OQL - Object Query Language

The goal of this file is to help you get started with OQL. The examples presented in this file refer to classes defined in the file "O2 Tutorial".

Please note that the syntax in some of the examples might need minor adjustment before they will work with the current version of O2. If you find any errors, or places which are unclear, or if you have any suggestions or comments, please let us know (email to Michalis - mpetropo@cs.ucsd.edu). Your help is greatly appreciated.

Introduction

OQL is the way to access data in an O2 database. OQL is a powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

Using OQL

- Setup your environment
- SELECT, FROM, WHERE
- Dot notation and path expressions
- Subqueries in FROM clause
- Subqueries in WHERE clause
- Set operations and Aggregation
- GROUP BY
- Embedded OQL
- More documentation

Setup your environment

We've been able to create classes and write some programs. So far, O2 appears to be like an object-oriented programming language like C++ instead of a database system. Probably the main difference is that O2 supports queries. The queries that you'll be creating will look very similar to that of SQL.

In order to perform queries, you'll need to enter query mode. From within the O2 client, do the command:

- **query;**

This will put you in a sub-shell for queries. From here, you can enter your queries followed by Ctrl-D. To exit query mode, just hit Ctrl-D without entering a query.

SELECT, FROM, WHERE

SELECT <list of values>
FROM <list of collections and variable assignments>
WHERE <condition>

The SELECT clause extracts those elements of a collection meeting a specific condition. By using the keyword DISTINCT duplicated elements in the resulting collection get eliminated. Collections in FROM can be either extents (persistent names - sets) or expressions that evaluate to a collection (a set). Strings are enclosed in double-quotes in OQL. We can rename a field by if we prefix the path with the desired name and a colon.

Example Query 1
Give the names of people who are older than 26 years old:

SELECT SName: p.name
FROM p in People
WHERE p.age > 26
(hit Ctrl-D)

Dot Notation & Path Expressions

We use the dot notation and path expressions to access components of complex values.

Let variables t and ta range over objects in extents (persistent names) of Tutors and TAs (i.e., range over objects in sets Tutors and TAs).

ta.salary -> real
t.students -> set of tuples of type tuple(name: string, fee: real) representing students
t.salary -> real

Cascade of dots can be used if all names represent objects and not a collection.

Example of Illegal Use of Dot

t.students.name, where ta is a TA object.

This is illegal, because ta.students is a set of objects, not a single object.

Example Query 2
Find the names of the students of all tutors:

SELECT s.name
FROM Tutors t, t.students s

Here we notice that the variable t that binds to the first collection of FROM is used to help us define the second collection s. Because students is a

collection, we use it in the FROM list, like t.students above, if we want to access attributes of students.

Subqueries in FROM Clause

Example Query 3
Give the names of the Tutors which have a salary greater than $300 and have a student paying more than $30:

```
SELECT t.name
FROM ( SELECT t FROM Tutors t WHERE t.salary > 300 ) r, r.students s
WHERE s.fee > 30
```

Subqueries in WHERE Clause

Example Query 4
Give the names of people who aren't TAs:

```
SELECT p.name
FROM p in People
WHERE not ( p.name in SELECT t.name FROM t in TAs )
```

Set Operations and Aggregation

The standard O2C operators for sets are + (union), * (intersection), and – (difference). In OQL, the operators are written as UNION, INTERSECT and EXCEPT , respectively.

Example Query 5
Give the names of TAs with the highest salary:

```
SELECT t.name
FROM t in TAs
WHERE t.salary = max ( select ta.salary from ta in TAs )
```

GROUP BY

The GROUP BY operator creates a set of tuples with two fields. The first has the type of the specified GROUP BY attribute. The second field is the set of tuples that match that attribute. By default, the second field is called PARTITION.

Example Query 6
Give the names of the students and the average fee they pay their Tutors:

```
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)
FROM t in Tutors, t.students s
GROUP BY sname: s.name
```

1. Initial collection

We begin from collection Tutors, but technically it is a bag of tuples of the form:

tuple(t: t1, s: tuple(name: string, fee: real))

where t1 is a Tutor object and s denotes a student tuple. In general, there are fields for all of the variable bindings in the FROM clause.

2. Intermediate collection

The GROUP BY attribute s.name maps the tuples of the initial collection to the value of the name of the student. The intermediate collection is a set of tuples of type:

tuple(sname: string, partition: set(tuple(t: Tutor, s: tuple(name: string, fee: real))))

For example:

tuple(sname = "Mike", partition = set(tuple(t1, tuple("Mike", 10)), tuple(t2, tuple("Mike", 20))))

where t1,t2,... are all the tutors of student "Mike".

3. Output collection

Consists of student-average fee pairs, one for each tuple in the intermediate collection. The type of tuples in the output is:

tuple(sname: string, avgFee: real)

Note that in the subquery of the SELECT clause:

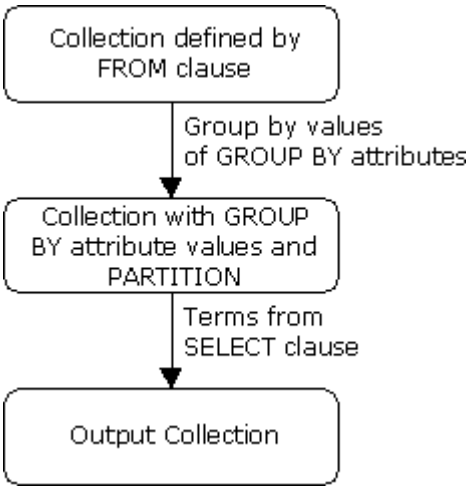
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)

We let p range over all tuples in partition. Each of these tuples contains a Tutor object and a student tuple. Thus, p.s.fee extracts the fee from one of the student tuples.

A typical output tuple looks like this:

tuple(sname = "Mike", avgFee = 15)

The whole procedure of GROUP BY operator's evaluation is summarized in the following figure:



Embedded OQL

Instead of using query mode, you can incorporate these queries in your O2 programs using the "o2query" command:

```
run body {
  o2 real total_salaries;
  o2query( total_salaries, "sum ( SELECT ta->get_salary \
  FROM ta in TAs )" );
  printf("TAs combined salary:  %.2f\n", total_salaries);
};
```

The first argument for o2query is the variable in which you want to store the query results. The second argument is a string that contains the query to be performed. If your query string takes up several lines, be sure to backslash (\) the carriage returns.

CAP Theorem and NoSQL Databases

What is the CAP theorem?

The CAP theorem is used to makes system designers aware of the trade-offs while designing networked shared-data systems. CAP theorem has influenced the design of many distributed data systems. It is very important to understand the CAP theorem as It makes the basics of choosing any NoSQL database based on the requirements.

CAP theorem states that in networked shared-data systems or distributed systems, we can only achieve at most two out of three guarantees for a database: Consistency, Availability and Partition Tolerance.

A distributed system is a network that stores data on more than one node (physical or virtual machines) at the same time.

Let’s first understand C, A, and P in simple words:

Consistency: means that all clients see the same data at the same time, no matter which node they connect to in a distributed system. To achieve consistency, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed successful.

Availability: means that every non-failing node returns a response for all read and write requests in a reasonable amount of time, even if one or more nodes are down. Another way to state this — all working nodes in the distributed system return a valid response for any request, without failing or exception.

Partition Tolerance: means that the system continues to operate despite arbitrary message loss or failure of part of the system. In other words, even if there is a network outage in the data center and some of the computers are unreachable, still the system continues to perform. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

The CAP theorem categorizes systems into three categories:

CP (Consistent and Partition Tolerant) database: A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node (i.e., make it unavailable) until the partition is resolved.

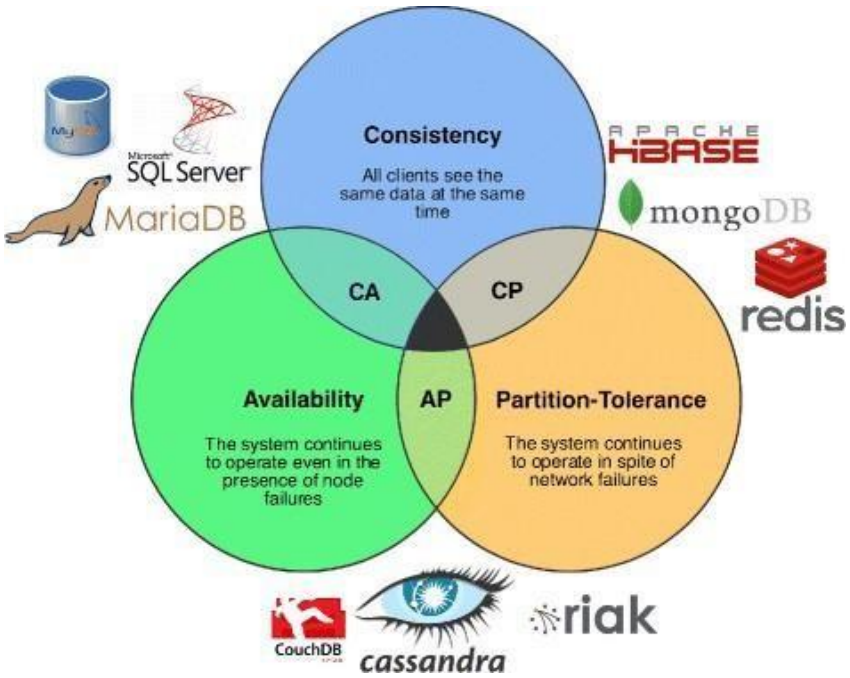
Partition *refers to a communication break between nodes within a distributed system. Meaning, if a node cannot receive any messages from another node in the system, there is a partition between the two nodes. Partition could have been because of network failure, server crash, or any other reason.*

AP (Available and Partition Tolerant) database: An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. When the partition is resolved, the AP databases typically resync the nodes to repair all inconsistencies in the system.

CA (Consistent and Available) database: A CA delivers consistency and availability in the absence of any network partition. Often a single node's DB servers are categorized as CA systems. Single node DB servers do not need to deal with partition tolerance and are thus considered CA systems.

In any networked shared-data systems or distributed systems partition tolerance is a must. Network partitions and dropped messages are a fact of life and must be handled appropriately. Consequently, system designers must choose between consistency and availability.

The following diagram shows the classification of different databases based on the CAP theorem.



System designers must take into consideration the CAP theorem while designing or choosing distributed storages as one needs to be sacrificed from **C** and **A** for others.
https://cloudxlab.com/assessment/displayslide/345/nosql-cap-theorem#:~:text=CAP%20theorem%20or%20Eric%20Brewers,data%20at%20the%20same%20time

Data Modeling in MongoDB

In MongoDB, data has a flexible schema. It is totally different from SQL database where you had to determine and declare a table's schema before inserting data. MongoDB collections do not enforce document structure.

The main challenge in data modeling is balancing the need of the application, the performance characteristics of the database engine, and the data retrieval patterns.

Data in MongoDB has a flexible schema.documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below −

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
        }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Radhika",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "radhika_sharma.123@gmail.com",
        phone: "9848022338"
}
```

Address:

```
{
        _id: <ObjectId104>,
        empDocID: " ObjectId101",
        city: "Hyderabad",
        Area: "Madapur",
        State: "Telangana"
}
```

Considerations while designing Schema in MongoDB

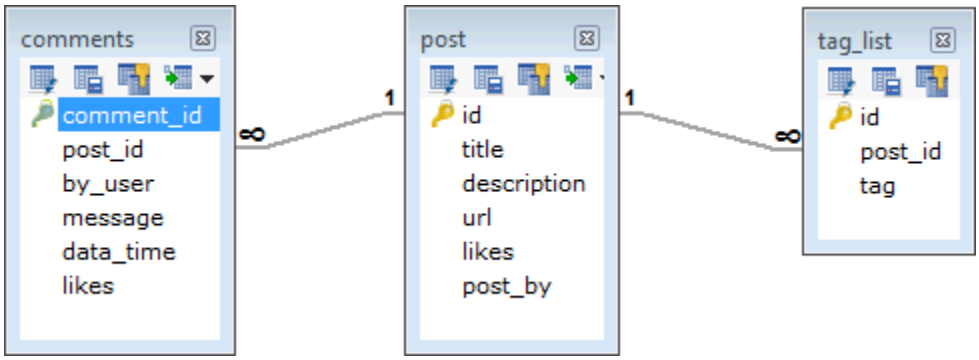
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure −

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
```

```
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

What is CRUD in MongoDB?

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database.

How to Perform CRUD Operations

Now that we've defined MongoDB CRUD operations, we can take a look at how to carry out the individual operations and manipulate documents in a MongoDB database. Let's go into the processes of creating, reading, updating, and deleting documents, looking at each operation in turn.

Create Operations

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

As the namesake, insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.RecordsDB.insertOne({
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

It's possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection (*RecordsDB*) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([{
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true},
     {name: "Kitana",
     age: "4 years",
     species: "Cat",
     ownerAddress: "521 E. Cortland",
     chipped: true}])
```

```
db.RecordsDB.insertMany([{ name: "Marsh", age: "6 years", species: "Dog",
ownerAddress: "380 W. Fir Ave", chipped: true}, {name: "Kitana", age: "4 years",
species: "Cat", ownerAddress: "521 E. Cortland", chipped: true}])
{
```

```
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

findOne()

In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null.

The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```
And, we run the following line of code:
```
db.RecordsDB.find({"age":"8  years"})
```
We would get the following result:
```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```
Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St.
A204"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species" :
"Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

replaceOne()

The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species"
```

HBase Data Model

The HBase Data Model is designed to handle semi-structured data that may differ in field size, which is a form of data and columns. The data model's layout partitions the data into simpler components and spread them across the cluster. HBase's Data Model consists of various logical components, such as a **table, line, column, family, column, column, cell,** and **edition.**

Row Key	Customer		Sales	
Customer id	Name	City	Product	Amount
101	Ram	Delhi	Chairs	4000.00
102	Shyam	Lucknow	Lamps	2000.00
103	Gita	M.P	Desk	5000.00
104	Sita	U.K	Bed	2600.00

Column Families

Table:

An HBase table is made up of several columns. The tables in HBase defines upfront during the time of the schema specification.

Row:

An HBase row consists of a row key and one or more associated value columns. Row keys are the bytes that are not interpreted. Rows are ordered lexicographically, with the first row appearing in a table in the lowest order. The layout of the row key is very critical for this purpose.

Column:

A column in HBase consists of a family of columns and a qualifier of columns, which is identified by a character: (colon).

Column Family:

Apache HBase columns are separated into the families of columns. The column families physically position a group of columns and their values to increase its performance. Every row in a table has a similar family of columns, but there may not be anything in a given family of columns.

The same prefix is granted to all column members of a column family. For example, Column **courses: history** and **courses: math**, are both members of the column family of courses. The character of the colon (:) distinguishes the family of columns from the qualifier of the family of columns. The prefix of the column family must be made up of printable characters.

During schema definition time, column families must be declared upfront while columns are not specified during schema time. They can be conjured on the fly when the table is up and running. Physically, all members of the column family are stored on the file system together.

Column Qualifier

The column qualifier is added to a column family. A column standard could be **content** (html and pdf), which provides the content of a column unit. Although column families are set up at table formation, column qualifiers are mutable and can vary significantly from row to row.

Cell:

A Cell store data and is quite a unique combination of row key, Column Family, and the Column. The data stored in a cell call its value and data types, which is every time treated as a byte[].

Timestamp:

In addition to each value, the timestamp is written and is the identifier for a given version of a number. The timestamp reflects the time when the data is written on the Region Server. But when we put data into the cell, we can assign a different timestamp value.

HBase CRUD Operations

General Commands

HBase provides shell commands to directly interact with the Database and below are a few most used shell commands.

status: This command will display the cluster information and health of the cluster.

```
1    hbase(main):>status
2    hbase(main):>status "detailed"
```

version: This will provide information about the version of HBase.

```
1    hbase(main):> version
```

whoami : This will list the current user.

```
1    hbase(main):> whoami
```

table_help : This will give the reference shell command for HBase.

```
1    hbase(main):009:> table_help
```

Create

Let's create an HBase table and insert data into the table. Now that we know, while creating a table user needs to create required Column Families.

Here we have created two-column families for table 'employee'. First Column Family is 'Personal Info' and Second Column Family is 'Professional Info'.

```
1    create 'employee', 'Personal info', 'Professional Info'
2    0 row(s) in 1.4750 seconds
3
4    => Hbase::Table - employee
```

Upon successful creation of the table, the shell will return 0 rows.

Create a table with Namespace:

A namespace is nothing but a logical grouping of tables.'company_empinfo' is the namespace id in the below command.

```
1    create 'company_empinfo:employee', 'Personal info', 'Professional Info'
```

Create a table with version:

By default, versioning is not enabled in HBase. So users need to specify while creating.

Given below is the syntax for creating an HBase table with versioning enabled.

```
1    create 'tableName',{NAME=>"CF1",VERSIONS=>5},{NAME=."CF2",VERSIONS=>5}
2    create 'bankdetails',{NAME=>"address",VERSIONS=>5}
```

Put:

Put command is used to insert records into HBase.

```
1    put 'employee', 1, 'Personal info:empId', 10
2    put 'employee', 1, 'Personal info:Name', 'Alex'
3    put 'employee', 1, 'Professional Info:Dept, 'IT'
```

Here in the above example all the rows having Row Key as 1 is considered to be one row in HBase.To add multiple rows

```
1    put 'employee', 2, 'Personal info:empId', 20
```

```
2       put 'employee', 2, 'Personal info:Name', 'Bob'
3       put 'employee', 2, 'Professional Info:Dept', 'Sales'
```

As discussed earlier, the user can add any number of columns as part of the row.

Read

'get' and 'scan' command is used to read data from HBase. Lets first discuss 'get' operation.

get: 'get' operation returns a single row from the HBase table. Given below is the syntax for the 'get' method.

```
1       get 'table Name', 'Row Key'
1       hbase(main):022:get 'employee', 1
```

```
COLUMN                          CELL
 Pe sonal info:Name             timestamp=1504600767520, value=Alex
Per sonal info:empId            timestamp=1504600767491, value=10
Pro fessional Info:Dept         timestamp=1504600767540, value=IT
3 rc w(s) in 0.0250 seconds
```

To retrieve a specific column of row:

Follow the command to read a specific column of a row.

```
1       get 'table Name', 'Row Key',{COLUMN => 'column family:column'}
2       get 'table Name', 'Row Key' {COLUMN => ['c1', 'c2', 'c3']
1       get 'employee', 1 ,{COLUMN => 'Personal info:empId'}
```

```
COLUMN                          CELL
 Pe sonal info:Name             timestamp=1504600767520, value=Alex
Per sonal info:empId            timestamp=1504600767491, value=10
Pro fessional Info:Dept         timestamp=1504600767540, value=IT
3 rc w(s) in 0.0250 seconds
```

Note: Notice that there is a timestamp attached to each cell. These timestamps will update for the cell whenever the cell value is updated. All the old values will be there but timestamp having the latest value will be displayed as output.

Get all version of a column

Below given command is used to find different versions. Here 'VERSIONS => 3' defines number of version to be retrieved.

```
1       get 'Table Name', 'Row Key', {COLUMN => 'Column Family', VERSIONS => 3}
```

scan:

'scan' command is used to retrieve multiple rows.

Select all:

The below command is an example of a basic search on the entire table.

```
1       scan 'Table Name'
1       hbase(main):074:> scan 'employee'
```

```
RO                      COLUMN+CELL
 1              column=Personal info:Name, timestamp=1504600767520, value=Alex
 1              column=Personal info:empId, timestamp=1504606480934, value=15
 1              column=Professional Info:Dept, timestamp=1504600767540, value=IT
 2              column=Personal info:Name, timestamp=1504600767588, value=Bob
 2              column=Personal info:empId, timestamp=1504600767568, value=20
 2              column=Professional Info:Dept, timestamp=1504600768266, value=Sales
```

```
2 row(s) in 0.0500 seconds
```

Note: All the Rows are arranged by Row Keys along with columns in each row.

Column Selection:
The below command is used to Scan any particular column.

```
1    hbase(main):001:>scan 'employee' ,{COLUMNS => 'Personal info:Name' }
```

```
RO                              COLUMN+CELL
 1                column=Personal info:Name, timestamp=1504600767520, value=Alex
 2                column=Personal info:Name, timestamp=1504600767588, value=Bob
2 rc w(s) in 0.3660 seconds
```

Limit Query:
The below command is used to Scan any particular column.

```
1    hbase(main):002:>scan 'employee' ,{COLUMNS => 'Personal info:Name',LIMIT =>1 }
```

```
RO                              COLUMN+CELL
 1                column=Personal info:Name, timestamp=1504600767520, value=Alex
1 rc w(s) in 0.0270 seconds
```

Update

To update any record HBase uses ‘put’ command. To update any column value, users need to put new values and HBase will automatically update the new record with the latest timestamp.

```
1    put 'employee', 1, 'Personal info:empId', 30
```

The old value will not be deleted from the HBase table. Only the updated record with the latest timestamp will be shown as query output.
To check the old value of any row use below command.

```
1    get 'Table Name', 'Row Key', {COLUMN => 'Column Family', VERSIONS => 3}
```

Delete

‘**delete**‘ command is used to delete individual cells of a record.
The below command is the syntax of delete command in the HBase Shell.

```
1    delete 'Table Name' ,'Row Key','Column Family:Column'
1    delete 'employee',1, 'Personal info:Name'
```

Drop Table:
To drop any table in HBase, first, it is required to disable the table. The query will return an error if the user is trying to delete the table without disabling the table. Disable removes the indexes from memory.
The below command is used to disable and drop the table.

```
1    disable 'employee'
```

Once the table is disabled, the user can drop using below syntax.

```
1    drop 'employee'
```

You can verify the table in using ‘exist’ command and enable table which is already disabled, just use ‘enable’ command.