X DevAPI User Guide

Abstract

User documentation for developers using the X DevAPI.

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

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Document generated on: 2017-09-18 (revision: 53987)

Table of Contents

Preface and Legal Notices

This is the X DevAPI 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 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

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 Quick-Start Guide: MySQL Shell for JavaScript and Quick-Start Guide: MySQL Shell for Python.

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. Consult Additional Documentation for the X DevAPI reference documentation for the implementation you are using to verify exact syntax and usage.

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*.

Additional Documentation

In addition to this guide, which provides examples in many languages, a comprehensive reference manual is available for each language which implements the X DevAPI.

- MySQL Shell X DevAPI and AdminAPI reference: JavaScript and Python.
- MySQL Connector/J X DevAPI Reference
- MySQL Connector/Node.js X DevAPI Reference
- MySQL Connector/Net X DevAPI Reference
- MySQL Connector/Python X DevAPI Reference

For general information about using MySQL Shell, which provides JavaScript and Python implementations of X DevAPI, see MySQL Shell User Guide.

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 and using sessions are provided.

This section provides an overview of the connection and session concepts used by the X DevAPI. An X Session is a high-level database session concept that is different from working with traditional low-level MySQL connections. X Sessions can encapsulate one or more actual MySQL connections. Use of this higher abstraction level decouples the physical MySQL setup from the application code. An application using the X DevAPI XSession class can be run against a single MySQL server or a group of MySQL servers with no code changes. When a low-level MySQL connection to a single MySQL instance is needed this is still supported by using a low-level NodeSession.

Before looking at the XSession and NodeSession concepts in more detail, the following examples show that working with a database remains easy.

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 nodes running the X Plugin. Sessions are produced by the `mysqlx` factory. The factory returns two types of Sessions. An XSession encapsulates access to a single MySQL server running the X Plugin or multiple nodes of cluster. A NodeSession serves as an abstraction for a physical connection to exactly one MySQL server running the X Plugin. It is recommended to use the `mysqlx.getSession()` method to obtain an XSession object. Applications that use XSession objects by default can be deployed on both single server setups and database clusters with no code changes. The method expects a list of connection parameters, very much like the code in one of the classic APIs.

The following example code shows how to connect to a MySQL server and get a document from the `my_collection` that has the field `name` starting with `S`. The example assumes that schema called test exists, and the my_collection collection exists. To make the example work, replace `mike` with your username, and `s3cr3t!` 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` .

Note

Please note that MySQL Shell JavaScript and Python code examples are specific to MySQL Shell and rely mostly on the exception handling done by MySQL Shell. For all other languages proper exception handling is required to catch errors. For more information see: Section 8.2, “Error Handling”.

MySQL Shell JavaScript Code

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

```
// Connect to server on localhost
var mySession = mysqlx.getNodeSession( {
                host: 'localhost', port: 33060,
                dbUser: 'mike', dbPassword: 's3cr3t!' } );

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', 'S%').execute();

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

mySession.close();
```

MySQL Shell Python Code

```
import mysqlx

# Connect to server on localhost
mySession = mysqlx.get_node_session( {
        'host': 'localhost', 'port': 33060,
        'dbUser': 'mike', 'dbPassword': 's3cr3t!' } )

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', 'S%').execute()

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

mySession.close()
```

Node.js JavaScript Code

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

// Connect to server on localhost
mysqlx.getSession( {
  host: 'localhost', port: '33060',
  dbUser: 'mike', dbPassword: 's3cr3t!'
}).then(function(session) {
  return session.getSchema('test');
}).then(function (db) {
  // Use the collection 'my_collection'
  var myColl = db.getCollection('my_collection');
  // Specify with document to find with Collection.find() and
  // fetch it from the database with .execute()
  return myColl.find('name like :1').limit(1).bind('S*').execute(function (document) {
    console.log(document);
  });
}).catch(function (err) {
  // Handle error
});
```

C# Code

```

```

```
  // Connect to server on localhost
  var mySession = MySQLX.GetSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");

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

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

  // Specify with 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", "S%").Execute();

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

  mySession.Close();
```

Java Code

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

// Connect to server on localhost
XSession mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password

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", "S%").execute();

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

mySession.close();
```

C++ Code

```
#include <mysql_devapi.h>

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

{
  XSession sess("localhost", 33060, "mike", "s3cr3t!");
  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","S%").execute();

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

By writing MySQL Shell code to a file, such as `test.js` (or `test.py`), you can then execute the code from MySQL Shell.

```
# Launch the MySQL Shell and execute the script
$ mysqlsh < test.js
```

For more information on MySQL Shell, see MySQL Shell User Guide.

2.2 Session Models

This section provides an overview of the XSession and NodeSession objects. The difference between XSession and NodeSession is that XSession abstracts the connection, meaning that it can be a connection to one *or more* MySQL Servers. A NodeSession does not abstract the connection and connects directly to only a single MySQL Server. An XSession is the default when connecting from MySQL Shell. This logical abstraction enables you to address a group of servers without requiring code changes. The following table illustrates the advantages of coding using XSession.

Feature	**XSession**	**NodeSession**
Session Type	Logical	Physical
Document support	✓	✓
Relational Table support	✓	✓
Full SQL Language support	-	✓

The following features are available:

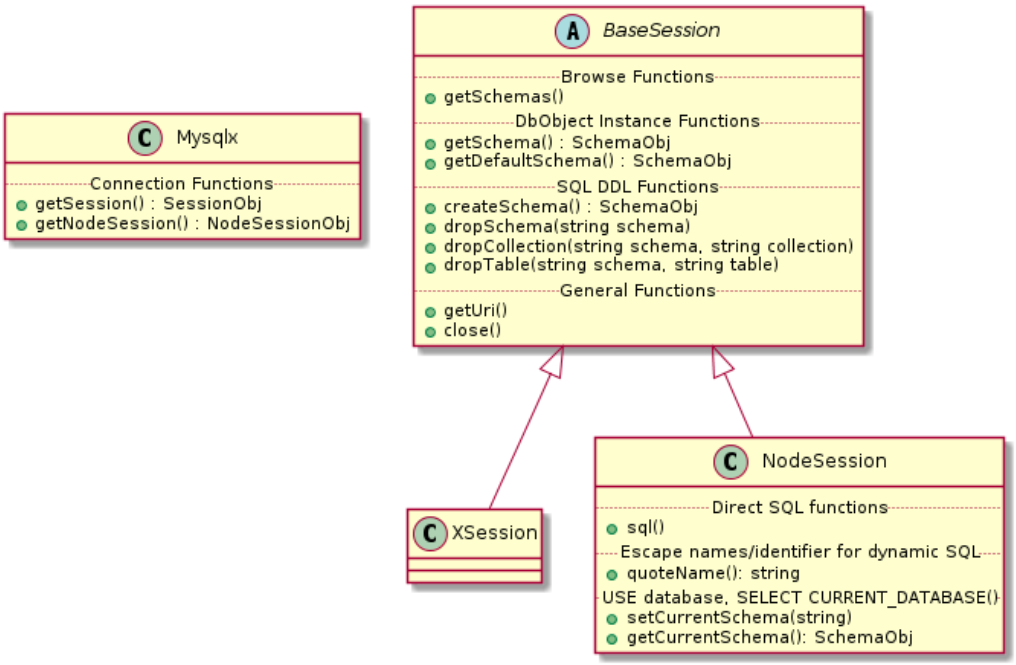
Feature	**XSession**	**NodeSession**
Session Type	Logical	Physical
Document support	✓	✓
Relational Table support	✓	✓
Transparent HA support	-	-
Vertical scaling support	-	-
Horizontal scaling support	-	-
Full SQL Language support	-	✓

MySQL Shell offers an additional ClassicSession class which can be used to execute SQL on MySQL Servers not running X Plugin, see also Section 13.2, “MySQL Shell X DevAPI extensions”.

2.3 Session Classes

The following class diagram gives an overview of the most important classes when working with a MySQL document store.

Figure 2.1 Session - Class Diagram



2.4 The XSession Object

Modern database applications often have to deal with a high number of reads and/or writes per second. This is why it is important to be able to scale the database to deal with increasing load. Even projects that are not initially designed with scaleout in mind may have to grow quickly if they become successful. Therefore it is very important that the database can be scaled at any point in time without having to rewrite the application code.

Using the XSession concept enables you to write code that can be scaled without changing any object. This could be used for transparent High availability (HA), vertical scaleout (database replication) and horizontal scaleout (sharding) for database applications. Although the current version of MySQL X supports XSessions that connect to a single node, future versions could extend this to cover multiple nodes. An application using XSession is prepared to be scaled: change some connect parameters and be done with code changes. An application using NodeSession establishes connections to individual nodes. To scale an application using NodeSession you would have to replace the use of NodeSession with XSession whenever possible. This is likely going to be more coding work than updating some connect parameters.Therefore it is strongly recommended to use XSessions wherever possible.

2.4.1 Connecting to a Session

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

2.4.1.1 Connecting to a Single MySQL Server

In some cases, like prototyping or applications running against a single MySQL Server, it can be helpful to directly specify the host explicitly in the connection string. The following examples show how this is done.

In this example a connection to a local MySQL Server running X Plugin on the default TCP/IP port 33060 is established using the MySQL user account "mike" with the password "s3cr3t!". 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,
  dbUser: 'mike',
  dbPassword: 's3cr3t!' } );

var db1 = dictSession.getSchema('test');

// Passing the parameters in the URL format
var uriSession = mysqlx.getSession('mike:s3cr3t!@localhost:33060');

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

MySQL Shell Python Code

```
# Passing the parameters in the { param: value } format
dictSession = mysqlx.get_node_session( {
        'host': 'localhost', 'port': 33060,
        'dbUser': 'mike', 'dbPassword': 's3cr3t!' } )

db1 = dictSession.get_schema('test')

# Passing the parameters in the URL format
uriSession = mysqlx.get_node_session('mike:s3cr3t!@localhost:33060')

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

The following example shows how to connect to a single MySQL Server 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 use for the user information
print("Please enter the database user information.");
var usr = prompt("Username: ", {defaultValue: "mike"});
var pwd = prompt("Password: ", {type: "password"});

// Connect to MySQL Server running X Plugin on a network machine
var mySession = mysqlx.getSession( {
  host: 'localhost', port: 33060,
  dbUser: usr, dbPassword: pwd} );

var 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': "mike"})
pwd = shell.prompt("Password: ", {'type': "password"})

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

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

C# Code

```
```

```
// Passing the parameters in the { param: value } format
// Query the use for the user information
Console.WriteLine("Please enter the database user information.");
Console.Write("Username: ");
var usr = Console.ReadLine();
if (string.IsNullOrEmpty(usr)) usr = "mike";
Console.Write("Password: ");
var pwd = Console.ReadLine();
if (string.IsNullOrEmpty(pwd)) pwd = "s3cr3t!";

// Connect to MySQL Server on a network machine
var mySession = MySQLX.GetSession(string.Format("host=localhost; port=33060; User={0}; Password={1};",

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

Java Code

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

// Connect to server on localhost using a connection URL
XSession mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password

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
XSession mySession(SessionSettings::HOST, "localhost",
                   SessionSettings::PORT, 33060,
                   SessionSettings::USER, usr,
                   SessionSettings::PWD, pwd);

// An alternative way of defining session settings.

SessionSettings settings;

settings[SessionSettings::HOST] = "localhost";
settings[SessionSettings::PORT] = 33060;
settings[SessionSettings::USER] = usr;
settings[SessionSettings::PWD]  = pwd;

XSession mySession(settings);

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

C Code

2.4.1.2 Transport Protocols

The only transport protocol which is supported is TCP/IP. SSL support is not included in the first release.

2.4.1.3 Connection Option Summary

When using a 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

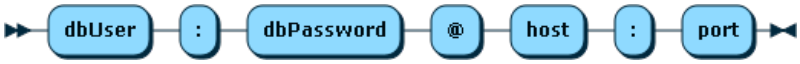
Option	Name	Optional	Default	Notes
TCP/IP Port	port	✓	33060	Standard X Plugin port is 33060
MySQL user	dbUser	-		MySQL database user
MySQL password	dbPassword	-		The MySQL user's password

Supported password encryptions are:

- MYSQL 4.1

URI elements and format.

Figure 2.2 Connection URI



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

2.4.2 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 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().
```

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 URL
var mySession = mysqlx.getNodeSession('mike:s3cr3t!@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
import mysqlx

# Connect to a dedicated MySQL server using a connection URL
mySession = mysqlx.get_node_session('mike:s3cr3t!@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('mysqlx');

// Connect to a dedicated MySQL server using a connection URL
mysqlx.getSession('mike:s3cr3t!@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
  for (schema in schemaList) {
    console.log(schema + '\n');
  }
});
```

C# Code

```
// Connect to a dedicated MySQL server node using a connection URL
var mySession = MySQLX.GetSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");

// 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);
}

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 URL
XSession mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password

// 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 <mysql_devapi.h>

// Connecting to MySQL and working with a Session

// Connect to a dedicated MySQL server using a connection URL
string url = "mysqlx://localhost:33060/test?user=mike&password=s3cr3t!";
{
  XSession 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;
  }
}
```

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.

2.5 NodeSession

When using an XSession, a simplified X DevAPI syntax is used for SQL queries. This exposes a subset of SQL that can be used in transparent high availability, replication, and sharding environments.

In some cases access to the full SQL language is needed, for example to directly connect to a specific MySQL Server to do operations specifically on this server. In those cases a direct, low-level connection needs to be opened. This is performed by using the `mysqlx.getNodeSession()` function, which returns a `NodeSession` object.

The `mysqlx.getNodeSession()` function can take a URL that specifies the connection information for a specific server or it can take a configuration provided by a Session.

2.5.1 Using SQL with NodeSession

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

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

MySQL Shell JavaScript Code

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

// Connect to server using a NodeSession
var mySession = mysqlx.getNodeSession('mike:s3cr3t!@localhost');

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

```
// In a NodeSession 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

```
import mysqlx

# Connect to server using a NodeSession
mySession = mysqlx.get_node_session('mike:s3cr3t!@localhost')

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

# In a NodeSession 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('mysqlx');

// Connect to server using a Low-Level NodeSession
mysqlx.getNodeSession('root:s3kr3t@localhost').then(function(session) {
  return session.getSchema('test');
}).then(function(nodeDb) {
  return Promise.all([
    // Switch to use schema 'test'
    nodeDb.executeSql("USE test").execute(),
    // In a NodeSession context the full SQL language can be used
    nodeDb.executeSql("CREATE PROCEDURE my_add_one_procedure " +
      " (INOUT incr_param INT) " +
      "BEGIN " +
      "  SET incr_param = incr_param + 1;" +
      "END;").execute(),
    nodeDb.executeSql("SET @my_var = ?;", 10).execute(),
    nodeDb.executeSql("CALL my_add_one_procedure(@my_var);").execute(),
```

```
    nodeDb.executeSql("DROP PROCEDURE my_add_one_procedure;").execute()
  ]).then(function() {
    // Use an SQL query to get the result
    return nodeDb.executeSql("SELECT @my_var");
  }).then(function(my_result) {
    // Print result
    console.log(my_result);
  });
});
```

C# Code

```
// Connect to server using a NodeSession
var mySession = MySQLX.GetNodeSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");

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

// In a NodeSession 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();
```

Java Code

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

// Connect to server on localhost
NodeSession mySession = new XSessionFactory().getNodeSession("mysqlx://localhost:33060/test?user=mike&passw

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

// In a NodeSession context the full SQL language can be used
mySession.sql("CREATE PROCEDURE my_add_one_procedure " + " (INOUT incr_param INT) " + "BEGIN " + "  SET inc
        .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 <mysql_devapi.h>

// Connect to server on localhost
```

```
string url = "mysqlx://localhost:33060/test?user=mike&password=s3cr3t!";
NodeSession mySession(url);

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

// In a NodeSession 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;
```

Note

As mentioned previously, literal or verbatim SQL can only be issued when connected to one node. Only the NodeSession class includes a function sql(). The XSession class does not because it encapsulates a connection to many nodes. When connected to multiple MySQL Servers, additional tasks arise over the single node case. Among these tasks is failover. The standard query language allows users to establish a stateful connection. Transparently failing over a stateful connection is a difficult and complex task. The X DevAPI XSession class implements a significant subset of the SQL language through API calls. The API calls offered do not allow building a connection state. This is a key enabler for automatic and transparent failover. Future versions of the X DevAPI might support literal SQL when working with the XSession class. Note that you are not prevented from using the full power of SQL but literal or verbatim SQL is only available with NodeSession based connections.

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.

2.5.2 Setting the Current Schema

A newly opened session has no schema set. There is no concept of an implicitly set default schema when a session is established. A default schema can be defined, but a session's current schema is not set to a default before being explicitly selected.

An explicit schema selection happens as part of using the CRUD functions of the X DevAPI. For example, Table and Collection objects are created by Schema objects. Schema objects, in turn, are created by Session objects. To insert rows into a table, call `session.getSchema().getTable().insert()`. To add a document to a collection from the default schema, call `session.getDefaultSchema().getCollection().add()`.

The `sql()` function is a method of the NodeSession class but not the Schema class. Upon creation, the NodeSession class has no schema selected by default. Therefore, the `sql()` function does not know which schema to execute the SQL statement against. Use the Session class `setCurrentSchema()` function to set or change the current schema.

MySQL Shell JavaScript Code

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

// Direct connect with no client side default schema defined
var mySession = mysqlx.getNodeSession('mike:s3cr3t!@localhost');
mySession.setCurrentSchema("test");
```

MySQL Shell Python Code

```
import mysqlx

# Direct connect with no client side default schema defined
mySession = mysqlx.get_node_session('mike:s3cr3t!@localhost')
mySession.set_current_schema("test")
```

Node.js JavaScript Code

```

```

C# Code

```
// Direct connect with no client side default schema defined
var mySession = MySQLX.GetNodeSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");
mySession.GetSchema("test");
```

```
var session = MySQLX.GetNodeSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");

var default_schema = session.GetSchema("test");

// print the current schema name
Console.WriteLine(session.Schema.Name);


private Table CreateTestTable(NodeSession 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);
}
```

Java Code

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

// Schema 'test' set in connection URL
NodeSession mySession = new XSessionFactory().getNodeSession("mysqlx://localhost:33060/test?user=mike&passw
```

C++ Code

```
#include <mysql_devapi.h>

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

```
string url = "mysqlx://localhost:33060/test?user=mike&password=s3cr3t!"
NodeSession mySession(url);
```

C Code

```
```

2.5.3 Dynamic SQL

A quoting function exists to escape SQL names and identifiers. `NodeSession.quoteName()` escapes the identifier given in accordance to the settings of the current connection. The escape function must not be used to escape values. Use the value bind syntax of `NodeSession.sql()` instead.

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.getNodeSession('mike:s3cr3t!@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)

import mysqlx

session = mysqlx.get_node_session('mike:s3cr3t!@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

```
function createTestTable(session, name) {

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

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

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

  session.executeSql(create).execute();

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

var mysqlx = require('mysqlx');

mysqlx.getNodeSession({
  dataSourceFile: 'mysqlxconfig.json', app: 'myapp',
  dbUser: 'mike', dbPassword: 's3cr3t!'
}).then(function (session) {

  session.executeSql("use myschema").execute();
  var default_schema = session.getDefaultSchema().name;
  session.setCurrentSchema(default_schema);

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

  session.close();
});
```

C# Code

```
var session = MySQLX.GetNodeSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");

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

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

Java Code

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

C++ Code

```
#include <mysql_devapi.h>

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

Table createTestTable(NodeSession &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);
}

NodeSession session(33060, "mike", "s3cr3t!");

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

Users of the X DevAPI do 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

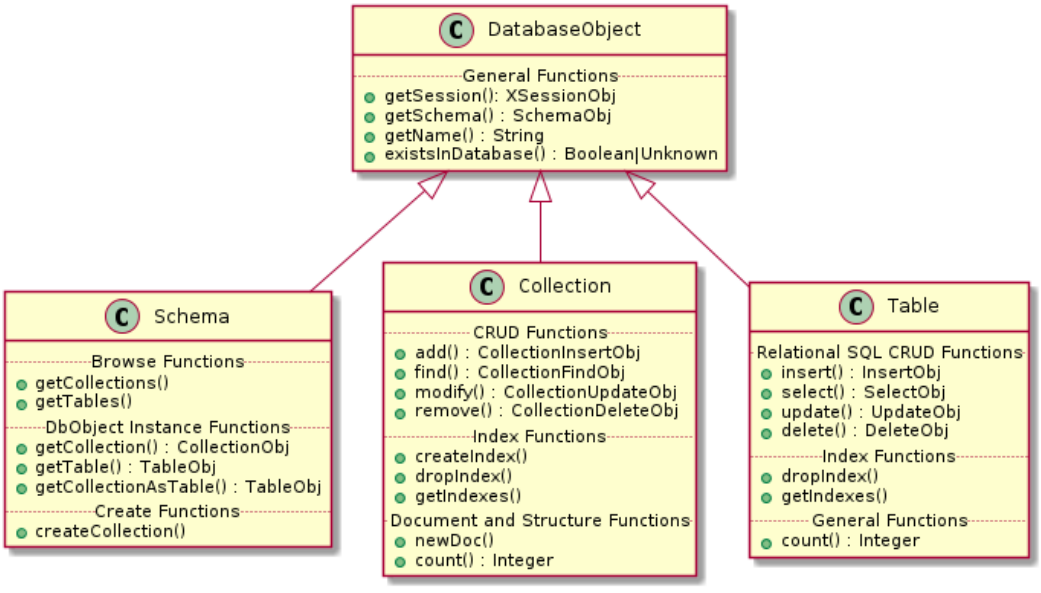
CRUD operations are available as methods, which operate on Schema objects. The available Schema objects consist of Collection objects, containing Documents, or Table objects consisting of rows and Collections containing Documents.

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

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

The following examples show how method chaining is used instead of an SQL string when working with XSession and NodeSession. 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
// This is only available when using a NodeSession
// 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
# This is only available when using a NodeSession
# 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
// This is only available when using a NodeSession
// It should only be used when absolutely necessary
var promise = db.executeSql('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
{
  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
  // This is only available when using a NodeSession
  // 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();
}
```

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
// This is only available when using a NodeSession
// 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
// This is only available when using a NodeSession
// 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 multi-threaded 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('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

Languages such as C# 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/Java 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.

Consult your language's Connector reference for more details, see Additional Documentation.

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 = 15').execute();

// Using the .bind() function to bind parameters
var myRes2 = myColl.find('name = :param1 AND age = :param2').bind('param1','jack').bind('param2', 17).execu

// Using named parameters
myColl.modify('name = :param').set('age', 37).
        bind('param', 'clare').execute();

// Binding works for all CRUD statements except add()
var myRes3 = myColl.find('name like :param').
        bind('param', 'J%').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 = 15').execute()

# Using the .bind() function to bind parameters
myRes2 = myColl.find('name = :param1 AND age = :param2').bind('param1','jack').bind('param2', 17).execute()

# Using named parameters
myColl.modify('name = :param').set('age', 37).bind('param', 'clare').execute()

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

C# Code

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

var myRes1 = myColl.Find("age = 15").Execute();

// Using the .bind() function to bind parameters
var myRes2 = myColl.Find("name = :param1 AND age = :param2").Bind("param1", "jack").Bind("param2", 17).

// Using named parameters
myColl.Modify("name = :param").Set("age", 37)
  .Bind("param", "clare").Execute();

// Binding works for all CRUD statements except add()
var myRes3 = myColl.Find("name like :param")
  .Bind("param", "J%").Execute();
```

Java Code

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

DocResult myRes1 = myColl.find("age = 15").execute();

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

// Using named parameters
myColl.modify("name = :param").set("age", 37)
  .bind("param", "clare").execute();

// Using named parameters with a Map
Map<String, Object> params = new HashMap<>();
params.put("name", "clare");
myColl.modify("name = :name").set(".age", 37).bind(params).execute();

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

C++ Code

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

auto myRes1 = myColl.find("age = 15").execute();

// Using the .bind() function to bind parameters
auto myRes2 = myColl.find("name = :param1 AND age = :param2")
                    .bind("param1","jack").bind("param2", 17)
                    .execute();

// Using named parameters
myColl.modify("name = :param").set("age", 37)
      .bind("param", "clare").execute();

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

Anonymous placeholders are not supported in the 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 occurences 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
var myColl = db.getCollection('relatives');
myColl.find('alias = "jr"').execute(function (junior) {
  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();
```

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', 'mike').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', 'S%').bind('param2', 18).execute();
var MyOtherDocs = myFind.bind('param1', 'M%').bind('param2', 24).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', 'mike').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', 'S%').bind('param2', 18).execute()
MyOtherDocs = myFind.bind('param1', 'M%').bind('param2', 24).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', 'mike').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', 'S%').bind('param2', 18).execute();
var MyOtherDocs = myFind.bind('param1', 'M%').bind('param2', 24).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", "mike").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", "S%").Bind("param2", 18).Execute();
var MyOtherDocs = myFind.Bind("param1", "M%").Bind("param2", 24).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", "mike").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", "S%");
params.put("age", 18);
DocResult myDocs = myFind.bind(params).execute();
params.put("name", "M%");
params.put("age", 24);
DocResult myOtherDocs = myFind.bind(params).execute();
```

C++ Code

```
// Ths functionality is not yet implemented in Connector/C++. At the
// moment a crud operation can be executed only once, with one set
// of parameter values.
```

3.5 MySQL Shell Automatic Code Execution

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

```
var result = session.sql('show databases').execute();
var city_res = db.cities.find().execute();
```

The call of the `execute()` function above causes the operation to be executed and return 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 the 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 User Guide.

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 examples above in MySQL Shell and removing the assignment operation shows they are automatically executed.

```
mysql-js> session.sql('show databases');
```

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 straight forward when using the X DevAPI. The following examples show the basic usage of CRUD operations when working with documents.

After establishing a connection to a MySQL Server, 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.getNodeSession( {
host: 'localhost', port: 33060,
dbUser: 'mike', dbPassword: 's3cr3t!'} );

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

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

// Insert documents
myColl.add({name: 'Sakila', age: 15}).execute();
myColl.add({name: 'Susanne', age: 24}).execute();
myColl.add({name: 'Mike', age: 39}).execute();

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

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

// Drop the collection
session.dropCollection('test','my_collection');
```

MySQL Shell Python Code

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

import mysqlx

# Connect to server
```

```
mySession = mysqlx.get_node_session( {
'host': 'localhost', 'port': 33060,
'dbUser': 'mike', 'dbPassword': 's3cr3t!'} )

myDb = mySession.get_schema('test')

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

# Insert documents
myColl.add({'name': 'Sakila', 'age': 15}).execute()
myColl.add({'name': 'Susanne', 'age': 24}).execute()
myColl.add({'name': 'Mike', 'age': 39}).execute()

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

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

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

Node.js JavaScript Code

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

var mysqlx = require('mysqlx');

// Connect to server
mysqlx.getSession( {
  host: 'localhost', port: '33060',
  dbUser: 'mike', dbPassword: 's3cr3t!'
}).then(function(session) {
  return session.getSchema('test');
}).then(function(db) {
  // Create a new collection 'my_collection'
  return db.createCollection('my_collection');
}).then(function(myColl) {
  // Insert documents
  return Promise.all([
    myColl.add({name: 'Sakila', age: 15}).execute(),
    myColl.add({name: 'Susanne', age: 24}).execute(),
    myColl.add({name: 'Mike', age: 39}).execute()
  ]).then(function() {
    // Find a document
    return myColl.find('name like :name AND age < :age')
      .bind({ name: 'S*', age: 20 }).limit(1).execute();
  }).then(function(docs) {
    // Print document
    console.log(docs.fetchOne());

    // Drop the collection
    return myColl.drop();
  });
}).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=mike;password=s3cr3t!;");

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

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

// Insert documents
myColl.Add(new { name = "Sakila", age = 15}).Execute();
myColl.Add(new { name = "Susanne", age = 24}).Execute();
myColl.Add(new { name = "Mike", age = 39}).Execute();

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

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

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

Java Code

```
// Connect to server
XSession mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password

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

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

// Insert documents
myColl.add("{\"name\":\"Sakila\", \"age\":15}").execute();
myColl.add("{\"name\":\"Susanne\", \"age\":24}").execute();
myColl.add("{\"name\":\"Mike\", \"age\":39}").execute();

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

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

// Drop the collection
mySession.dropCollection("test", "my_collection");
```

C++ Code

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

#include <mysql_devapi.h>

// Connect to server
XSession session(33060, "mike", "s3cr3t!");
Schema db = session.getSchema("test");

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

// Insert documents
myColl.add(R"({"name": "Sakila", "age": 15})").execute();
myColl.add(R"({"name": "Susanne", "age": 24})").execute();
myColl.add(R"({"name": "Mike", "age": 39})").execute();

// Find a document
```

```
DocResult docs = myColl.find("name like :param1 AND age < :param2").limit(1)
                       .bind("param1","S%").bind("param2",20).execute();

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

// Drop the collection
session.dropCollection("test", "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. The 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);
}
```

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 promise = db.getCollection('my_collection');

// Get a collection object but also ensure it exists in the database
var promise = db.getCollection('my_collection', { validateExistence: true } );
```

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);
}
```

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 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.

4.3 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) statements. 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( { name: 'Sakila', age: 15 } ).execute();

// Insert several documents at once
myColl.add( [
{ name: 'Susanne', age: 24 },
{ name: 'Mike', age: 39 } ] ).execute();
```

MySQL Shell Python Code

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

# Insert a document
myColl.add( { 'name': 'Sakila', 'age': 15 } ).execute()

# Insert several documents at once
myColl.add( [
{ 'name': 'Susanne', 'age': 24 },
{ 'name': 'Mike', 'age': 39 } ] ).execute()
```

Node.js JavaScript Code

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

  // Insert a document
  myColl.add( { name: 'Sakila', age: 15 } ).execute();
```

```
  // Insert several documents at once
  myColl.add( [
    { name: 'Susanne', age: 24 },
    { name: 'Mike', age: 39 }
  ] ).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 = "Sakila", age = 15 }).Execute();

// Insert several documents at once
myColl.Add(new[] {
new { name = "Susanne", age = 24 },
new { name = "Mike", age = 39 } }).Execute();
}
```

Java Code

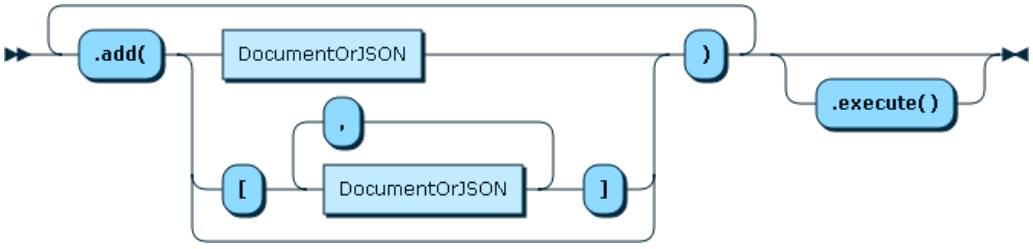
```
// Create a new collection
Collection coll = db.createCollection("payments");

// Insert a document
coll.add("{\"name\":\"Sakila\", \"age\":15}");

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

C++ Code

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

// Insert a document
coll.add(R"({"name":"Sakila", "age":15})").execute();

// Insert several documents at once
std::list<DbDoc> docs = {
  R"({"name":"Susanne", "age":24})",
  R"({"name":"Mike", "age":39})"
};
coll.add(docs).execute();
```

The following diagram shows the full syntax definition.

Figure 4.1 Collection.add() Syntax Diagram



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 string with a maximum length of 32 characters.

You do not need to provide the `_id` field manually. As shown, documents can be inserted without the field. If no `_id` field is given, a unique value is generated. Auto generated values are reported by the `Result.getLastDocumentId()` function. The 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();
print("User Provided Id:", result.getLastDocumentId());

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

MySQL Shell Python Code

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

Node.js JavaScript Code

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

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

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);
}
```

Java Code

```
// If the _id is provided, it will be honored
Result result = coll.add("{\"_id\":\"custom_id\",\"a\":1}").execute();
System.out.println("User Provided Id:" + result.getLastDocumentIds().get(0));

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

C++ Code

```
// If the _id is provided, it will be honored
Result result = myColl.add(R"({ "_id": "custom_id", "a" : 1 })").execute();
cout << "User Provided Id:" << result.getDocumentId() << endl;

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

MySQL Shell JavaScript Code

```
// using a book's unique ISBN as the document 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:

MySQL Shell JavaScript Code

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

Currently, the 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' starts with an 'S'
var docs = myColl.find('name like :param').
           limit(1).bind('param', 'S%').execute();

print(docs.fetchOne());

// Get all documents with a field 'name' that starts with an 'S'
docs = myColl.find('name like :param').
        bind('param','S%').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' starts with an 'S'
docs = myColl.find('name like :param').limit(1).bind('param', 'S%').execute()

print docs.fetch_one()

# Get all documents with a field 'name' that starts with an 'S'
docs = myColl.find('name like :param').bind('param','S%').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' starts with an 'S'
myColl.find('name like :name').bind('S%').limit(1).execute(function (doc) {
  console.log(doc);
});

// Get all documents with a field 'name' that starts with an 'S'
myColl.find('name like :name')
.bind('S%').execute(function (myDoc) {
  console.log(myDoc);
});
```

C# Code

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

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

Console.WriteLine(docs.FetchOne());

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

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

Java Code

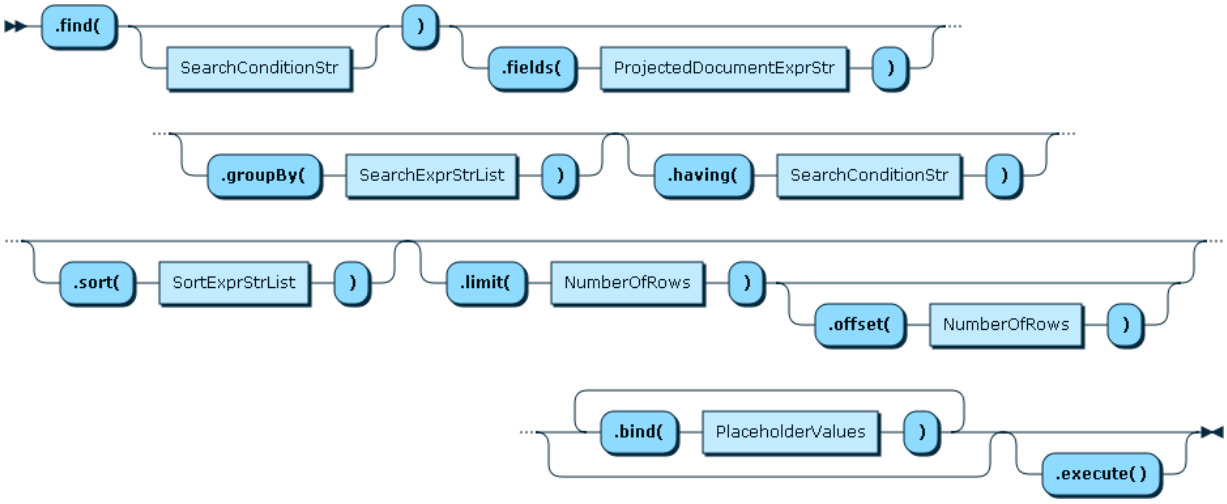
```
// Use the collection 'my_collection'
Collection myColl = db.getCollection("my_collection");

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

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

// Get all documents with a field 'name' that starts with an 'S'
docs = myColl.find("name like :name").bind("name", "S%").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' starts with an 'S'
DocResult docs = myColl.find("name like :param")
                       .limit(1).bind("param", "S%").execute();

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

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

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

The following diagram shows the full syntax definition.

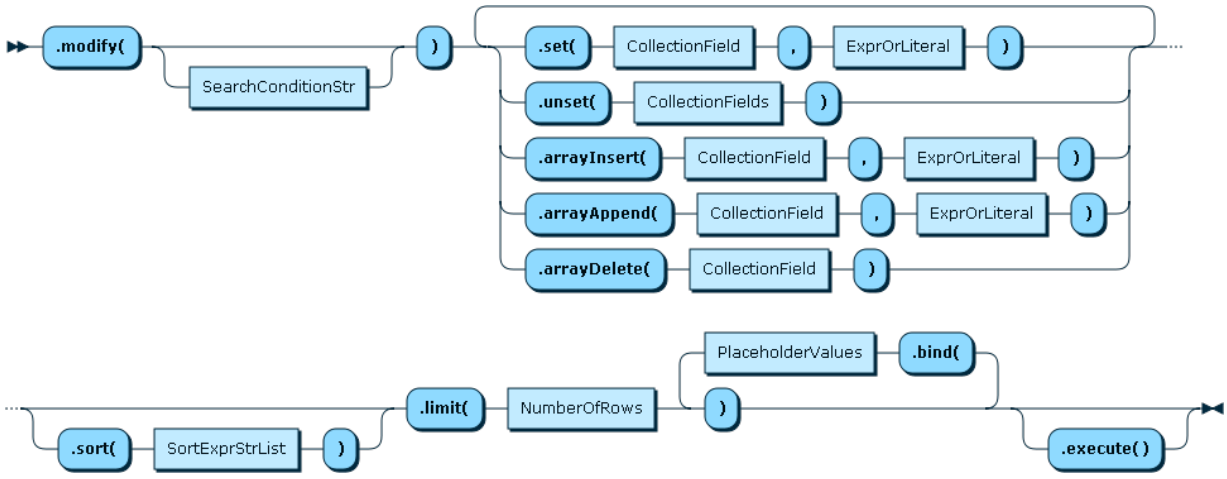
Figure 4.2 Collection.find() Syntax Diagram



For more information, see Section 11.7, “Other EBNF Definitions”.

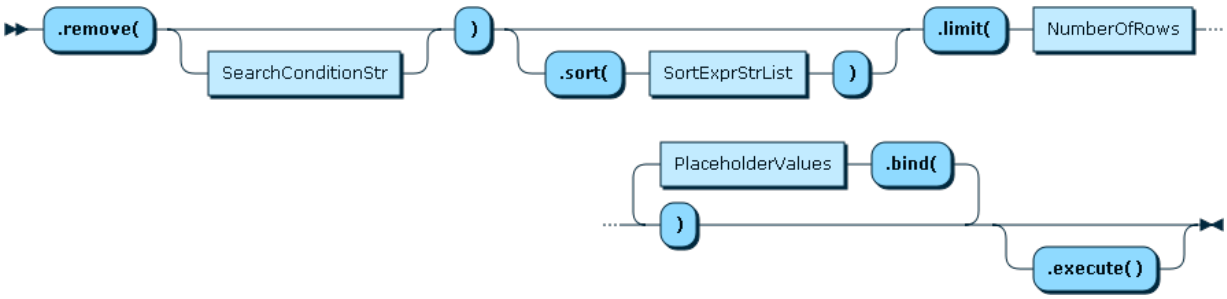
Collection.modify()

Figure 4.3 Collection.modify() Syntax Diagram



Collection.remove()

Figure 4.4 Collection.remove() Syntax Diagram



Chapter 5 Working with Documents

Once a collection has been created in the database, it can store JSON documents. Storing documents is as easy as 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, 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({name: 'Sakila', age: 15});

// Inserting several docs at once
myColl.add([ {name: 'Susanne', age: 24},
  {name: 'Mike', age: 39} ]);
```

MySQL Shell Python Code

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

# Insert JSON data directly
myColl.add({'name': 'Sakila', 'age': 15})

# Inserting several docs at once
myColl.add([ {'name': 'Susanne', 'age': 24},
  {'name': 'Mike', 'age': 39} ])
```

Node.js JavaScript Code

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

  // Insert JSON data directly
  myColl.add({name: 'Sakila', age: 15});

  // Inserting several docs at once
  myColl.add([ {name: 'Susanne', age: 24},
    {name: 'Mike', age: 39} ]);

  // Insert Documents
```

```
  var myDoc = {};
  myDoc.name = 'James';
  myDoc.age = 47;
  myColl.add(myDoc);
})
```

C# Code

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

  // Insert JSON data directly
  myColl.Add(new { name = "Sakila", age = 15 }).Execute();

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

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

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


}
```

Java Code

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

// Insert JSON data directly
coll.add("{\"name\":\"Sakila\", \"age\":15}");

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

// Insert Documents
DbDoc myDoc = new DbDoc();
myDoc.add("name", new JsonString().setValue("James"));
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": "Sakila", "age": 15})").execute();

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

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.getNodeSession( {
  host: 'localhost', port: 33060,
  dbUser: 'mike', dbPassword: 's3cr3t!'} )

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('Sakila', mysqlx.dateValue(2000, 5, 27), 16).execute();

// Find a row in the SQL Table
var myResult = myTable.select(['_id', 'name', 'birthday']).
  where('name like :name AND age < :age').
  bind('name', 'S%').bind('age', 20).execute();

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

MySQL Shell Python Code

```
# Working with Relational Tables
import mysqlx

# Connect to server using a connection URL
mySession = mysqlx.get_node_session( {
  'host': 'localhost', 'port': 33060,
  'dbUser': 'mike', 'dbPassword': 's3cr3t!'} )

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('Sakila', mysqlx.date_value(2000, 5, 27), 16).execute()

# Find a row in the SQL Table
myResult = myTable.select(['_id', 'name', 'birthday']) \
  .where('name like :name AND age < :age') \
  .bind('name', 'S%') \
  .bind('age', 20).execute()
```

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

Node.js JavaScript Code

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

// Connect to server using a connection URL
mysqlx.getSession({
  host: 'localhost', port: 33060,
  dbUser: 'mike', dbPassword: 's3cr3t!'
}).then(function (session) {
  var db = session.getSchema('test');

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

  // Insert SQL Table data
  myTable.insert(['name', 'birthday', 'age']).
    values('Sakila', mysqlx.dateValue(2000, 5, 27), 16).execute();

  // Find a row in the SQL Table
  var myResult = myTable.select(['_id', 'name', 'birthday']).
    where('name like :name AND age < :age').
    bind('name', 'S%').bind('age', 20).execute(function (row) {
      console.log(row);
  });
```

C# Code

```
{
  // Working with Relational Tables

  // Connect to server using a connection
  var db = MySQLX.GetSession("server=localhost;port=33060;user=mike;password=s3cr3t!;")
 .GetSchema("test");

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

  // Insert SQL Table data
  myTable.Insert("name", "age")
 .Values("Sakila", "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 = "S%", age = 20 }).Execute();

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

Java Code

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

// Connect to server using a connection URL
XSession mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password=s3c
Schema db = mySession.getSchema("test");

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

```
// Insert SQL Table data
myTable.insert("name", "birthday").values("Sakila", "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", "S*").bind("age", 20).execute();

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

C++ Code

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

// Connect to server using a connection URL
XSession mySession(33060, "mike", "s3cr3t!");

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("Sakila", "2000-5-27", 16).execute();

// Find a row in the SQL Table
RowResult myResult = myTable.select("_id", "name", "birthday")
  .where("name like :name AND age < :age")
  .bind("name", "S%").bind("age", 20).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, 'Mike').
        values(2, 'Jack').
        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, 'Mike').values(2, 'Jack').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, 'Mike').
        values(2, 'Jack').
        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, "Mike")
 .Values(2, "Jack")
 .Execute();
}
```

Java Code

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

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

C++ Code

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

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

Figure 6.1 Table.insert() Syntax Diagram

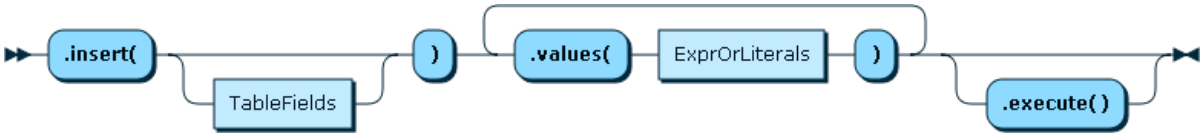


Table.select()

`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

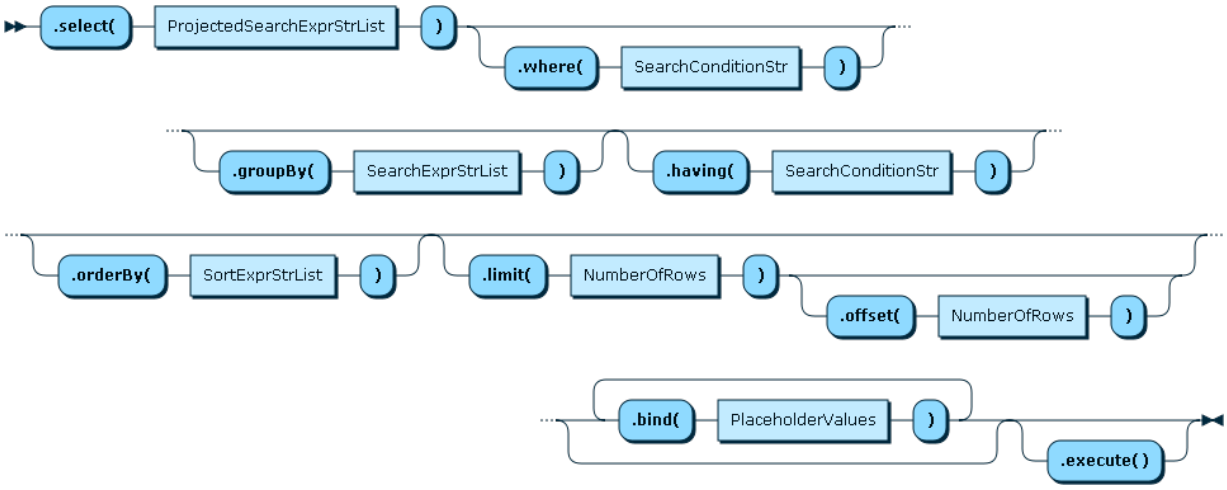


Table.update()

Figure 6.3 Table.update() Syntax Diagram

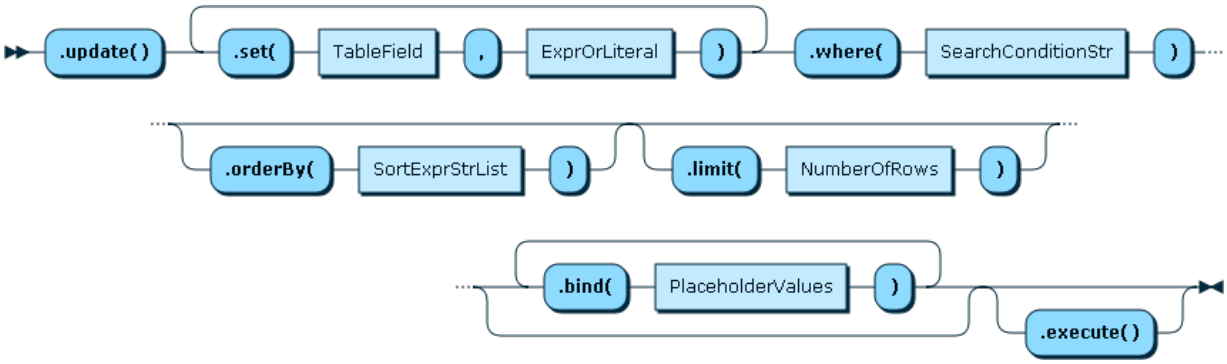
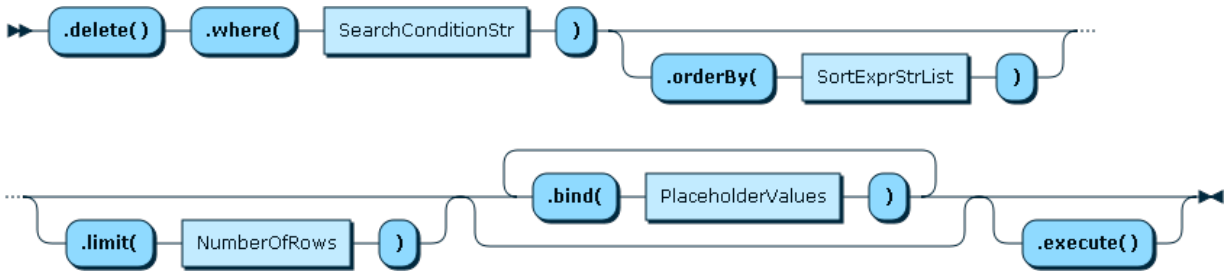


Table.delete()

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 examples show 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": "mike", "last_name": "connor"}').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": "mike", "last_name": "connor"}').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": "mike"}').execute();
```

C# Code

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

Java Code

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

C++ Code

```
// Get the customers collection as a table
Table customers = db.getCollectionAsTable("customers");
customers.insert("doc")
         .values(R"({"_id":"001", "name": "mike", "last_name": "connor"})").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.getNodeSession( {
  host: 'localhost', port: 33060,
  dbUser: 'mike', dbPassword: 's3cr3t!' } );

// 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: 'Jack', age: 15, height: 1.76, weight: 69.4}).execute();
  myColl.add({name: 'Susanne', age: 24, height: 1.65}).execute();
  myColl.add({name: 'Mike', age: 39, height: 1.9, weight: 74.3}).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

```
import mysqlx

# Connect to server
session = mysqlx.get_node_session( {
        'host': 'localhost', 'port': 33060,
```

```
        'dbUser': 'mike', 'dbPassword': 's3cr3t!' } )

# Get the Schema test
db = session.get_schema('test')

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

# Start a transaction
session.start_transaction()
try:
        myColl.add({'name': 'Jack', 'age': 15, 'height': 1.76, 'weight': 69.4}).execute()
        myColl.add({'name': 'Susanne', 'age': 24, 'height': 1.65}).execute()
        myColl.add({'name': 'Mike', 'age': 39, 'height': 1.9, 'weight': 74.3}).execute()

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

        print 'Data inserted successfully.'
except Exception, err:
        # Rollback the transaction in case of an error
        session.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=mike;password=s3cr3t!;");

  // 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 = "Jack", age = 15, height = 1.76, weight = 69.4}).Execute();
 myColl.Add(new { name = "Susanne", age = 24, height = 1.65}).Execute();
 myColl.Add(new { name = "Mike", age = 39, height = 1.9, weight = 74.3}).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);
  }
}
```

Java Code

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

// Connect to server
XSession mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password=s3c
```

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

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

// Start a transaction
this.session.startTransaction();
try {
    myColl.add("{\"name\":\"Jack\", \"age\":15}", "{\"name\":\"Susanne\", \"age\":24}", "{\"name\":\"Mi

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

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

C++ Code

```
// Connect to server
XSession session(SessionSettings::HOST, "localhost",
                 SessionSettings::PORT, 33060,
                 SessionSettings::USER, "mike",
                 SessionSettings::PWD, "s3cr3t!");

// 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": "Jack", "age": 15, "height": 1.76, "weight": 69.4})").execute();
  myColl.add(R"({"name": "Susanne", "age": 24, "height": 1.65})").execute();
  myColl.add(R"({"name": "Mike", "age": 39, "height": 1.9, "weight": 74.3})").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.getNodeSession( {
```

```
        host: 'localhost', port: 33060,
        dbUser: 'mike', dbPassword: 's3cr3t!' } );

// 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': 'Jack', 'age': 15, 'height': 1.76, 'weight': 69.4}).execute();
    myColl.add({'name': 'Susanne', 'age': 24, 'height': 1.65}).execute();
    myColl.add({'name': 'Mike', 'age': 39, 'height': 1.9, 'weight': 74.3}).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

```
import mysqlx

# Connect to server
mySession = mysqlx.get_node_session( {
        'host': 'localhost', 'port': 33060,
        'dbUser': 'mike', 'dbPassword': 's3cr3t!' } )

# 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': 'Jack', 'age': 15, 'height': 1.76, 'weight': 69.4}).execute()
    myColl.add({'name': 'Susanne', 'age': 24, 'height': 1.65}).execute()
    myColl.add({'name': 'Mike', 'age': 39, 'height': 1.9, 'weight': 74.3}).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, 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=mike;password=s3cr3t!;");

  // 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 = "Jack", age = 15, height = 1.76, weight = 69.4}).Execute();
 warningCount += result.Warnings.Count;
 result = myColl.Add(new { name = "Susanne", age = 24, height = 1.65}).Execute();
 warningCount += result.Warnings.Count;
 result = myColl.Add(new { name = "Mike", age = 39, height = 1.9, weight = 74.3}).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);
  }
}
```

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.getNodeSession( {
  host: 'localhost', port: 33060,
  dbUser: 'mike', dbPassword: 's3cr3t!' } );

// 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

```
import mysqlx

def process_warnings(result):
  if result.get_warning_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_node_session( {
  'host': 'localhost', 'port': 33060,
  'dbUser': 'mike', 'dbPassword': 's3cr3t!' } );

# 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)
```

Node.js JavaScript Code

C# Code

```
{
  // Assumptions: test schema exists, my_collection exists

  XSession mySession = null;
  try
  {
 // Connect to server on localhost
 mySession = MySQLX.GetSession("server=localhost;port=33060;user=mike;password=s3cr3t!;");
  }
  catch (Exception err)
  {
 Console.WriteLine("The database session could not be opened: " + err.Message);
 throw;
  }

  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", "S%").Execute();

 // Print document
 PrintResult(myDoc.FetchOne());
  }
  catch (Exception err)
  {
 Console.WriteLine("The following error occurred: " + err.Message);
  }
  finally
  {
 // Close the session in any case
 mySession.Close();
  }
}
```

Java Code

C++ Code

```
// setFetchWarnings() not yet implemented in Connector/C++
```

8.2 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 examples show how to perform proper error handling for the various languages. 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.getNodeSession( {
    host: 'localhost', port: 33060,
    dbUser: 'mike', dbPassword: 's3cr3t!' } );
}
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','S%').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

```
import mysqlx

# Connect to server
mySession = mysqlx.get_node_session( {
        'host': 'localhost', 'port': 33060,
        'dbUser': 'mike', 'dbPassword': 's3cr3t!' } )

# 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': 'Jack', 'age': 15, 'height': 1.76, 'weight': 69.4}).execute()
    myColl.add({'name': 'Susanne', 'age': 24, 'height': 1.65}).execute()
    myColl.add({'name': 'Mike', 'age': 39, 'height': 1.9, 'weight': 74.3}).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, 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)
```

Node.js JavaScript Code

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

// Connect to server on localhost
mysqlx.getSession( {
    host: 'localhost', port: 33060,
    dbUser: 'mike', dbPassword: 's3cr3t!'
}).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
  myColl.find('name like :param').limit(1)
  .bind('param','S%').execute(function (row) {
    console.log(row);
  }).catch(function (err) {
    print('The following error occurred: ' + err.message);
  });

  session.close();
}).catch (err) {
  console.log('The database session could not be opened: ' + err.message);
});
```

C# Code

```

```

Java Code

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

XSession mySession;

try {
    // Connect to server on localhost
    mySession = new XSessionFactory().getSession("mysqlx://localhost:33060/test?user=mike&password=s3cr

    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", "S%").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.h>

try
{
  // Connect to server on localhost
  XSession session(33060, "mike", "s3cr3t!");

  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", "S%").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);
}
```

Chapter 9 Working with Result Sets

Table of Contents

This section explains how to work with the results of processing.

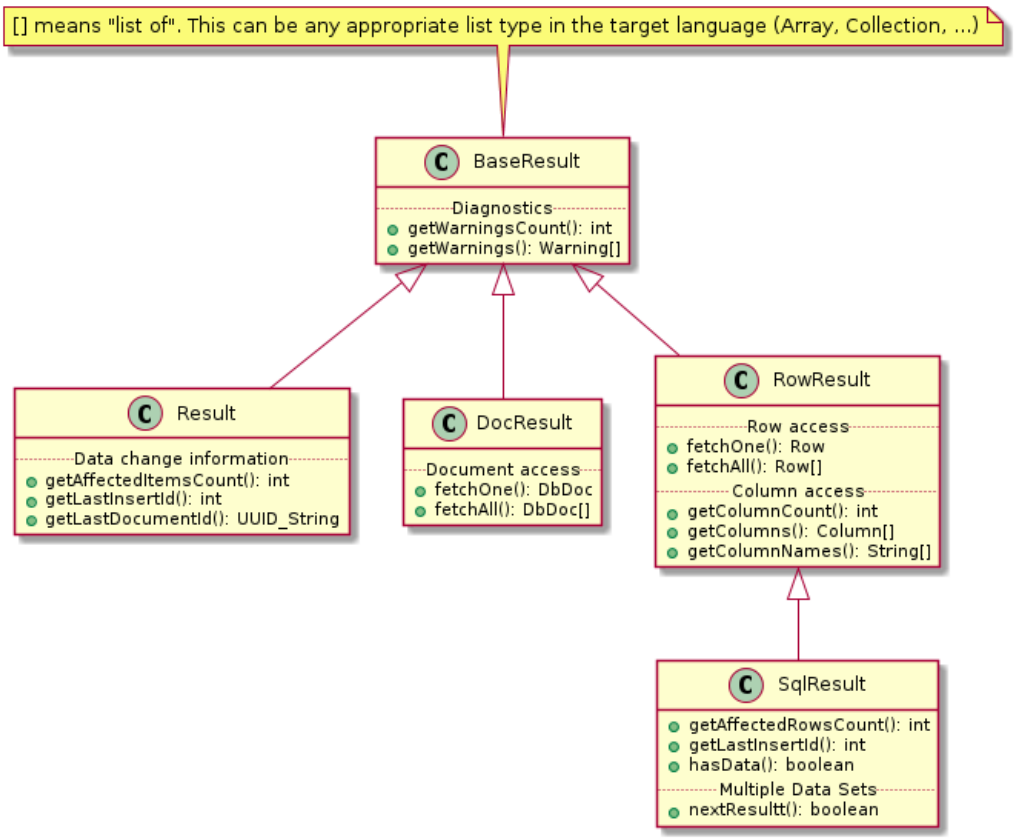
9.1 Result Set Classes

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 Document IDs

Every document has a unique identifier called the document ID, which can be thought of as the equivalent of a tables' primary key. The document ID value can be manually assigned when adding a document. If no value is assigned, a document ID is generated and assigned to the document automatically. Without knowledge of the generated IDs you cannot reliably fetch and update any of the previously inserted documents. The following methods enable you to access the document ID value from the return value of `collection.add()`:

- `getDocumentId()`
- `getDocumentIds()`

Note the difference in the plural. `getDocumentId()` is used to get the unique identifier value when a single document is added. For example assuming that a test schema is assigned to the variable db and the collection my_collection exists:

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

// Insert a document
var res = myColl.add({ name: 'Jack', age: 15, height: 1.76 }).execute();

// Print the document ID assigned to the document
print('Document Id:', res.getDocumentId());
```

`getDocumentIds()` returns a list of all document IDs for documents added. For example:

```
var res = collection.add({ _id: 1, name: 'Jack'}).add({ _id: 223, name: 'Jim'}).execute();
```

```
ids = res.getDocumentIds());
ids.forEach(function(id) { print(id); });

// prints 1
// prints 223
```

These two methods are necessary because X DevAPI supports chaining of `collection.add()` and `table.insert()` calls as one command. For example:

```
coll.add({name: 'Jack'}).add({age: 13}).execute();
```

When the above code is run two documents are added to the collection, which creates two document IDs implicitly. You can not execute `getDocumentID()` in this case, because multiple document IDs are returned. To access all the resulting document IDs from a chain of calls, use `getDocumentIds()`. To get the document ID of the most recently added document, use `getDocumentId()`.

To retrieve the document IDs of certain documents that have been added, use `getDocumentIds()` and address the specific document IDs. For example:

```
coll.add({name: 'Jack'}).add({age: 13}).execute();
ids = res.getDocumentIds());
// [0] - first, [1] - second and last, ...
print(ids[1]);
```

9.3 Working with AUTO-INCREMENT Values

A common MySQL task is to use `AUTO_INCREMENT` columns, for example generating primary key values. 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 following methods to return `AUTO_INCREMENT` column values from the return value of `table.insert()`:

- `getFirstAutoIncrementValue()`
- `getAutoIncrementValues()`

In the following examples it is assumed that the table contains a column for which the `AUTO_INCREMENT` attribute is set. Furthermore it is assumed that all insertions succeed. The `getFirstAutoIncrementValue()` function is used when adding rows individually, or in other words when not chaining `table.insert()` calls. For example:

```
res = tab.insert(['name']).values('Sakila'}.execute();
print(res.getFirstAutoIncrementValue());
```

When you chain multiple `table.insert()` calls, there are potentially multiple `AUTO_INCREMENT` values returned. The `getAutoIncrementValues()` function returns a list of all `AUTO_INCREMENT` values generated when inserting multiple rows:

```
res = tab.insert(['name']).values('Sakila').values('Otto').execute();
print(res.getAutoIncrementValues());

// prints a list of values for 'Sakila' and 'Otto'
```

Note that `AUTO_INCREMENT` columns may be used for generating primary key or `id` values but are not limited to them.

9.4 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 `NodeSession.sql()`. All three methods return data sets which encapsulate data items. `Collection.find()` returns a data set with documents and `Table.select()` respectively `NodeSession.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 usning `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. Consult your language's Connector reference for more details, see Additional Documentation.

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','S%').
        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','S%').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", "S%")
  .Execute();

DbDoc doc;
while ((doc = res.FetchOne()) != null)
{
  Console.WriteLine(doc);
}
```

Java Code

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

DocResult res = myColl.find("name like :name").bind("name", "S%")
  .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", "S%").execute();
DbDoc doc;
while ((doc = res.fetchOne()))
{
  cout <<*doc <<endl;
}
```

When using Node.js results are returned to a callback function, which is passed to `execute()` in an asychronous manner whenever results from the server arrive.

Node.js JavaScript Code

```
myColl.find('name like :name').bind('S%').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 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','S%').
        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','S%').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','S%').
 execute(function (row) {

  // Accessing the fields by array
  console.log('Name: ' + row['name']);

  // Accessing the fields by dynamic attribute
  console.log(' Age: ' + row.age);
});
```

C# Code

```
var myRows = myTable.Select("name", "age")
  .Where("name like :name").Bind("name", "S%")
  .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"]);
}
```

Java Code

```
RowResult myRows = myTable.select("name, age")
  .where("name like :name").bind("name", "S%")
  .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", "S%")
                          .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 young = row[age].get<int>() < 18;
  // Alternative formulation
  bool young = (int)row[age] < 18;
```

```
}
```

9.5 Fetching All Data Items at Once

Data sets feature two iteration patterns available with all Connectors. The first pattern using `fetchOne()` enables applications to consume data items one by one. The second pattern using `fetchAll()` passes all data items of a data set as a list to the application. Drivers use appropriate data types of their programming language for the list. Because different data types are used, the language's native constructs are supported to access the list elements. Consult your language's Connector reference for more details, see Additional Documentation. The 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','S%').
  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','S%') \
  .execute()

myRows = myResult.fetch_all()

for row in myRows:
  print "%s is %s years old." % (row.name, row.age)
```

C# Code

```
var myRows = myTable.Select("name", "age")
  .Where("name like :name").Bind("name", "S%")
  .Execute();
var rows = myRows.FetchAll();
```

Java Code

```
RowResult myRows = myTable.select("name, age")
  .where("name like :name").bind("name", "S%")
  .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", "S%")
                          .execute();

std::list<Row> rows = myRows.fetchAll();
for (Row row : rows)
```

```
{
  cout << row[1] << endl;
}

// Directly iterate over rows, without stroing 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.

9.6 Working with SQL Result Sets

When executing an SQL operation on a NodeSession with `NodeSession.sql()` an SqlResult is returned.

Result iteration is identical to working with results from CRUD operations. The example assumes that the users table exists.

MySQL Shell JavaScript Code

```
var res = nodeSession.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 = nodeSession.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

```
var res = nodeSession.sql('SELECT name, age FROM users').execute(function (row) {
  console.log('Name: ' + row['name']);
  console.log(' Age: ' + row.age);
});
```

C# Code

```
var res = nodeSession.SQL("SELECT name, age FROM users").Execute();

while (res.Next())
{
  Console.WriteLine("Name: " + res.Current["name"]);
  Console.WriteLine("Age: " + res.Current["age"]);
}
```

Java Code

```
SqlResult res = nodeSession.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 = nodeSession.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 may be empty. It is empty if `fetchOne()` returns NULL or `fetchAll()` returns an empty list. The example assumes that the procedure `my_proc` exists.

MySQL Shell JavaScript Code

```
var res = nodeSession.sql('CALL my_proc()').execute();

if (res.hasData()){

  var row = res.fetchOne();
  if (row){
    print('List of row 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 = nodeSession.sql('CALL my_proc()').execute()

if res.has_data():

        row = res.fetch_one()
        if row:
                print 'List of row available for fetching.'
                while row:
                        print row
                        row = res.fetch_one()
        else:
                print 'Empty list of rows.'
else:
        print 'No row result.'
```

C# Code

```
var res = nodeSession.SQL("CALL my_proc()").Execute();

if (res.HasData)
{

  var row = res.FetchOne();
  if (row != null)
  {
    Console.WriteLine("List of row available for fetching.");
    do
    {
      PrintResult(row);
    } while ((row = res.FetchOne()) != null);
  }
  else
  {
    Console.WriteLine("Empty list of rows.");
  }
}
else
{
  Console.WriteLine("No row result.");
}
```

Java Code

```
SqlResult res = nodeSession.sql("CALL my_proc()").execute();

if (res.hasData()){

  Row row = res.fetchOne();
  if (row != null){
    print("List of row available for fetching.");
    do {
      System.out.println(row);
    } while ((row = res.fetchOne()) != null);
  }
  else{
    System.out.println("Empty list of rows.");
  }
}
else {
  System.out.println("No row result.");
}
```

C++ Code

```
SqlResult res = nodeSession.sql("CALL my_proc()").execute();

if (res.hasData())
{
  Row row = res.fetchOne();
  if (row)
  {
    cout << "List of row 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.getAffectedRowCount());
  }
}

print_result(nodeSession.sql('DELETE FROM users WHERE age > 40').execute());
print_result(nodeSession.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_row_count()


print_result(nodeSession.sql('DELETE FROM users WHERE age > 40').execute())
print_result(nodeSession.sql('SELECT * FROM users WHERE age = 40').execute())
```

C# Code

```
private void print_result(SqlResult res)
{
  if (res.HasData)
  {
    // SELECT
  }
  else
  {
    // INSERT, UPDATE, DELETE, ...
    Console.WriteLine("Rows affected: " + res.RecordsAffected);
  }
}

print_result(nodeSession.SQL("DELETE FROM users WHERE age > 40").Execute());
print_result(nodeSession.SQL("SELECT COUNT(*) AS oldies FROM users WHERE age = 40").Execute());
```

Java Code

```
private void print_result(SqlResult res) {
  if (res.hasData()) {
    // SELECT
  } else {
    // INSERT, UPDATE, DELETE, ...
    System.out.println("Rows affected: " + res.getAffectedItemsCount());
  }
}

print_result(nodeSession.sql("DELETE FROM users WHERE age > 40").execute());
print_result(nodeSession.sql("SELECT COUNT(*) AS oldies 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
    std::list<Column> columns = res.getColumns();
    Row record = res.fetchOne();

    while (record)
    {
      for (unsigned index=0; index < columns.size(); ++index)
      {
        cout << columns[index].getColumnName() << ": "
             << record[index] << endl;
      }

      // Get the next record
      record = res.fetchOne();
```

```
    }

  }
  else
  {
    // INSERT, UPDATE, DELETE, ...
    // Note: getAffectedRowCount() not yet implemented in Connector/C++.
    cout << "No rows in the result" << endl;
  }
}

print_result(nodeSession.sql("DELETE FROM users WHERE age > 40").execute());
print_result(nodeSession.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.getAffectedRowCount());
  }
}


var res = nodeSession.sql('CALL my_proc()').execute();

// Prints each returned result
var more = true;
while (more){
  print_result(res);

  more = res.nextDataSet();
}
```

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_row_count()

res = nodeSession.sql('CALL my_proc()').execute()

# Prints each returned result
more = True
while more:
  print_result(res)

  more = res.next_data_set()
```

C# Code

```
var res = nodeSession.SQL("CALL my_proc()").Execute();

if (res.HasData)
{
  do
  {
    Console.WriteLine("New resultset");
    while (res.Next())
    {
      Console.WriteLine(res.Current);
    }
  } while (res.NextResult());
}
```

Java Code

```
SqlResult res = nodeSession.sql("CALL my_proc()").execute();
```

C++ Code

```
SqlResult res = nodeSession.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 are returned to a callback, which has to be provided to the `execute()` method. To identify individual result sets you can provide a second callback, which will be called for meta data which marks the beginning of a result set.

Node.js JavaScript Code

```
var resultcount = 0;
var res = nodeSession.sql('CALL my_proc()').execute(function (
  function (row) {
    console.log("Row: ", row);
  }, function (meta) {
    resultcount++;
    cosole.log("Begin of result set number ", resultcount);
  }
);
```

The number of result sets is not know immediately after the query execution. Query results may be streamed to the client or buffered at the client. In the streaming or partial buffering mode a client cannot tell whether a query will emit more than one result set.

9.7 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
```

9.8 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 straight forward 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.

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();
```

All implementations can use the syntax illustrated above.

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().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().set("counter", "counter + 1").execute()
```

Chapter 11 CRUD EBNF Definitions

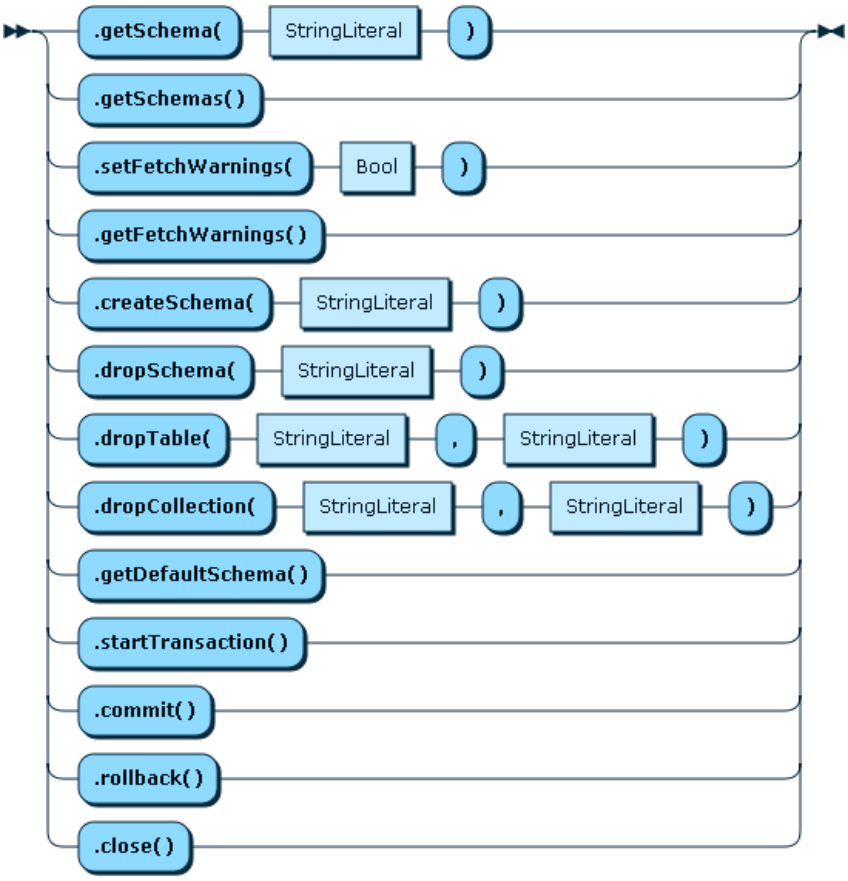
Table of Contents

This chapter provides a visual reference guide to the objects and functions available in the X DevAPI. For more information on the specific objects and functions available in the language you are using which implements the X DevAPI, see the Additional Documentation.

11.1 Session Objects and Functions

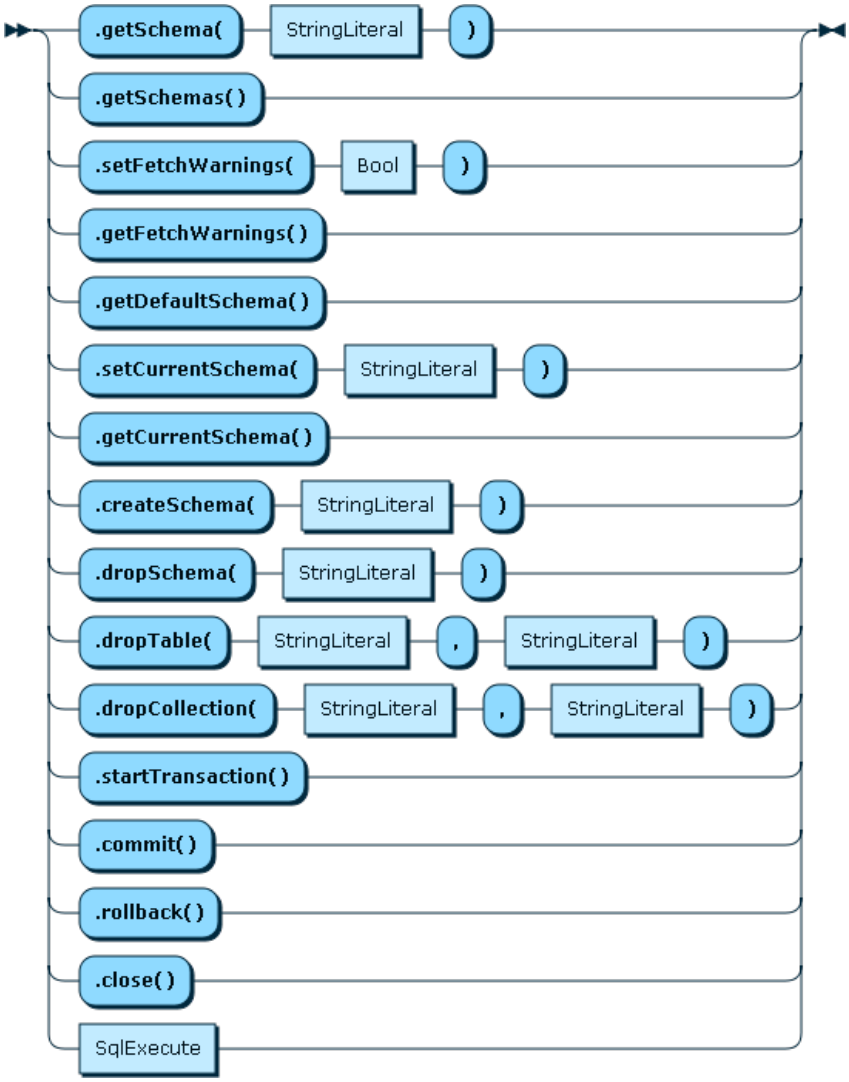
XSession

Figure 11.1 XSession



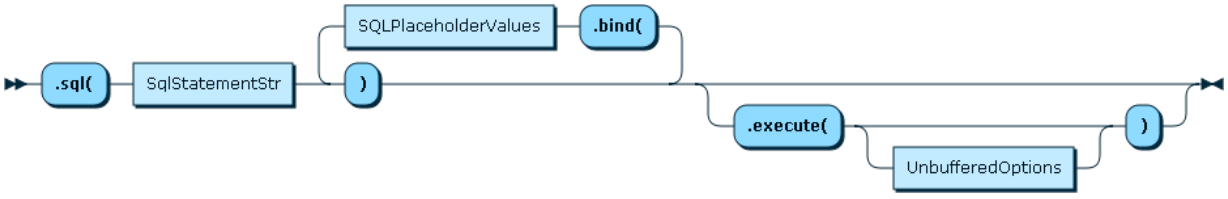
NodeSession

Figure 11.2 NodeSession



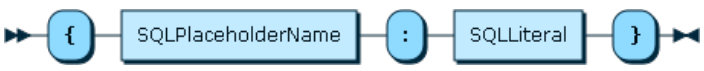
SqlExecute

Figure 11.3 SqlExecute



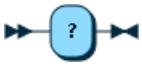
SQLPlaceholderValues

Figure 11.4 SQLPlaceholderValues



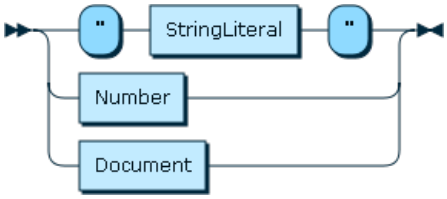
SQLPlaceholderName

Figure 11.5 SQLPlaceholderName



SQLLiteral

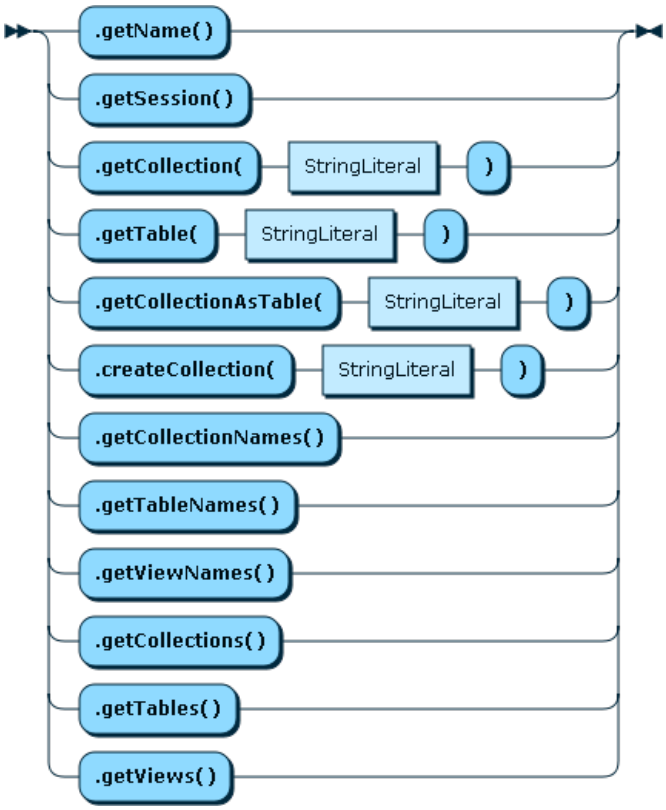
Figure 11.6 SQLLiteral



11.2 Schema Objects and Functions

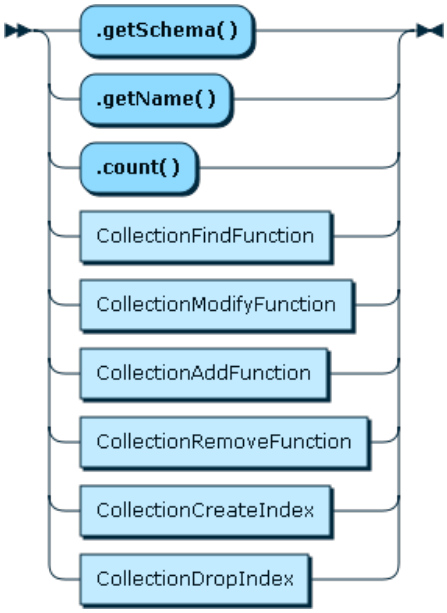
Schema

Figure 11.7 Schema



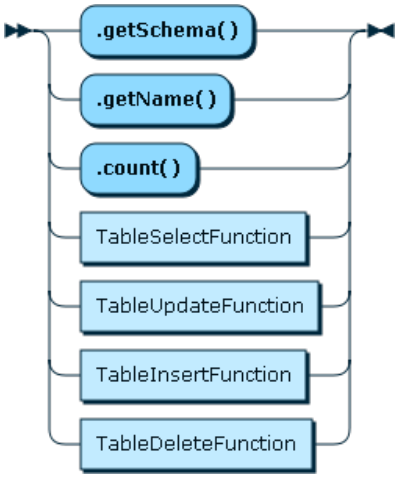
Collection

Figure 11.8 Collection



Table

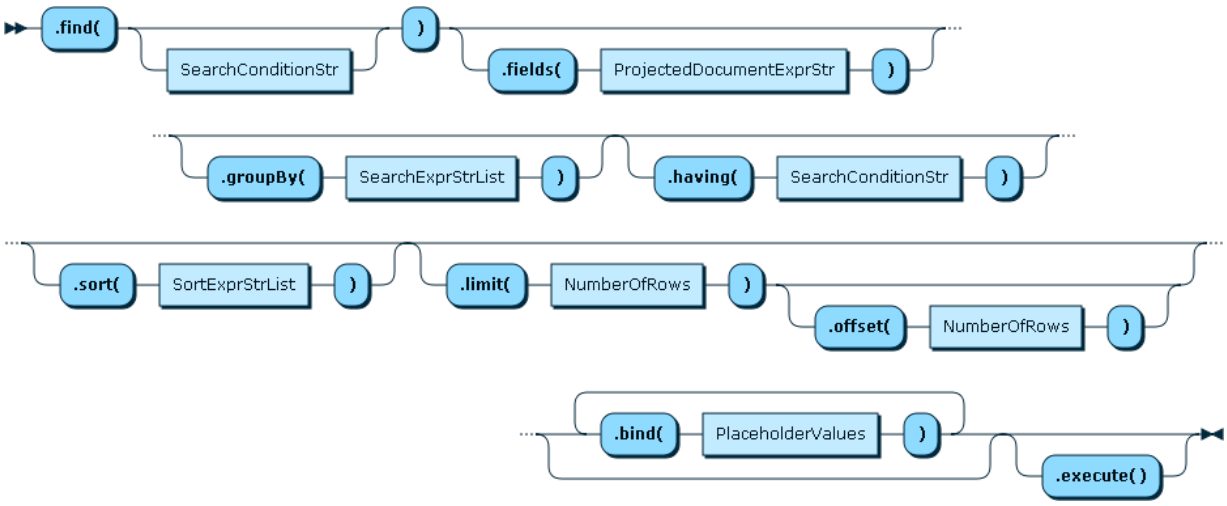
Figure 11.9 Table



11.3 Collection CRUD Functions

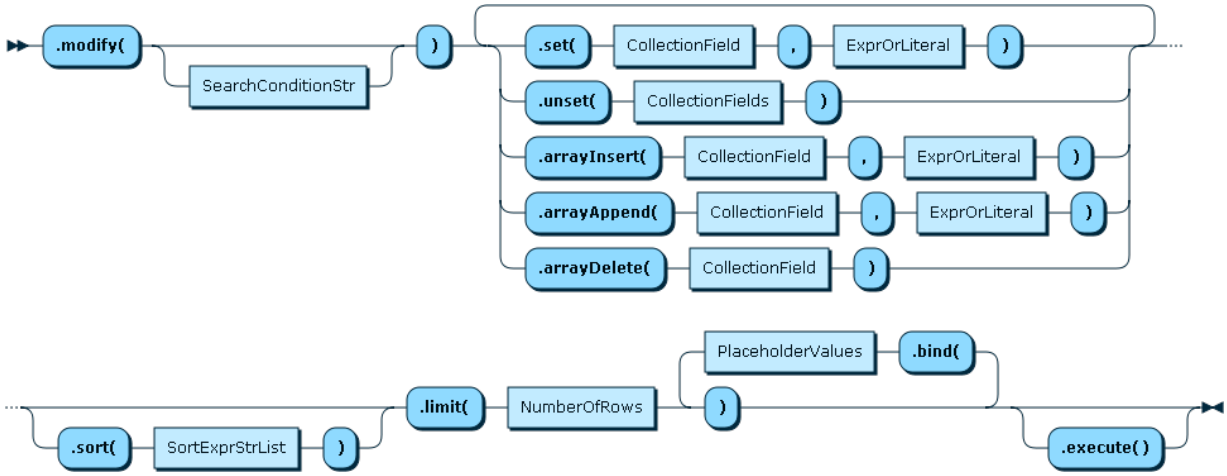
CollectionFindFunction

Figure 11.10 CollectionFindFunction



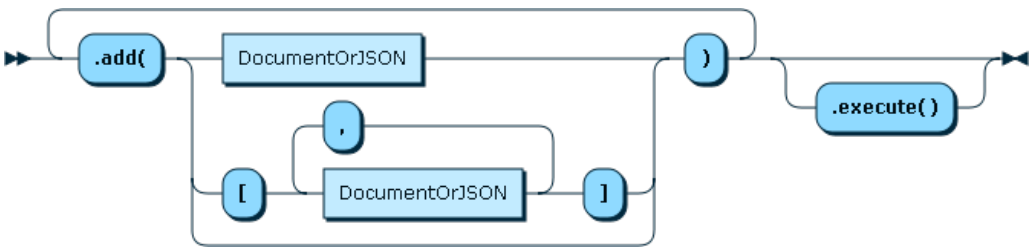
CollectionModifyFunction

Figure 11.11 CollectionModifyFunction



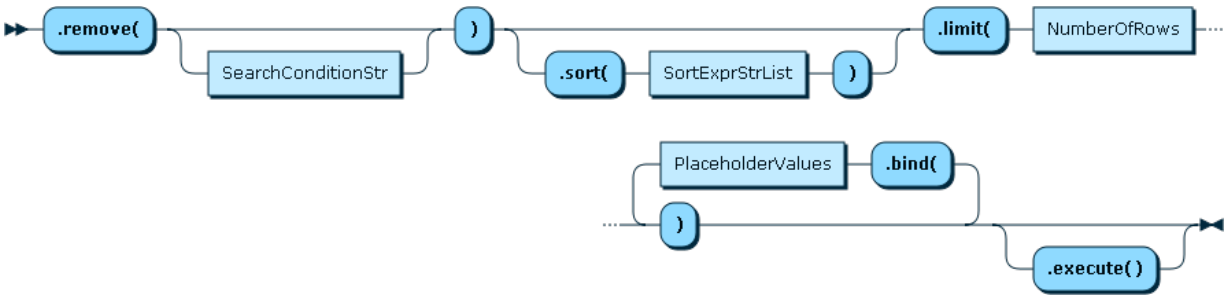
CollectionAddFunction

Figure 11.12 CollectionAddFunction



CollectionRemoveFunction

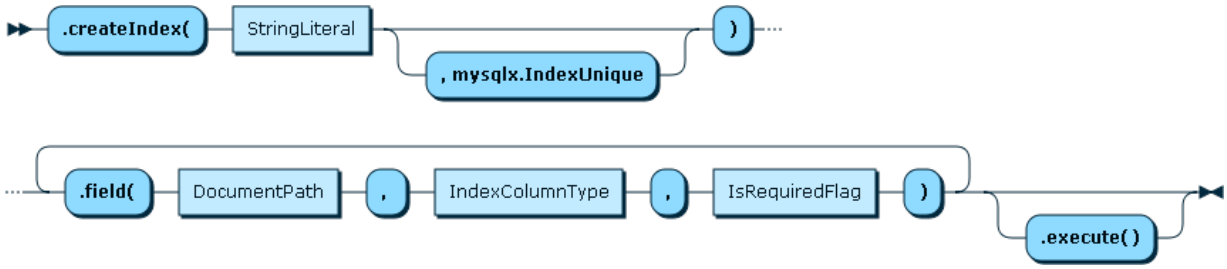
Figure 11.13 CollectionRemoveFunction



11.4 Collection Index Management Functions

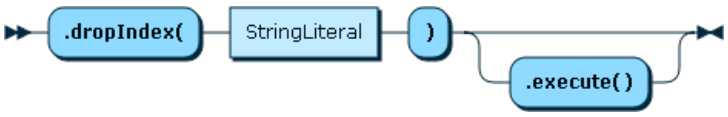
CollectionCreateIndex

Figure 11.14 CollectionCreateIndex



CollectionDropIndex

Figure 11.15 CollectionDropIndex

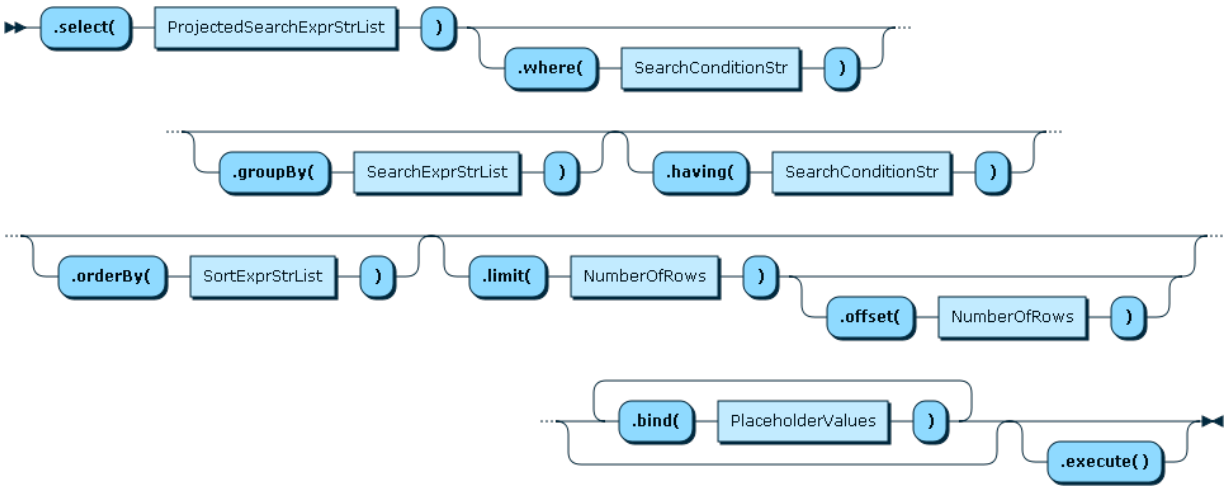


11.5 Table CRUD Functions

TableSelectFunction

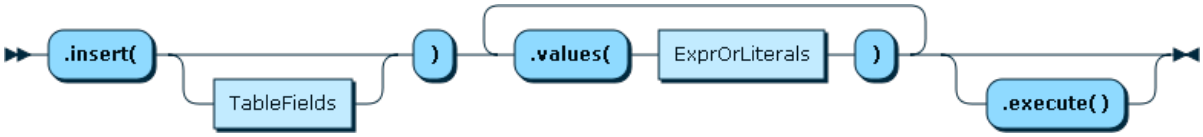
`Table.select()` and `collection.find()` use different methods for sorting results. `Table.select()` follows the SQL language naming and calls the sort method `orderBy()`. `Collection.find()` does not. Use the method `sort()` to sort the results returned by `Collection.find()`. Proximity with the SQL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

Figure 11.16 TableSelectFunction



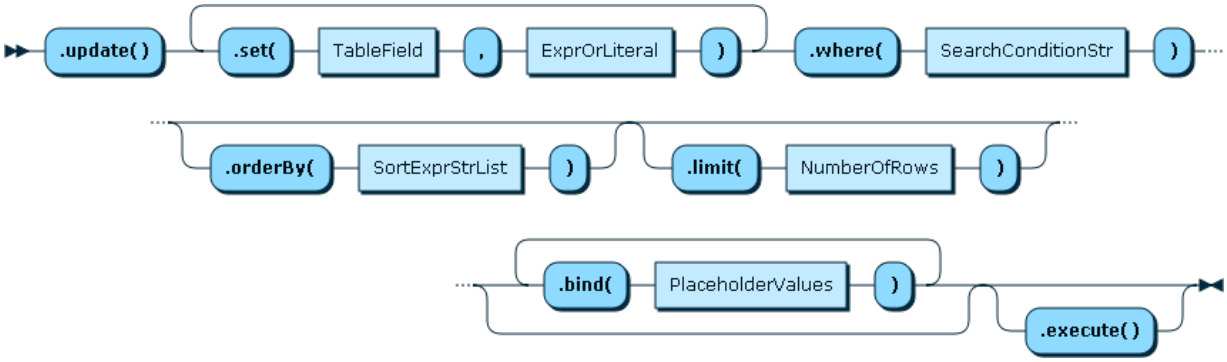
TableInsertFunction

Figure 11.17 TableInsertFunction



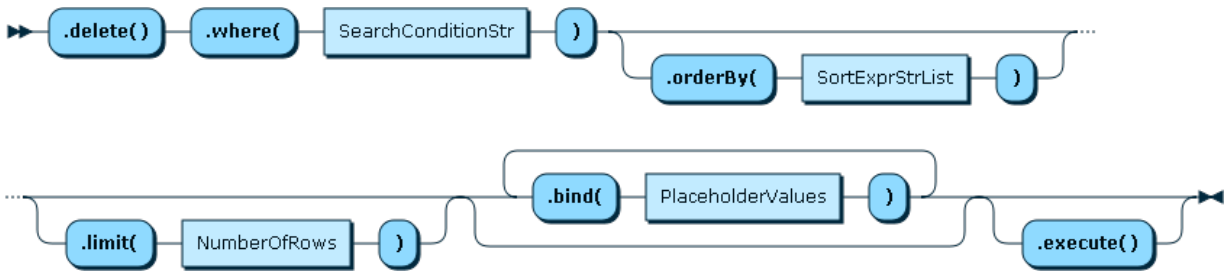
TableUpdateFunction

Figure 11.18 TableUpdateFunction



TableDeleteFunction

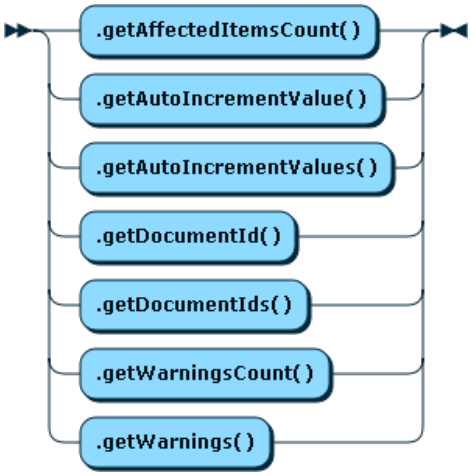
Figure 11.19 TableDeleteFunction



11.6 Result Functions

Result

Figure 11.20 Result

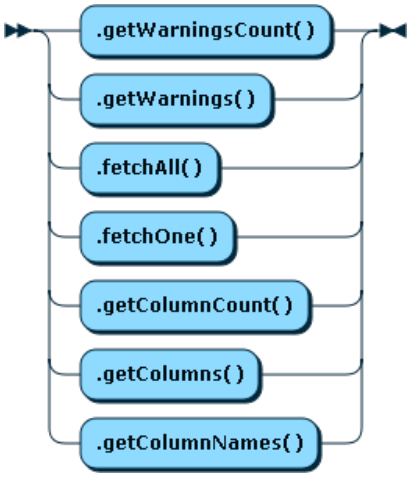


DocResult

Figure 11.21 DocResult

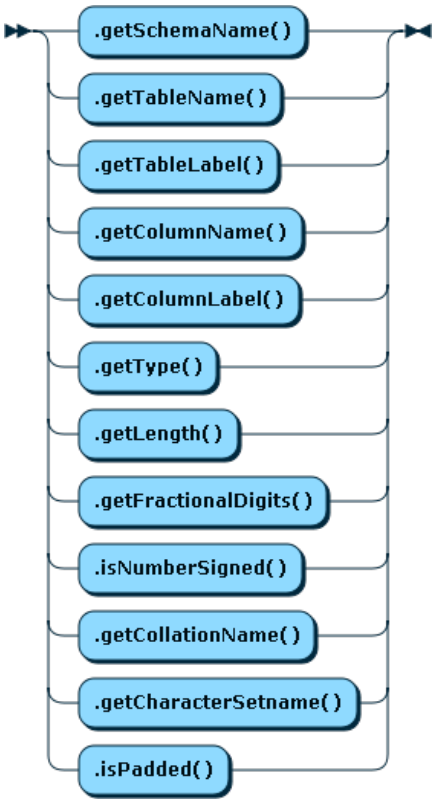
RowResult

Figure 11.22 RowResult



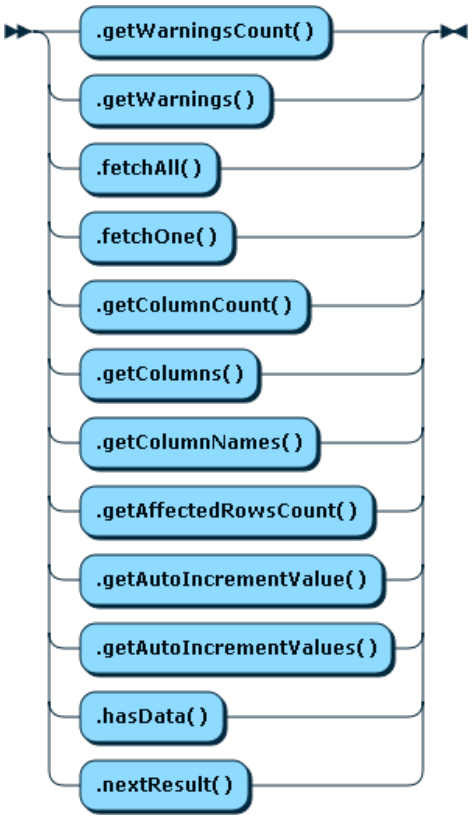
Column

Figure 11.23 Column



SqlResult

Figure 11.24 SqlResult



11.7 Other EBNF Definitions

SearchConditionStr

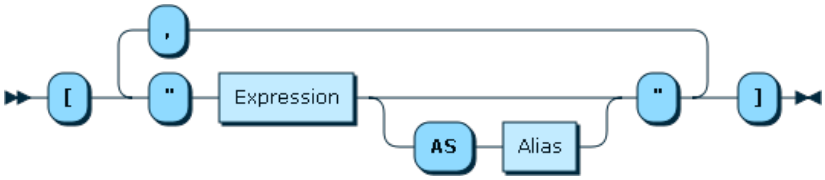
Figure 11.25 SearchConditionStr

SearchExprStrList

Figure 11.26 SearchExprStrList

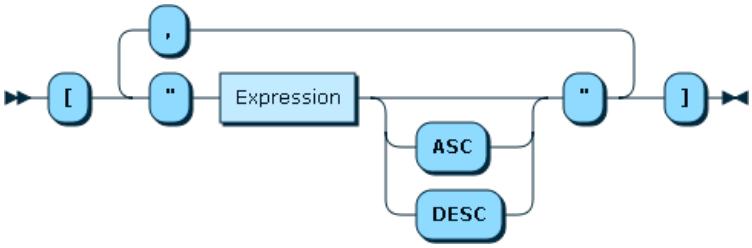
ProjectedSearchExprStrList

Figure 11.27 ProjectedSearchExprStrList



SortExprStrList

Figure 11.28 SortExprStrList

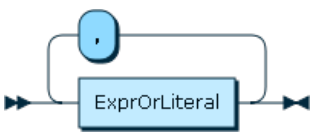


ExprOrLiteral

Figure 11.29 ExprOrLiteral

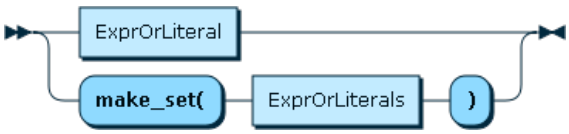
ExprOrLiterals

Figure 11.30 ExprOrLiterals



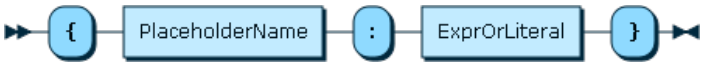
ExprOrLiteralOrOperand

Figure 11.31 ExprOrLiteralOrOperand



PlaceholderValues

Figure 11.32 PlaceholderValues



PlaceholderName

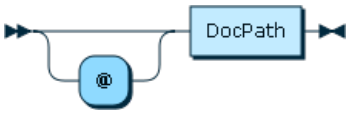
Figure 11.33 PlaceholderName

CollectionFields

Figure 11.34 CollectionFields

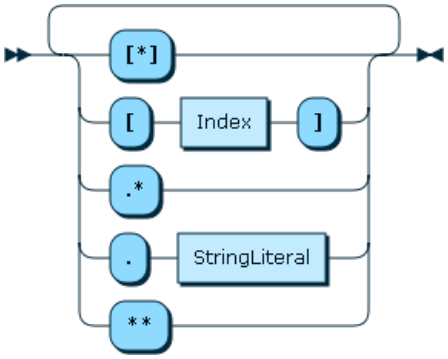
CollectionField

Figure 11.35 CollectionField



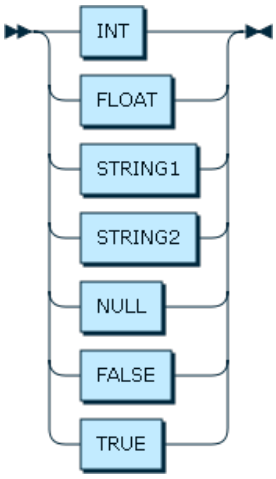
DocPath

Figure 11.36 DocPath



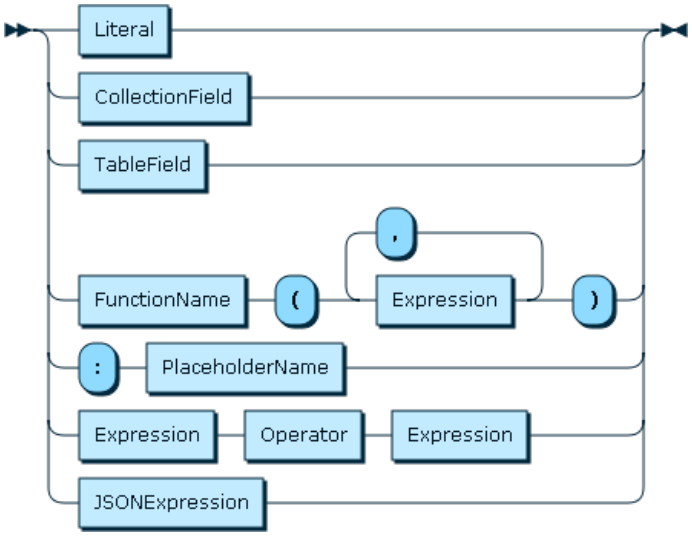
Literal

Figure 11.37 Literal



Expression

Figure 11.38 Expression



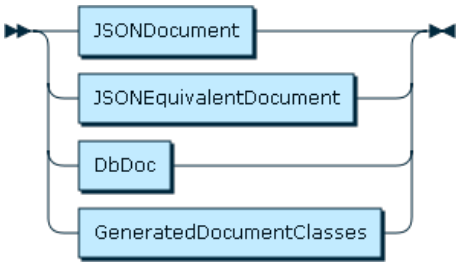
Document

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

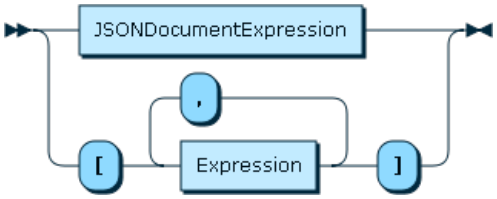
All implementations allow expressing an document by the special DbDoc type and as a string. Consult your language's Connector reference for more details, see Additional Documentation.

Figure 11.39 Document



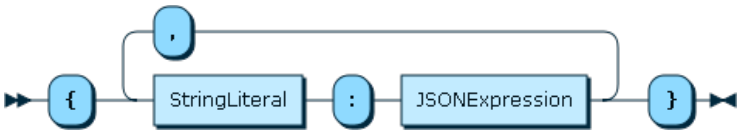
JSONExpression

Figure 11.40 JSONExpression



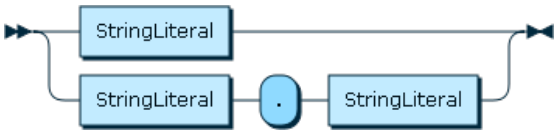
JSONDocumentExpression

Figure 11.41 JSONDocumentExpression



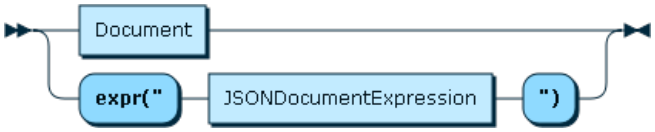
FunctionName

Figure 11.42 FunctionName



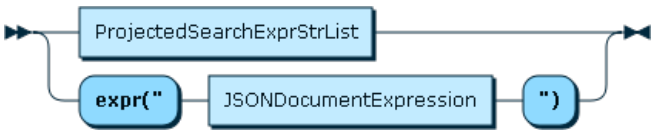
DocumentOrJSON

Figure 11.43 DocumentOrJSON



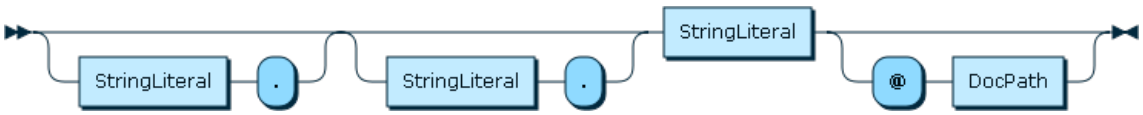
ProjectedDocumentExprStr

Figure 11.44 ProjectedDocumentExprStr



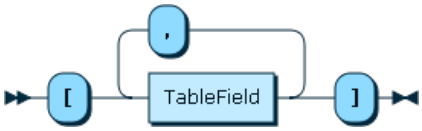
TableField

Figure 11.45 TableField



TableFields

Figure 11.46 TableFields

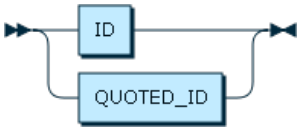


Chapter 12 Expressions EBNF Definitions

This section provides a visual reference guide to the grammar for the expression language used in the X DevAPI.

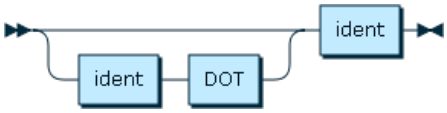
ident

Figure 12.1 ident



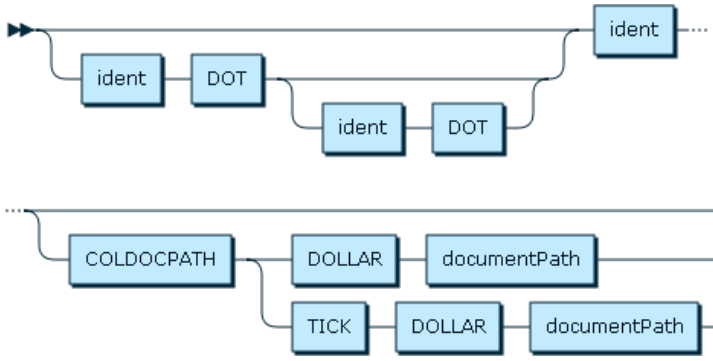
schemaQualifiedIdent

Figure 12.2 schemaQualifiedIdent



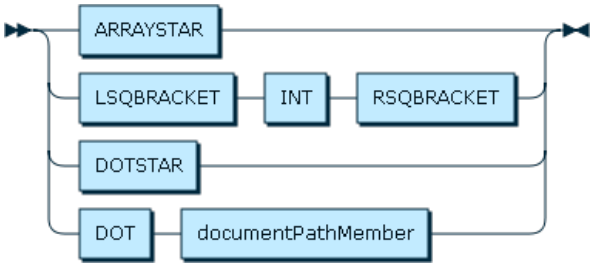
columnIdent

Figure 12.3 columnIdent



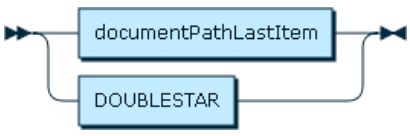
documentPathLastItem

Figure 12.4 documentPathLastItem



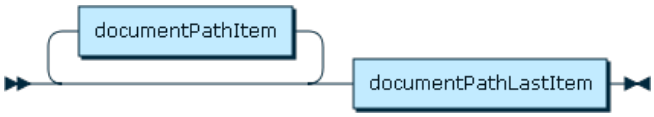
documentPathItem

Figure 12.5 documentPathItem



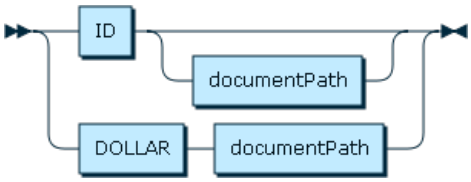
documentPath

Figure 12.6 documentPath



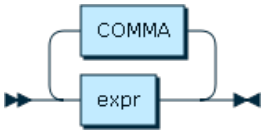
documentField

Figure 12.7 documentField



argsList

Figure 12.8 argsList

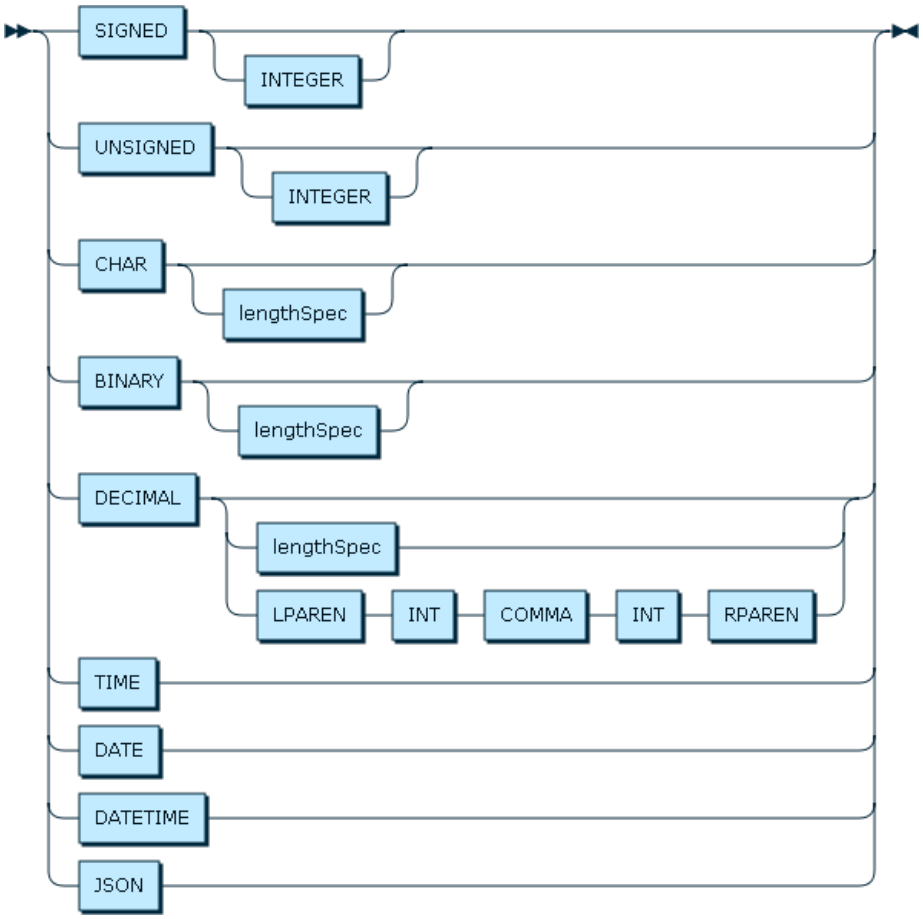


lengthSpec

Figure 12.9 lengthSpec

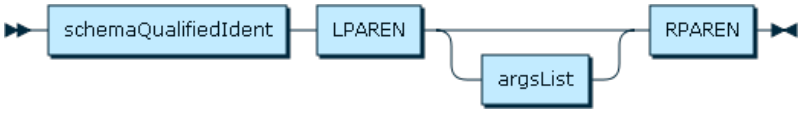
castType

Figure 12.10 castType



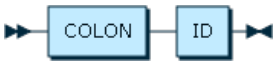
functionCall

Figure 12.11 functionCall



placeholder

Figure 12.12 placeholder

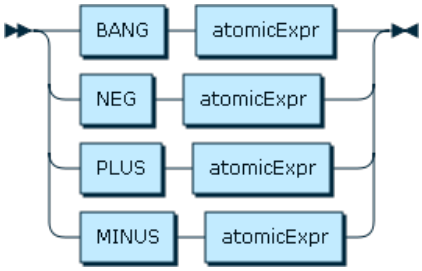


groupedExpr

Figure 12.13 groupedExpr

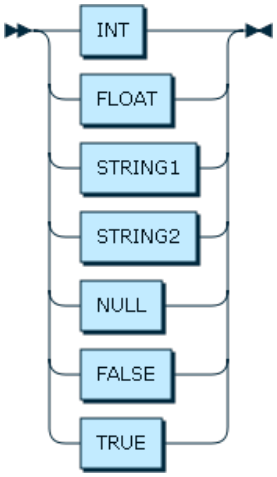
unaryOp

Figure 12.14 unaryOp



literal

Figure 12.15 literal

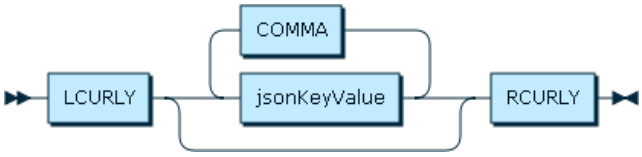


jsonKeyValue

Figure 12.16 jsonKeyValue

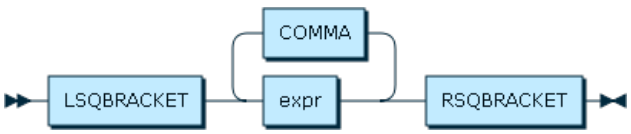
jsonDoc

Figure 12.17 jsonDoc



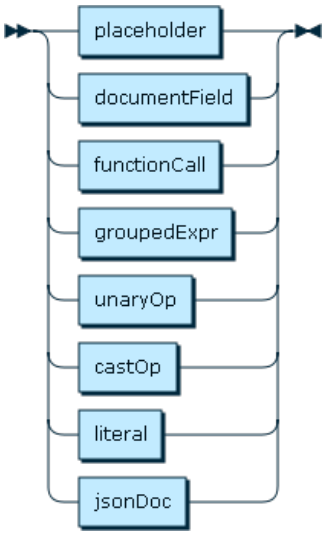
array

Figure 12.18 array



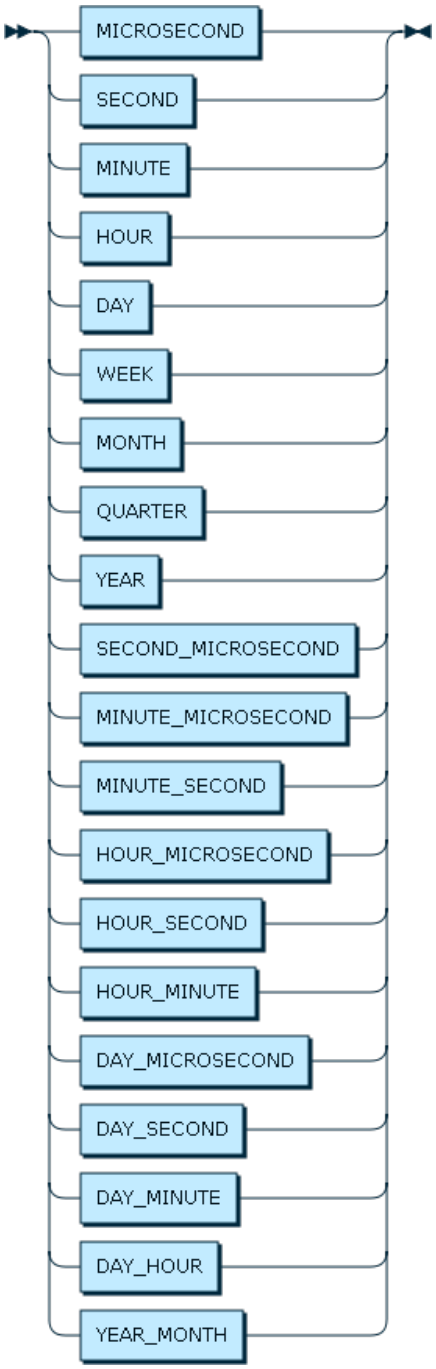
atomicExpr

Figure 12.19 atomicExpr



intervalUnit

Figure 12.20 intervalUnit

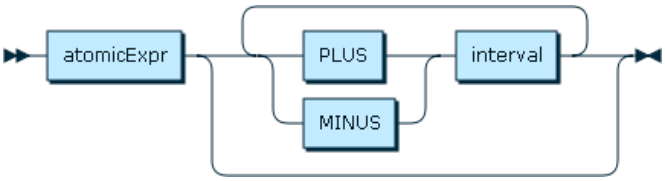


interval

Figure 12.21 interval

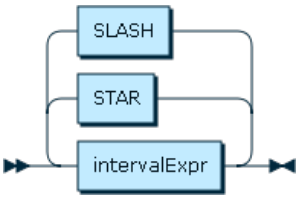
intervalExpr

Figure 12.22 intervalExpr



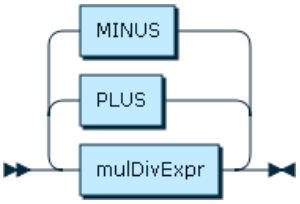
mulDivExpr

Figure 12.23 mulDivExpr



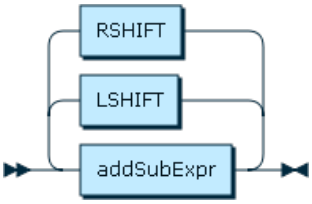
addSubExpr

Figure 12.24 addSubExpr



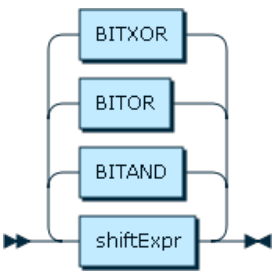
shiftExpr

Figure 12.25 shiftExpr



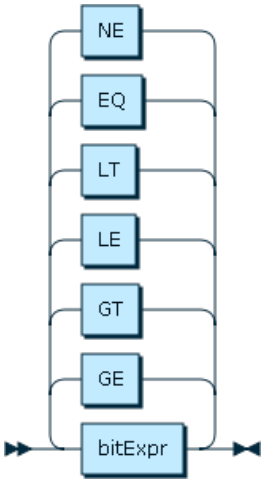
bitExpr

Figure 12.26 bitExpr



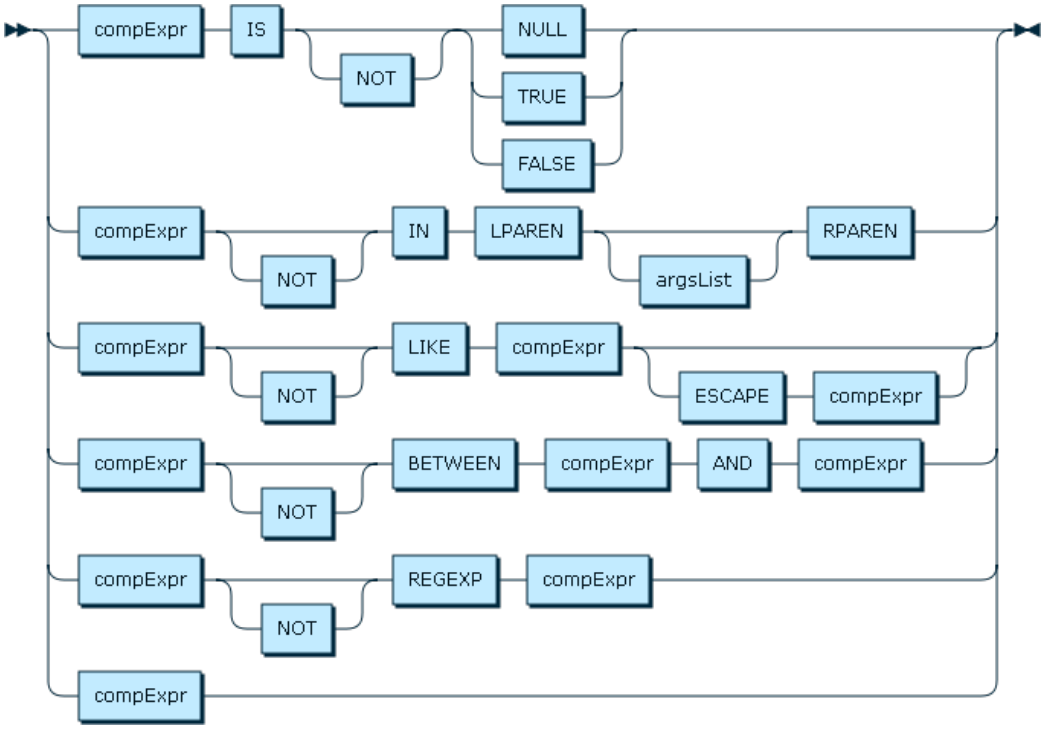
compExpr

Figure 12.27 compExpr



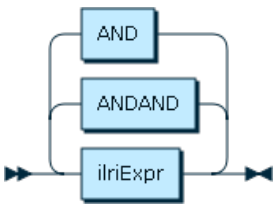
ilriExpr

Figure 12.28 ilriExpr



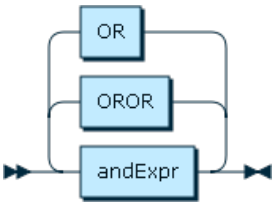
andExpr

Figure 12.29 andExpr



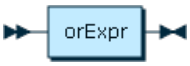
orExpr

Figure 12.30 orExpr



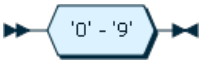
expr

Figure 12.31 expr



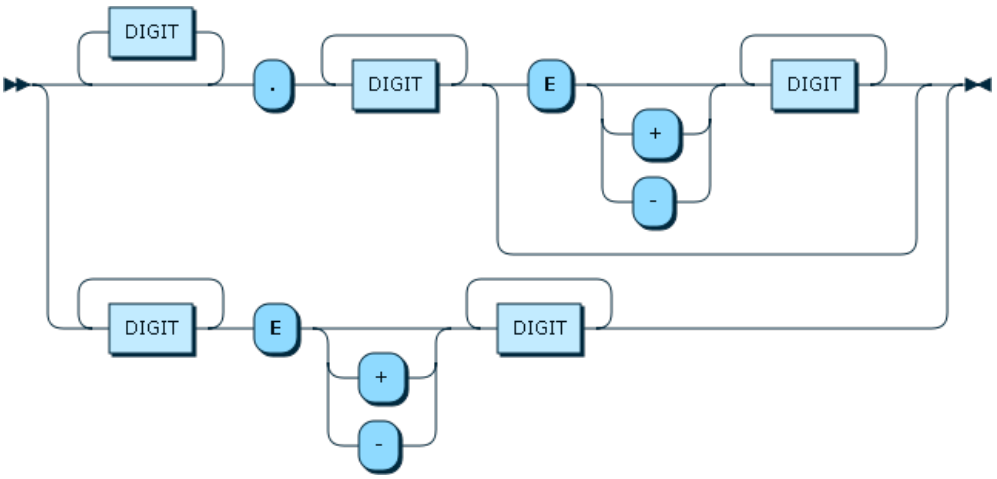
fragment DIGIT

Figure 12.32 fragment DIGIT



FLOAT

Figure 12.33 FLOAT

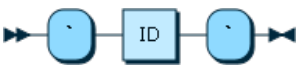


INT

Figure 12.34 INT

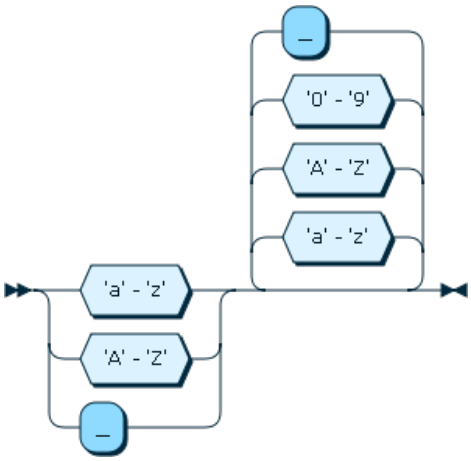
QUOTED_ID

Figure 12.35 QUOTED_ID



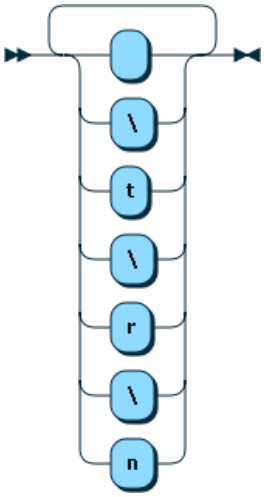
ID

Figure 12.36 ID

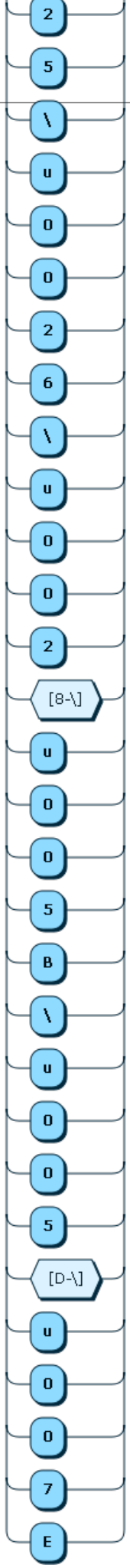


WS

Figure 12.37 WS

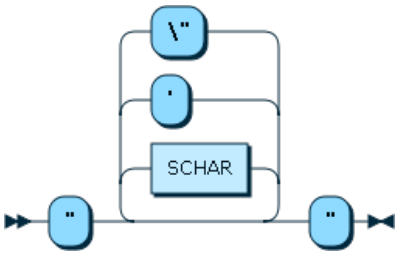


SCHAR



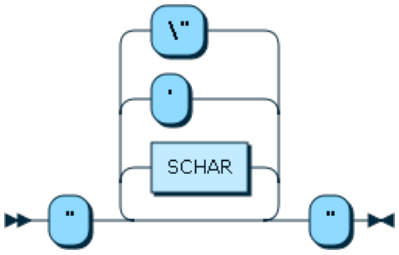
STRING1

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 the X DevAPI.

13.1 MySQL Connector Notes

Each driver implementation of the 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 concept 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.

Consult your language's Connector reference for more details, see Additional Documentation.

13.2 MySQL Shell X DevAPI extensions

MySQL Shell deviates from the Connector implementations in certain places. A Connector can connect to MySQL Servers running the X Plugin only by means of the X Protocol. MySQL Shell contains an extension of the X DevAPI to access MySQL Servers through the X Protocol. An additional ClassicSession class is available to establish a connection to a single MySQL node using the X Protocol. The functionality of the ClassicSession is limited to basic schema browsing and SQL execution.

See MySQL Shell User Guide, 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.