its "rich query language", "fast in-place update", etc. The chapter will cover the CRUD (Create, Read, Update, and Delete) operations in detail.

To gain the maximum from the chapter, please attempt the Test Me exercises given at the end of the chapter.

## 6.1 WHAT IS MONGODB?

MongoDB is

1. Cross-platform.
2. Open source.
3. Non-relational.
4. Distributed.
5. NoSQL.
6. Document-oriented data store.

## 6.2 WHY MONGODB?

Few of the major challenges with traditional RDBMS are dealing with large volumes of data, rich variety of data – particularly unstructured data, and meeting up to the scale needs of enterprise data. The need is for a database that can scale out or scale horizontally to meet the scale requirements, has flexibility with respect to schema, is fault tolerant, is consistent and partition tolerant, and can be easily distributed over a multitude of nodes in a cluster. Refer Figure 6.1.

### 6.2.1 Using Java Script Object Notation (JSON)

JSON is extremely expressive. MongoDB actually does not use JSON but BSON (pronounced Bee Son) – it is Binary JSON. It is an open standard. It is used to store complex data structures.

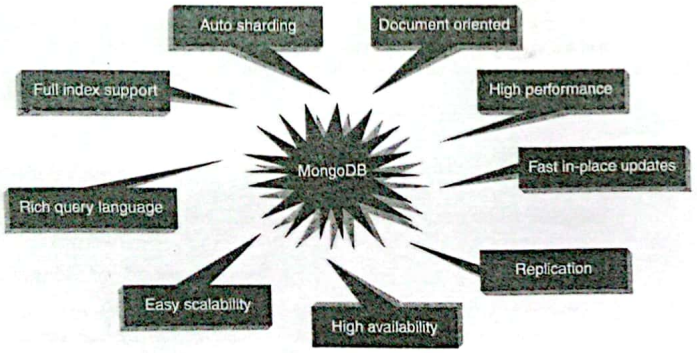



**Figure 6.1** Why MongoDB?



(NoSQL – III)

***Let us trace the journey from .csv to XML to JSON:*** Let us look at how data is stored in .csv file. Assume that this data is about the employees of an organization named "XYZ". As can be seen below, the column values are separated using commas and the rows are separated by a carriage return.

John, Mathews, +123 4567 8900
Andrews, Symmonds, +456 7890 1234
Mable, Mathews, +789 1234 5678

This looks good! However let us make it slightly more legible by adding column heading.

FirstName, LastName, ContactNo
John, Mathews, +123 4567 8900
Andrews, Symmonds, +456 7890 1234
Mable, Mathews, +789 1234 5678

Now assume that few employees have more than one ContactNo. It can be neatly classified as OfficeContactNo and HomeContactNo. But what if few employees have more than one OfficeContactNo and more than one HomeContactNo? Ok, so this is the first issue we need to address.

Let us look at just another piece of data that you wish to store about the employees. You need to store their email addresses as well. Here again we have the same issues, few employees have two email addresses, few have three and there are a few employees with more than three email addresses as well.

As we come across these fields or columns, we realize that it gets messy with .csv. CSV are known to store data well if it is flat and does not have repeating values.

The problem becomes even more complex when different departments maintain the details of their employees. The formats of .csv (columns, etc.) could vastly differ and it will call for some efforts before we can merge the files from the various departments to make a single file.

This problem can be solved by XML. But as the name suggests XML is highly extensible. It does not just call for defining a data format, rather it defines how you define a data format. You may be prepared to undertake this cumbersome task for highly complex and structured data; however, for simple data exchange it might just be too much work.

Enter JSON! Let us look at how it reacts to the problem at hand.

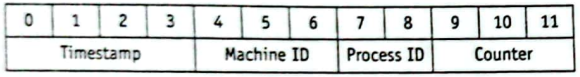
{
FirstName: John,
LastName: Mathews,
ContactNo: [+123 4567 8900, +123 4444 5555]
}
{
FirstName: Andrews,
LastName: Symmonds,
ContactNo: [+456 7890 1234, +456 6666 7777]
}
{
FirstName Mable,
LastName: Mathews,
ContactNo: +789 1234 5678
}

As you can see it is quite easy to read a JSON. There is absolutely no confusion now. One can have a list of *n* contact numbers, and they can be stored with ease.

JSON is very expressive. It provides the much needed ease to store and retrieve documents in their real form. The binary form of JSON is BSON. BSON is an open standard. In most cases it consumes less space as compared to the text-based JSON. There is yet another advantage with BSON. It is much easier and quicker to convert BSON to a programming language's native data format. There are MongoDB drivers available for a number of programming languages such as C, C++, Ruby, PHP, Python, C#, etc., and each works slightly differently. Using the basic binary format enables the native data structures to be built quickly for each language without going through the hassle of first processing JSON.

### 6.2.2 Creating or Generating a Unique Key

Each JSON document should have a unique identifier. It is the _id key. It is similar to the primary key in relational databases. This facilitates search for documents based on the unique identifier. An index is automatically built on the unique identifier. It is your choice to either provide unique values yourself or have the mongo shell generate the same.




#### 6.2.2.1 Database

It is a collection of collections. In other words, it is like a container for collections. It gets created the first time that your collection makes a reference to it. This can also be created on demand. Each database gets its own set of files on the file system. A single MongoDB server can house several databases.

#### 6.2.2.2 Collection

A collection is analogous to a table of RDBMS. A collection is created on demand. It gets created the first time that you attempt to save a document that references it. A collection exists within a single database. A collection holds several MongoDB documents. A collection does not enforce a schema. This implies that documents within a collection can have different fields. Even if the documents within a collection have same fields, the order of the fields can be different.

#### 6.2.2.3 Document

A document is analogous to a row/record/tuple in an RDBMS table. A document has a dynamic schema. This implies that a document in a collection need not necessarily have the same set of fields/key-value pairs. Shown in Figure 6.2 is a collection by the name "students" containing three documents.

### 6.2.3 Support for Dynamic Queries

MongoDB has extensive support for dynamic queries. This is in keeping with traditional RDBMS wherein we have static data and dynamic queries. CouchDB, another document-oriented, schema-less NoSQL database and MongoDB's biggest competitor, works on quite the reverse philosophy. It has support for dynamic data and static queries.



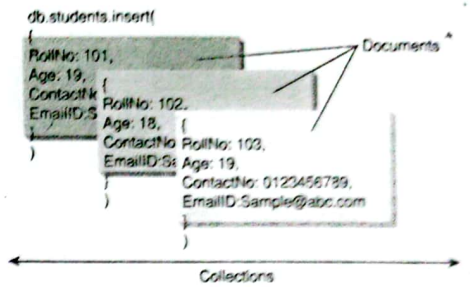



**Figure 6.2** A collection "students" containing 3 documents.

### 6.2.4 Storing Binary Data

MongoDB provides GridFS to support the storage of binary data. It can store up to 4 MB of data. This usually suffices for photographs (such as a profile picture) or small audio clips. However, if one wishes to store movie clips, MongoDB has another solution.

It stores the metadata (data about data along with the context information) in a collection called "file". It then breaks the data into small pieces called chunks and stores it in the "chunks" collection. This process takes care about the need for easy scalability.

### 6.2.5 Replication

Why replication? It provides data redundancy and high availability. It helps to recover from hardware failure and service interruptions. In MongoDB, the replica set has a single primary and several secondaries. Each write request from the client is directed to the primary. The primary logs all write requests into its Oplog (operations log). The Oplog is then used by the secondary replica members to synchronize their data. This way there is strict adherence to consistency. Refer Figure 6.3. The clients usually read from the primary. However, the client can also specify a read preference that will then direct the read operations to the secondary.

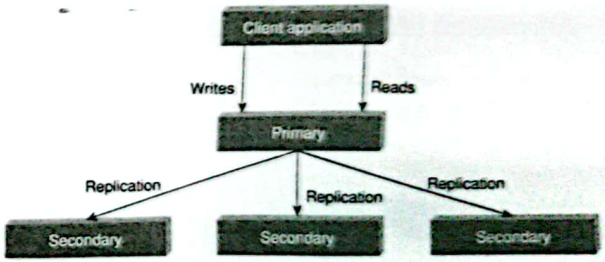



**Figure 6.3** The process of **REPLICATION** in MongoDB.

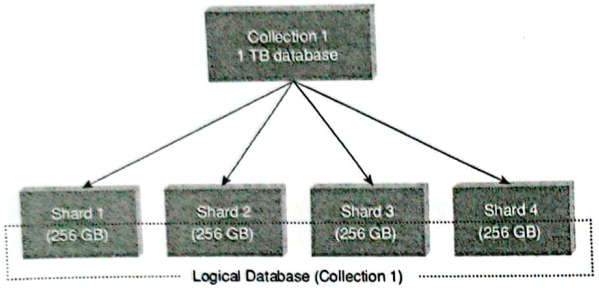



**Figure 6.4** The process of **SHARDING** in MongoDB.

### 6.2.6 Sharding

Sharding is akin to horizontal scaling. It means that the large dataset is divided and distributed over multiple servers or shards. Each shard is an independent database and collectively they would constitute a logical database.

The prime advantages of sharding are as follows:

1. Sharding reduces the amount of data that each shard needs to store and manage. For example, if the dataset was 1 TB in size and we were to distribute this over four shards, each shard would house just 256 GB data. Refer Figure 6.4. As the cluster grows, the amount of data that each shard will store and manage will decrease.
2. Sharding reduces the number of operations that each shard handles. For example, if we were to insert data, the application needs to access only that shard which houses that data.

### 6.2.7 Updating Information In-Place

MongoDB updates the information in-place. This implies that it updates the data wherever it is available. It does not allocate separate space and the indexes remain unaltered.

MongoDB is all for lazy-writes. It writes to the disk once every second. Reading and writing to disk is a slow operation as compared to reading and writing from memory. The fewer the reads and writes that we perform to the disk, the better is the performance. This makes MongoDB faster than its other competitors who write almost immediately to the disk. However, there is a tradeoff. MongoDB makes no guarantee that data will be stored safely on the disk.

***Guess Me***

***A. Who am I?***

- I am blindingly fast
- I am massively scalable
- I am easy to use
- I work with documents rather than rows



***B. Who am I?***

- I am not for everyone
- I am good with complex data structures such as blog posts and comments
- I am good with analytics such as a real time google analytics
- I am comfortable with Linux, Mac OS, Solaris, and windows

***C. Who am I?***

- I have support for transactions
- I have static data
- I allow dynamic queries to be run on me

***D. Who am I?***

- I am one of the biggest competitor for MongoDB
- I have dynamic data
- Only static queries can be run on me
- I am document-oriented too

**Answers:**

**A.** MongoDB
**B.** MongoDB
**C.** Traditional RDBMS
**D.** CouchDB

## 6.3 TERMS USED IN RDBMS AND MONGODB

| RDBMS | MongoDB |
|---|---|
| Database | Database |
| Table | Collection |
| Record | Document |
| Columns | Fields / Key Value pairs |
| Index | Index |
| Joins | Embedded documents |
| Primary Key | Primary key (_id is a identifier) |

| | MySQL | Oracle | MongoDB |
|---|---|---|---|
| Database Server | MySqld | Oracle | Mongod |
| Database Client | MySql | SQL Plus | mongo |

### 6.3.1 Create Database

The syntax for creating database is as follows:

```
use DATABASE_Name
```

To create a database by the name "myDB" the syntax is

```
use myDB
```

```
> use myDB;
switched to db myDB
>
```

To confirm the existence of your database, type the command at the MongoDB shell:

```
db
```

```
> db;
myDB
>
```

To get a list of all databases, type the below command:

```
show dbs
```

```
> show dbs;
admin  (empty)
local  0.078GB
test   0.078GB
>
```

Notice that the newly created database, "myDB" does not show up in the list above. The reason is that the database needs to have at least one document to show up in the list.

The default database in MongoDB is test. If one does not create any database, all collections are by default stored in the test database.

### 6.3.2 Drop Database

The syntax to drop database is as follows:

```
db.dropDatabase();
```

To drop the database, "myDB", first ensure that you are currently placed in "myDB" database and then use the db.dropDatabase() command to drop the database.

```
use myDB;
db.dropDatabase();
```

Confirm if the database "myDB" has been dropped.

```
> db.dropDatabase();
{ "dropped" : "myDB", "ok" : 1 }
>
```

If no database is selected, the default database "test" is dropped.

## 6.4 DATA TYPES IN MONGODB

The following are various data types in MongoDB.



| | |
|---|---|
| String | Must be UTF-8 valid.<br>Most commonly used data type. |
| Integer | Can be 32-bit or 64-bit (depends on the server). |
| Boolean | To store a true/false value. |
| Double | To store floating point (real values). |
| Min/Max keys | To compare a value against the lowest or highest BSON elements. |
| Arrays | To store arrays or list or multiple values into one key. |
| Timestamp | To record when a document has been modified or added. |
| Null | To store a NULL value. A NULL is a missing or unknown value. |
| Date | To store the current date or time in Unix time format. One can create object of date and pass day, month and year to it. |
| Object ID | To store the document's id. |
| Binary data | To store binary data (images, binaries, etc.). |
| Code | To store javascript code into the document. |
| Regular expression | To store regular expression. |

A few commands worth looking at are as follows (try them!!!).

***To report the name of the current database:***

```
C:\windows\system32\cmd.exe - mongo
> db
test
>
```

***To display the list of databases:***

```
C:\windows\system32\cmd.exe - mongo
> show dbs
admin  (empty)
local  0.078GB
myDB1  0.078GB
>
```

***To switch to a new database, for example, myDB1:***

```
C:\windows\system32\cmd.exe - mongo
> use myDB1
switched to db myDB1
>
```

***To display the list of collections (tables) in the current database:***

```
C:\windows\system32\cmd.exe - mongo
> show collections
system.indexes
system.js
>
```

***To display the current version of the MongoDB server:***

```
C:\windows\system32\cmd.exe - mongo
> db.version()
2.6.1
>
```

## WHAT'S IN STORE?

This chapter will cover another NoSQL database called "Cassandra". We will explore the features of Cassandra that has made it so immensely popular. The chapter will cover the basic CRUD (Create, Read, Update, and Delete) operations using cqlsh.

Please attempt the Test Me exercises given at the end of the chapter to practice, learn, and comprehend Cassandra effectively.

## 7.1 APACHE CASSANDRA – AN INTRODUCTION

We shall start this chapter with few points that a reader should know about Cassandra.

1. Apache Cassandra was born at Facebook. After Facebook open sourced the code in 2008, Cassandra became an Apache Incubator project in 2009 and subsequently became a top-level Apache project in 2010.
2. It is built on Amazon's dynamo and Google's BigTable.
3. Cassandra does NOT compromise on availability. Since it does not have a master-slave architecture, there is no question of single point of failure. This proves beneficial for business critical applications that need to be up and running always and cannot afford to go down ever.
4. It is highly scalable (it scales out), high performance distributed database. It distributes and manages gigantic amount of data across commodity servers.

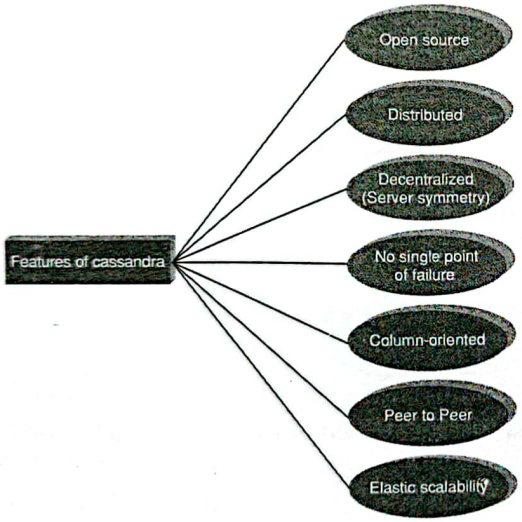



**Figure 7.1** Features of Cassandra.



5. It is a column-oriented database designed to support peer-to-peer symmetric nodes instead of the master-slave architecture.
6. It has adherence to the Availability and Partition Tolerance properties of CAP theorem. It takes care of consistency using BASE (Basically Available Soft State Eventual Consistency) approach.

Refer Figure 7.1. Few companies that have successfully deployed Cassandra and have benefitted immensely from it are as follows:

1. Twitter
2. Netflix
3. Cisco
4. Adobe
5. eBay
6. Rackspace

## 7.2 FEATURES OF CASSANDRA

### 7.2.1 Peer-to-Peer Network

As with any other NoSQL database, Cassandra is designed to distribute and manage large data loads across multiple nodes in a cluster constituted of commodity hardware. Cassandra does NOT have a master-slave architecture which means that it does NOT have single point of failure. A node in Cassandra is structurally identical to any other node. Refer Figure 7.2. In case a node fails or is taken offline, it definitely impacts the throughput. However, it is a case of graceful degradation where everything does not come crashing at any given instant owing to a node failure. One can still go about business as usual. It tides over the problem of failure by employing a peer-to-peer distributed system across homogeneous nodes. It ensures that data is distributed across all nodes in the cluster. Each node exchanges information across the cluster every second.

Let us look at how a Cassandra node writes. Each write is written to the commit log sequentially. A write is taken to be successful only if it is written to the commit log. Data is then indexed and pushed to an in-memory structure called "Memtable". When the in-memory data structure, "the Memtable", is full, the contents are flushed to "SSTable" (Sorted String) data file on the disk. The SSTable is immutable and is

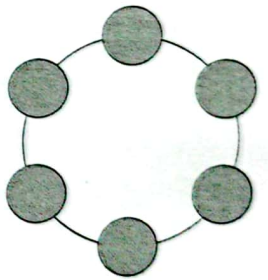


**Figure 7.2** Sample Cassandra cluster.

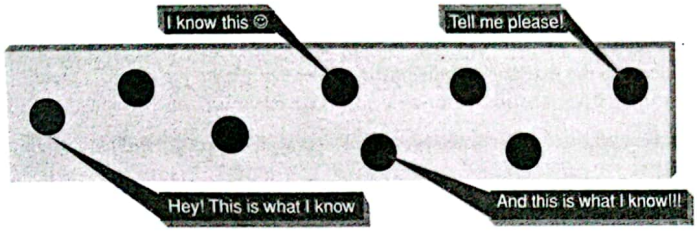



**Figure 7.3** Gossip protocol.

append-only. It is stored on disk sequentially and is maintained for each Cassandra table. The partitioning and replication of all writes are performed automatically across the cluster.

### 7.2.2 Gossip and Failure Detection

Gossip protocol is used for intra-ring communication. It is a peer-to-peer communication protocol which eases the discovery and sharing of location and state information with other nodes in the cluster. Refer Figure 7.3. Although there are quite a few subtleties involved, but at its core it's a simple and robust system. A node only has to send out the communication to a subset of other nodes. For repairing unread data, Cassandra uses what's called an anti-entropy version of the gossip protocol.

### 7.2.3 Partitioner

A partitioner takes a call on how to distribute data on the various nodes in a cluster. It also determines the node on which to place the very first copy of the data. Basically a partitioner is a hash function to compute the token of the partition key. The partition key helps to identify a row uniquely.

### 7.2.4 Replication Factor

The replication factor determines the number of copies of data (replicas) that will be stored across nodes in a cluster. If one wishes to store only one copy of each row on one node, they should set the replication factor to one. However, if the need is for two copies of each row of data on two different nodes, one should go with a replication factor of two. The replication factor should ideally be more than one and not more than the number of nodes in the cluster. A replication strategy is employed to determine which nodes to place the data on. Two replication strategies are available:

1. SimpleStrategy.
2. NetworkToplogyStrategy.

The preferred one is NetworkTopologyStrategy as it is simple and supports easy expansion to multiple data centers, should there be a need.

### 7.2.5 Anti-Entropy and Read Repair

A cluster is made up of several nodes. Since the cluster is constituted of commodity hardware, it is prone to failure. In order to achieve fault tolerance, a given piece of data is replicated on one or more nodes. A client



can connect to any node in the cluster to read data. How many nodes will be read before responding to the client is based on the consistency level specified by the client. If the client-specified consistency is not met, the read operation blocks. There is a possibility that few of the nodes may respond with an out-of-date value. In such a case, Cassandra will initiate a read repair operation to bring the replicas with stale values up to date.

For repairing unread data, Cassandra uses an anti-entropy version of the gossip protocol. Anti-entropy implies comparing all the replicas of each piece of data and updating each replica to the newest version. The read repair operation is performed either before or after returning the value to the client as per the specified consistency level.

### 7.2.6 Writes in Cassandra

Let us look at behind the scene activities. Here is a client that initiates a write request. Where does his write get written to? It is first written to the commit log. A write is taken as successful only if it is written to the commit log. The next step is to push the write to a memory resident data structure called Memtable. A threshold value is defined in the Memtable. When the number of objects stored in the Memtable reaches a threshold, the contents of Memtable are flushed to the disk in a file called SSTable (Sorted String Table). Flushing is a non-blocking operation. It is possible to have multiple Memtables for a single column family. One out of them is current and the rest are waiting to be flushed.

### 7.2.7 Hinted Handoffs

The first question that arises is: Why Cassandra is all for availability? It works on the philosophy that it will always be available for writes.

Assume that we have a cluster of three nodes – Node A, Node B, and Node C. Node C is down for some reason. Refer Figure 7.4. We are maintaining a replication factor of 2 which implies that two copies of each row will be stored on two different nodes. The client makes a write request to Node A. Node A is the coordinator and serves as a proxy between the client and the nodes on which the replica is to be placed. The client

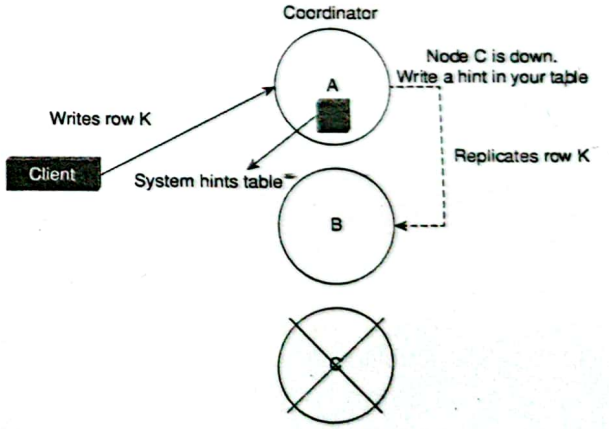



**Figure 7.4** Depiction of hinted handoffs.

writes Row K to Node A. Node A then writes Row K to Node B and stores a hint for Node C. The hint will have the following information:

1. Location of the node on which the replica is to be placed.
2. Version metadata.
3. The actual data.

When Node C recovers and is back to the functional self, Node A reacts to the hint by forwarding the data to Node C.

### 7.2.8 Tunable Consistency

One of the features of Cassandra that has made it immensely popular is its ability to utilize tunable consistency. The database systems can go for either strong consistency or eventual consistency. Cassandra can cash in on either flavor of consistency depending on the requirements. In a distributed system, we work with several servers in the system. Few of these servers are in one data center and others in other data centers. Let us take a look at what it means by strong consistency and eventual consistency.

1. **Strong consistency:** If we work with strong consistency, it implies that each update propagates to all locations where that piece of data resides. Let us assume a single data center setup. Strong consistency will ensure that all of the servers that should have a copy of the data, will have it, before the client is acknowledged with a success. If we are wondering whether it will impact performance, yes it will. It will cost a few extra milliseconds to write to all servers.
2. **Eventual consistency:** If we work with eventual consistency, it implies that the client is acknowledged with a success as soon as a part of the cluster acknowledges the write. When should one go for eventual consistency? The choice is fairly obvious... when application performance matters the most. Example: A single server acknowledges the write and then begins propagating the data to other servers.

#### 7.2.8.1 *Read Consistency*

Let us understand what the read consistency level means. It means how many replicas must respond before sending out the result to the client application. There are several read consistency levels as mentioned in Table 7.1.

#### 7.2.8.2 *Write Consistency*

Let us understand what the write consistency level means. It means on how many replicas write must succeed before sending out an acknowledgement to the client application. There are several write consistency levels as mentioned in Table 7.2.

**Table 7.1** Read consistency levels in Cassandra

| | |
|---|---|
| ONE | Returns a response from the closest node (replica) holding the data. |
| QUORUM | Returns a result from a quorum of servers with the most recent timestamp for the data. |
| LOCAL_QUORUM | Returns a result from a quorum of servers with the most recent timestamp for the data in the same data center as the coordinator node. |
| EACH_QUORUM | Returns a result from a quorum of servers with the most recent timestamp in all data centers. |
| ALL | This provides the highest level of consistency of all levels and the lowest level of availability of all levels. It responds to a read request from a client after all the replica nodes have responded. |



**Table 7.2** Write consistency levels in Cassandra

| | |
|---|---|
| ALL | This is the highest level of consistency of all levels as it necessitates that a write must be written to the commit log and Memtable on all replica nodes in the cluster. |
| EACH_QUORUM | A write must be written to the commit log and Memtable on a quorum of replica nodes in *all* data centers. |
| QUORUM | A write must be written to the commit log and Memtable on a quorum of replica nodes. |
| LOCAL_QUORUM | A write must be written to the commit log and Memtable on a quorum of replica nodes in the same data center as the coordinator node. This is to avoid latency of inter-data center communication. |
| ONE | A write must be written to the commit log and Memtable of at least one replica node. |
| TWO | A write must be written to the commit log and Memtable of at least two replica nodes. |
| THREE | A write must be written to the commit log and Memtable of at least three replica nodes. |
| LOCAL_ONE | A write must be sent to, and successfully acknowledged by, at least one replica node in the local data center. |

## 7.3 CQL DATA TYPES

Refer Table 7.3 for built-in data types for columns in CQL.

**Table 7.3** Built-in data types in Cassandra

| | |
|---|---|
| Int | 32 bit signed integer |
| Bigint | 64 bit signed long |
| Double | 64-bit IEEE-754 floating point |
| Float | 32-bit IEEE-754 floating point |
| Boolean | True or false |
| Blob | Arbitrary bytes, expressed in hexadecimal |
| Counter | Distributed counter value |
| Decimal | Variable – precision integer |
| List | A collection of one or more ordered elements |
| Map | A JSON style array of elements |
| Set | A collection of one or more elements |
| Timestamp | Date plus time |
| Varchar | UTF 8 encoded string |
| Varint | Arbitrary-precision integers |
| Text | UTF 8 encoded string |