)	Easy to use
JSON equivalent syntax	C# (Anonymous Types, ExpandoObject)	Easy to use
DbDoc	All languages	Unified across languages
Generated Doc Classes	Strictly typed languages (C#)	Natural to use

The following example shows the different methods of inserting documents into a collection.

MySQL Shell JavaScript Code

```
// Create a new collection 'my_collection'
var myColl = db.createCollection('my_collection');

// Insert JSON data directly
myColl.add({_id: '8901', name: 'Mats', age: 21});

// Inserting several docs at once
myColl.add([ {_id: '8902', name: 'Lotte', age: 24},
  {_id: '8903', name: 'Vera', age: 39} ]);
```

MySQL Shell Python Code

```
// Create a new collection 'my_collection'
var myColl = db.createCollection('my_collection');

// Insert JSON data directly
myColl.add({_id: '8901', name: 'Mats', age: 21});

// Inserting several docs at once
myColl.add([ {_id: '8902', name: 'Lotte', age: 24},
  {_id: '8903', name: 'Vera', age: 39} ]);
```

Node.js JavaScript Code

```
// Create a new collection 'my_collection'
db.createCollection('my_collection').then(function (myColl) {

  // Add a document to insert
  var insert = myColl.add({ name: 'Mats', age: 21 });
  // Add multiple documents to insert
  insert.add([
    { name: 'Lotte', age: 24 },
    { name: 'Vera', age: 39 }
  ]);

  // Add one more document to insert
  var myDoc = {};
  myDoc.name = 'Jamie';
  myDoc.age = 47;
  insert.add(myDoc);

  // run the operation
  return insert.execute();
});
```

C# Code

```
// Create a new collection "my_collection"
var myColl = db.CreateCollection("my_collection");

// Insert JSON data directly
myColl.Add(new { name = "Mats", age = 21 }).Execute();

// Inserting several docs at once
myColl.Add(new[] {new { name = "Lotte", age = 24},
new { name = "Vera", age = 39} }).Execute();

// Insert Documents
var myDoc = new DbDoc();
myDoc.SetValue("name", "Jamie");
myDoc.SetValue("age", 47);
myColl.Add(myDoc).Execute();

//Fetch all docs
var docResult = myColl.Find().Execute();
var docs = docResult.FetchAll();
```

Python Code

```
# Create a new collection 'my_collection'
my_coll = my_schema.create_collection('my_collection')

# Insert JSON data directly
my_coll.add({'name': 'Mats', 'age': 21})

# Inserting several docs at once
my_coll.add([
    {'name': 'Lotte', 'age': 24},
    {'name': 'Vera', 'age': 39}
])
```

Java Code

```
// Create a new collection 'my_collection'
Collection coll = db.createCollection("my_collection");

// Insert JSON data directly
coll.add("{\"name\":\"Mats\", \"age\":21}");

// Insert several documents at once
coll.add("{\"name\":\"Lotte\", \"age\":24}",
        "{\"name\":\"Vera\", \"age\":39}");

// Insert Documents
DbDoc myDoc = new coll.newDoc();
myDoc.add("name", new JsonString().setValue("Jamie"));
myDoc.add("age", new JsonNumber().setValue("47"));
coll.add(myDoc);
```

C++ Code

```
// Create a new collection 'my_collection'
Collection myColl = db.createCollection("my_collection");

// Insert JSON data directly
myColl.add(R"({"name": "Mats", "age": 21})").execute();

// Inserting several docs at once
std::list<DbDoc> docs = {
  DbDoc(R"({"name": "Lotte", "age": 24})"),
  DbDoc(R"({"name": "Vera", "age": 39})")
};
myColl.add(docs).execute();
```

5.1 Working with Document IDs

This section describes how to work with document IDs. Every document has a unique identifier called the document ID, which can be thought of as the equivalent of a table's primary key. The document ID value is usually automatically generated by the server when the document is added, but can also be manually assigned. The assigned document ID is returned in the result of the `collection.add()` operation. See Section 5.1.1, “Understanding Document IDs” for more background information on document IDs.

The following example in JavaScript code shows adding a document to a collection, retrieving the added document's IDs and testing that duplicate IDs cannot be added.

```
mysql-js > mycollection.add({test:'demo01'})
Query OK, 1 item affected (0.00 sec)

mysql-js > mycollection.add({test:'demo02'}).add({test:'demo03'})
Query OK, 2 items affected (0.00 sec)

mysql-js > mycollection.find()
[
    {
        "_id": "00005a640138000000000000002c",
        "test": "demo01"
    },
    {
        "_id": "00005a640138000000000000002d",
        "test": "demo02"
    },
    {
        "_id": "00005a640138000000000000002e",
        "test": "demo03"
    }
]
3 documents in set (0.00 sec)

mysql-js > mycollection.add({_id:'00005a640138000000000000002f', test:'demo04'})
Query OK, 1 item affected (0.00 sec)

mysql-js > mycollection.find()
[
    {
        "_id": "00005a640138000000000000002c",
        "test": "demo01"
    },
    {
        "_id": "00005a640138000000000000002d",
        "test": "demo02"
    },
    {
        "_id": "00005a640138000000000000002e",
        "test": "demo03"
    },
    {
        "_id": "00005a640138000000000000002f",
        "test": "demo04"
    }
]
4 documents in set (0.00 sec)

mysql-js > mycollection.add({test:'demo05'})
ERROR: 5116: Document contains a field value that is not unique but required to be
```

5.1.1 Understanding Document IDs

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. `InnoDB` uses the document ID as a primary key, therefore these sequential primary keys for all clients result in efficient page splits and tree reorganizations.

This section describes the properties and format of the automatically generated document IDs.

Document ID Properties

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.

Important

If you are using manual document IDs, you must ensure that IDs from the server's automatically generated document ID sequence are never used. X Plugin is not aware of the data inserted into the collection, including any IDs you use. Thus in future inserts, 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.

Whenever 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 cluster. Use the `mysqlx_document_id_unique_prefix` option to ensure that the `unique_prefix` part of the document ID is unique to the cluster.

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

This section describes how document IDs are formatted. 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 instances 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.

Chapter 6 Working with Relational Tables

Table of Contents

This section explains how to use X DevAPI SQL CRUD functions to work with relational tables.

The following example code compares the operations previously shown for collections and how they can be used to work with relational tables using SQL. The simplified X DevAPI syntax is demonstrated using SQL in a Session and showing how it is similar to working with documents.

MySQL Shell JavaScript Code

```
// Working with Relational Tables
var mysqlx = require('mysqlx');

// Connect to server using a connection URL
var mySession = mysqlx.getSession( {
  host: 'localhost', port: 33060,
  user: 'user', password: 'password'} )

var myDb = mySession.getSchema('test');

// Accessing an existing table
var myTable = myDb.getTable('my_table');

// Insert SQL Table data
myTable.insert(['name', 'birthday', 'age']).
  values('Laurie', mysqlx.dateValue(2000, 5, 27), 19).execute();

// Find a row in the SQL Table
var myResult = myTable.select(['_id', 'name', 'birthday']).
  where('name like :name AND age < :age').
  bind('name', 'L%').bind('age', 30).execute();

// Print result
print(myResult.fetchOne());
```

MySQL Shell Python Code

```
# Working with Relational Tables
from mysqlsh import mysqlx

# Connect to server using a connection URL
mySession = mysqlx.get_session( {
  'host': 'localhost', 'port': 33060,
  'user': 'user', 'password': 'password'} )

myDb = mySession.get_schema('test')

# Accessing an existing table
myTable = myDb.get_table('my_table')

# Insert SQL Table data
myTable.insert(['name','birthday','age']) \
  .values('Laurie', mysqlx.date_value(2000, 5, 27), 19).execute()

# Find a row in the SQL Table
myResult = myTable.select(['_id', 'name', 'birthday']) \
  .where('name like :name AND age < :age') \
  .bind('name', 'L%') \
  .bind('age', 30).execute()
```

```
# Print result
print(myResult.fetch_all())
```

Node.js JavaScript Code

```
// Working with Relational Tables
var mysqlx = require('@mysql/xdevapi');
var myTable;

// Connect to server using a connection URL
mysqlx
  .getSession({
    user: 'user',
    password: 'password',
    host: 'localhost',
    port: 33060
  })
  .then(function (session) {
    // Accessing an existing table
    myTable = session.getSchema('test').getTable('my_table');

    // Insert SQL Table data
    return myTable
      .insert(['name', 'birthday', 'age'])
      .values(['Laurie', '2000-5-27', 19])
      .execute()
  })
  .then(function () {
    // Find a row in the SQL Table
    return myTable
        .select(['_id', 'name', 'birthday'])
        .where('name like :name && age < :age)')
        .bind('name', 'L%')
        .bind('age', 30)
        .execute(function (row) {
          console.log(row);
        });
  })
  .then(function (myResult) {
    console.log(myResult);
  });
```

C# Code

```
// Working with Relational Tables

// Connect to server using a connection
var db = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password")
.GetSchema("test");

// Accessing an existing table
var myTable = db.GetTable("my_table");

// Insert SQL Table data
myTable.Insert("name", "age")
.Values("Laurie", "19").Execute();

// Find a row in the SQL Table
var myResult = myTable.Select("_id, name, age")
.Where("name like :name AND age < :age")
.Bind(new { name = "L%", age = 30 }).Execute();

// Print result
PrintResult(myResult.FetchAll());
```

Python Code

```
# Working with Relational Tables
import mysqlx
```

```
# Connect to server using a connection URL
my_session = mysqlx.get_session({
    'host': 'localhost', 'port': 33060,
    'user': 'user', 'password': 'password'
})

my_schema = my_session.get_schema('test')

# Accessing an existing table
my_table = my_schema.get_table('my_table')

# Insert SQL Table data
my_table.insert(['name', 'birthday', 'age']) \
    .values('Laurie', mysqlx.date_value(2000, 5, 27), 19).execute()

# Find a row in the SQL Table
result = my_table.select(['_id', 'name', 'birthday']) \
    .where('name like :name AND age < :age') \
    .bind('name', 'L%') \
    .bind('age', 30).execute()

# Print result
print(result.fetch_all())
```

Java Code

```
// Working with Relational Tables
import com.mysql.cj.xdevapi.*;

// Connect to server using a connection URL
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=p
Schema db = mySession.getSchema("test");

// Accessing an existing table
Table myTable = db.getTable("my_table");

// Insert SQL Table data
myTable.insert("name", "birthday").values("Laurie", "2000-05-27").execute();

// Find a row in the SQL Table
RowResult myResult = myTable.select("_id, name, birthday")
  .where("name like :name AND age < :age")
  .bind("name", "L%").bind("age", 30).execute();

// Print result
System.out.println(myResult.fetchAll());
```

C++ Code

```
// Working with Relational Tables
#include <mysqlx/xdevapi.h>

// Connect to server using a connection URL
Session mySession(33060, "user", "password");

Schema myDb = mySession.getSchema("test");

// Accessing an existing table
Table myTable = myDb.getTable("my_table");

// Insert SQL Table data
myTable.insert("name", "birthday", "age")
       .values("Laurie", "2000-5-27", 19).execute();

// Find a row in the SQL Table
RowResult myResult = myTable.select("_id", "name", "birthday")
  .where("name like :name AND age < :age")
  .bind("name", "L%").bind("age", 30).execute();

// Print result
Row row = myResult.fetchOne();
```

```
cout << "     _id: " << row[0] << endl;
cout << "    name: " << row[1] << endl;
cout << "birthday: " << row[2] << endl;
```

6.1 SQL CRUD Functions

The following SQL CRUD functions are available in X DevAPI.

Table.insert()

The `Table.insert()` function is used to store data in a relational table in the database. It is executed by the `execute()` function.

The following example shows how to use the Table.insert() function. The example assumes that the `test` schema exists and is assigned to the variable `db`, and that an empty table called `my_table` exists.

MySQL Shell JavaScript Code

```
// Accessing an existing table
var myTable = db.getTable('my_table');

// Insert a row of data.
myTable.insert(['id', 'name']).
        values(1, 'Imani').
        values(2, 'Adam').
        execute();
```

MySQL Shell Python Code

```
# Accessing an existing table
myTable = db.get_table('my_table')

# Insert a row of data.
myTable.insert(['id', 'name']).values(1, 'Imani').values(2, 'Adam').execute()
```

Node.js JavaScript Code

```
// Accessing an existing table
var myTable = db.getTable('my_table');

// Insert a row of data.
myTable.insert(['id', 'name']).
        values(1, 'Imani').
        values(2, 'Adam').
        execute();
```

C# Code

```
// Assumptions: test schema assigned to db, empty my_table table exists

// Accessing an existing table
var myTable = db.GetTable("my_table");

// Insert a row of data.
myTable.Insert("id", "name")
.Values(1, "Imani")
.Values(2, "Adam")
.Execute();
```

Python Code

```
# Accessing an existing table
my_table = db.get_table('my_table')

# Insert a row of data.
```

```
my_table.insert(['id', 'name']).values(1, 'Imani').values(2, 'Adam').execute()
```

Java Code

```
// Accessing an existing table
Table myTable = db.getTable("my_table");

// Insert a row of data.
myTable.insert("id", "name")
  .values(1, "Imani")
  .values(2, "Adam")
  .execute();
```

C++ Code

```
// Accessing an existing table
var myTable = db.getTable("my_table");

// Insert a row of data.
myTable.insert("id", "name")
       .values(1, "Imani")
       .values(2, "Adam")
       .execute();
```

Figure 6.1 Table.insert() Syntax Diagram

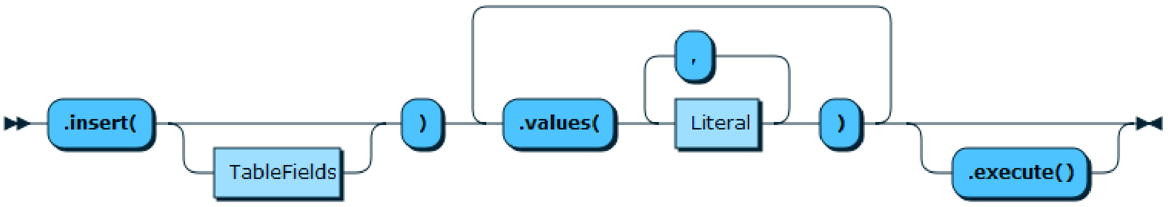


Table.select()

`Table.select()` and `collection.find()` use different methods for sorting results. `Table.select()` follows the SQL language naming and calls the sort method `orderBy()`. `Collection.find()` does not. Use the method `sort()` to sort the results returned by `Collection.find()`. Proximity with the SQL standard is considered more important than API uniformity here.

Figure 6.2 Table.select() Syntax Diagram

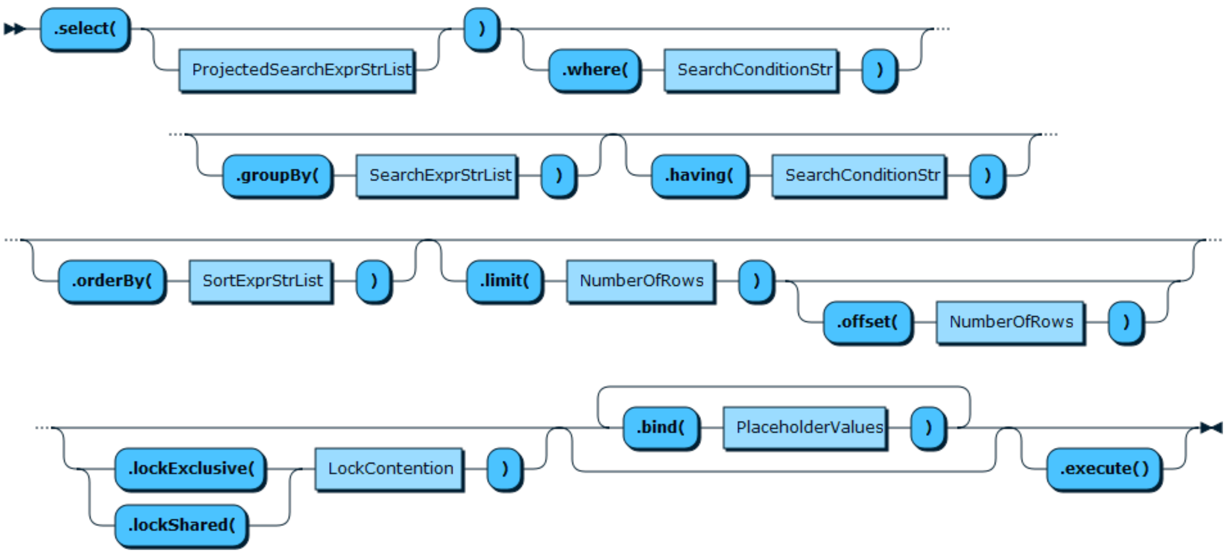


Table.update()

Figure 6.3 Table.update() Syntax Diagram

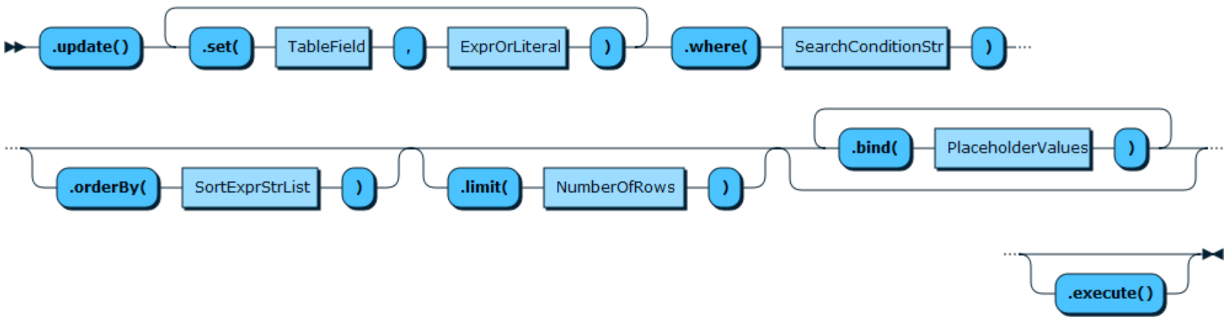
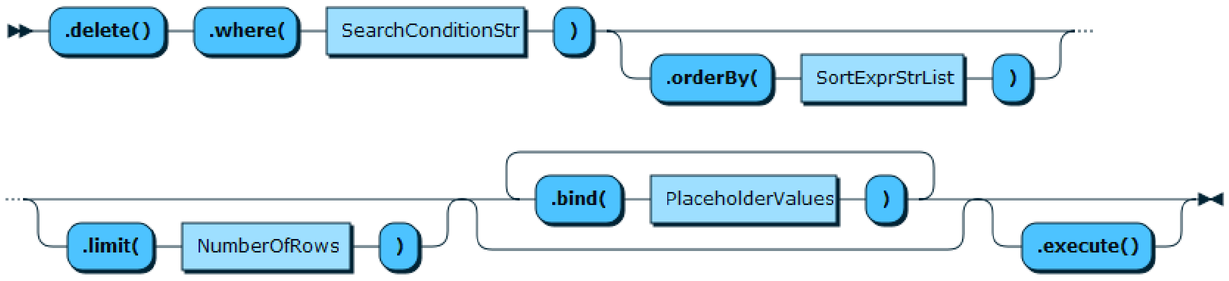


Table.delete()

Figure 6.4 Table.delete() Syntax Diagram



Chapter 7 Working with Relational Tables and Documents

Table of Contents

After seeing how to work with documents and how to work with relational tables, this section explains how to combine the two and work with both at the same time.

It can be beneficial to use documents for very specific tasks inside an application and rely on relational tables for other tasks. Or a very simple document only application can outgrow the document model and incrementally integrate or move to a more powerful relational database. This way the advantages of both documents and relational tables can be combined. SQL tables contribute strictly typed value semantics, predictable and optimized storage. Documents contribute type flexibility, schema flexibility and non-scalar types.

7.1 Collections as Relational Tables

Applications that seek to store standard SQL columns with Documents can cast a collection to a table. In this case a collection can be fetched as a Table object with the `Schema.getCollectionAsTable()` function. From that moment on it is treated as a regular table. Document values can be accessed in SQL CRUD operations using the following syntax:

```
doc->'$.field'
```

`doc->'$.field'` is used to access the document top level fields. More complex paths can be specified as well.

```
doc->'$.some.field.like[3].this'
```

Once a collection has been fetched as a table with the `Schema.getCollectionAsTable()` function, all SQL CRUD operations can be used. Using the syntax for document access, you can select data from the Documents of the Collection and the extra SQL columns.

The following example shows how to insert a JSON document string into the `doc` field.

MySQL Shell JavaScript Code

```
// Get the customers collection as a table
var customers = db.getCollectionAsTable('customers');
customers.insert('doc').values('{"_id":"001", "name": "Ana", "last_name": "Silva"}').execute();

// Now do a find operation to retrieve the inserted document
var result = customers.select(["doc->'$.name'", "doc->'$.last_name'"]).where("doc->'$._id' = '001'").ex

var record = result.fetchOne();

print ("Name : "  + record[0]);
print ("Last Name : "  + record[1]);
```

MySQL Shell Python Code

```
# Get the customers collection as a table
customers = db.get_collection_as_table('customers')
customers.insert('doc').values('{"_id":"001", "name": "Ana", "last_name": "Silva"}').execute()

# Now do a find operation to retrieve the inserted document
result = customers.select(["doc->'$.name'", "doc->'$.last_name'"]).where("doc->'$._id' = '001'").execut

record = result.fetch_one()

print("Name : %s\n"  % record[0])
print("Last Name : %s\n"  % record[1])
```

Node.js JavaScript Code

```
// Get the customers collection as a table
var customers = db.getCollectionAsTable('customers');
customers.insert('doc').values('{"_id":"001", "name": "Ana"}').execute();
```

C# Code

```
// Get the customers collection as a table
var customers = db.GetCollectionAsTable("customers");
customers.Insert("doc").Values("{ \"_id\": 1, \"name\": \"Ana\" }").Execute();
```

Python Code

```
# Get the customers collection as a table
customers = db.get_collection_as_table("customers")
customers.insert('doc').values({'_id':'001', 'name': 'Ana', 'last_name': 'Silva'}).execute()

# Now do a find operation to retrieve the inserted document
result = customers.select(["doc->'$.name'", "doc->'$.last_name'"]).where("doc->'$._id' = '001'").execute()

record = result.fetch_one()

print('Name : {0}'.format(record[0]))
print('Last Name : {0}'.format(record[1]))
```

Java Code

```
// Get the customers collection as a table
Table customers = db.getCollectionAsTable("customers");
customers.insert("doc").values("{\"name\": \"Ana\"}").execute();
```

C++ Code

```
// Get the customers collection as a table
Table customers = db.getCollectionAsTable("customers");
customers.insert("doc")
         .values(R"({"_id":"001", "name": "Ana", "last_name": "Silva"})").execute();

// Now do a find operation to retrieve the inserted document
RowResult result = customers.select("doc->'$.name'", "doc->'$.last_name'")
                            .where("doc->'$._id' = '001'").execute();

Row record = result.fetchOne();
cout << "Name : " << record[0] << endl;
cout << "Last Name : " << record[1] << endl;
```

Chapter 8 Statement Execution

Table of Contents

This section explains statement execution, with information on how to handle transactions and errors.

8.1 Transaction Handling

Transactions can be used to group operations into an atomic unit. Either all operations of a transaction succeed when they are committed, or none. It is possible to roll back a transaction as long as it has not been committed.

Transactions can be started in a session using the `startTransaction()` method, committed with `commitTransaction()` and cancelled or rolled back with `rollbackTransaction()`. This is illustrated in the following example. The example assumes that the `test` schema exists and that the collection `my_collection` does not exist.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

// Connect to server
var session = mysqlx.getSession( {
  host: 'localhost', port: 33060,
  user: 'user', password: 'password' } );

// Get the Schema test
var db = session.getSchema('test');

// Create a new collection
var myColl = db.createCollection('my_collection');

// Start a transaction
session.startTransaction();
try {
  myColl.add({name: 'Rohit', age: 18, height: 1.76}).execute();
  myColl.add({name: 'Misaki', age: 24, height: 1.65}).execute();
  myColl.add({name: 'Leon', age: 39, height: 1.9}).execute();

  // Commit the transaction if everything went well
  session.commit();

  print('Data inserted successfully.');
}
catch (err) {
  // Rollback the transaction in case of an error
  session.rollback();

  // Printing the error message
  print('Data could not be inserted: ' + err.message);
}
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

# Connect to server
```

```
mySession = mysqlx.get_session( {
        'host': 'localhost', 'port': 33060,
        'user': 'user', 'password': 'password' } )

# Get the Schema test
myDb = mySession.get_schema('test')

# Create a new collection
myColl = myDb.create_collection('my_collection')

# Start a transaction
mySession.start_transaction()
try:
    myColl.add({'name': 'Rohit', 'age': 18, 'height': 1.76}).execute()
    myColl.add({'name': 'Misaki', 'age': 24, 'height': 1.65}).execute()
    myColl.add({'name': 'Leon', 'age': 39, 'height': 1.9}).execute()
    # Commit the transaction if everything went well
    mySession.commit()
    print('Data inserted successfully.')
except Exception as err:
    # Rollback the transaction in case of an error
    mySession.rollback()

    # Printing the error message
    print('Data could not be inserted: %s' % str(err))
```

C# Code

```
// Connect to server
var session = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");

// Get the Schema test
var db = session.GetSchema("test");

// Create a new collection
var myColl = db.CreateCollection("my_collection");

// Start a transaction
session.StartTransaction();
try
{
 myColl.Add(new { name = "Rohit", age = 18, height = 1.76}).Execute();
 myColl.Add(new { name = "Misaki", age = 24, height = 1.65}).Execute();
 myColl.Add(new { name = "Leon", age = 39, height = 1.9}).Execute();

 // Commit the transaction if everything went well
 session.Commit();

 Console.WriteLine("Data inserted successfully.");
}
catch(Exception err)
{
 // Rollback the transaction in case of an error
 session.Rollback();

 // Printing the error message
 Console.WriteLine("Data could not be inserted: " + err.Message);
}
```

Python Code

```
import mysqlx

# Connect to server
my_session = mysqlx.get_session({
    'host': 'localhost', 'port': 33060,
    'user': 'user', 'password': 'password'
})

# Get the Schema test
my_schema = my_session.get_schema('test')
```

```
# Create a new collection
my_coll = my_schema.create_collection('my_collection')

# Start a transaction
session.start_transaction()
try:
    my_coll.add({'name': 'Rohit', 'age': 18, 'height': 1.76}).execute()
    my_coll.add({'name': 'Misaki', 'age': 24, 'height': 1.65}).execute()
    my_coll.add({'name': 'Leon', 'age': 39, 'height': 1.9}).execute()

    # Commit the transaction if everything went well
    my_session.commit()

    print('Data inserted successfully.')
except Exception as err:
    # Rollback the transaction in case of an error
    my_session.rollback()

    # Printing the error message
    print('Data could not be inserted: {0}'.format(str(err)))
```

Java Code

```
import com.mysql.cj.xdevapi.*;

// Connect to server
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=p

Schema db = mySession.getSchema("test");

// Create a new collection
Collection myColl = db.createCollection("my_collection");

// Start a transaction
mySession.startTransaction();
try {
    myColl.add("{\"name\":\"Rohit\", \"age\":18}", "{\"name\":\"Misaki\", \"age\":24}", "{\"name\":\"Le

    mySession.commit();
    System.out.println("Data inserted successfully.");
} catch (Exception err) {
    // Rollback the transaction in case of an error
    mySession.rollback();

    // Printing the error message
    System.out.println("Data could not be inserted: " + err.getMessage());
}
```

C++ Code

```
// Connect to server
Session session(SessionOption::HOST, "localhost",
                SessionOption::PORT, 33060,
                SessionOption::USER, "user",
                SessionOption::PWD, "password");

// Get the Schema test
Schema db = session.getSchema("test");

// Create a new collection
Collection myColl = db.createCollection("my_collection");

// Start a transaction
session.startTransaction();
try {
  myColl.add(R"({"name": "Rohit", "age": 18, "height": 1.76})").execute();
  myColl.add(R"({"name": "Misaki", "age": 24, "height": 1.65})").execute();
  myColl.add(R"({"name": "Leon", "age": 39, "height": 1.9})").execute();

  // Commit the transaction if everything went well
  session.commit();

  cout << "Data inserted successfully." << endl;
}
catch (const Error &err) {
  // Rollback the transaction in case of an error
  session.rollback();

  // Printing the error message
  cout << "Data could not be inserted: " << err << endl;
}
```

8.1.1 Processing Warnings

Similar to the execution of single statements committing or rolling back a transaction can also trigger warnings. To be able to process these warnings the replied result object of `Session.commit();` or `Session.rollback();` needs to be checked.

This is shown in the following example. The example assumes that the test schema exists and that the collection `my_collection` does not exist.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

// Connect to server
var mySession = mysqlx.getSession( {
        host: 'localhost', port: 33060,
        user: 'user', password: 'password' } );

// Get the Schema test
var myDb = mySession.getSchema('test');

// Create a new collection
var myColl = myDb.createCollection('my_collection');

// Start a transaction
mySession.startTransaction();
try
{
    myColl.add({'name': 'Rohit', 'age': 18, 'height': 1.76}).execute();
    myColl.add({'name': 'Misaki', 'age': 24, 'height': 1.65}).execute();
    myColl.add({'name': 'Leon', 'age': 39, 'height': 1.9}).execute();

    // Commit the transaction if everything went well
    var reply = mySession.commit();

    // handle warnings
    if (reply.warningCount){
      var warnings = reply.getWarnings();
      for (index in warnings){
        var warning = warnings[index];
        print ('Type ['+ warning.level + '] (Code ' + warning.code + '): ' + warning.message + '\n');
      }
    }

    print ('Data inserted successfully.');
}
catch(err)
{
    // Rollback the transaction in case of an error
    reply = mySession.rollback();

    // handle warnings
    if (reply.warningCount){
      var warnings = reply.getWarnings();
      for (index in warnings){
        var warning = warnings[index];
        print ('Type ['+ warning.level + '] (Code ' + warning.code + '): ' + warning.message + '\n');
      }
    }

    // Printing the error message
    print ('Data could not be inserted: ' + err.message);
}
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

# Connect to server
mySession = mysqlx.get_session( {
        'host': 'localhost', 'port': 33060,
        'user': 'user', 'password': 'password' } )

# Get the Schema test
myDb = mySession.get_schema('test')

# Create a new collection
myColl = myDb.create_collection('my_collection')

# Start a transaction
mySession.start_transaction()
try:
    myColl.add({'name': 'Rohit', 'age': 18, 'height': 1.76}).execute()
    myColl.add({'name': 'Misaki', 'age': 24, 'height': 1.65}).execute()
    myColl.add({'name': 'Leon', 'age': 39, 'height': 1.9}).execute()

    # Commit the transaction if everything went well
    reply = mySession.commit()

    # handle warnings
    if reply.warning_count:
      for warning in result.get_warnings():
        print('Type [%s] (Code %s): %s\n' % (warning.level, warning.code, warning.message))

    print('Data inserted successfully.')
except Exception as err:
    # Rollback the transaction in case of an error
    reply = mySession.rollback()

    # handle warnings
    if reply.warning_count:
      for warning in result.get_warnings():
        print('Type [%s] (Code %s): %s\n' % (warning.level, warning.code, warning.message))

    # Printing the error message
    print('Data could not be inserted: %s' % str(err))
```

C# Code

```
// Connect to server
var session = MySQLX.GetSession("server=localhost;port=33060;user=user;password=password;");

// Get the Schema test
var db = session.GetSchema("test");

// Create a new collection
var myColl = db.CreateCollection("my_collection");

// Start a transaction
session.StartTransaction();
int warningCount = 0;
try
{
 var result = myColl.Add(new { name = "Rohit", age = 18, height = 1.76}).Execute();
 warningCount += result.Warnings.Count;
 result = myColl.Add(new { name = "Misaki", age = 24, height = 1.65}).Execute();
 warningCount += result.Warnings.Count;
 result = myColl.Add(new { name = "Leon", age = 39, height = 1.9}).Execute();
 warningCount += result.Warnings.Count;

 // Commit the transaction if everything went well
 session.Commit();
 if(warningCount > 0)
 {
   // handle warnings
 }

 Console.WriteLine("Data inserted successfully.");
}
catch (Exception err)
{
 // Rollback the transaction in case of an error
 session.Rollback();
 if(warningCount > 0)
 {
   // handle warnings
 }

 // Printing the error message
 Console.WriteLine("Data could not be inserted: " + err.Message);
}
```

Python Code

```
import mysqlx

# Connect to server
my_session = mysqlx.get_session({
    'host': 'localhost', 'port': 33060,
    'user': 'user', 'password': 'password'
})

# Get the Schema test
my_schema = my_session.get_schema('test')

# Create a new collection
my_coll = my_schema.create_collection('my_collection')

# Start a transaction
my_session.start_transaction()
try:
    my_coll.add({'name': 'Rohit', 'age': 18, 'height': 1.76}).execute()
    my_coll.add({'name': 'Misaki', 'age': 24, 'height': 1.65}).execute()
    my_coll.add({'name': 'Leon', 'age': 39, 'height': 1.9}).execute()

    # Commit the transaction if everything went well
    result = my_session.commit()

    # handle warnings
    if result.get_warnings_count() > 0:
        for warning in result.get_warnings():
            print('Type [{0}] (Code {1}): {2}'.format(warning['level'], warning['code'], warning['msg']))

    print('Data inserted successfully.')
except Exception as err:
    # Rollback the transaction in case of an error
    result = my_session.rollback()

    # handle warnings
    if result.get_warnings_count() > 0:
        for warning in result.get_warnings():
            print('Type [{0}] (Code {1}): {2}'.format(warning['level'], warning['code'], warning['msg']))

    # Printing the error message
    print('Data could not be inserted: {0}'.format(err))
```

Java Code

```
// c.f. "standard transaction handling"
```

C++ Code

```
/*
  Connector/C++ does not yet provide access to transaction warnings
  -- Session methods commit() and rollback() do not return a result object.
*/
```

By default all warnings are sent from the server to the client. If an operation is known to generate many warnings and the warnings are of no value to the application then sending the warnings can be suppressed. This helps to save bandwith. `session.setFetchWarnings()` controls whether warnings are discarded at the server or are sent to the client. `session.getFetchWarnings()` is used to learn the currently active setting.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

function process_warnings(result){
  if (result.getWarningCount()){
    var warnings = result.getWarnings();
    for (index in warnings){
      var warning = warnings[index];
      print ('Type ['+ warning.level + '] (Code ' + warning.code + '): ' + warning.message + '\n');
    }
  }
  else{
    print ("No warnings were returned.\n");
  }
}

// Connect to server
var mySession = mysqlx.getSession( {
  host: 'localhost', port: 33060,
  user: 'user', password: 'password' } );

// Disables warning generation
mySession.setFetchWarnings(false);
var result = mySession.sql('drop schema if exists unexisting').execute();
process_warnings(result);

// Enables warning generation
mySession.setFetchWarnings(true);
var result = mySession.sql('drop schema if exists unexisting').execute();
process_warnings(result);
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

def process_warnings(result):
  if result.get_warnings_count():
    for warning in result.get_warnings():
      print('Type [%s] (Code %s): %s\n' % (warning.level, warning.code, warning.message))
  else:
    print("No warnings were returned.\n")


# Connect to server
mySession = mysqlx.get_session( {
  'host': 'localhost', 'port': 33060,
  'user': 'user', 'password': 'password' } );

# Disables warning generation
mySession.set_fetch_warnings(False)
result = mySession.sql('drop schema if exists unexisting').execute()
process_warnings(result)

# Enables warning generation
mySession.set_fetch_warnings(True)
result = mySession.sql('drop schema if exists unexisting').execute()
process_warnings(result)
```

Java Code

```
// Connect to server
Session mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=passw

Schema db = mySession.getSchema("test");

// Create a new collection
Collection myColl = db.createCollection("my_collection");

// Start a transaction
mySession.startTransaction();
try {
    Result res = myColl.add("{\"name\":\"Rohit\", \"age\":18}", "{\"name\":\"Misaki\", \"age\":24}", "{\"na

    System.out.println(res.getWarningsCount());

    Iterator<Warning> warnings = res.getWarnings();
    while (warnings.hasNext()) {
        Warning warn = warnings.next();
        System.out.println(warn.getCode() + ", " + warn.getLevel() + ", " + warn.getMessage());
    }

    mySession.commit();
    System.out.println("Data inserted successfully.");
} catch (Exception err) {
    // Rollback the transaction in case of an error
    mySession.rollback();

    // Printing the error message
    System.out.println("Data could not be inserted: " + err.getMessage());
}
```

8.1.2 Error Handling

When writing scripts for MySQL Shell you can often simply rely on the exception handling done by MySQL Shell. For all other languages either proper exception handling is required to catch errors or the traditional error handling pattern needs to be used if the language does not support exceptions.

The default error handling can be changed by creating a custom `SessionContext` and passing it to the `mysqlx.getSession()` function. This enables switching from exceptions to result based error checking.

The following example shows how to perform proper error handling. The example assumes that the test schema exists and that the collection `my_collection` exists.

MySQL Shell JavaScript Code

```
var mysqlx = require('mysqlx');

var mySession;

try {
  // Connect to server on localhost
  mySession = mysqlx.getSession( {
    host: 'localhost', port: 33060,
    user: 'user', password: 'password' } );
}
catch (err) {
  print('The database session could not be opened: ' + err.message);
}

try {
  var myDb = mySession.getSchema('test');

  // Use the collection 'my_collection'
  var myColl = myDb.getCollection('my_collection');
  // Find a document
  var myDoc = myColl.find('name like :param').limit(1)
    .bind('param','L%').execute();

  // Print document
  print(myDoc.first());
}
catch (err) {
  print('The following error occurred: ' + err.message);
}
finally {
  // Close the session in any case
  mySession.close();
}
```

MySQL Shell Python Code

```
from mysqlsh import mysqlx

mySession

try:
        # Connect to server on localhost
        mySession = mysqlx.get_session( {
                'host': 'localhost', 'port': 33060,
                'user': 'user', 'password': 'password' } )

except Exception as err:
        print('The database session could not be opened: %s' % str(err))

try:
        myDb = mySession.get_schema('test')

        # Use the collection 'my_collection'
        myColl = myDb.get_collection('my_collection')

        # Find a document
        myDoc = myColl.find('name like :param').limit(1).bind('param','L%').execute()

        # Print document
        print(myDoc.first())
except Exception as err:
        print('The following error occurred: %s' % str(err))
finally:
        # Close the session in any case
        mySession.close()
```

Node.js JavaScript Code

```
var mysqlx = require('@mysql/xdevapi');

// Connect to server on localhost
mysqlx
  .getSession({
    host: 'localhost',
    port: 33060,
    user: 'user',
    password: 'password'
  })
  .then(function (mySession) {
    // This can't throw an error as we check existence at a later operation only
    var myDb = mySession.getSchema('test');

    // Use the collection 'my_collection'
    // This can't throw an error as we check existence at a later operation only
    var myColl = myDb.getCollection('my_collection');

    // Find a document
    return myColl
```

```
      .find('name like :param')
      .limit(1)
      .bind('param','L%')
      .execute(function (row) {
        console.log(row);
      })
      .then(function () {
        return session.close();
      })
      .catch(function (err) {
        console.log('The following error occurred: ' + err.message);
      });
  })
  .catch (err) {
    console.log('The database session could not be opened: ' + err.message);
  });
```

C# Code

```
Session mySession = null;
try
{
  // Connect to server on localhost
  mySession = MySQLX.GetSession("mysqlx://user:password@localhost:33060");

  try
  {
    Schema myDb = mySession.GetSchema("test");

    // Use the collection 'my_collection'
    Collection myColl = myDb.GetCollection("my_collection");

    // Find a document
    DocResult myDoc = myColl.Find("name like :param").Limit(1).Bind("param", "L%").Execute();

    // Print document
    Console.WriteLine(myDoc.FetchOne());
  }
  catch (Exception err)
  {
    Console.WriteLine("The following error occurred: " + err.Message);
  }
  finally
  {
    // Close the session in any case
    mySession.Close();
  }
}
catch (Exception err)
{
  Console.WriteLine("The database session could not be opened: " + err.Message);
}
```

Python Code

```
import mysqlx

# Connect to server
my_session = mysqlx.get_session({
    'host': 'localhost', 'port': 33060,
    'user': 'user', 'password': 'password'
})

# Get the Schema test
my_schema = my_session.get_schema('test')

# Create a new collection
my_coll = my_schema.create_collection('my_collection')

# Start a transaction
my_session.start_transaction()
try:
    my_coll.add({'name': 'Rohit', 'age': 18, 'height': 1.76}).execute()
    my_coll.add({'name': 'Misaki', 'age': 24, 'height': 1.65}).execute()
    my_coll.add({'name': 'Leon', 'age': 39, 'height': 1.9}).execute()

    # Commit the transaction if everything went well
    result = my_session.commit()

    # handle warnings
    if result.get_warnings_count() > 0:
        for warning in result.get_warnings():
            print('Type [{0}] (Code {1}): {2}'.format(warning['level'], warning['code'], warning['msg']

    print('Data inserted successfully.')
except Exception as err:
    # Rollback the transaction in case of an error
    my_session.rollback()

    # handle warnings
    if reply.get_warnings_count() > 0:
        for warning in result.get_warnings():
            print('Type [{0}] (Code {1}): {2}'.format(warning['level'], warning['code'], warning['msg']

    # Printing the error message
    print('Data could not be inserted: {0}'.format(err))
```

Java Code

```
import com.mysql.cj.xdevapi.*;

Session mySession;

try {
    // Connect to server on localhost
    mySession = new SessionFactory().getSession("mysqlx://localhost:33060/test?user=user&password=passw

    try {
        Schema myDb = mySession.getSchema("test");

        // Use the collection 'my_collection'
        Collection myColl = myDb.getCollection("my_collection");

        // Find a document
        DocResult myDoc = myColl.find("name like :param").limit(1).bind("param", "L%").execute();

        // Print document
        System.out.println(myDoc.fetchOne());
    } catch (XDevAPIError err) { // special exception class for server errors
        System.err.println("The following error occurred: " + err.getMessage());
    } finally {
        // Close the session in any case
        mySession.close();
    }
} catch (Exception err) {
    System.err.println("The database session could not be opened: " + err.getMessage());
}
```

C++ Code

```
#include <mysqlx/xdevapi.h>

try
{
  // Connect to server on localhost
  Session session(33060, "user", "password");

  try
  {
    Schema db = session.getSchema("test");

    // Use the collection 'my_collection'
    Collection myColl = db.getCollection("my_collection");
    // Find a document
    auto myDoc = myColl.find("name like :param").limit(1)
                       .bind("param", "L%").execute();

    // Print document
    cout << myDoc.fetchOne() << endl;

    // Exit with success code
    exit(0);
  }
  catch (const Error &err)
  {
    cout << "The following error occurred: " << err << endl;
    exit(1);
  }

  // Note: session is closed automatically when session object
  // is destructed.
}
catch (const Error &err)
{
  cout << "The database session could not be opened: " << err << endl;

  // Exit with error code
  exit(1);
}
```

8.2 Working with Savepoints

X DevAPI supports savepoints, which enable you to set a named point within a transaction that you can revert to. By setting savepoints within a transaction, you can later use the rollback functionality to undo any statements issued after setting the savepoint. Savepoints can be released if you no longer require them. This section documents how to work with savepoints in X DevAPI. See `SAVEPOINT` for background information.

Setting a Savepoint

Savepoints are identified by a string name. The string can contain any character allowed for an identifier. To create a savepoint, use the `session.setSavepoint()` operation, which maps to the SQL statement `SAVEPOINT` *`name`*`;`. If you do not specify a `name`, one is automatically generated. For example by issuing:

```
session.setSavepoint()
```

a transaction savepoint is created with an automatically generated name and a string is returned with the name of the savepoint. This name can be used with the `session.rollbackTo()` or `session.releaseSavepoint()` operations. The `session.setSavepoint()` operation can be called multiple times within a session and each time a unique savepoint name is generated.

It is also possible to manually define the name of the savepoint by passing in a string *`name`*. For example issuing:

```
session.setSavepoint('name')
```

results in a transaction savepoint with the specified *`name`*, which is returned by the operation as a string. The `session.setSavepoint('`*`name`*`')` operation can be called multiple times in this way, and if the *`name`* has already been used for a savepoint then the previous savepoint is is deleted and a new one is set.

Rolling Back to a Savepoint

When a session has transaction savepoints, you can undo any subsequent transactions using the `session.rollbackTo()` operation, which maps to the `ROLLBACK TO name` statement. For example, issuing:

```
session.rollbackTo('name')
```

rolls back to the transaction savepoint *name*. This operation succeeds as long as the given savepoint has not been released. Rolling back to a savepoint which was created prior to other savepoints results in the subsequent savepoints being either released or rolled back. For example:

```
session.startTransaction()
(some data modifications occur...)

session.setSavepoint('point1')     <---- succeeds
(some data modifications occur...)

session.setSavepoint('point2')     <---- succeeds
(some data modifications occur...)

session.rollbackTo('point1')       <---- succeeds
session.rollbackTo('point1')       <---- still succeeds, but position stays the same
session.rollbackTo('point2')       <---- generates an error because lines above already cleared point2
session.rollbackTo('point1')       <---- still succeeds
```

Releasing a Savepoint

To cancel a savepoint, for example when it is no longer needed, use `releaseSavepoint()` and pass in the name of the savepoint you want to release. For example, issuing:

```
session.releaseSavepoint('name')
```

releases the savepoint *name*.

Savepoints and Implicit Transaction Behavior

The exact behavior of savepoints is defined by the server, and specifically how autocommit is configured. See autocommit, Commit, and Rollback.

For example, consider the following statements with no explicit `BEGIN`, `session.startTransaction()` or similar call:

```
session.setSavepoint('testsavepoint');
session.releaseSavepoint('testsavepoint');
```

If autocommit mode is enabled on the server, these statements result in an error because the savepoint named `testsavepoint` does not exist. This is because the call to `session.setSavepoint()` creates a transaction, then the savepoint and directly commits it. The result is that savepoint does not exist by the time the call to `releaseSavepoint()` is issued, which is instead in its own transaction. In this case, for the savepoint to survive you need to start an explicit transaction block first.

8.3 Working with Locking

X DevAPI supports MySQL locking through the `lockShared()` and `lockExclusive()` methods for the Collection.find() and Table.select() methods. This enables you to control row locking to ensure safe, transactional document updates on collections and to avoid concurrency problems, for example when using the modify() method. This section describes how to use the `lockShared()` and `lockExclusive()` methods for both the Collection.find() and Table.select() methods. For more background information on locking, see Locking Reads.

The `lockShared()` and `lockExclusive()` methods have the following properties, whether they are used with a Collection or a Table.

- Multiple calls to the lock methods are permitted. If a locking statement executes while a different transaction holds the same lock, it blocks until the other transaction releases it. If multiple calls to the lock methods are made, the last called lock method takes precedence. In other words `find().lockShared().lockExclusive()` is equivalent to `find().lockExclusive()`.

- `lockShared()` has the same semantics as `SELECT ... LOCK IN SHARE MODE`. Sets a shared mode lock on any rows that are read. Other sessions can read the rows, but cannot modify them until your transaction commits. If any of these rows were changed by another transaction that has not yet committed, your query waits until that transaction ends and then uses the latest values.

- `lockExclusive()` has the same semantics as `SELECT ... FOR UPDATE`. For any index records the search encounters, it locks the rows and any associated index entries, in the same way as if you issued an `UPDATE` statement for those rows. Other transactions are blocked from updating those rows, from doing `SELECT ... LOCK IN SHARE MODE`, or from reading the data in certain transaction isolation levels. Consistent reads ignore any locks set on the records that exist in the read view. Old versions of a record cannot be locked; they are reconstructed by applying undo logs on an in-memory copy of the record.

- Locks are held for as long as the transaction which they were acquired in exists. They are immediately released after the statement finishes unless a transaction is open or autocommit mode is turned off.

Both locking methods support the `NOWAIT` and `SKIP LOCKED` `InnoDB` locking modes. For more information see Locking Read Concurrency with NOWAIT and SKIP LOCKED. To use these locking modes with the locking methods, pass in one of the following:

- `NOWAIT` - if the function encounters a row lock it aborts and generates an `ER_LOCK_NOWAIT` error

- `SKIP_LOCKED` - if the function encounters a row lock it skips the row and continues

- `DEFAULT` - if the function encounters a row lock it waits until there is no lock. The equivalent of calling the lock method without a mode.

Locking considerations

When working with locking modes note the following:

- `autocommit` mode means that there is always a transaction open, which is commited automatically when a SQL statement executes.

- By default sessions are in autocommit mode.

- You disable autocommit mode implicitly when you call `startTransaction()`.

- When in autocommit mode, if a lock is acquired, it is released after the statement finishes. This could lead you to conclude that the locks were not acquired, but that is not the case.

- Similarly, if you try to acquire a lock that is already owned by someone else, the statement blocks until the other lock is released.

8.4 Working with Prepared Statements

Implemented in MySQL 8.0.16 and later: X DevAPI improves performance for each CRUD statement that is executed repeatedly by using a server-side prepared statement for its second and subsequent executions. This happens internally—applications do not need to do anything extra to utilize the feature, as long as the same operation object is reused.

When a statement is executed for a second time with changes only in data values or in values that refine the execution results (for example, different `offset()` or `limit()` values), the server prepares the statement for subsequent executions, so that there is no need to reparse the statement when it is being run again. New values for re-executions of the prepared statement are provided with parameter binding. When the statement is modified by chaining to it a method that refines the result (for example, `sort()`, `skip()`, `limit()`, or `offset()`), the statement is reprepared. The following pseudocode and the comments on them demonstrate the feature:

```
var f = coll.find("field = :field");
```

```
f.bind("field", 1).execute(); // Normal execution
f.bind("field", 2).execute(); // Same statement executed with a different parameter value triggers stat
f.bind("field", 3).execute(); // Prepared statement executed with a new value
f.bind("field", 3).limit(10).execute(); // Statement reprepared as it is modified with limit()
f.bind("field", 4).limit(20).execute(); // Reprepared statement executed with new parameters
```

Notice that to take advantage of the feature, the same operation object must be reused in the repetitions of the statement. Look at this example

```
for (i=0; i<100; ++i) {
    collection.find().execute();
}
```

This loop cannot take advantage of the prepared statement feature, because the operation object of `collection.find()` is recreated at each iteration of the `for` loop. Now, look at this example:

```
for (i=0; i<100; ++i) {
    var op = collection.find()
    op.execute();
}
```

The repeated statement is prepared once and then reused, because the same operation object of `collection.find()` is re-executed for each iteration of the `for` loop.

Prepared statements are part of a `Session`. When a `Client` resets the `Session` (by using, for example, `Mysqlx.Session.Reset`), the prepared statements are dropped.

Chapter 9 Working with Result Sets

Table of Contents

This section explains how to work with the results of processing.

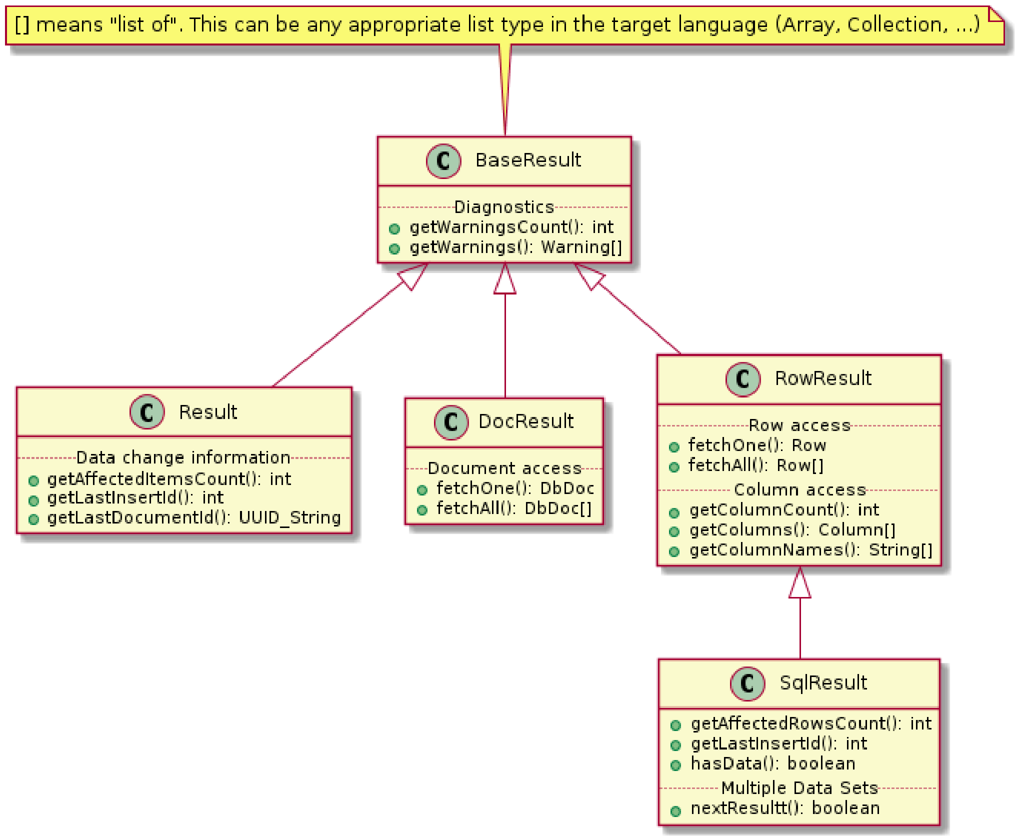
9.1 Result Set Classes

All database operations return a result. The type of result returned depends on the operation which was executed. The different types of results returned are outlined in the following table.

Result Class	Returned By	Provides
`Result`	`add().execute()`, `insert().execute()`, ...	`affectedRows`, `lastInsertId`, warnings
`SqlResult`	`session.sql()`	`affectedRows`, `lastInsertId`, warnings, fetched data sets
`DocResult`	`find().execute()`	fetched data set
`RowResult`	`select.execute()`	fetched data set

The following class diagram gives a basic overview of the result handling.

Figure 9.1 Results - Class Diagram



9.2 Working with AUTO-INCREMENT Values

AUTO_INCREMENT columns can be used in MySQL for generating primary key or id values, but are not limited to these uses. This section explains how to retrieve AUTO_INCREMENT values when adding rows using X DevAPI. For more background information, see Using AUTO_INCREMENT.

X DevAPI provides the getAutoIncrementValue() method to return the first AUTO_INCREMENT column value that was successfully inserted by the operation, taken from the return value of table.insert(). In the following example it is assumed that the table contains a column for which the AUTO_INCREMENT attribute is set:

```
res = myTable.insert(['name']).values('Mats').values('Otto').execute();
print(res.getAutoIncrementValue());
```

This table.insert() operation inserted multiple rows. getAutoIncrementValue() returns the AUTO_INCREMENT column value generated for the first inserted row only, so in this example, for the row containing “Mats”. The reason for this is to make it possible to reproduce easily the same operation against some other server.

9.3 Working with Data Sets

Operations that fetch data items return a data set as opposed to operations that modify data and return a result set. Data items can be read from the database using Collection.find(), Table.select() and Session.sql(). All three methods return data sets which encapsulate data items. Collection.find() returns a data set with documents and Table.select() respectively Session.sql() return a data set with rows.

All data sets implement a unified way of iterating their data items. The unified syntax supports fetching items one by one using fetchOne() or retrieving a list of all items using fetchAll(). fetchOne()

and `fetchAll()` follow forward-only iteration semantics. Connectors implementing the X DevAPI can offer more advanced iteration patterns on top to match common native language patterns.

The following example shows how to access the documents returned by a `Collection.find()` operation by using `fetchOne()` to loop over all documents.

The first call to `fetchOne()` returns the first document found. All subsequent calls increment the internal data item iterator cursor by one position and return the item found making the second call to `fetchOne()` return the second document found, if any. When the last data item has been read and `fetchOne()` is called again a NULL value is returned. This ensures that the basic while loop shown works with all languages which implement the X DevAPI if the language supports such an implementation.

When using `fetchOne()` it is not possible to reset the internal data item cursor to the first data item to start reading the data items again. An data item - here a Document - that has been fetched once using `fetchOne()` can be discarded by the Connector. The data item's life time is decoupled from the data set. From a Connector perspective items are consumed by the caller as they are fetched. This example assumes that the test schema exists.

MySQL Shell JavaScript Code

```
var myColl = db.getCollection('my_collection');

var res = myColl.find('name like :name').bind('name','L%').
        execute();

var doc;
while (doc = res.fetchOne()) {
  print(doc);
}
```

MySQL Shell Python Code

```
myColl = db.get_collection('my_collection')

res = myColl.find('name like :name').bind('name','L%').execute()

doc = res.fetch_one()
while doc:
        print(doc)
        doc = res.fetch_one()
```

C# Code

```
var myColl = db.GetCollection("my_collection");

var res = myColl.Find("name like :name").Bind("name", "L%")
  .Execute();

DbDoc doc;
while ((doc = res.FetchOne()) != null)
{
  Console.WriteLine(doc);
}
```

Python Code

```
my_coll = db.get_collection('my_collection')

res = my_coll.find('name like :name').bind('name', 'L%').execute()

doc = res.fetch_one()
while doc:
    print(doc)
    doc = res.fetch_one()
```

Java Code

```
Collection myColl = db.getCollection("my_collection");

DocResult res = myColl.find("name like :name").bind("name", "L%")
  .execute();

DbDoc doc;
while ((doc = res.fetchOne()) != null) {
  System.out.println(doc);
}
```

C++ Code

```
Collection myColl = db.getCollection("my_collection");

DocResult res = myColl.find("name like :name").bind("name", "L%").execute();
DbDoc doc;
while ((doc = res.fetchOne()))
{
  cout << doc <<endl;
}
```

Node.js JavaScript Code

```
var myColl = db.getCollection('my_collection');

myColl.find('name like :name')
  .bind('name', 'L%')
  .execute()
  .then(res => {
    while (doc = res.fetchOne()) {
      console.log(doc);
    }
  });


myColl.find('name like :name')
  .bind('name', 'L%')
  .execute(function (doc) {
    console.log(doc);
});
```

When using Node.js, results can also be returned to a callback function, which is passed to `execute()` in an asychronous manner whenever results from the server arrive.

```
var myColl = db.getCollection('my_collection');

myColl.find('name like :name')
  .bind('name', 'L%')
  .execute(function (doc) {
    console.log(doc);
});
```

The following example shows how to directly access the rows returned by a `Table.select()` operation. The basic code pattern for result iteration is the same. The difference between the following and the previous example is in the data item handling. Here, `fetchOne()` returns Rows. The exact syntax to access the column values of a Row is language dependent. Implementations seek to provide a language native access pattern. The example assumes that the `test` schema exists and that the employee table exists in `myTable`.

MySQL Shell JavaScript Code

```
var myRows = myTable.select(['name', 'age']).
        where('name like :name').bind('name','L%').
        execute();

var row;
while (row = myRows.fetchOne()) {
  // Accessing the fields by array
  print('Name: ' + row['name'] + '\n');
  // Accessing the fields by dynamic attribute
  print(' Age: ' + row.age + '\n');
}
```

MySQL Shell Python Code

```
myRows = myTable.select(['name', 'age']).where('name like :name').bind('name','L%').execute()

row = myRows.fetch_one()
while row:
        # Accessing the fields by array
        print('Name: %s\n' % row[0])
        # Accessing the fields by dynamic attribute
        print('Age: %s\n' % row.age)
        row = myRows.fetch_one()
```

Node.js JavaScript Code

```
var myRows = myTable
  .select(['name', 'age'])
  .where('name like :name')
  .bind('name','L%')
  .execute(function (row) {
    // Connector/Node.js does not support referring to row columns by their name yet.
    // One needs to access fields by their array index.
    console.log('Name: ' + row[0]);
    console.log(' Age: ' + row[1]);
  });
```

Alternatively, you can use callbacks:

```
myTable.select(['name', 'age'])
  .where('name like :name')
  .bind('name', 'L%')
  .execute()
  .then(myRows => {
    while (var row = myRows.fetchOne()) {
      // Accessing the fields by array
      console.log('Name: ' + row[0] + '\n');
      console.log('Age: ' + row[1] + '\n');
    }
  });
```

C# Code

```
var myRows = myTable.Select("name", "age")
  .Where("name like :name").Bind("name", "L%")
  .Execute();

Row row;
while ((row = myRows.FetchOne()) != null)
{
  // Accessing the fields by array
  Console.WriteLine("Name: " + row[0]);

  // Accessing the fields by name
  Console.WriteLine("Age: " + row["age"]);
}
```

Python Code

```
rows = my_table.select(['name', 'age']).where('name like :name').bind('name','L%').execute()

row = rows.fetch_one()
while row:
    # Accessing the fields by array
    print('Name: {0}'.format(row[0]))

    # Accessing the fields by dynamic attribute
    print('Age: {0}'.format(row['age'])

    row = rows.fetch_one()
```

Java Code

```
RowResult myRows = myTable.select("name, age")
  .where("name like :name").bind("name", "L%")
  .execute();

Row row;
while ((row = myRows.fetchOne()) != null) {
  // Accessing the fields
  System.out.println(" Age: " + row.getInt("age") + "\n");
}
```

C++ Code

```
RowResult myRows = myTable.select("name", "age")
                          .where("name like :name")
                          .bind("name", "L%")
                          .execute();

Row row;
while ((row = myRows.fetchOne()))
{
  // Connector/C++ does not support referring to row columns by their name yet.
  cout <<"Name: " << row[0] <<endl;
  cout <<" Age: " << row[1] <<endl;
  int age = row[1];
  // One needs explicit .get<int>() as otherwise operator<() is ambiguous
  bool uforty = row[age].get<int>() < 40;
  // Alternative formulation
  bool uforty = (int)row[age] < 40;
}
```

9.4 Fetching All Data Items at Once

In addition to the pattern of using `fetchOne()` explained at Section 9.3, “Working with Data Sets”, which enables applications to consume data items one by one, X DevAPI also provides a pattern using `fetchAll()`, which passes all data items of a data set as a list to the application. The different X DevAPI implementations use appropriate data types for their programming language for the list. Because different data types are used, the language's native constructs are supported to access the list elements. The following example assumes that the `test` schema exists and that the employee table exists in `myTable`.

MySQL Shell JavaScript Code

```
var myResult = myTable.select(['name', 'age']).
  where('name like :name').bind('name','L%').
  execute();

var myRows = myResult.fetchAll();

for (index in myRows){
  print (myRows[index].name + " is " + myRows[index].age + " years old.");
}
```

MySQL Shell Python Code

```
myResult = myTable.select(['name', 'age']) \
  .where('name like :name').bind('name','L%') \
  .execute()

myRows = myResult.fetch_all()

for row in myRows:
  print("%s is %s years old." % (row.name, row.age))
```

Node.js JavaScript Code

```
myTable.select(['name', 'age'])
  .where('name like :name')
  .bind('name', 'L%')
  .execute()
  .then(myResult => {
    var myRows = myResult.fetchAll();

    myRows.forEach(row => {
      console.log(`${row[0]} is ${row[1]} years old.`);
    });
  });
```

C# Code

```
var myRows = myTable.Select("name", "age")
  .Where("name like :name").Bind("name", "L%")
  .Execute();
var rows = myRows.FetchAll();
```

Python Code

```
result = myTable.select(['name', 'age']) \
    .where('name like :name').bind('name', 'L%') \
    .execute()

rows = result.fetch_all()

for row in rows:
    print("{0} is {1} years old.".format(row["name"], row["age"]))
```

Java Code

```
RowResult myRows = myTable.select("name, age")
  .where("name like :name").bind("name", "L%")
  .execute();

List<Row> rows = myRows.fetchAll();
for (Row row : rows) {
  // Accessing the fields
  System.out.println(" Age: " + row.getInt("age") + "\n");
}
```

C++ Code

```
RowResult myRows = myTable.select("name, age")
                          .where("name like :name")
                          .bind("name", "L%")
                          .execute();

std::list<Row> rows = myRows.fetchAll();
for (Row row : rows)
{
  cout << row[1] << endl;
}

// Directly iterate over rows, without storing them in a container

for (Row row : myRows.fetchAll())
{
  cout << row[1] << endl;
}
```

When mixing `fetchOne()` and `fetchAll()` to read from one data set keep in mind that every call to `fetchOne()` or `fetchAll()` consumes the data items returned. Items consumed cannot be requested again. If, for example, an application calls `fetchOne()` to fetch the first data item of a data set, then a subsequent call to `fetchAll()` returns the second to last data item. The first item is not part of the list of data items returned by `fetchAll()`. Similarly, when calling `fetchAll()` again for a data set after calling it previously, the second call returns an empty collection.

The use of `fetchAll()` forces a Connector to build a list of all items in memory before the list as a whole can be passed to the application. The life time of the list is independent from the life of the data set that has produced it.

Asynchronous query executions return control to caller once a query has been issued and prior to receiving any reply from the server. Calling `fetchAll()` to read the data items produced by an asynchronous query execution may block the caller. `fetchAll()` cannot return control to the caller before reading results from the server is finished.

9.5 Working with SQL Result Sets

When you execute an SQL operation on a Session using the `sql()` method an SqlResult is returned. Iterating an SqlResult is identical to working with results from CRUD operations. The following example assumes that the users table exists.

MySQL Shell JavaScript Code

```
var res = mySession.sql('SELECT name, age FROM users').execute();

var row;
while (row = res.fetchOne()) {
  print('Name: ' + row['name'] + '\n');
  print(' Age: ' + row.age + '\n');
}
```

MySQL Shell Python Code

```
res = mySession.sql('SELECT name, age FROM users').execute()

row = res.fetch_one()

while row:
    print('Name: %s\n' % row[0])
    print(' Age: %s\n' % row.age)
    row = res.fetch_one()
```

Node.js JavaScript Code

```
mySession.sql('SELECT name, age FROM users')
  .execute()
  .then(res => {
    while (row = res.fetchOne()) {
      console.log('Name: ' + row[0] + '\n');
      console.log(' Age: ' + row[1] + '\n');
    }
  });
```

Alternatively, you can use callbacks:

```
mySession.sql('SELECT name, age FROM users')
  .execute(function (row) {
    console.log('Name: ' + row[0] + '\n');
    console.log(' Age: ' + row[1] + '\n');
});
```

C# Code

```
var res = Session.SQL("SELECT name, age FROM users").Execute();

while (res.Next())
{
  Console.WriteLine("Name: " + res.Current["name"]);
  Console.WriteLine("Age: " + res.Current["age"]);
}
```

Python Code

```
# Connector/Python
res = mySession.sql('SELECT name, age FROM users').execute()

row = res.fetch_one()

while row:
        print('Name: %s\n' % row[0])
        print(' Age: %s\n' % row.age)
        row = res.fetch_one()
```

Java Code

```
SqlResult res = mySession.sql("SELECT name, age FROM users").execute();

Row row;
while ((row = res.fetchOne()) != null) {
  System.out.println(" Name: " + row.getString("name") + "\n");
  System.out.println(" Age: " + row.getInt("age") + "\n");
}
```

C++ Code

```
RowResult res = mysession.sql("SELECT name, age FROM users").execute();

Row row;
while ((row = res.fetchOne())) {
  cout << "Name: " << row[0] << endl;
  cout << " Age: " << row[1] << endl;
}
```

SqlResult differs from results returned by CRUD operations in the way how result sets and data sets are represented. A SqlResult combines a result set produced by, for example, `INSERT`, and a data set, produced by, for example, `SELECT` in one. Unlike with CRUD operations there is no distinction between the two types. A SqlResult exports methods for data access and to retrieve the last inserted id or number of affected rows.

Use the `hasData()` method to learn whether a SqlResult is a data set or a result. The method is useful when code is to be written that has no knowledge about the origin of a SqlResult. This can be the case when writing a generic application function to print query results or when processing stored procedure results. If `hasData()` returns true, then the SqlResult origins from a `SELECT` or similar command that can return rows.

A return value of true does not indicate whether the data set contains any rows. The data set can be empty, for example it is empty if `fetchOne()` returns NULL or `fetchAll()` returns an empty list. The following example assumes that the procedure `my_proc` exists.

MySQL Shell JavaScript Code

```
var res = mySession.sql('CALL my_proc()').execute();

if (res.hasData()){

  var row = res.fetchOne();
  if (row){
    print('List of rows available for fetching.');
    do {
      print(row);
    } while (row = res.fetchOne());
  }
  else{
    print('Empty list of rows.');
  }
}
else {
  print('No row result.');
}
```

MySQL Shell Python Code

```
res = mySession.sql('CALL my_proc()').execute()

if res.has_data():

    row = res.fetch_one()
    if row:
        print('List of rows available for fetching.')
        while row:
            print(row)
            row = res.fetch_one()
    else:
        print('Empty list of rows.')
else:
    print('No row result.')
```

Node.js JavaScript Code

```
mySession.sql('CALL my_proc()')
  .execute()
  .then(function (res) {
    if (!res.hasData()) {
      return console.log('No row result.');
    }

    var row = res.fetchOne();

    if (!row) {
      return console.log('Empty list of rows.');
    }

    console.log('List of rows available for fetching.');

    do {
      console.log(row);
    } while (row = res.fetchOne());
})
```

C# Code

```
var res = Session.SQL("CALL my_proc()").Execute();

if (res.HasData)
{

  var row = res.FetchOne();
  if (row != null)
  {
    Console.WriteLine("List of rows available for fetching.");
    do
    {
      PrintResult(row);
    } while ((row = res.FetchOne()) != null);
  }
  else
  {
    Console.WriteLine("Empty list of rows.");
  }
}
else
{
  Console.WriteLine("No row result.");
}
```

Python Code

```
# Connector/Python
```

```
res = mySession.sql('CALL my_proc()').execute()

if res.has_data():

    row = res.fetch_one()
    if row:
        print('List of rows available for fetching.')
        while row:
            print(row)
            row = res.fetch_one()
    else:
        print('Empty list of rows.')
else:
    print('No row result.')
```

Java Code

```
SqlResult res = mySession.sql("CALL my_proc()").execute();

if (res.hasData()){

  Row row = res.fetchOne();
  if (row != null){
    System.out.println("List of rows available for fetching.");
    do {
     for (int c = 0; c < res.getColumnCount(); c++) {
      System.out.println(row.getString(c));
      }
    } while ((row = res.fetchOne()) != null);
  }
  else{
    System.out.println("Empty list of rows.");
  }
}
else {
  System.out.println("No row result.");
}
```

C++ Code

```
SqlResult res = mysession.sql("CALL my_proc()").execute();

if (res.hasData())
{
  Row row = res.fetchOne();
  if (row)
  {
    cout << "List of rows available for fetching." << endl;
    do {
      cout << "next row: ";
      for (unsigned i=0 ; i < row.colCount(); ++i)
        cout << row[i] << ", ";
      cout << endl;
    } while ((row = res.fetchOne()));
  }
  else
  {
    cout << "Empty list of rows." << endl;
  }
}
else
{
  cout << "No row result." << endl;
}
```

It is an error to call either `fetchOne()` or `fetchAll()` when `hasResult()` indicates that a SqlResult is not a data set.

MySQL Shell JavaScript Code

```
function print_result(res) {
  if (res.hasData()) {
    // SELECT
    var columns = res.getColumns();
    var record = res.fetchOne();

    while (record){
      for (index in columns){
        print (columns[index].getColumnName() + ": " + record[index] + "\n");
      }

      // Get the next record
      record = res.fetchOne();
    }

  } else {
    // INSERT, UPDATE, DELETE, ...
    print('Rows affected: ' + res.getAffectedItemsCount());
  }
}

print_result(mySession.sql('DELETE FROM users WHERE age < 30').execute());
print_result(mySession.sql('SELECT * FROM users WHERE age = 40').execute());
```

MySQL Shell Python Code

```
def print_result(res):
  if res.has_data():
    # SELECT
    columns = res.get_columns()
    record = res.fetch_one()

    while record:
      index = 0

      for column in columns:
        print("%s: %s \n" % (column.get_column_name(), record[index]))
        index = index + 1

      # Get the next record
      record = res.fetch_one()
  else:
    #INSERT, UPDATE, DELETE, ...
    print('Rows affected: %s' % res.get_affected_items_count())

print_result(mySession.sql('DELETE FROM users WHERE age < 30').execute())
print_result(mySession.sql('SELECT * FROM users WHERE age = 40').execute())
```

Node.js JavaScript Code

```
function print_result(res) {
  if (res.hasData()) {
    // SELECT
    var columns = res.getColumns();
    var record = res.fetchOne();

    while (record) {
      for (index in columns) {
        console.log(columns[index].getColumnName() + ": " + record[index]);
      }

      // Get the next record
      record = res.fetchOne();
    }

  } else {
    // INSERT, UPDATE, DELETE, ...
    console.log('Rows affected: ' + res.getAffectedItemsCount());
  }
}

mySession.sql(`DELETE FROM users WHERE age < 30`)
```

```
  .execute()
  .then(function (res) {
    print_result(res);
  });

mySession.sql(`SELECT * FROM users WHERE age = 40`)
  .execute()
  .then(function (res) {
    print_result(res);
  });
```

C# Code

```
private void print_result(SqlResult res)
{
  if (res.HasData)
  {
    // SELECT
  }
  else
  {
    // INSERT, UPDATE, DELETE, ...
    Console.WriteLine("Rows affected: " + res.RecordsAffected);
  }
}

print_result(Session.SQL("DELETE FROM users WHERE age < 30").Execute());
print_result(Session.SQL("SELECT COUNT(*) AS forty FROM users WHERE age = 40").Execute());
```

Python Code

```
# Connector/Python
def print_result(res):
  if res.has_data():
    # SELECT
    columns = res.get_columns()
    record = res.fetch_one()

    while record:
      index = 0

      for column in columns:
        print("%s: %s \n" % (column.get_column_name(), record[index]))
        index = index + 1

      # Get the next record
      record = res.fetch_one()

  else:
    #INSERT, UPDATE, DELETE, ...
    print('Rows affected: %s' % res.get_affected_items_count())


print_result(mySession.sql('DELETE FROM users WHERE age < 30').execute())
print_result(mySession.sql('SELECT * FROM users WHERE age = 40').execute())
```

Java Code

```
private void print_result(SqlResult res) {
  if (res.hasData()) {
    // SELECT
     Row row;
        while ((row = res.fetchOne()) != null){
            for (int c = 0; c < res.getColumnCount(); c++) {
                System.out.println(row.getString(c));
            }
        }
  } else {
    // INSERT, UPDATE, DELETE, ...
    System.out.println("Rows affected: " + res.getAffectedItemsCount());
  }
}

print_result(mySession.sql("DELETE FROM users WHERE age < 30").execute());
print_result(mySession.sql("SELECT COUNT(*) AS forty FROM users WHERE age = 40").execute());
```

C++ Code

```
void print_result(SqlResult &&_res)
{
  // Note: We need to store the result somewhere to be able to process it.

  SqlResult res(std::move(_res));

  if (res.hasData())
  {
    // SELECT
    const Columns &columns = res.getColumns();
    Row record = res.fetchOne();

    while (record)
    {
      for (unsigned index=0; index < res.getColumnCount(); ++index)
      {
        cout << columns[index].getColumnName() << ": "
             << record[index] << endl;
      }

      // Get the next record
      record = res.fetchOne();
    }

  }
  else
  {
    // INSERT, UPDATE, DELETE, ...
    // Note: getAffectedItemsCount() not yet implemented in Connector/C++.
    cout << "No rows in the result" << endl;
  }
}

print_result(mysession.sql("DELETE FROM users WHERE age < 30").execute());
print_result(mysession.sql("SELECT * FROM users WHERE age = 40").execute());
```

Calling a stored procedure might result in having to deal with multiple result sets as part of a single execution. As a result for the query execution a SqlResult object is returned, which encapsulates the first result set. After processing the result set you can call `nextResult()` to move forward to the next result, if any. Once you advanced to the next result set, it replaces the previously loaded result which then becomes unavailable.

MySQL Shell JavaScript Code

```
function print_result(res) {
  if (res.hasData()) {
    // SELECT
    var columns = res.getColumns();
    var record = res.fetchOne();

    while (record){
      for (index in columns){
        print (columns[index].getColumnName() + ": " + record[index] + "\n");
      }

      // Get the next record
      record = res.fetchOne();
    }

  } else {
    // INSERT, UPDATE, DELETE, ...
    print('Rows affected: ' + res.getAffectedItemsCount());
  }
}
```

```
var res = mySession.sql('CALL my_proc()').execute();

// Prints each returned result
var more = true;
while (more){
  print_result(res);

  more = res.nextResult();
}
```

MySQL Shell Python Code

```
def print_result(res):
  if res.has_data():
    # SELECT
    columns = res.get_columns()
    record = res.fetch_one()

    while record:
      index = 0

      for column in columns:
        print("%s: %s \n" % (column.get_column_name(), record[index]))
        index = index + 1

      # Get the next record
      record = res.fetch_one()
  else:
    #INSERT, UPDATE, DELETE, ...
    print('Rows affected: %s' % res.get_affected_items_count())

res = mySession.sql('CALL my_proc()').execute()

# Prints each returned result
more = True
while more:
  print_result(res)
  more = res.next_result()
```

Node.js JavaScript Code

```
function print_result(res) {
  if (res.hasData()) {
    // SELECT
    var columns = res.getColumns();
    var record = res.fetchOne();

    while (record) {
      for (index in columns) {
        console.log(columns[index].getColumnName() + ": " + record[index]);
      }

      // Get the next record
      record = res.fetchOne();
    }

  } else {
    // INSERT, UPDATE, DELETE, ...
    console.log('Rows affected: ' + res.getAffectedItemsCount());
  }
}

mySession.sql('CALL my_proc()')
  .execute()
  .then(function (res) {
    // Prints each returned result
    var more = true;

    while (more) {
      print_result(res);

      more = res.nextResult();
    }
  })
```

C# Code

```
var res = Session.SQL("CALL my_proc()").Execute();

if (res.HasData)
{
  do
  {
    Console.WriteLine("New resultset");
    while (res.Next())
    {
      Console.WriteLine(res.Current);
    }
  } while (res.NextResult());
}
```

Python Code

```
# Connector/Python
def print_result(res):
  if res.has_data():
    # SELECT
    columns = res.get_columns()
    record = res.fetch_one()

    while record:
      index = 0

      for column in columns:
        print("%s: %s \n" % (column.get_column_name(), record[index]))
        index = index + 1

      # Get the next record
      record = res.fetch_one()
  else:
    #INSERT, UPDATE, DELETE, ...
    print('Rows affected: %s' % res.get_affected_items_count())

res = mySession.sql('CALL my_proc()').execute()

# Prints each returned result
more = True
while more:
  print_result(res)

  more = res.next_result()
```

Java Code

```
SqlResult res = mySession.sql("CALL my_proc()").execute();
```

C++ Code

```
SqlResult res = mysession.sql("CALL my_proc()").execute();

while (true)
{
  if (res.hasData())
  {
    cout << "List of rows in the resultset." << endl;
    for (Row row; (row = res.fetchOne());)
    {
      cout << "next row: ";
      for (unsigned i = 0; i < row.colCount(); ++i)
```

```
        cout << row[i] << ", ";
        cout << endl;
    }
  }
  else
  {
    cout << "No rows in the resultset." << endl;
  }

  if (!res.nextResult())
    break;

  cout << "Next resultset." << endl;
}
```

When using Node.js, individual rows can be returned immediately using a callback, which has to be provided to the `execute()` method. To identify individual result sets you can provide a second callback, which is called for meta data that marks the beginning of a result set.

Node.js JavaScript Code

```
var resultcount = 0;
var res = session
  .sql('CALL my_proc()')
  .execute(
    function (row) {
      console.log(row);
    },
    function (meta) {
      console.log('Begin of result set number ', resultCount++);
    });
```

The number of result sets is not known immediately after the query execution. Query results can be streamed to the client or buffered at the client. In the streaming or partial buffering mode a client cannot tell whether a query emits more than one result set.

9.6 Working with Metadata

Results contain metadata related to the origin and types of results from relational queries. This metadata can be used by applications that need to deal with dynamic query results or format results for transformation or display. Result metadata is accessible via instances of `Column`. An array of columns can be obtained from any RowResult using the `getColumns()` method.

For example, the following metadata is returned in response to the query `SELECT 1+1 AS a, b FROM mydb.some_table_with_b AS b_table`.

```
Column[0].databaseName = NULL
Column[0].tableName = NULL
Column[0].tableLabel = NULL
Column[0].columnName = NULL
Column[0].columnLabel = "a"
Column[0].type = BIGINT
Column[0].length = 3
Column[0].fractionalDigits = 0
Column[0].numberSigned = TRUE
Column[0].collationName = "binary"
Column[0].characterSetName = "binary"
Column[0].padded = FALSE

Column[1].databaseName = "mydb"
Column[1].tableName = "some_table_with_b"
Column[1].tableLabel = "b_table"
Column[1].columnName = "b"
Column[1].columnLabel = "b"
Column[1].type = STRING
Column[1].length = 20 (e.g.)
Column[1].fractionalDigits = 0
Column[1].numberSigned = TRUE
```

```
Column[1].collationName = "utf8mb4_general_ci"
Column[1].characterSetName = "utf8mb4"
Column[1].padded = FALSE
```

9.7 Support for Language Native Iterators

All implementations of the DevAPI feature the methods shown in the UML diagram at the beginning of this chapter. All implementations allow result set iteration using `fetchOne(), fetchAll()` and `nextResult().` In addition to the unified API drivers should implement language native iteration patterns. This applies to any type of data set (DocResult, RowResult, SqlResult) and to the list of items returned by `fetchAll()`. You can choose whether you want your X DevAPI based application code to offer the same look and feel in all programming languages used or opt for the natural style of a programming language.

Chapter 10 Building Expressions

Table of Contents

This section explains how to build expressions using X DevAPI.

When working with MySQL expressions used in CRUD, statements can be specified in two ways. The first is to use strings to formulate the expressions which should be familiar if you have developed code with SQL before. The other method is to use Expression Builder functionality.

10.1 Expression Strings

Defining string expressions is straightforward as these are easy to read and write. The disadvantage is that they need to be parsed before they can be transfered to the MySQL server. In addition, type checking can only be done at runtime. All implementations can use the syntax illustrated here, which is shown as MySQL Shell JavaScript code.

```
// Using a string expression to get all documents that
// have the name field starting with 'S'
var myDocs = myColl.find('name like :name').bind('name', 'S%').execute();
```

10.1.1 Boolean Expression Strings

Boolean expression strings can be used when filtering collections or tables using operations, such as `find()` and `remove()`. The expression is evaluated once for each document or row.

The following example of a boolean expression string uses `find()` to search for all documents with a “red” color attribute from the collection “apples”:

```
apples.find('color = "red"').execute()
```

Similarly, to delete all red apples:

```
apples.remove('color = "red"').execute()
```

10.1.2 Value Expression Strings

Value expression strings are used to compute a value which can then be assigned to a given field or column. This is necessary for both `modify()` and `update()`, as well as computing values in documents at insertion time.

An example use of a value expression string would be to increment a counter. The `expr()` function is used to wrap strings where they would otherwise be interpreted literally. For example, to increment a counter:

```
// the expression is evaluated on the server
collection.modify('true').set("counter", expr("counter + 1")).execute()
```

If you do not wrap the string with `expr()`, it would be assigning the literal string "counter + 1" to the "counter" member:

```
// equivalent to directly assigning a string: counter = "counter + 1"
collection.modify('true').set("counter", "counter + 1").execute()
```

Chapter 11 CRUD EBNF Definitions

Table of Contents

This chapter provides a visual reference guide to the objects and functions available in the X DevAPI.

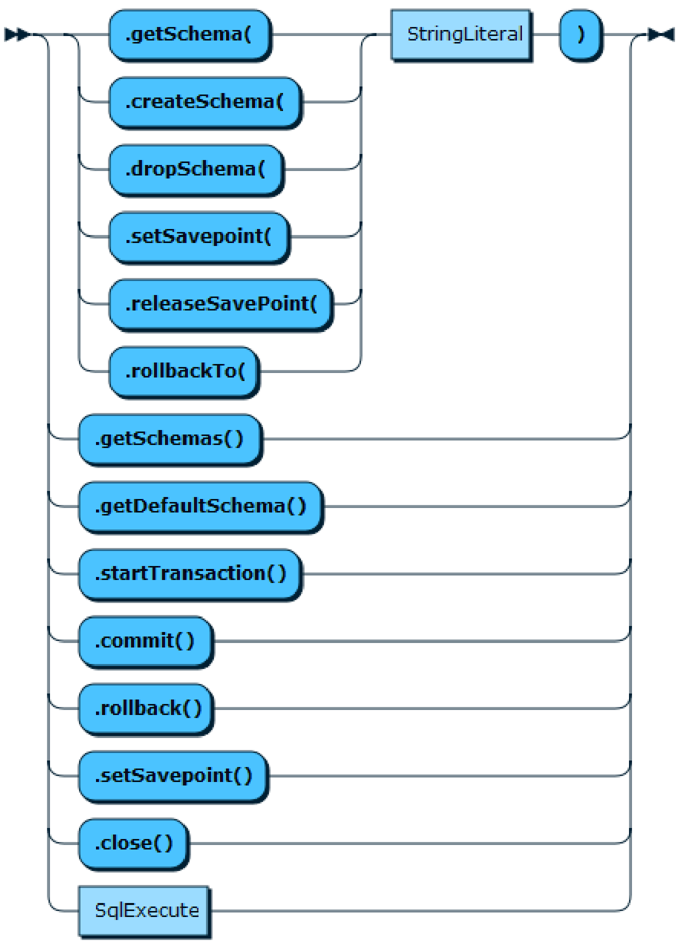
11.1 Session Objects and Functions

Session

The syntax for this object shown in EBNF is:

```
Session
    ::= '.getSchema(' StringLiteral ')'
        | '.getSchemas()'
        | '.createSchema(' StringLiteral ')'
        | '.dropSchema(' StringLiteral ')'
        | '.getDefaultSchema()'
        | '.startTransaction()'
        | '.commit()'
        | '.rollback()'
        | '.setSavepoint()'
        | '.setSavepoint(' StringLiteral ')'
        | '.releaseSavePoint(' StringLiteral ')'
        | '.rollbackTo(' StringLiteral ')'
        | '.close()'
        | SqlExecute
```

Figure 11.1 Session

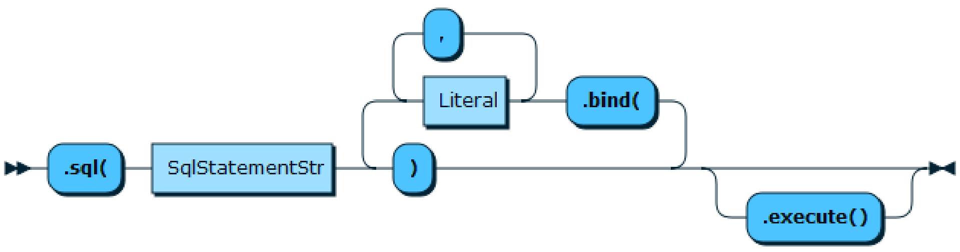


SqlExecute

The syntax for this function shown in EBNF is:

```
SqlExecute
    ::= '.sql(' SqlStatementStr ')'
        ( '.bind(' Literal (',' Literal)* ')')*
        ( '.execute()' )?
```

Figure 11.2 SqlExecute

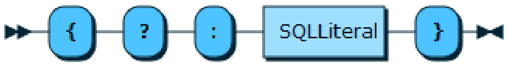


SQLPlaceholderValues

The syntax for this function shown in EBNF is:

```
SQLPlaceholderValues
  ::= '{' SQLPlaceholderName ':' ( SQLLiteral ) '}'
```

Figure 11.3 SQLPlaceholderValues

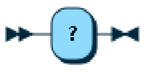


SQLPlaceholderName

The syntax for this function shown in EBNF is:

```
SQLPlaceholderName
  ::= '?'
```

Figure 11.4 SQLPlaceholderName

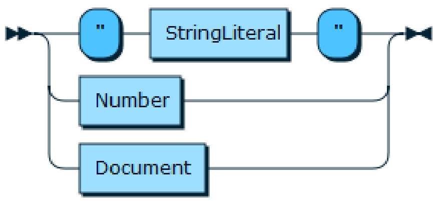


SQLLiteral

The syntax for this function shown in EBNF is:

```
SQLLiteral
  ::= '"' StringLiteral '"' | Number | Document
```

Figure 11.5 SQLLiteral



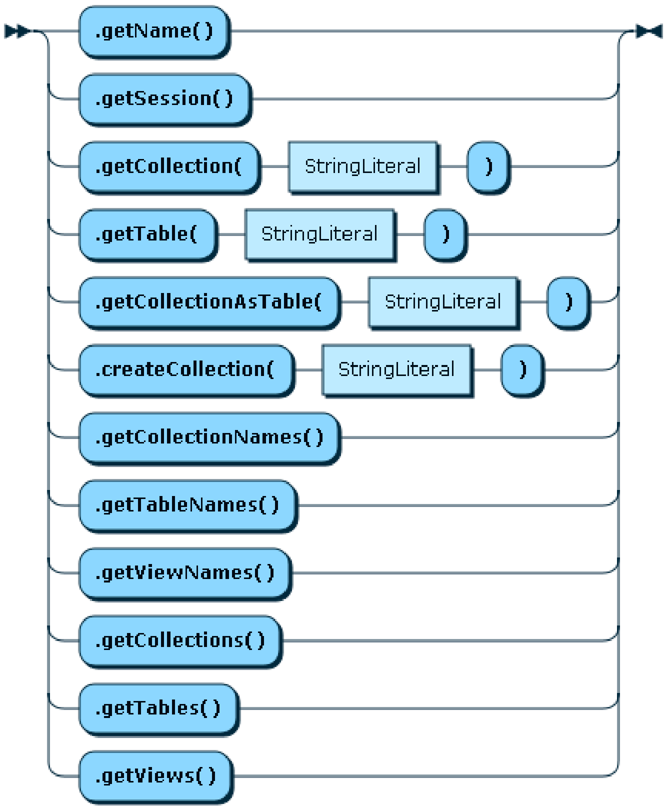
11.2 Schema Objects and Functions

Schema

The syntax for this function shown in EBNF is:

```
Schema
    ::= '.getName()'
        | '.existsInDatabase()'
        | '.getSession()'
        | '.getCollection(' StringLiteral ')'
        | '.getCollections()'
        | '.getCollectionAsTable(' StringLiteral ')'
        | '.dropCollection(' StringLiteral ')'
        | '.getTable(' StringLiteral ')'
        | '.getTables()'
        | '.createCollection(' StringLiteral ')'
```

Figure 11.6 Schema

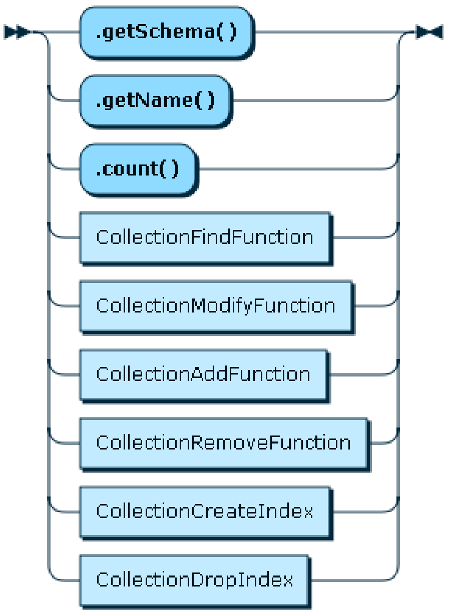


Collection

The syntax for this function shown in EBNF is:

```
Collection
    ::= '.getSchema()'
        | '.getName()'
        | '.getSession()'
        | '.existsInDatabase()'
        | '.replaceOne(' DocumentId ',' DocumentOrJSON  ')'
        | '.addOrReplaceOne(' DocumentId ',' DocumentOrJSON  ')'
        | '.getOne(' DocumentId ')'
        | '.removeOne(' DocumentId ')'
        | CollectionFindFunction
        | CollectionModifyFunction
        | CollectionAddFunction
        | CollectionRemoveFunction
        | CollectionCreateIndex
        | CollectionDropIndex
```

Figure 11.7 Collection

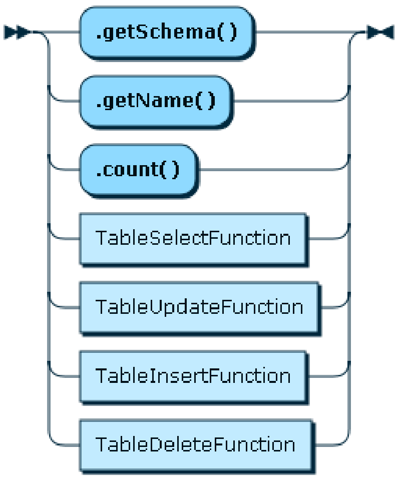


Table

The syntax for this function shown in EBNF is:

```
Table
    ::= '.getSchema()'
        | '.getName()'
        | '.getSession()'
        | '.existsInDatabase()'
        | '.isView()'
        | TableSelectFunction
        | TableUpdateFunction
        | TableInsertFunction
        | TableDeleteFunction
```

Figure 11.8 Table



11.3 Collection CRUD Functions

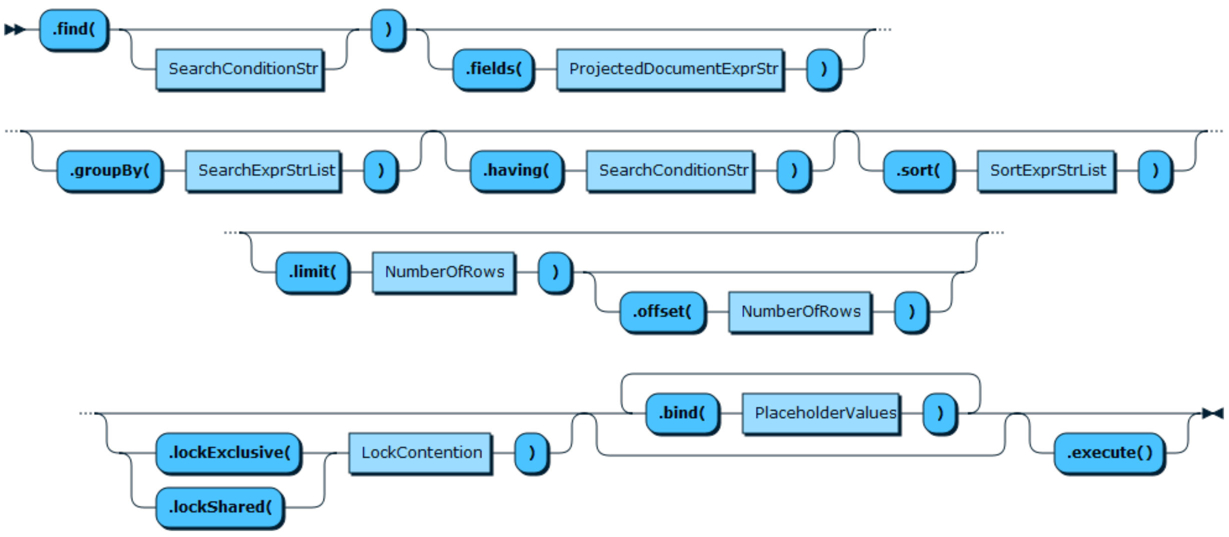
CollectionFindFunction

The syntax for this function in EBNF is:

```
CollectionFindFunction
  ::= '.find(' SearchConditionStr? ')' ( '.fields(' ProjectedDocumentExprStr ')' )?
```

```
      ( '.groupBy(' SearchExprStrList ')' )? ( '.having(' SearchConditionStr ')' )?
      ( '.sort(' SortExprStrList ')' )? ( '.limit(' NumberOfRows ')' ( '.offset(' NumberOfRows ')' )? )?
      ( '.lockExclusive(' LockContention ')' | '.lockShared(' LockContention ')' )?
      ( '.bind(' PlaceholderValues ')' )*
      ( '.execute()' )?
```

Figure 11.9 CollectionFindFunction

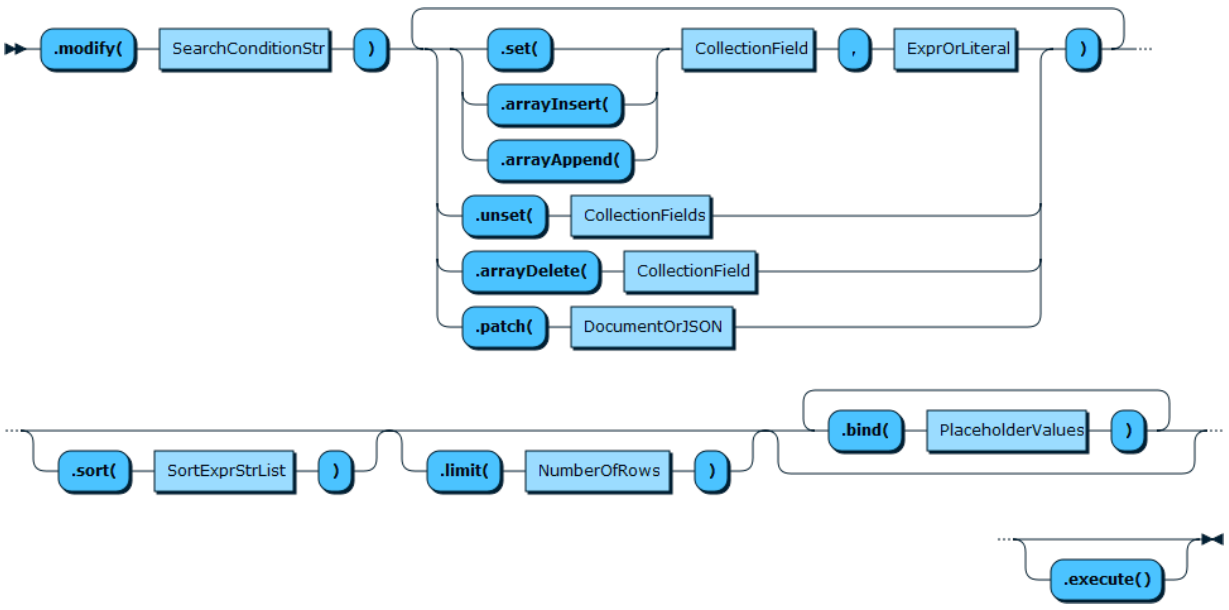


CollectionModifyFunction

The syntax for this function shown in EBNF is:

```
CollectionModifyFunction
  ::= '.modify(' SearchConditionStr ')'
      ( '.set(' CollectionField ',' ExprOrLiteral ')' |
        '.unset(' CollectionFields ')' |
        '.arrayInsert(' CollectionField ',' ExprOrLiteral ')' |
        '.arrayAppend(' CollectionField ',' ExprOrLiteral ')' |
        '.arrayDelete(' CollectionField ')' |
        '.patch(' DocumentOrJSON ')'
      )+
      ( '.sort(' SortExprStrList ')' )? ( '.limit(' NumberOfRows ')' )?
      ( '.bind(' PlaceholderValues ')' )*
      ( '.execute()' )?
```

Figure 11.10 CollectionModifyFunction

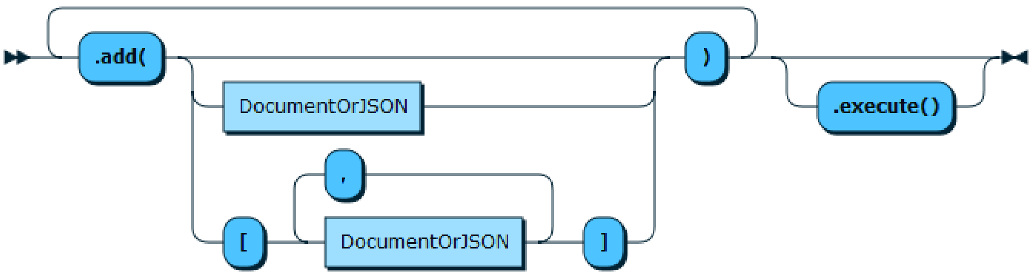


CollectionAddFunction

The syntax for this function shown in EBNF is:

```
CollectionAddFunction
    ::= ( '.add(' ( DocumentOrJSON | '[' DocumentOrJSON ( ',' DocumentOrJSON )* ']' )? ')' )+
        ( '.execute()' )?
```

Figure 11.11 CollectionAddFunction

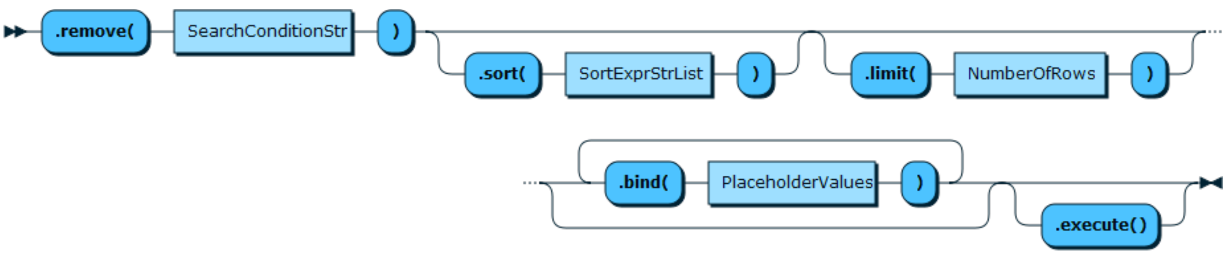


CollectionRemoveFunction

The syntax for this function shown in EBNF is:

```
CollectionRemoveFunction
    ::= '.remove(' SearchConditionStr ')'
        ( '.sort(' SortExprStrList ')' )? ( '.limit(' NumberOfRows ')' )?
        ( '.bind(' PlaceholderValues ')' )*
        ( '.execute()' )?
```

Figure 11.12 CollectionRemoveFunction



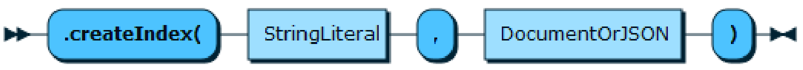
11.4 Collection Index Management Functions

Collection.createIndex() Function

The syntax for this function shown in EBNF is:

```
CollectionCreateIndex
    ::= '.createIndex(' StringLiteral ',' DocumentOrJSON ')'
```

Figure 11.13 CollectionCreateIndexFunction



CollectionDropIndex

The syntax for this function shown in EBNF is:

```
CollectionDropIndex
    ::= '.dropIndex(' StringLiteral ')'
```

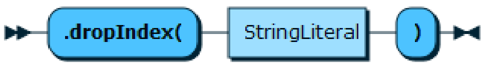


Figure 11.14 CollectionDropIndex

11.5 Table CRUD Functions

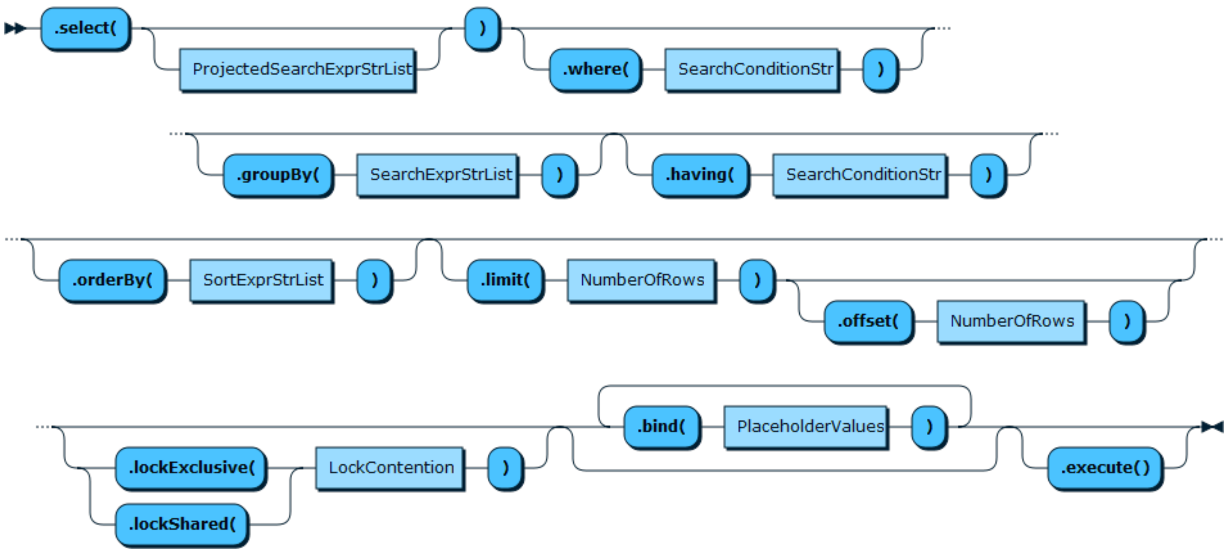
TableSelectFunction

`Table.select()` and `collection.find()` use different methods for sorting results. `Table.select()` follows the SQL language naming and calls the sort method `orderBy()`. `Collection.find()` does not. Use the method `sort()` to sort the results returned by `Collection.find()`. Proximity with the SQL standard is considered more important than API uniformity here.

The syntax for this function shown in EBNF is:

```
TableSelectFunction
  ::= '.select(' ProjectedSearchExprStrList? ')' ( '.where(' SearchConditionStr ')' )?
      ( '.groupBy(' SearchExprStrList ')' )? ( '.having(' SearchConditionStr ')' )?
      ( '.orderBy(' SortExprStrList ')' )? ( '.limit(' NumberOfRows ')' ( '.offset(' NumberOfRows ')' )? )
      ( '.lockExclusive(' LockContention ')' | '.lockShared(' LockContention ')' )?
      ( '.bind(' ( PlaceholderValues ) ')' )*
      ( '.execute()' )?
```

Figure 11.15 TableSelectFunction

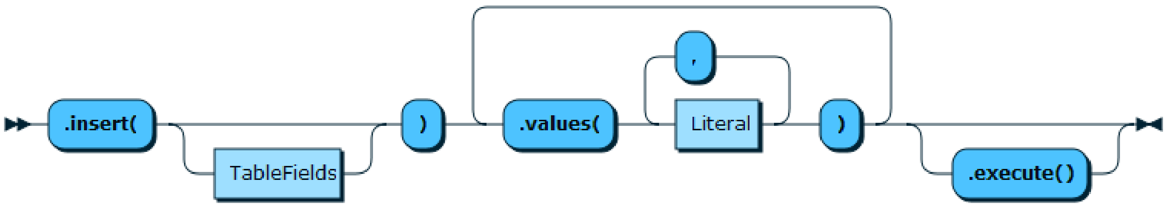


TableInsertFunction

The syntax for this function shown in EBNF is:

```
TableInsertFunction
  ::= '.insert(' ( TableFields )? ')'
        ( '.values(' Literal (',' Literal)* ')' )+
        ( '.execute()' )?
```

Figure 11.16 TableInsertFunction

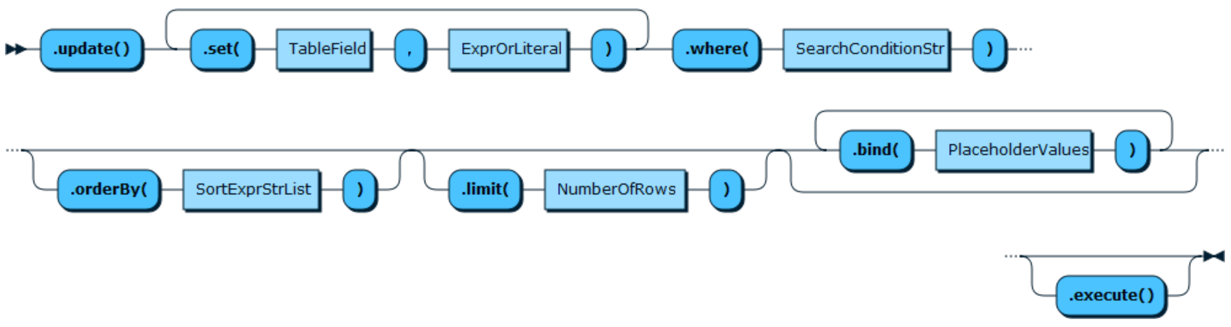


TableUpdateFunction

The syntax for this function shown in EBNF is:

```
TableUpdateFunction
  ::= '.update()'
      ( '.set(' TableField ',' ExprOrLiteral ')' )+ '.where(' SearchConditionStr ')'
      ( '.orderBy(' SortExprStrList ')' )? ( '.limit(' NumberOfRows ')' )?
      ( '.bind(' ( PlaceholderValues ) ')' )*
      ( '.execute()' )?
```

Figure 11.17 TableUpdateFunction

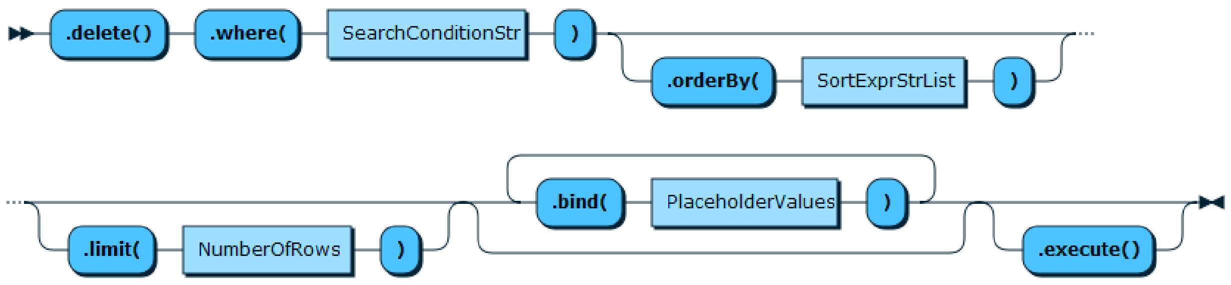


TableDeleteFunction

The syntax for this function shown in EBNF is:

```
TableDeleteFunction
  ::= '.delete()' '.where(' SearchConditionStr ')'
      ( '.orderBy(' SortExprStrList ')' )? ( '.limit(' NumberOfRows ')' )?
      ( '.bind(' ( PlaceholderValues ) ')' )*
      ( '.execute()' )?
```

Figure 11.18 TableDeleteFunction



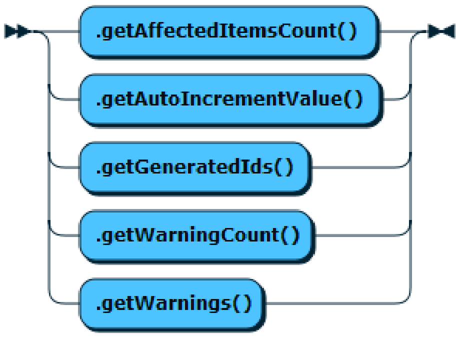
11.6 Result Functions

Result

The syntax for this function shown in EBNF is:

```
Result
    ::= '.getAffectedItemsCount()'
        | '.getAutoIncrementValue()'
        | '.getGeneratedIds()'
        | '.getWarningCount()'
        | '.getWarnings()'
```

Figure 11.19 Result

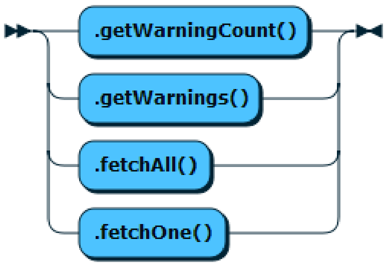


DocResult

The syntax for this function shown in EBNF is:

```
DocResult
    ::= '.getWarningCount()'
        | '.getWarnings()'
        | '.fetchAll()'
        | '.fetchOne()'
```

Figure 11.20 DocResult

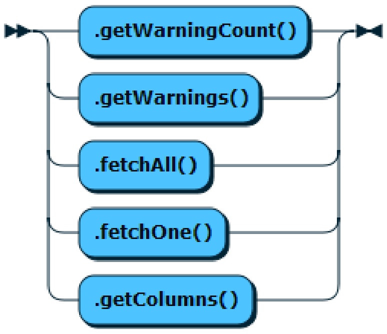


RowResult

The syntax for this function shown in EBNF is:

```
RowResult
    ::= '.getWarningCount()'
        | '.getWarnings()'
        | '.fetchAll()'
        | '.fetchOne()'
        | '.getColumns()'
```

Figure 11.21 RowResult

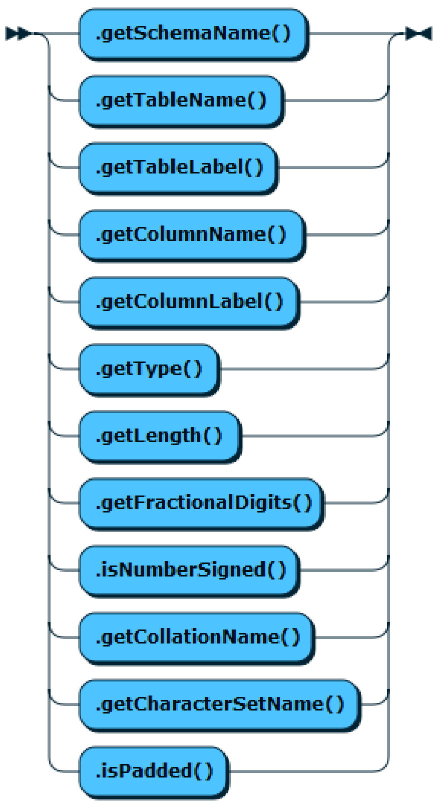


Column

The syntax for this function shown in EBNF is:

```
Column
    ::= '.getSchemaName()'
        | '.getTableName()'
        | '.getTableLabel()'
        | '.getColumnName()'
        | '.getColumnLabel()'
        | '.getType()'
        | '.getLength()'
        | '.getFractionalDigits()'
        | '.isNumberSigned()'
        | '.getCollationName()'
        | '.getCharacterSetName()'
        | '.isPadded()'
```

Figure 11.22 Column

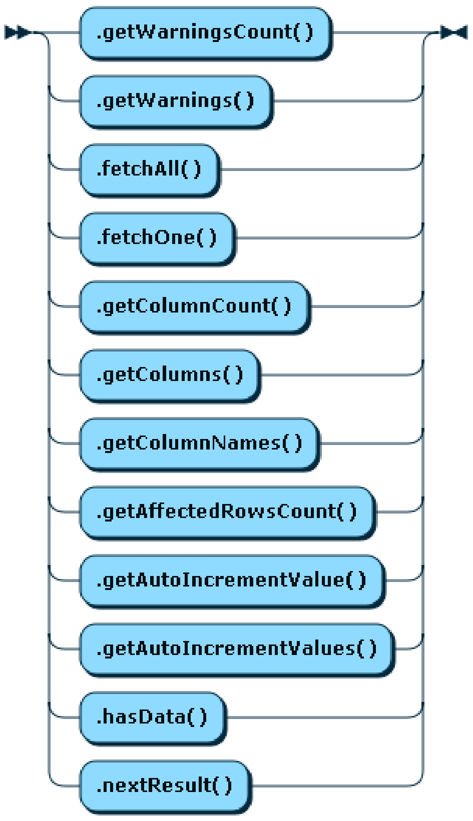


SqlResult

The syntax for this function shown in EBNF is:

```
SqlResult
    ::= '.getWarningCount()'
        | '.getWarnings()'
        | '.fetchAll()'
        | '.fetchOne()'
        | '.getColumns()'
        | '.getAutoIncrementValue()'
        | '.hasData()'
        | '.nextResult()'
```

Figure 11.23 SqlResult



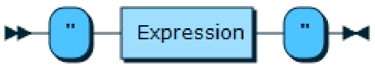
11.7 Other EBNF Definitions

SearchConditionStr

The syntax for this function shown in EBNF is:

```
SearchConditionStr
  ::= '"' Expression '"'
```

Figure 11.24 SearchConditionStr

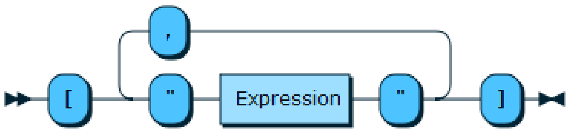


SearchExprStrList

The syntax for this function shown in EBNF is:

```
SearchExprStrList
  ::= '[' '"' Expression '"' ( ',' '"' Expression '"' )* ']'
```

Figure 11.25 SearchExprStrList

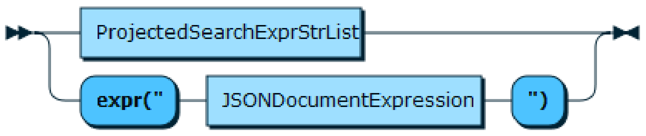


ProjectedDocumentExprStr

The syntax for this function shown in EBNF is:

```
ProjectedDocumentExprStr
  ::= ProjectedSearchExprStrList | 'expr("' JSONDocumentExpression '")'
```

Figure 11.26 ProjectedDocumentExprStr

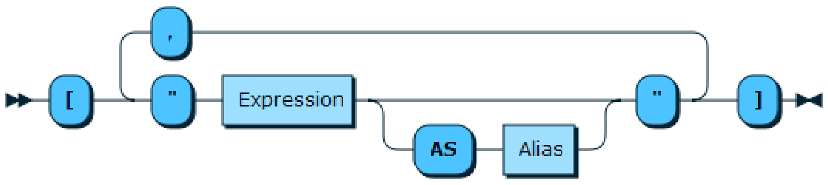


ProjectedSearchExprStrList

The syntax for this function shown in EBNF is:

```
ProjectedSearchExprStrList
  ::= '[' '"' Expression ( 'AS' Alias )? '"' ( ',' '"' Expression ( 'AS' Alias )? '"' )* ']'
```

Figure 11.27 ProjectedSearchExprStrList

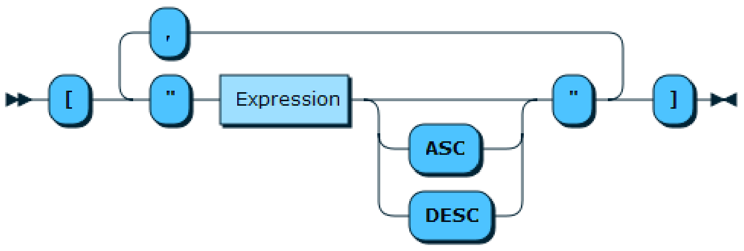


SortExprStrList

The syntax for this function shown in EBNF is:

```
SortExprStrList
  ::= '[' '"' Expression ( 'ASC' | 'DESC' )? '"' ( ',' '"' Expression ( 'ASC' | 'DESC' )? '"' )* ']'
```

Figure 11.28 SortExprStrList

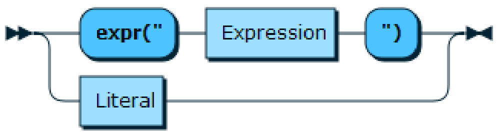


ExprOrLiteral

The syntax for this function shown in EBNF is:

```
ExprOrLiteral
  ::= 'expr("' Expression '")' | Literal
```

Figure 11.29 ExprOrLiteral

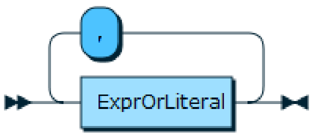


ExprOrLiterals

The syntax for this function shown in EBNF is:

```
ExprOrLiterals
  ::= ExprOrLiteral ( ',' ExprOrLiteral )*
```

Figure 11.30 ExprOrLiterals



ExprOrLiteralOrOperand

The syntax for this function shown in EBNF is:

```
ExprOrLiteralOrOperand
  ::= ExprOrLiteral
```

Figure 11.31 ExprOrLiteralOrOperand

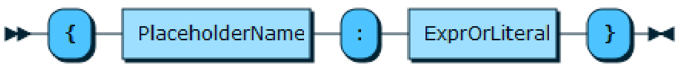
ExprOrLiteral

PlaceholderValues

The syntax for this function shown in EBNF is:

```
PlaceholderValues
  ::= '{' PlaceholderName ':' ( ExprOrLiteral ) '}'
```

Figure 11.32 PlaceholderValues



PlaceholderName

The syntax for this function shown in EBNF is:

```
PlaceholderName
  ::= NamedPlaceholderNotQuestionmarkNotNumbered
```

Figure 11.33 PlaceholderName

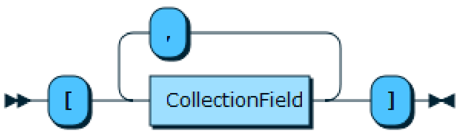
NamedPlaceholderNotQuestionmarkNotNumbered

CollectionFields

The syntax for this function shown in EBNF is:

```
CollectionFields
  ::= ( '[' CollectionField ( ',' CollectionField )* ']' )
```

Figure 11.34 CollectionFields

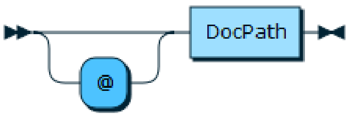


CollectionField

The syntax for this function shown in EBNF is:

```
CollectionField
  ::= '@'? DocPath
```

Figure 11.35 CollectionField

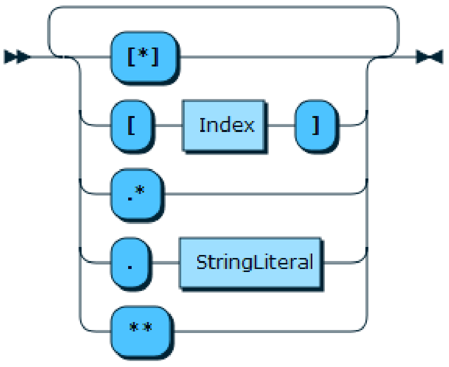


DocPath

The syntax for this function shown in EBNF is:

```
DocPath
  ::= ( '[*]' | ( '[' Index ']' ) | '.*' | ( '.' StringLiteral ) | '**' )+
```

Figure 11.36 DocPath

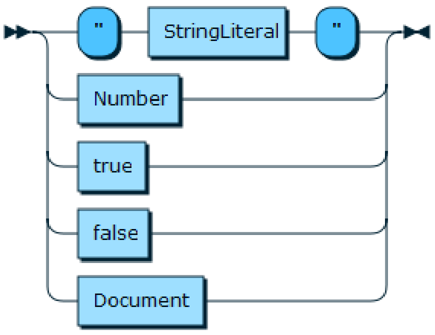


Literal

The syntax for this function shown in EBNF is:

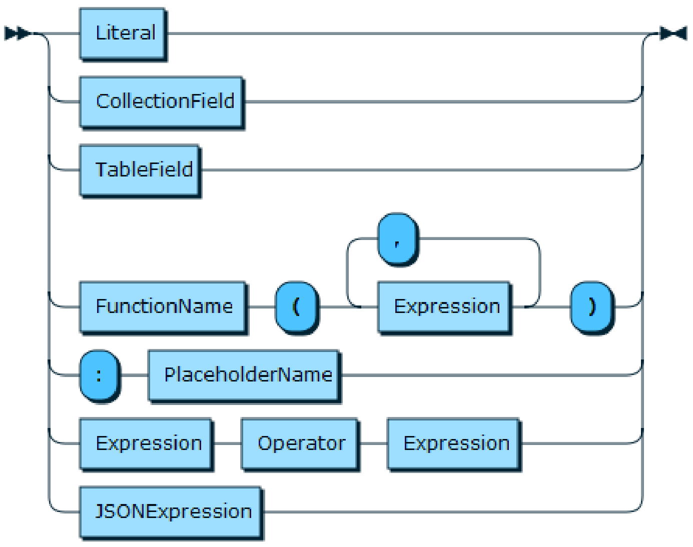
```
Literal
  ::= '"' StringLiteral '"' | Number | true | false | Document
```

Figure 11.37 Literal



Expression

Figure 11.38 Expression



Document

An API call expecting a JSON document allows the use of many data types to describe the document. Depending on the X DevAPI implementation and language any of the following data types can be used:

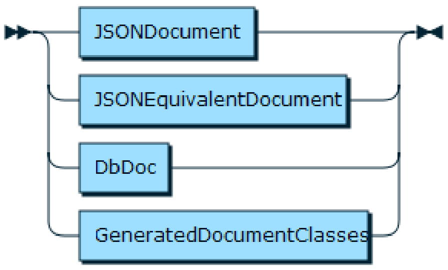
- String
- Native JSON
- JSON equivalent syntax
- DbDoc
- Generated Doc Classes

All implementations of X DevAPI allow expressing a document by the special DbDoc type and as a string.

The syntax for this function shown in EBNF is:

```
Document
  ::= JSONDocument | JSONEquivalentDocument | DbDoc | GeneratedDocumentClasses
```

Figure 11.39 Document

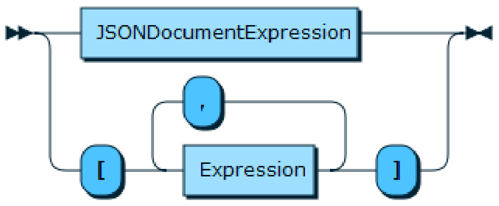


JSONExpression

The syntax for this function shown in EBNF is:

```
JSONExpression
  ::= JSONDocumentExpression | '[' Expression ( ',' Expression )* ']'
```

Figure 11.40 JSONExpression

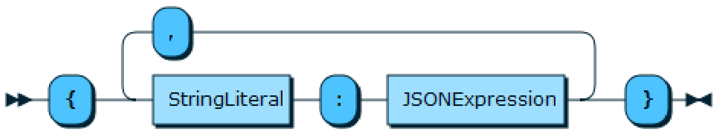


JSONDocumentExpression

The syntax for this function shown in EBNF is:

```
JSONDocumentExpression
  ::= '{' StringLiteral ':' JSONExpression (',' StringLiteral ':' JSONExpression)* '}'
```

Figure 11.41 JSONDocumentExpression

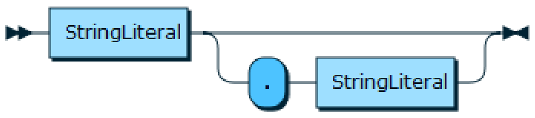


FunctionName

The syntax for this function shown in EBNF is:

```
FunctionName
  ::= StringLiteral | StringLiteral '.' StringLiteral
```

Figure 11.42 FunctionName

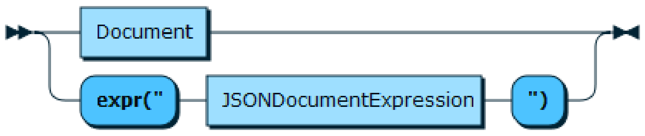


DocumentOrJSON

The syntax for this function shown in EBNF is:

```
DocumentOrJSON
  ::= Document | 'expr("' JSONDocumentExpression '")'
```

Figure 11.43 DocumentOrJSON

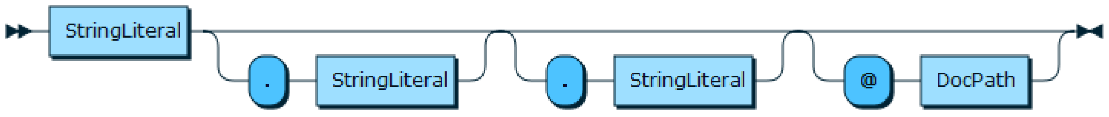


TableField

The syntax for this function shown in EBNF is:

```
TableField
  ::= ( StringLiteral '.' )? ( StringLiteral '.' )? StringLiteral ( '@' DocPath )?
```

Figure 11.44 TableField

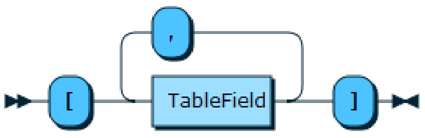


TableFields

The syntax for this function shown in EBNF is:

```
TableFields
  ::= ( '[' TableField ( ',' TableField )* ']' )
```

Figure 11.45 TableFields

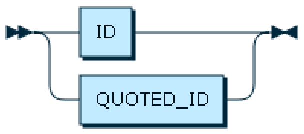


Chapter 12 Expressions EBNF Definitions

This section provides a visual reference guide to the grammar for the expression language used in X DevAPI.

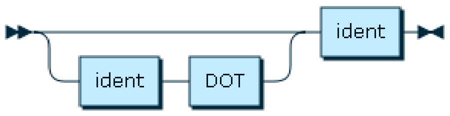
ident

Figure 12.1 ident



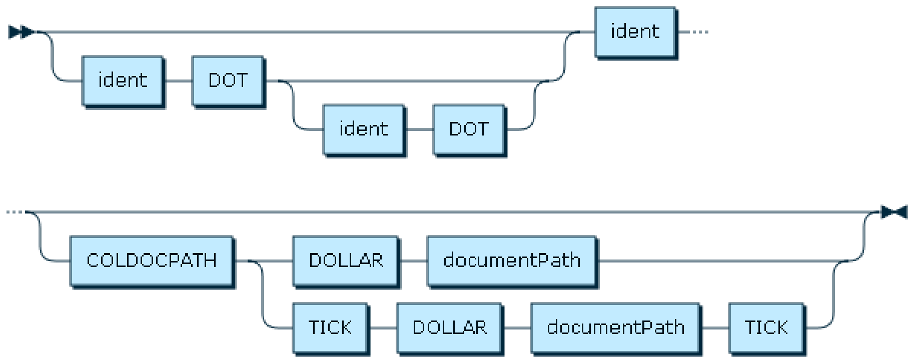
schemaQualifiedIdent

Figure 12.2 schemaQualifiedIdent



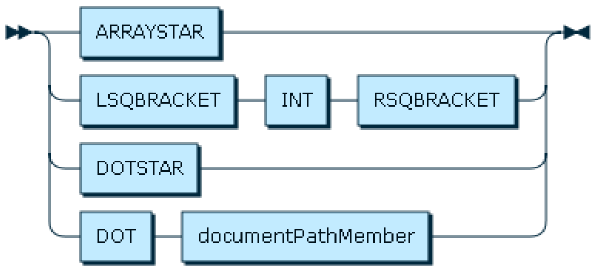
columnIdent

Figure 12.3 columnIdent



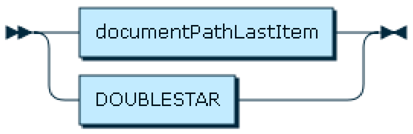
documentPathLastItem

Figure 12.4 documentPathLastItem



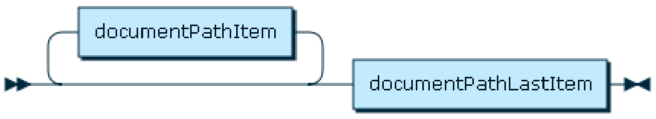
documentPathItem

Figure 12.5 documentPathItem



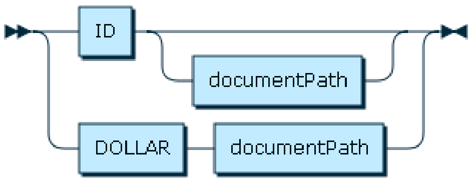
documentPath

Figure 12.6 documentPath



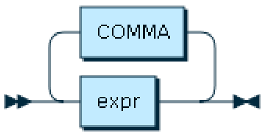
documentField

Figure 12.7 documentField



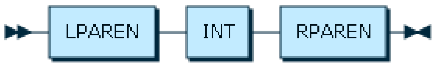
argsList

Figure 12.8 argsList



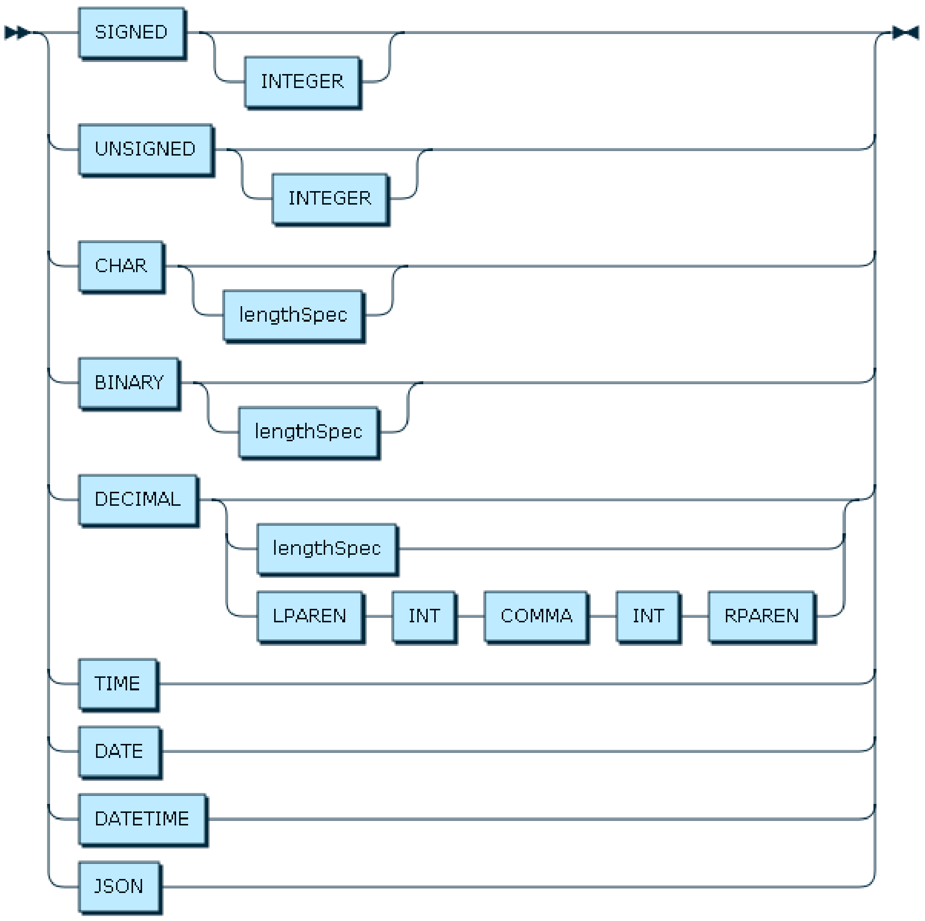
lengthSpec

Figure 12.9 lengthSpec



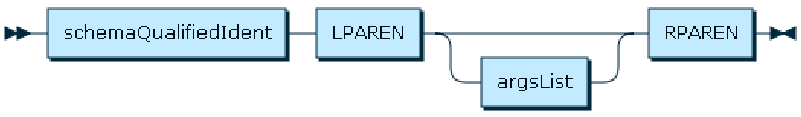
castType

Figure 12.10 castType



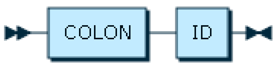
functionCall

Figure 12.11 functionCall



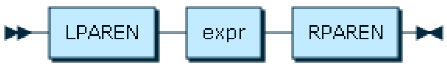
placeholder

Figure 12.12 placeholder



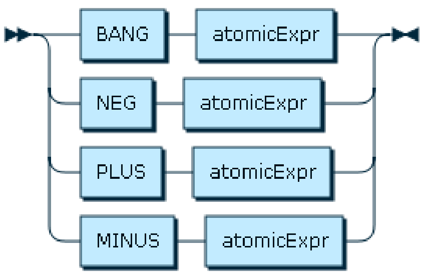
groupedExpr

Figure 12.13 groupedExpr



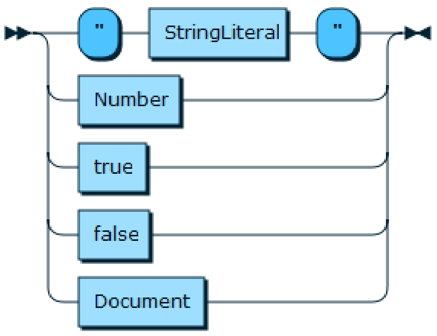
unaryOp

Figure 12.14 unaryOp



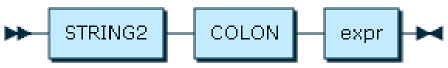
literal

Figure 12.15 literal



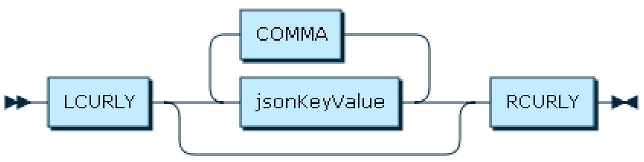
jsonKeyValue

Figure 12.16 jsonKeyValue



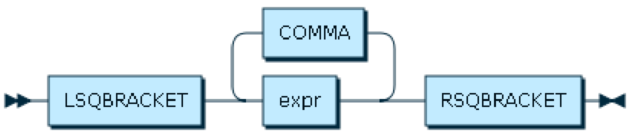
jsonDoc

Figure 12.17 jsonDoc



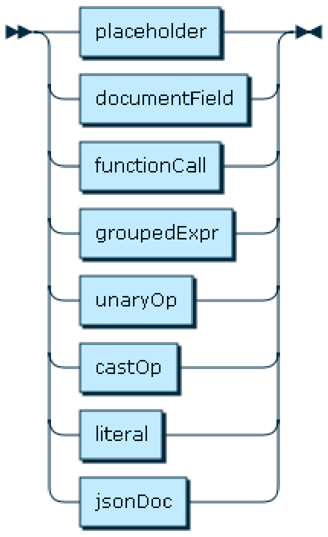
array

Figure 12.18 array



atomicExpr

Figure 12.19 atomicExpr



intervalUnit

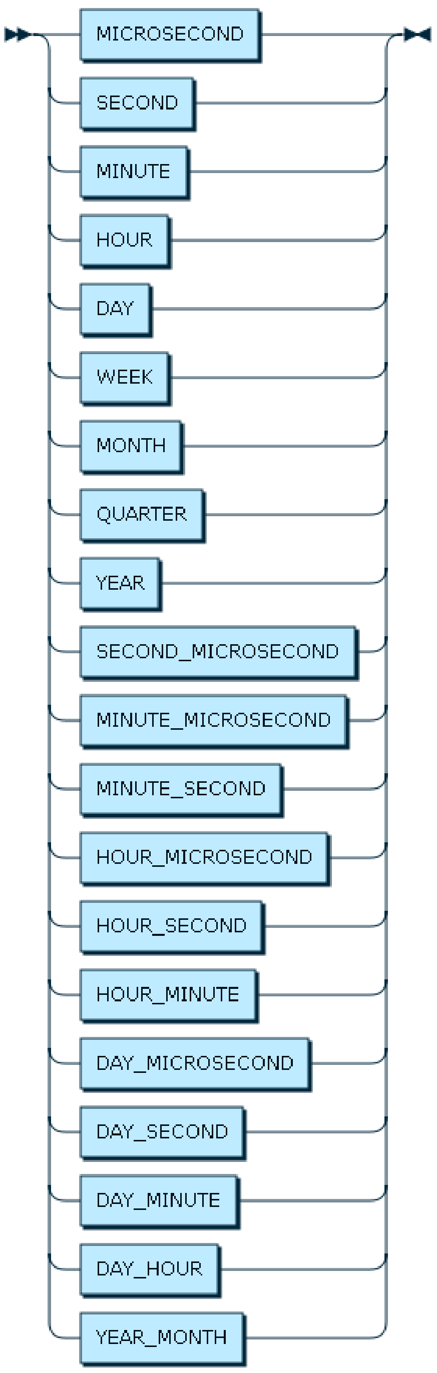


Figure 12.20 intervalUnit

interval

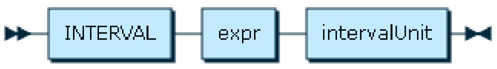
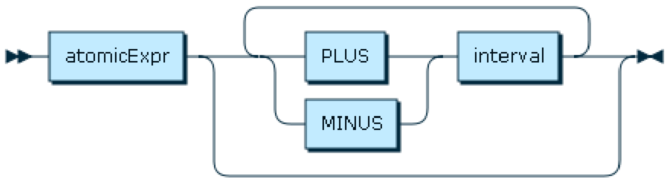


Figure 12.21 interval

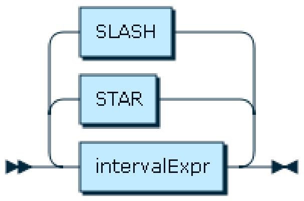
intervalExpr

Figure 12.22 intervalExpr



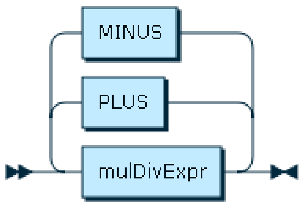
mulDivExpr

Figure 12.23 mulDivExpr



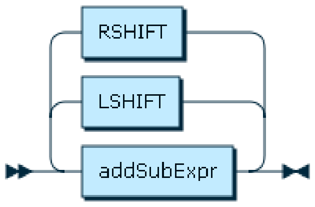
addSubExpr

Figure 12.24 addSubExpr



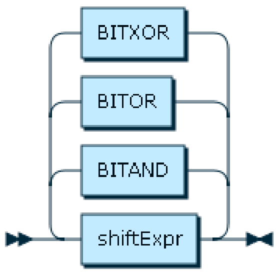
shiftExpr

Figure 12.25 shiftExpr



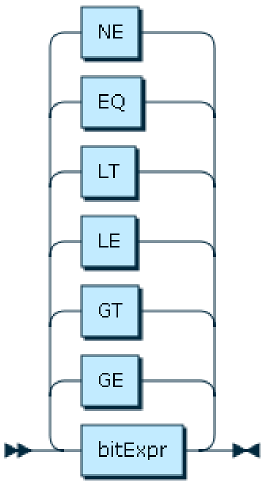
bitExpr

Figure 12.26 bitExpr



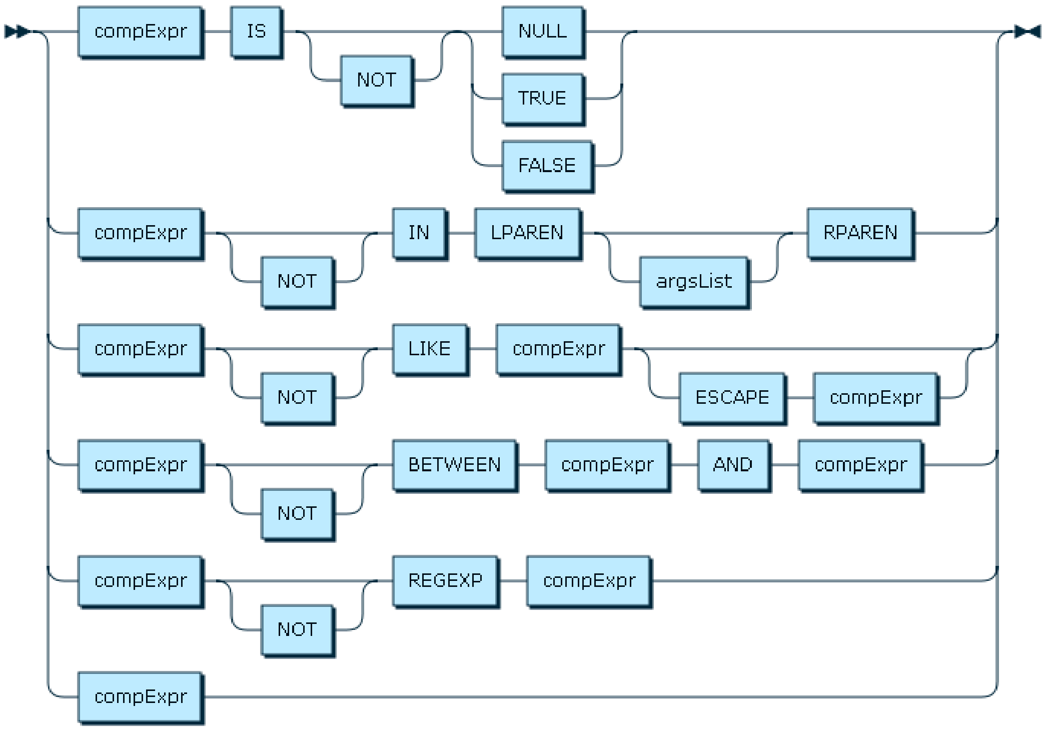
compExpr

Figure 12.27 compExpr



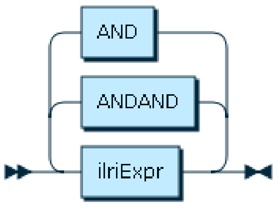
ilriExpr

Figure 12.28 ilriExpr



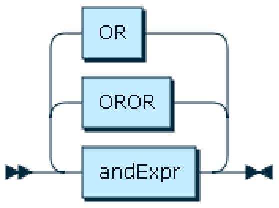
andExpr

Figure 12.29 andExpr



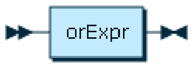
orExpr

Figure 12.30 orExpr



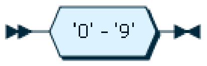
expr

Figure 12.31 expr



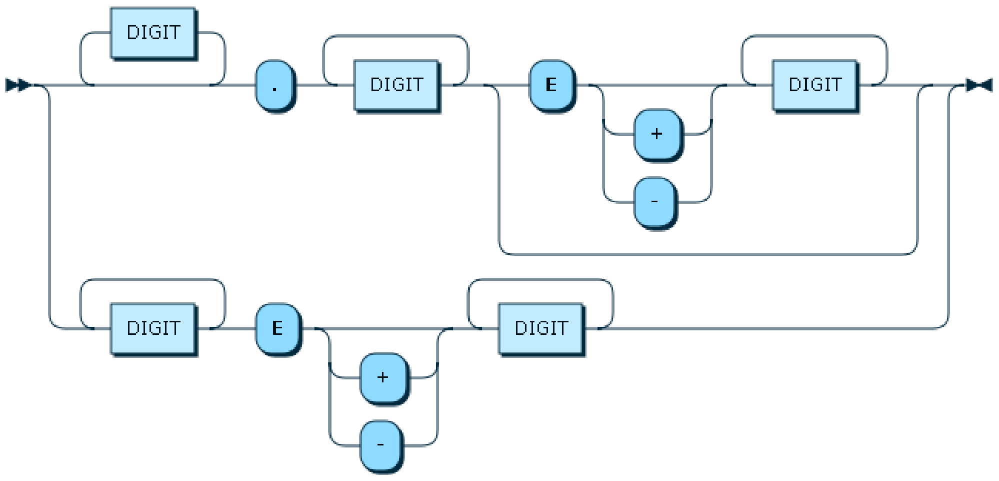
fragment DIGIT

Figure 12.32 fragment DIGIT



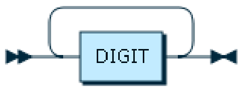
FLOAT

Figure 12.33 FLOAT



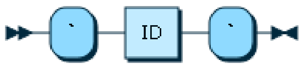
INT

Figure 12.34 INT



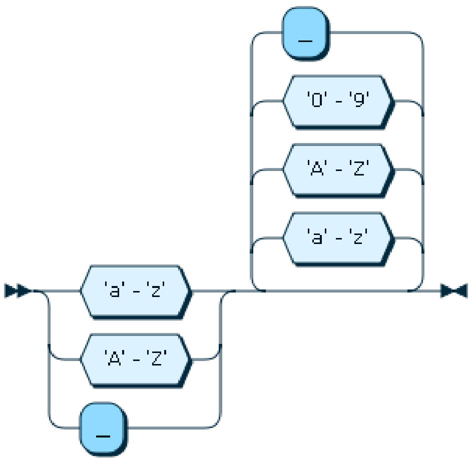
QUOTED_ID

Figure 12.35 QUOTED_ID



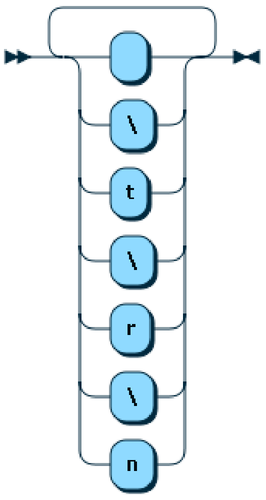
ID

Figure 12.36 ID

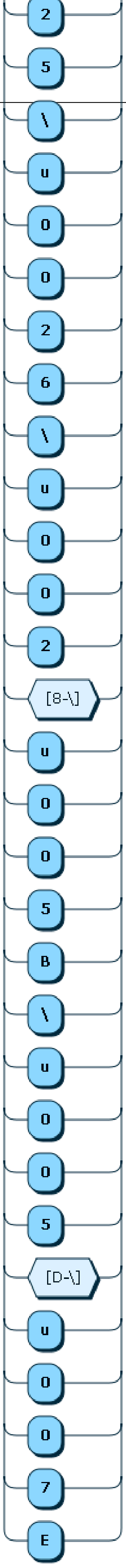


WS

Figure 12.37 WS

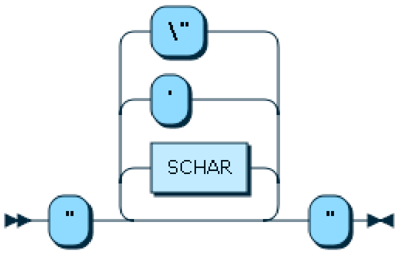


SCHAR



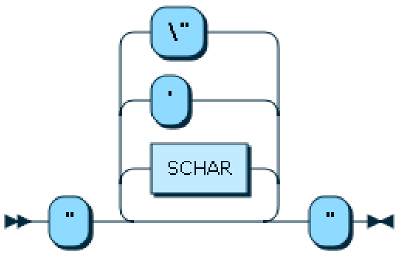
STRING1

Figure 12.39 STRING1



STRING2

Figure 12.40 STRING2



Chapter 13 Implementation Notes

Table of Contents

This section provides notes on the different language-specific implementations of X DevAPI.

13.1 MySQL Connector Notes

Each driver implementation of X DevAPI may deviate from the description in marginal details to align the implementation to the common pattern and styles of the host language. All class names are identical among drivers and all drivers support the same core concepts such as `find()` or the chaining supported for `find()` to ensure developers experience similar APIs in all implementations.

The following implementation differences are possible:

- Function names can be postfixed to add specialisation. For example, implementations can choose between 'execute([<flag_async>])' and/or 'executeAsync()'.
- Functions can have prefixes such as 'get'.
- Connectors may offer native language result set iteration patterns in addition to a basic `while()` loop shown in many examples. For example, drivers may define iterator interfaces or classes.

13.2 MySQL Shell X DevAPI extensions

MySQL Shell deviates from the Connector implementations of X DevAPI in certain places. A Connector can connect to MySQL Server instances running X Plugin only by means of X Protocol. MySQL Shell contains an extension of X DevAPI to access MySQL Server instances through X Protocol. An additional ClassicSession class is available to establish a connection to a single MySQL Server instance using classic MySQL protocol. The functionality of the ClassicSession is limited to basic schema browsing and SQL execution.

See MySQL Shell 8.0 (part of MySQL 8.0), for more information.

13.3 MySQL Connector/Node.js Notes

MySQL Connector/Node.js is built with ECMAScript 6 Promise objects to provide an asynchronous API. All network operations return a Promise, which resolves when the server responds. Please refer to the information on the ES6 Promise implementation.

13.4 MySQL Connector/J Notes

The following are some features unique to X DevAPI for Connector/J:

- The creation of new `Session` objects with `SessionFactory`.
- The creation of new `Client` objects with `ClientFactory`.
- The `JsonValue` interface and its implementations, including `JsonObject`, `JsonString`, `JsonArray`, and `JsonNumber`.
- X Protocol connection properties are prefixed by `xdevapi:`.

- An extended connection string syntax; see Connection URL Syntax for details.