MongoDB CRUD Operations Introduction

Release 2.2.2

MongoDB Documentation Project

December 20, 2012

CONTENTS

I Read and Write Operations **3**

1 Read Operations **7**
- 1.1 Queries in MongoDB . . . 7
- 1.2 Indexes . . . 12
- 1.3 Cursors . . . 14
- 1.4 Architecture . . . 17

2 Write Operations **19**
- 2.1 Write Operators . . . 19
- 2.2 Write Concern . . . 19
- 2.3 Bulk Inserts . . . 22
- 2.4 Indexing . . . 22
- 2.5 Isolation . . . 22
- 2.6 Architecture . . . 23

3 BSON Documents **25**
- 3.1 Structure . . . 25
- 3.2 Document Types in MongoDB . . . 27
- 3.3 BSON Type Considerations . . . 30

II CRUD Operations **33**

4 Create **37**
- 4.1 Overview . . . 37
- 4.2 Insert . . . 38
- 4.3 Create with Save . . . 42
- 4.4 Create with Upsert . . . 43

5 Read **45**
- 5.1 Overview . . . 45
- 5.2 Find . . . 45
- 5.3 Find One . . . 51

6 Update **53**
- 6.1 Overview . . . 53
- 6.2 Update . . . 53
- 6.3 Save . . . 56

7 Delete **59**

7.1 Overview 59
7.2 Remove 59

Index **61**

CRUD stands for *create*, *read*, *update*, and *delete*, which are the four core database operations used in database driven application development. The *CRUD Operations* (page 35) section provides introduction to each class of operation along with complete examples of each operation. The documents in the *Read and Write Operations* (page 5) section provide a higher level overview of the behavior and available functionality of these operations.

Part I

Read and Write Operations

The *Read Operations* (page 7) and *Write Operations* (page 19) documents provide higher level introductions and description of the behavior and operations of read and write operations for MongoDB deployments. The *BSON Documents* (page 25) provides an overview of *documents* and document-orientation in MongoDB.

CHAPTER

ONE

READ OPERATIONS

Read operations include all operations that return a cursor in response to application request datas (i.e. *queries*,) and also include a number of `aggregation` operations that do not return a cursor but have similar properties as queries. These commands include `aggregate`, `count`, and `distinct`.

This document describes the syntax and structure of the queries applications use to request data from MongoDB and how different factors affect the efficiency of reads.

Note: All of the examples in this document use the `mongo` shell interface. All of these operations are available in an idiomatic interface for each language by way of the `MongoDB Driver`. See your driver documentation for full API documentation.

1.1 Queries in MongoDB

In the `mongo` shell, the `find()` and `findOne()` methods perform read operations. The `find()` method has the following syntax: [1]

```
db.collection.find( <query>, <projection> )
```

- The `db.collection` object specifies the database and collection to query. All queries in MongoDB address a *single* collection.

  You can enter `db` in the `mongo` shell to return the name of the current database. Use the `show collections` operation in the `mongo` shell to list the current collections in the database.

- Queries in MongoDB are *BSON* objects that use using a set of `query operators` to describe query parameters.

  The `<query>` argument of the `find()` method holds this query document. A read operation without a query document will return all documents in the collection.

- The `<projection>` argument describes the result set in the form of a document. Projections specify or limit the fields to return.

  Without a projection, the operation will return all fields of the documents. Specify a projection if your documents are larger, or when your application only needs a subset of available fields.

- The order of documents returned by a query is not defined and is not necessarily consistent unless you specify a sort (`sort()`).

[1] `db.collection.find()` is a wrapper for the more formal query structure with the `$query` operator.

For example, the following operation on the `inventory` collection selects all documents where the `type` field equals `'food'` and the `price` field has a value less than `9.95`. The projection limits the response to the `item` and `qty`, and `_id` field:

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } },
                   { item: 1, qty: 1 } )
```

The `findOne()` method is similar to the `find()` method except the `findOne()` method returns a single document from a collection rather than a cursor. The method has the syntax:

```
db.collection.findOne( <query>, <projection> )
```

For additional documentation and examples of the main MongoDB read operators, refer to the *Read* (page 45) page of the *CRUD* (page 1) section.

1.1.1 Query Document

This section provides an overview of the query document for MongoDB queries. See the preceding section for more information on *queries in MongoDB* (page 7).

The following examples demonstrate the key properties of the query document in MongoDB queries, using the `find()` method from the `mongo` shell, and a collection of documents named `inventory`:

- An empty query document (`{}`) selects all documents in the collection:

  ```
  db.inventory.find( {} )
  ```

  Not specifying a query document to the `find()` is equivalent to specifying an empty query document. Therefore the following operation is equivalent to the previous operation:

  ```
  db.inventory.find()
  ```

- A single-clause query selects all documents in a collection where a field has a certain value. These are simple "equality" queries.

  In the following example, the query selects all documents in the collection where the `type` field has the value `snacks`:

  ```
  db.inventory.find( { type: "snacks" } )
  ```

- A single-clause query document can also select all documents in a collection given a condition or set of conditions for one field in the collection's documents. Use the *query operators* to specify conditions in a MongoDB query.

  In the following example, the query selects all documents in the collection where the value of the `type` field is either `'food'` or `'snacks'`:

  ```
  db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
  ```

  Note: Although you can express this query using the `$or` operator, choose the `$in` operator rather than the `$or` operator when performing equality checks on the same field.

- A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical `AND` conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

  In the following example, the query document specifies an equality match on a single field, followed by a range of values for a second field using a *comparison operator*:

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

This query selects all documents where the `type` field has the value `'food'` **and** the value of the `price` field is less than (`$lt`) `9.95`.

- Using the `$or` operator, you can specify a compound query that joins each clause with a logical `OR` conjunction so that the query selects the documents in the collection that match at least one condition.

  In the following example, the query document selects all documents in the collection where the field `qty` has a value greater than (`$gt`) `100` **or** the value of the `price` field is less than (`$lt`) `9.95`:

```
db.inventory.find( { $or: [ { qty: { $gt: 100 } },
                            { price: { $lt: 9.95 } } ]
                   } )
```

- With additional clauses, you can specify precise conditions for matching documents. In the following example, the compound query document selects all documents in the collection where the value of the `type` field is `'food'` **and** *either* the `qty` has a value greater than (`$gt`) `100` *or* the value of the `price` field is less than (`$lt`) `9.95`:

```
db.inventory.find( { type: 'food', $or: [ { qty: { $gt: 100 } },
                                          { price: { $lt: 9.95 } } ]
                   } )
```

Subdocuments

When the field holds an embedded document (i.e. subdocument), you can either specify the entire subdocument as the value of a field, or "reach into" the subdocument using *dot notation*, to specify values for individual fields in the subdocument:

- Equality matches within subdocuments select documents if the subdocument matches *exactly* the specified subdocument, including the field order.

  In the following example, the query matches all documents where the value of the field `producer` is a subdocument that contains *only* the field `company` with the value `'ABC123'` and the field `address` with the value `'123 Street'`, in the exact order:

```
db.inventory.find( {
                     producer: {
                                 company: 'ABC123',
                                 address: '123 Street'
                               }
                   }
                 )
```

- Equality matches for specific fields within subdocuments select documents when the field in the subdocument contains a field that matches the specified value.

  In the following example, the query uses the *dot notation* to match all documents where the value of the field `producer` is a subdocument that contains a field `company` with the value `'ABC123'` and may contain other fields:

```
db.inventory.find( { 'producer.company': 'ABC123' } )
```

Arrays

When the field holds an array, you can query for values in the array, and if the array holds subdocuments, you query for specific fields within the these subdocuments using *dot notation*:

- Equality matches can specify an entire array, to select an array that matches exactly. In the following example, the query matches all documents where the value of the field `tags` is an array and holds three elements, `'fruit'`, `'food'`, and `'citrus'`, in this order:

  ```
  db.inventory.find( { tags: [ 'fruit', 'food', 'citrus' ] } )
  ```

- Equality matches can specify a single element in the array. If the array contains at least *one* element with the specified value, as in the following example: the query matches all documents where the value of the field `tags` is an array that contains, as one of its elements, the element `'fruit'`:

  ```
  db.inventory.find( { tags: 'fruit' } )
  ```

  Equality matches can also select documents by values in an array using the array index (i.e. position) of the element in the array, as in the following example: the query uses the *dot notation* to match all documents where the value of the `tags` field is an array whose first element equals `'fruit'`:

  ```
  db.inventory.find( { 'tags.0' : 'fruit' } )
  ```

In the following examples, consider an array that contains subdocuments:

- If you know the array index of the subdocument, you can specify the document using the subdocument's position.

  The following example selects all documents where the `memos` contains an array whose first element (i.e. index is `0`) is a subdocument with the field `by` with the value `'shipping'`:

  ```
  db.inventory.find( { 'memos.0.by': 'shipping' } )
  ```

- If you do not know the index position of the subdocument, concatenate the name of the field that contains the array, with a dot (`.`) and the name of the field in the subdocument.

  The following example selects all documents where the `memos` field contains an array that contains at least one subdocument with the field `by` with the value `'shipping'`:

  ```
  db.inventory.find( { 'memos.by': 'shipping' } )
  ```

- To match by multiple fields in the subdocument, you can use either dot notation or the `$elemMatch` operator:

  The following example uses dot notation to query for documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to `'on time'` and the field `by` equal to `'shipping'`:

  ```
  db.inventory.find(
                     {
                       'memos.memo': 'on time',
                       'memos.by': 'shipping'
                     }
                   )
  ```

  The following example uses `$elemMatch` to query for documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to `'on time'` and the field `by` equal to `'shipping'`:

  ```
  db.inventory.find( { memos: {
                                $elemMatch: {
                                              memo : 'on time',
  ```

```
                                                    by: 'shipping'
                                                }
                                            }
                                        }
                                    )
```

Refer to the `http://docs.mongodb.org/manual/reference/operator` document for the complete list of query operators.

1.1.2 Result Projections

The *projection* specification limits the fields to return for all matching documents. Constraining the result set by restricting the fields to return can minimize network transit costs and the costs of deserializing documents in the application layer.

The second argument to the `find()` method is a projection, and it takes the form of a *document* with a list of fields for inclusion or exclusion from the result set. You can either specify the fields to include (e.g. `{ field: 1 }`) or specify the fields to exclude (e.g. `{ field: 0 }`). The `_id` field is implicitly included, unless explicitly excluded.

Note: You cannot combine inclusion and exclusion semantics in a single projection with the *exception* of the `_id` field.

Consider the following projection specifications in `find()` operations:

- If you specify no projection, the `find()` method returns all fields of all documents that match the query.

  ```
  db.inventory.find( { type: 'food' } )
  ```

  This operation will return all documents in the `inventory` collection where the value of the `type` field is `'food'`.

- A projection can explicitly include several fields. In the following operation, `find()` method returns all documents that match the query as well as `item` and `qty` fields. The results also include the `_id` field:

  ```
  db.inventory.find( { type: 'food' }, { item: 1, qty: 1 } )
  ```

- You can remove the `_id` field by excluding it from the projection, as in the following example:

  ```
  db.inventory.find( { type: 'food' }, { item: 1, qty: 1, _id:0 } )
  ```

  This operation returns all documents that match the query, and *only* includes the `item` and `qty` fields in the result set.

- To exclude a single field or group of fields you can use a projection in the following form:

  ```
  db.inventory.find( { type: 'food' }, { type:0 } )
  ```

  This operation returns all documents where the value of the `type` field is `food`, but does not include the `type` field in the output.

  With the exception of the `_id` field you cannot combine inclusion and exclusion statements in projection documents.

The `$elemMatch` and `$slice` projection operators provide more control when projecting only a portion of an array.

1.2 Indexes

Indexes improve the efficiency of read operations by reducing the amount of data that query operations need to process and thereby simplifying the work associated with fulfilling queries within MongoDB. The indexes themselves are a special data structure that MongoDB maintains when inserting or modifying documents, and any given index can: support and optimize specific queries, sort operations, and allow for more efficient storage utilization. For more information about indexes in MongoDB see: `http://docs.mongodb.org/manual/indexes` and `http://docs.mongodb.org/manual/core/indexes`.

You can create indexes using the `db.collection.ensureIndex()` method in the `mongo` shell, as in the following prototype operation:

```
db.collection.ensureIndex( { <field1>: <order>, <field2>: <order>, ... } )
```

- The `field` specifies the field to index. The field may be a field from a subdocument, using *dot notation* to specify subdocument fields.

  You can create an index on a single field or a *compound index* that includes multiple fields in the index.

- The `order` option is specifies either ascending (`1`) or descending (`-1`).

  MongoDB can read the index in either direction. In most cases, you only need to specify *indexing order* to support sort operations in compound queries.

1.2.1 Measuring Index Use

The `explain()` cursor method allows you to inspect the operation of the query system, and is useful for analyzing the efficiency of queries, and for determining how the query uses the index. Call the `explain()` method on a cursor returned by `find()`, as in the following example:

```
db.inventory.find( { type: 'food' } ).explain()
```

Note: Only use `explain()` to test the query operation, and *not* the timing of query performance. Because `explain()` attempts multiple query plans, it does not reflect accurate query performance.

If the above operation could not use an index, the output of `explain()` would resemble the following:

```
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 5,
  "nscannedObjects" : 4000006,
  "nscanned" : 4000006,
  "nscannedObjectsAllPlans" : 4000006,
  "nscannedAllPlans" : 4000006,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 2,
  "nChunkSkips" : 0,
  "millis" : 1591,
  "indexBounds" : { },
  "server" : "mongodb0.example.net:27017"
}
```

The `BasicCursor` value in the `cursor` field confirms that this query does not use an index. The `nscannedObjects` value shows that MongoDB must scan 4,000,006 documents to return only 5 documents. To increase the efficiency of the query, create an index on the `type` field, as in the following example:

```
db.inventory.ensureIndex( { type: 1 } )
```

Run the `explain()` operation, as follows, to test the use of the index:

```
db.inventory.find( { type: 'food' } ).explain()
```

Consider the results:

```
{
  "cursor" : "BtreeCursor type_1",
  "isMultiKey" : false,
  "n" : 5,
  "nscannedObjects" : 5,
  "nscanned" : 5,
  "nscannedObjectsAllPlans" : 5,
  "nscannedAllPlans" : 5,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : { "type" : [
                                [ "food",
                                  "food" ]
                              ] },
  "server" : "mongodbo0.example.net:27017" }
```

The `BtreeCursor` value of the `cursor` field indicates that the query used an index. This query:

- returned 5 documents, as indicated by the `n` field;
- scanned 5 documents from the index, as indicated by the `nscanned` field;
- then read 5 full documents from the collection, as indicated by the `nscannedObjects` field.

  This indicates that the query was not "covered," or able to complete only using the index, as reflected in the `indexOnly`. See *indexes-covered-queries* for more information.

1.2.2 Query Optimization

The MongoDB query optimizer processes queries and chooses the most efficient query plan for a query given the available indexes. The query system then uses this query plan each time the query runs. The query optimizer occasionally reevaluates query plans as the content of the collection changes to ensure optimal query plans.

To create a new query plan, the query optimizer:

1. runs the query against several indexes in parallel.
2. records the matches in a single common buffer, as though the results all came from the same index.

   If an index returns a result already returned by another index, the optimizer skips the duplicate match.
3. selects an index when either of the following occur:
   - The optimizer exhausts an index, which means that the index has provided the full result set. At this point, the optimizer stops querying.
   - The optimizer reaches 101 results. At this point, the optimizer chooses the index that has provided the most results *first* and continues reading only from that index.

The selected index becomes the index specified in the query plan; future iterations of this query or queries with the same query pattern will use this index. Query pattern refers to query select conditions that differ only in the values, as in the following two queries with the same query pattern:

```
db.inventory.find( { type: 'food' } )
db.inventory.find( { type: 'utensil' } )
```

To manually compare the performance of a query using more than one index, you can use the `hint()` and `explain()` methods in conjunction, as in the following prototype:

```
db.collection.find().hint().explain()
```

The following operations each run the same query but will reflect the use of the different indexes:

```
db.inventory.find( { type: 'food' } ).hint( { type: 1 } ).explain()
db.inventory.find( { type: 'food' } ).hint( { type: 1, name: 1 }).explain()
```

This returns the statistics regarding the execution of the query. For more information on the output of `explain()` see the `http://docs.mongodb.org/manual/reference/explain`.

Note: If you run `explain()` without including `hint()`, the query optimizer reevaluates the query and runs against multiple indexes before returning the query statistics.

As collections change over time, the query optimizer deletes a query plan and reevaluates the after any of the following events:

- the collection receives 1,000 write operations.
- the `reIndex` rebuilds the index.
- you add or drop an index.
- the `mongod` process restarts.

For more information, see `http://docs.mongodb.org/manual/applications/indexes`.

1.2.3 Query Operations that Cannot Use Indexes Effectively

Some query operations cannot use indexes effectively or cannot use indexes at all. Consider the following situations:

- The inequality operators `$nin` and `$ne` are not very selective, as they often match a large portion of the index.

  As a result, in most cases, a `$nin` or `$ne` query with an index may perform no better than a `$nin` or `$ne` query that must scan all documents in a collection.

- Queries that specify regular expressions, with inline JavaScript regular expressions or `$regex` operator expressions, cannot use an index. *However*, the regular expression with anchors to the beginning of a string *can* use an index.

1.3 Cursors

The `find()` method returns a *cursor* to the results; however, in the `mongo` shell, if the returned cursor is not assigned to a variable, then the cursor is automatically iterated up to 20 times [2] to print up to the first 20 documents that match the query, as in the following example:

[2] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`.

```
db.inventory.find( { type: 'food' } );
```

When you assign the `find()` to a variable:

- you can call the cursor variable in the shell to iterate up to 20 times [2] and print the matching documents, as in the following example:

  ```
  var myCursor = db.inventory.find( { type: 'food' } );

  myCursor
  ```

- you can use the cursor method `next()` to access the documents, as in the following example:

  ```
  var myCursor = db.inventory.find( { type: 'food' } );
  var myDocument = myCursor.hasNext() ? myCursor.next() : null;

  if (myDocument) {
      var myItem = myDocument.item;
      print(tojson(myItem));
  }
  ```

  As an alternative print operation, cosider the the `printjson()` helper method to replace `print(tojson())`:

  ```
  if (myDocument) {
      var myItem = myDocument.item;
      printjson(myItem);
  }
  ```

- you can use the cursor method `forEach()` to iterate the cursor and access the documents, as in the following example:

  ```
  var myCursor =  db.inventory.find( { type: 'food' } );

  myCursor.forEach(printjson);
  ```

See *JavaScript cursor methods* and your `driver` documentation for more information on cursor methods.

1.3.1 Iterator Index

In the `mongo` shell, you can use the `toArray()` method to iterate the cursor and return the documents in an array, as in the following:

```
var myCursor = db.inventory.find( { type: 'food' } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[3];
```

The `toArray()` method loads into RAM all documents returned by the cursor; the `toArray()` method exhausts the cursor.

Additionally, some `drivers` provide access to the documents by using an index on the cursor (i.e. `cursor[index]`). This is a shortcut for first calling the `toArray()` method and then using an index on the resulting array.

Consider the following example:

```
var myCursor = db.inventory.find( { type: 'food' } );
var myDocument = myCursor[3];
```

The `myCursor[3]` is equivalent to the following example:

```
myCursor.toArray() [3];
```

1.3.2 Cursor Behaviors

Consider the following behaviors related to cursors:

- By default, the server will automatically close the cursor after 10 minutes of inactivity or if client has exhausted the cursor. To override this behavior, you can specify the `noTimeout` wire protocol flag in your query; however, you should either close the cursor manually or exhaust the cursor. In the `mongo` shell, you can set the `noTimeout` flag:

  ```
  var myCursor = db.inventory.find().addOption(DBQuery.Option.noTimeout);
  ```

  See your `driver` documentation for information on setting the `noTimeout` flag. See *Cursor Flags* (page 17) for a complete list of available cursor flags.

- Because the cursor is not isolated during its lifetime, intervening write operations may result in a cursor that returns a single document [3] more than once. To handle this situation, see the information on snapshot mode.

- The MongoDB server returns the query results in batches:

  - For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. To override the default size of the batch, see `cursor.batchSize()` and `cursor.limit()`.
  - For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort and will return all documents in the first batch.
  - Batch size will not exceed the *maximum BSON document size*.
  - As you iterate through the cursor and reach the end of the returned batch, if there are more results, `cursor.next()` will perform a `getmore operation` to retrieve the next batch.

    To see how many documents remain in the batch as you iterate the cursor, you can use the `cursor.objsLeftInBatch()` method, as in the following example:

    ```
    var myCursor = db.inventory.find();

    var myFirstDocument = myCursor.hasNext() ? myCursor.next() : null;

    myCursor.objsLeftInBatch();
    ```

- You can use the command `cursorInfo` to retrieve the following information on cursors:

  - total number of open cursors
  - size of the client cursors in current use
  - number of timed out cursors since the last server restart

  Consider the following example:

  ```
  db.runCommand( { cursorInfo: 1 } )
  ```

  The result from the command returns the following documentation:

[3] A single document relative to value of the `_id` field. A cursor cannot return the same document more than once *if* the document has not changed.

```
{ "totalOpen" : <number>, "clientCursors_size" : <number>, "timedOut" : <number>, "ok" : 1 }
```

1.3.3 Cursor Flags

The `mongo` shell provides the following cursor flags:

- `DBQuery.Option.tailable`
- `DBQuery.Option.slaveOk`
- `DBQuery.Option.oplogReplay`
- `DBQuery.Option.noTimeout`
- `DBQuery.Option.awaitData`
- `DBQuery.Option.exhaust`
- `DBQuery.Option.partial`

1.3.4 Aggregation

Changed in version 2.2. MongoDB can perform some basic data aggregation operations on results before returning data to the application. These operations are not queries, they use *database commands* rather than queries, and they do not return a cursor; however, they are still require MongoDB to read data.

Running aggregation operations on the database side can be more efficient than running them in the application layer and can reduce the amount of data MongoDB needs to send to the application. These aggregation operations include basic grouping, counting, and even processing data using a map reduce framework. Additionally, in 2.2 MongoDB provides a complete aggregation framework for more rich aggregation operations.

The aggregation framework provides users with a "pipeline" like framework: documents enter from a collection and then pass through a series of steps by a sequence of *pipeline operators* that manipulate and transform the documents until they're output at the end. The aggregation framework is accessible via the `aggregate` command or the `db.collection.aggregate()` helper in the `mongo` shell.

For more information on the aggregation framework see `http://docs.mongodb.org/manual/aggregation`.

Additionally, MongoDB provides a number of simple data aggregation operations for more basic data aggregation operations:

- `count()`
- `distinct()`
- `group()`
- `mapReduce()` (See also MapReduce.)

1.4 Architecture

1.4.1 Read Operations From Sharded Clusters

Sharded clusters allow you to partition a data set among a cluster of program:*mongod* in a way that is nearly transparent to the application. See the `http://docs.mongodb.org/manual/sharding` section of this manual for additional information about these deployments.

For a sharded cluster, you issue all operations to one of the `mongos` instances associated with the cluster. `mongos` instances route operations to the `mongod` in the cluster and behave like `mongod` instances to the application. Read operations to a sharded collection in a sharded cluster are largely the same as operations to a *replica set* or *standalone* instances. See the section on *Read Operations in Sharded Clusters* for more information.

In sharded deployments, the `mongos` instance routes the queries from the clients to the `mongod` instances that hold the data, using the cluster metadata stored in the *config database*.

For sharded collections, if queries do not include the *shard key*, the `mongos` must direct the query to all shards in a collection. These *scatter gather* queries can be inefficient, particularly on larger clusters, and are unfeasible for routine operations.

For more information on read operations in sharded clusters, consider the following resources:

- *An Introduction to Shard Keys*
- *Shard Key Internals and Operations*
- *Querying Sharded Clusters*
- *sharding-mongos*

1.4.2 Read Operations from Replica Sets

Replica sets use *read preferences* to determine where and how to route read operations to members of the replica set. By default, MongoDB always reads data from a replica set's *primary*. You can modify that behavior by changing the *read preference mode*.

You can configure the *read preference mode* on a per-connection or per-operation basis to allow reads from *secondaries* to:

- reduce latency in multi-data-center deployments,
- improve read throughput by distributing high read-volumes (relative to write volume),
- for backup operations, and/or
- to allow reads during *failover* situations.

Read operations from secondary members of replica sets are not guaranteed to reflect the current state of the primary, and the state of secondaries will trail the primary by some amount of time. Often, applications don't rely on this kind of strict consistency, but application developers should always consider the needs of their application before setting read preference.

For more information on *read preferences* or on the read preference modes, see *read-preference* and *replica-set-read-preference-modes*.

CHAPTER

TWO

WRITE OPERATIONS

All operations that create or modify data in the MongoDB instance are write operations. MongoDB represents data as *BSON documents* stored in *collections*. Write operations target one collection and are atomic on the level of a single document: no single write operation can atomically affect more than one document or more than one collection.

This document introduces the write operators available in MongoDB as well as presents strategies to increase the efficiency of writes in applications.

2.1 Write Operators

For information on write operators and how to write data to a MongoDB database, see the following pages:

- *Create* (page 37)
- *Update* (page 53)
- *Delete* (page 59)

For information on specific methods used to perform write operations in the `mongo` shell, see the following:

- `db.collection.insert()`
- `db.collection.update()`
- `db.collection.save()`
- `db.collection.findAndModify()`
- `db.collection.remove()`

For information on how to perform write operations from within an application, see the `http://docs.mongodb.org/manual/applications/drivers` documentation or the documentation for your client library.

2.2 Write Concern

Note: The `driver write concern` change created a new connection class in all of the MongoDB drivers, called `MongoClient` with a different default write concern. See the `release notes` for this change, and the release notes for the driver you're using for more information about your driver's release.

2.2.1 Operational Considerations and Write Concern

Clients issue write operations with some level of *write concern*, which describes the level of concern or guarantee the server will provide in its response to a write operation. Consider the following levels of conceptual write concern:

- *errors ignored*: Write operations are not acknowledged by MongoDB, and may not succeed in the case of connection errors that the client is not yet aware of, or if the `mongod` produces an exception (e.g. a duplicate key exception for *unique indexes*.) While this operation is efficient because it does not require the database to respond to every write operation, it also incurs a significant risk with regards to the persistence and durability of the data.

  Warning: Do not use this option in normal operation.

- *unacknowledged*: MongoDB does not acknowledge the receipt of write operation as with a write concern level of *ignore*; however, the driver will receive and handle network errors, as possible given system networking configuration.

  Before the releases outlined in *driver-write-concern-change*, this was the default write concern.

- receipt *acknowledged*: The `mongod` will confirm the receipt of the write operation, allowing the client to catch network, duplicate key, and other exceptions. After the releases outlined in *driver-write-concern-change*, this is the default write concern. [1]

- *journaled*: The `mongod` will confirm the write operation only after it has written the operation to the *journal*. This confirms that the write operation can survive a `mongod` shutdown and ensures that the write operation is durable.

  While receipt *acknowledged* without *journaled* provides the fundamental basis for write concern, there is an up-to 100 millisecond window between journal commits where the write operation is not fully durable. Require *journaled* as part of the write concern to provide this durability guarantee.

Replica sets present an additional layer of consideration for write concern. Basic write concern level affect the write operation on only one `mongod` instance. The `w` argument to `getLastError` provides a *replica acknowledged* level of write concern. With *replica acknowledged* you can guarantee that the write operation has propagated to the members of a replica set. See the *Write Concern for Replica Sets* for more information.

Note: Requiring *journaled* write concern in a replica set only requires a journal commit of the write operation to the *primary* of the set regardless of the level of *replica acknowledged* write concern.

2.2.2 Internal Operation of Write Concern

To provide write concern, `drivers` issue the `getLastError` command after a write operation and receive a document with information about the last operation. This document's `err` field contains either:

- `null`, which indicates the write operations have completed successfully, or
- a description of the last error encountered.

The definition of a "successful write" depends on the arguments specified to `getLastError`, or in replica sets, the configuration of `getLastErrorDefaults`. When deciding the level of write concern for your application, become familiar with the *Operational Considerations and Write Concern* (page 20).

The `getLastError` command has the following options to configure write concern requirements:

[1] The default write concern is to call `getLastError` with no arguments. For replica sets, you can define the default write concern settings in the `getLastErrorDefaults`. If `getLastErrorDefaults` does not define a default write concern setting, `getLastError` defaults to basic receipt acknowledgment.

- `j` or "journal" option

  This option confirms that the `mongod` instance has written the data to the on-disk journal and ensures data is not lost if the `mongod` instance shuts down unexpectedly. Set to `true` to enable, as shown in the following example:

  ```
  db.runCommand( { getLastError: 1, j: "true" } )
  ```

  If you set `journal` to true, and the `mongod` does not have journaling enabled, as with `nojournal`, then `getLastError` will provide basic receipt acknowledgment, and will include a `jnote` field in its return document.

- `w` option

  This option provides the ability to disable write concern entirely *as well as* specifies the write concern operations for *replica sets*. See *Operational Considerations and Write Concern* (page 20) for an introduction to the fundamental concepts of write concern. By default, the `w` option is set to `1`, which provides basic receipt acknowledgment on a single `mongod` instance or on the *primary* in a replica set.

  The `w` option takes the following values:

  - `-1`:

    Disables all acknowledgment of write operations, and suppresses all including network and socket errors.

  - `0`:

    Disables basic acknowledgment of write operations, but returns information about socket excepts and networking errors to the application.

    Note: If you disable basic write operation acknowledgment but require journal commit acknowledgment, the journal commit prevails, and the driver will require that `mongod` will acknowladge the replica set.

  - `1`:

    Provides acknowledgment of write operations on a standalone `mongod` or the *primary* in a replica set.

  - *A number greater than 1*:

    Guarantees that write operations have propagated successfully to the specified number of replica set members including the primary. If you set `w` to a number that is greater than the number of set members that hold data, MongoDB waits for the non-existent members to become available, which means MongoDB blocks indefinitely.

  - `majority`:

    Confirms that write operations have propagated to the majority of configured replica set: nodes must acknowledge the write operation before it succeeds. This ensures that write operation will *never* be subject to a rollback in the course of normal operation, and furthermore allows you to prevent hard coding assumptions about the size of your replica set into your application.

  - *A tag set*:

    By specifying a *tag set* you can have fine-grained control over which replica set members must acknowledge a write operation to satisfy the required level of write concern.

`getLastError` also supports a `wtimeout` setting which allows clients to specify a timeout for the write concern: if you don't specify `wtimeout` and the `mongod` cannot fulfill the write concern the `getLastError` will block, potentially forever.

For more information on write concern and replica sets, see *Write Concern for Replica Sets* for more information..

In sharded clusters, `mongos` instances will pass write concern on to the shard `mongod` instances.

2.3 Bulk Inserts

In some situations you may need to insert or ingest a large amount of data into a MongoDB database. These *bulk inserts* have some special considerations that are different from other write operations.

The `insert()` method, when passed an array of documents, will perform a bulk insert, and inserts each document atomically. `Drivers` provide their own interface for this kind of operation. New in version 2.2: `insert()` in the `mongo` shell gained support for bulk inserts in version 2.2. Bulk insert can significantly increase performance by amortizing *write concern* (page 19) costs. In the drivers, you can configure write concern for batches rather than on a per-document level.

Drivers also have a `ContinueOnError` option in their insert operation, so that the bulk operation will continue to insert remaining documents in a batch even if an insert fails.

Note: New in version 2.0: Support for `ContinueOnError` depends on version 2.0 of the core `mongod` and `mongos` components.

If the bulk insert process generates more than one error in a batch job, the client will only receive the most recent error. All bulk operations to a *sharded collection* run with `ContinueOnError`, which applications cannot disable. See *sharding-bulk-inserts* section for more information on consideration for bulk inserts in sharded clusters.

For more information see your `driver documentation` for details on performing bulk inserts in your application. Also consider the following resources: *Sharded Clusters* (page 23), *sharding-bulk-inserts*, and `http://docs.mongodb.org/manual/administration/import-export`.

2.4 Indexing

After every insert, update, or delete operation, MongoDB must update *every* index associated with the collection in addition to the data itself. Therefore, every index on a collection adds some amount of overhead for the performance of write operations. [2]

In general, the performance gains that indexes provide for *read operations* are worth the insertion penalty; however, when optimizing write performance, be careful when creating new indexes and always evaluate the indexes on the collection and ensure that your queries are actually using these indexes.

For more information on indexes in MongoDB consider `http://docs.mongodb.org/manual/indexes` and `http://docs.mongodb.org/manual/applications/indexes`.

2.5 Isolation

When a single write operation modifies multiple documents, the operation as a whole is not atomic, and other operations may interleave. The modification of a single document, or record, is always atomic, even if the write operation modifies multiple sub-document *within* the single record.

No other operations are atomic; however, you can attempt to isolate a write operation that affects multiple documents using the `isolation operator`.

To isolate a sequence of write operations from other read and write operations, see `http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits`.

[2] The overhead for *sparse indexes* inserts and updates to un-indexed fields is less than for non-sparse indexes. Also for non-sparse indexes, updates that don't change the record size have less indexing overhead.

2.6 Architecture

2.6.1 Replica Sets

In *replica sets*, all write operations go to the set's *primary*, which applies the write operation then records the operations on the primary's operation log or *oplog*. The oplog is a reproducible sequence of operations to the data set. *Secondary* members of the set are continuously replicating the oplog and applying the operations to themselves in an asynchronous process.

Large volumes of write operations, particularly bulk operations, may create situations where the secondary members have difficulty applying the replicating operations from the primary at a sufficient rate: this can cause the secondary's state to fall behind that of the primary. Secondaries that are significantly behind the primary present problems for normal operation of the replica set, particularly *failover* in the form of *rollbacks* as well as general *re2ad consistency*.

To help avoid this issue, you can customize the *write concern* (page 19) to return confirmation of the write operation to another member [3] of the replica set every 100 or 1,000 operations. This provides an opportunity for secondaries to catch up with the primary. Write concern can slow the overall progress of write operations but ensure that the secondaries can maintain a largely current state with respect to the primary.

For more information on replica sets and write operations, see *replica-set-write-concern*, *replica-set-oplog-sizing*, *replica-set-oplog*, and *replica-set-procedure-change-oplog-size*.

2.6.2 Sharded Clusters

In a *sharded cluster*, MongoDB directs a given write operation to a *shard* and then performs the write on a particular *chunk* on that shard. Shards and chunks are range-based. *Shard keys* affect how MongoDB distributes documents among shards. Choosing the correct shard key can have a great impact on the performance, capability, and functioning of your database and cluster.

For more information, see `http://docs.mongodb.org/manual/administration/sharding` and *Bulk Inserts* (page 22).

[3] Calling `getLastError` intermittently with a `w` value of `2` or `majority` will slow the throughput of write traffic; however, this practice will allow the secondaries to remain current with the state of the primary.

CHAPTER

THREE

BSON DOCUMENTS

MongoDB is a document-based database system, and as a result, all records, or data, in MongoDB are documents. Documents are the default representation of most user accessible data structures in the database. Documents provide structure for data in the following MongoDB contexts:

- the *records* (page 27) stored in *collections*
- the *query selectors* (page 28) that determine which records to select for read, update, and delete operations
- the *update actions* (page 28) that specify the particular field updates to perform during an update operation
- the specification of *indexes* (page 29) for collection.
- arguments to several MongoDB methods and operators, including:
  - *sort order* (page 30) for the `sort()` method.
  - *index specification* (page 29) for the `hint()` method.
- the output of a number of MongoDB commands and operations, including:
  - the `output` of `collStats` command, and
  - the `output` of the `serverStatus` command.

3.1 Structure

The document structure in MongoDB are *BSON* objects with support for the full range of *BSON types*; however, BSON documents are conceptually, similar to *JSON* objects, and have the following structure:

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

Having support for the full range of BSON types, MongoDB documents may contain field and value pairs where the value can be another document, an array, an array of documents as well as the basic types such as `Double`, `String`, and `Date`. See also *BSON Type Considerations* (page 30).

Consider the following document that contains values of varying types:

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

The document contains the following fields:

- `_id` that holds an *ObjectId*.
- `name` that holds a *subdocument* that contains the fields `first` and `last`.
- `birth` and `death`, which both have *Date* types.
- `contribs` that holds an *array of strings*.
- `views` that holds a value of *NumberLong* type.

All field names are strings in *BSON* documents. Be aware that there are some `restrictions on field names` for *BSON* documents: field names cannot contain null characters, dots (`.`), or dollar signs (`$`).

3.1.1 Type Operators

To determine the type of fields, the `mongo` shell provides the following operators:

- *instanceof*' returns a boolean to test if a value has a specific type.
- `typeof` returns the type of a field.

Example

Consider the following operations using `instanceof` and `typeof`:

- The following operation tests whether the `_id` field is of type `ObjectId`:

  ```
  mydoc._id instanceof ObjectId
  ```

  The operation returns `true`.

- The following operation returns the type of the `_id` field:

  ```
  typeof mydoc._id
  ```

  In this case `typeof` will return the more generic `object` type rather than `ObjectId` type.

3.1.2 Dot Notation

MongoDB uses the *dot notation* to access the elements of an array and to access the fields of a subdocument.

To access an element of an array by the zero-based index position, you concatenate the array name with the dot (`.`) and zero-based index position:

```
'<array>.<index>'
```

To access a field of a subdocument with *dot-notation*, you concatenate the subdocument name with the dot (`.`) and the field name:

'<subdocument>.<field>'

See Also:

- *Subdocuments* (page 9) for dot notation examples with subdocuments.
- *Arrays* (page 10) for dot notation examples with arrays.

3.2 Document Types in MongoDB

3.2.1 Record Documents

Most documents in MongoDB in *collections* store data from users' applications.

These documents have the following attributes:

- The maximum BSON document size is 16 megabytes.

  The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` and the documentation for your `driver` for more information about GridFS.

- *Documents* (page 25) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.

Note: Most MongoDB driver clients will include the `_id` field and generate an `ObjectId` before sending the insert operation to MongoDB; however, if the client sends a document without an `_id` field, the `mongod` will add the `_id` field and generate the `ObjectId`.

The following document specifies a record in a collection:

```
{
  _id: 1,
  name: { first: 'John', last: 'Backus' },
  birth: new Date('Dec 03, 1924'),
  death: new Date('Mar 17, 2007'),
  contribs: [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
  awards: [
            { award: 'National Medal of Science',
              year: 1975,
              by: 'National Science Foundation' },
            { award: 'Turing Award',
              year: 1977,
              by: 'ACM' }
          ]
}
```

The document contains the following fields:

- `_id`, which must hold a unique value and is *immutable*.

- `name` that holds another *document*. This sub-document contains the fields `first` and `last`, which both hold *strings*.
- `birth` and `death` that both have *date* types.
- `contribs` that holds an *array of strings*.
- `awards` that holds an *array of documents*.

Consider the following behavior and constraints of the `_id` field in MongoDB documents:

- In documents, the `_id` field is always indexed for regular collections.
- The `_id` field may contain values of any BSON data type other than an array.

Consider the following options for the value of an `_id` field:

- Use an `ObjectId`. See the `ObjectId` documentation.

  Although it is common to assign `ObjectId` values to `_id` fields, if your objects have a natural unique identifier, consider using that for the value of `_id` to save space and to avoid an additional index.
- Generate a sequence number for the documents in your collection in your application and use this value for the `_id` value. See the `http://docs.mongodb.org/manual/tutorial/create-an-auto-incrementing-field` tutorial for an implementation pattern.
- Generate a UUID in your application code. For efficiency, store the UUID as a value of the BSON `BinData` type to reduce the size of UUID values as stored in the collection and in the `_id` index.
- Use your driver's BSON UUID facility to generate UUIDs. Be aware that driver implementations may implement UUID serialization and deserialization logic differently, which may not be fully compatible with other drivers. See your driver documentation for information concerning UUID interoperability.

3.2.2 Query Specification Documents

Query documents specify the conditions that determine which records to select for read, update, and delete operations. You can use `<field>:<value>` expressions to specify the equality condition and `query operator` expressions to specify additional conditions.

When passed as an argument to methods such as the `find()` method, the `remove()` method, or the `update()` method, the query document selects documents for MongoDB to return, remove, or update, as in the following:

```
db.bios.find( { _id: 1 } )
db.bios.remove( { _id: { $gt: 3 } } )
db.bios.update( { _id: 1, name: { first: 'John', last: 'Backus' } },
                <update>,
                <options> )
```

See Also:

- *Query Document* (page 8) and *Read* (page 45) for more examples on selecting documents for reads.
- *Update* (page 53) for more examples on selecting documents for updates.
- *Delete* (page 59) for more examples on selecting documents for deletes.

3.2.3 Update Specification Documents

Update documents specify the data modifications to perform during an `update()` operation to modify existing records in a collection. You can use *update operators* to specify the exact actions to perform on the document fields.

Consider the update document example:

```
{
  $set: { 'name.middle': 'Warner' },
  $push: { awards: { award: 'IBM Fellow',
                     year: '1963',
                     by: 'IBM' }
         }
}
```

When passed as an argument to the `update()` method, the update actions document:

- Modifies the field `name` whose value is another document. Specifically, the `$set` operator updates the `middle` field in the `name` subdocument. The document uses *dot notation* (page 26) to access a field in a subdocument.
- Adds an element to the field `awards` whose value is an array. Specifically, the `$push` operator adds another document as element to the field `awards`.

```
db.bios.update(
   { _id: 1 },
   {
     $set: { 'name.middle': 'Warner' },
     $push: { awards: {
                        award: 'IBM Fellow',
                        year: '1963',
                        by: 'IBM'
                      }
            }
   }
)
```

See Also:

- *update operators* page for the available update operators and syntax.
- *update* (page 53) for more examples on update documents.

For additional examples of updates that involve array elements, including where the elements are documents, see the `$` positional operator.

3.2.4 Index Specification Documents

Index specification documents describe the fields to index on during the `index creation`. See `indexes` for an overview of indexes. [1]

Index documents contain field and value pairs, in the following form:

```
{ field: value }
```

- `field` is the field in the documents to index.
- `value` is either 1 for ascending or -1 for descending.

The following document specifies the *multi-key index* on the `_id` field and the `last` field contained in the subdocument `name` field. The document uses *dot notation* (page 26) to access a field in a subdocument:

```
{ _id: 1, 'name.last': 1 }
```

When passed as an argument to the `ensureIndex()` method, the index documents specifies the index to create:

[1] Indexes optimize a number of key *read* (page 7) and *write* (page 19) operations.

```
db.bios.ensureIndex( { _id: 1, 'name.last': 1 } )
```

3.2.5 Sort Order Specification Documents

Sort order documents specify the order of documents that a `query()` returns. Pass sort order specification documents as an argument to the `sort()` method. See the `sort()` page for more information on sorting.

The sort order documents contain field and value pairs, in the following form:

```
{ field: value }
```

- `field` is the field by which to sort documents.
- `value` is either 1 for ascending or -1 for descending.

The following document specifies the sort order using the fields from a sub-document `name` first sort by the `last` field ascending, then by the `first` field also ascending:

```
{ 'name.last': 1, 'name.first': 1 }
```

When passed as an argument to the `sort()` method, the sort order document sorts the results of the `find()` method:

```
db.bios.find().sort( { 'name.last': 1, 'name.first': 1 } )
```

3.3 BSON Type Considerations

The following BSON types require special consideration:

3.3.1 ObjectId

ObjectIds are: small, likely unique, fast to generate, and ordered. These values consists of 12-bytes, where the first 4-bytes is a timestamp that reflects the ObjectId's creation. Refer to the `ObjectId` documentation for more information.

3.3.2 String

BSON strings are UTF-8. In general, drivers for each programming language convert from the language's string format to UTF-8 when serializing and deserializing BSON. This makes it possible to store most international characters in BSON strings with ease. [2] In addition, MongoDB `$regex` queries support UTF-8 in the regex string.

3.3.3 Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular *Date* (page 31) type. Timestamp values are a 64 bit value where:

- the first 32 bits are a `time_t` value (seconds since the Unix epoch)
- the second 32 bits are an incrementing `ordinal` for operations within a given second.

[2] Given strings using UTF-8 character sets, using `sort()` on strings will be reasonably correct; however, because internally `sort()` uses the C++ `strcmp` api, the sort order may handle some characters incorrectly.

Within a single `mongod` instance, timestamp values are always unique.

In replication, the *oplog* has a `ts` field. The values in this field reflect the operation time, which uses a BSON timestamp value.

Note: The BSON Timestamp type is for *internal* MongoDB use. For most cases, in application development, you will want to use the BSON date type. See *Date* (page 31) for more information.

If you create a BSON Timestamp using the empty constructor (e.g. `new Timestamp()`), MongoDB will only generate a timestamp *if* you use the constructor in the first field of the document. [3] Otherwise, MongoDB will generate an empty timestamp value (i.e. `Timestamp(0, 0)`.) Changed in version 2.1: `mongo` shell displays the Timestamp value with the wrapper:

```
Timestamp(<time_t>, <ordinal>)
```

Prior to version 2.1, the `mongo` shell display the Timestamp value as a document:

```
{ t : <time_t>, i : <ordinal> }
```

3.3.4 Date

BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). The official BSON specification refers to the BSON Date type as the *UTC datetime*. Changed in version 2.0: BSON Date type is signed. [4] Negative values represent dates before 1970. Consider the following examples of BSON Date:

- Construct a Date using the `new Date()` constructor in the `mongo` shell:

  ```
  var mydate1 = new Date()
  ```

- Construct a Date using the `ISODate()` constructor in the `mongo` shell:

  ```
  var mydate2 = ISODate()
  ```

- Return the `Date` value as string:

  ```
  mydate1.toString()
  ```

- Return the month portion of the Date value; months are zero-indexed, so that January is month `0`:

  ```
  mydate1.getMonth()
  ```

[3] If the first field in the document is `_id`, then you can generate a timestamp in the *second* field of a document.

[4] Prior to version 2.0, `Date` values were incorrectly interpreted as *unsigned* integers, which affected sorts, range queries, and indexes on `Date` fields. Because indexes are not recreated when upgrading, please re-index if you created an index on `Date` values with an earlier version, and dates before 1970 are relevant to your application.

Part II

CRUD Operations

These documents provide an overview and examples of CRUD operations in MongoDB.

CHAPTER

FOUR

CREATE

Of the four basic database operations (i.e. CRUD), *create* operations are those that add new records or *documents* to a *collection* in MongoDB. For general information about write operations and the factors that affect their performance, see *Write Operations* (page 19); for documentation of the other CRUD operations, see the *CRUD* (page 1) page.

4.1 Overview

You can create documents in a MongoDB collection using any of the following basic operations.

- *insert* (page 38)
- *create with save* (page 42)
- *create with upsert* (page 43)

All insert operations in MongoDB exhibit the following properties:

- If you attempt to insert a document without the *_id* field, the client library *or* the `mongod` instance will add an `_id` field and populate the field with a unique *ObjectId*.
- For operations with *write concern* (page 19), if you specify an `_id` field, the `_id` field must be unique within the collection; otherwise the `mongod` will return a duplicate key exception.
- The maximum BSON document size is 16 megabytes.

  The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` and the documentation for your `driver` for more information about GridFS.
- *Documents* (page 25) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.

Note: As of these *driver versions*, all write operations will issue a `getLastError` command to confirm the result of the write operation:

```
{ getLastError: 1 }
```

Refer to the documentation on *write concern* (page 19) in the *Write Operations* (page 19) document for more information.

4.2 Insert

The `insert()` is the primary method to insert a document or documents into a MongoDB collection, and has the following syntax:

```
db.collection.insert( <document> )
```

Corresponding Operation in SQL

The `insert()` method is analogous to the `INSERT` statement.

Consider the following examples that illustrate the behavior of `insert()`:

- If the collection does not exist [1], then the `insert()` method creates the collection during the first insert. Specifically in the example, if the collection `bios` does not exist , then the insert operation will create this collection:

```
db.bios.insert(
   {
      _id: 1,
      name: { first: 'John', last: 'Backus' },
      birth: new Date('Dec 03, 1924'),
      death: new Date('Mar 17, 2007'),
      contribs: [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
      awards: [
                {
                  award: 'W.W. McDowell Award',
                  year: 1967,
                  by: 'IEEE Computer Society'
                },
                {
                  award: 'National Medal of Science',
                  year: 1975,
                  by: 'National Science Foundation'
                },
                {
                  award: 'Turing Award',
                  year: 1977,
                  by: 'ACM'
                },
                {
                  award: 'Draper Prize',
                  year: 1993,
                  by: 'National Academy of Engineering'
                }
      ]
   }
)
```

  You can confirm the insert by *querying* (page 45) the `bios` collection:

[1] You can also view a list of the existing collections in the database using the `show collections` operation in the `mongo` shell.

```
db.bios.find()
```

This operation returns the following document from the `bios` collection:

```
{
  "_id" : 1,
  "name" : { "first" : "John", "last" : "Backus" },
  "birth" : ISODate("1924-12-03T05:00:00Z"),
  "death" : ISODate("2007-03-17T04:00:00Z"),
  "contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
  "awards" : [
                {
                  "award" : "W.W. McDowell Award",
                  "year" : 1967,
                  "by" : "IEEE Computer Society"
                },
                {
                  "award" : "National Medal of Science",
                  "year" : 1975,
                  "by" : "National Science Foundation"
                },
                {
                  "award" : "Turing Award",
                  "year" : 1977,
                  "by" : "ACM"
                },
                { "award" : "Draper Prize",
                  "year" : 1993,
                  "by" : "National Academy of Engineering"
                }
  ]
}
```

- If the new document does not contain an `_id` field, then the `insert()` method adds the `_id` field to the document and generates a unique `ObjectId` for the value.

```
db.bios.insert(
   {
      name: { first: 'John', last: 'McCarthy' },
      birth: new Date('Sep 04, 1927'),
      death: new Date('Dec 24, 2011'),
      contribs: [ 'Lisp', 'Artificial Intelligence', 'ALGOL' ],
      awards: [
                {
                  award: 'Turing Award',
                  year: 1971,
                  by: 'ACM'
                },
                {
                  award: 'Kyoto Prize',
                  year: 1988,
                  by: 'Inamori Foundation'
                },
                {
                  award: 'National Medal of Science',
                  year: 1990,
                  by: 'National Science Foundation'
                }
      ]
```

```
    }
)
```

You can verify the inserted document by the querying the `bios` collection:

```
db.bios.find( { name: { first: 'John', last: 'McCarthy' } } )
```

The returned document contains an `_id` field with the generated `ObjectId` value:

```
{
  "_id" : ObjectId("50a1880488d113a4ae94a94a"),
  "name" : { "first" : "John", "last" : "McCarthy" },
  "birth" : ISODate("1927-09-04T04:00:00Z"),
  "death" : ISODate("2011-12-24T05:00:00Z"),
  "contribs" : [ "Lisp", "Artificial Intelligence", "ALGOL" ],
  "awards" : [
                {
                  "award" : "Turing Award",
                  "year" : 1971,
                  "by" : "ACM"
                },
                {
                  "award" : "Kyoto Prize",
                  "year" :1988,
                  "by" : "Inamori Foundation"
                },
                {
                  "award" : "National Medal of Science",
                  "year" : 1990,
                  "by" : "National Science Foundation"
                }
  ]
}
```

- If you pass an array of documents to the `insert()` method, the `insert()` performs a bulk insert into a collection.

  The following operation inserts three documents into the `bios` collection. The operation also illustrates the *dynamic schema* characteristic of MongoDB. Although the document with `_id: 3` contains a field `title` which does not appear in the other documents, MongoDB does not require the other documents to contain this field:

```
db.bios.insert(
   [
      {
        _id: 3,
        name: { first: 'Grace', last: 'Hopper' },
        title: 'Rear Admiral',
        birth: new Date('Dec 09, 1906'),
        death: new Date('Jan 01, 1992'),
        contribs: [ 'UNIVAC', 'compiler', 'FLOW-MATIC', 'COBOL' ],
        awards: [
                  {
                    award: 'Computer Sciences Man of the Year',
                    year: 1969,
                    by: 'Data Processing Management Association'
                  },
                  {
                    award: 'Distinguished Fellow',
```

```
                year: 1973,
                by: ' British Computer Society'
              },
              {
                award: 'W. W. McDowell Award',
                year: 1976,
                by: 'IEEE Computer Society'
              },
              {
                award: 'National Medal of Technology',
                year: 1991,
                by: 'United States'
              }
    ]
  },
  {
    _id: 4,
    name: { first: 'Kristen', last: 'Nygaard' },
    birth: new Date('Aug 27, 1926'),
    death: new Date('Aug 10, 2002'),
    contribs: [ 'OOP', 'Simula' ],
    awards: [
              {
                award: 'Rosing Prize',
                year: 1999,
                by: 'Norwegian Data Association'
              },
              {
                award: 'Turing Award',
                year: 2001,
                by: 'ACM'
              },
              {
                award: 'IEEE John von Neumann Medal',
                year: 2001,
                by: 'IEEE'
              }
    ]
  },
  {
    _id: 5,
    name: { first: 'Ole-Johan', last: 'Dahl' },
    birth: new Date('Oct 12, 1931'),
    death: new Date('Jun 29, 2002'),
    contribs: [ 'OOP', 'Simula' ],
    awards: [
              {
                award: 'Rosing Prize',
                year: 1999,
                by: 'Norwegian Data Association'
              },
              {
                award: 'Turing Award',
                year: 2001,
                by: 'ACM'
              },
              {
                award: 'IEEE John von Neumann Medal',
```

```
                    year: 2001,
                    by: 'IEEE'
                  }
            ]
        }
    ]
)
```

4.3 Create with Save

The `save()` method is a specialized upsert that use the `_id` field in the `<document>` argument to determine whether to perform an insert or an update:

- If the `<document>` argument does not contain the `_id` field or contains an `_id` field with a value not in the collection, the `save()` method performs an insert of the document.
- Otherwise, the `save()` method performs an update.

The `save()` method has the following syntax:

```
db.collection.save( <document> )
```

Consider the following examples that illustrate the use of the `save()` method to perform inserts:

- If the `<document>` does not contain the `_id` field, the `save()` method performs an insert. Refer to the *insert* (page 38) section for details of the insert operation of a document without an `_id` field.

  The following operation performs an insert into the `bios` collection since the document does not contain the `_id` field:

  ```
  db.bios.save(
     {
        name: { first: 'Guido', last: 'van Rossum'},
        birth: new Date('Jan 31, 1956'),
        contribs: [ 'Python' ],
        awards: [
                  {
                    award: 'Award for the Advancement of Free Software',
                    year: 2001,
                    by: 'Free Software Foundation'
                  },
                  {
                    award: 'NLUUG Award',
                    year: 2003,
                    by: 'NLUUG'
                  }
        ]
     }
  )
  ```

- If the `<document>` contains an `_id` field but has a value not found in the collection, the `save()` method performs an insert. Refer to the *insert* (page 38) section for details of the insert operation.

  The following operation performs an insert into the `bios` collection since the document contains an `_id` field whose value `10` is not found in the `bios` collection:

  ```
  db.bios.save(
     {
  ```

```
        _id: 10,
        name: { first: 'Yukihiro', aka: 'Matz', last: 'Matsumoto'},
        birth: new Date('Apr 14, 1965'),
        contribs: [ 'Ruby' ],
        awards: [
                  {
                    award: 'Award for the Advancement of Free Software',
                    year: '2011',
                    by: 'Free Software Foundation'
                  }
        ]
      }
)
```

4.4 Create with Upsert

An *upsert* eliminates the need to perform a separate database call to check for the existence of a record before performing either an update or an insert operation. Typically update operations *update* (page 53) existing documents, but in MongoDB, the `update()` operation can accept an `<upsert>` option as an argument. Upserts are a hybrid operation that use the `<query>` argument to determine the write operation:

- If the query matches an existing document(s), the upsert performs an update.
- If the query matches no document in the collection, the upsert inserts a single document.

Consider the following syntax for an upsert operation:

```
db.collection.update( <query>,
                      <update>,
                      { upsert: true } )
```

The following examples illustrate the use of the upsert to perform create operations:

- If no document matches the `<query>` argument, the `upsert` performs an insert. If the `<update>` argument includes only field and value pairs, the new document contains the fields and values specified in the `<update>` argument. If the `_id` field is omitted, the operation adds the `_id` field and generates a unique `ObjectId` for its value.

  The following `upsert` operation inserts a new document into the `bios` collection:

  ```
  db.bios.update(
     { name: { first: 'Dennis', last: 'Ritchie'} },
     {
       name: { first: 'Dennis', last: 'Ritchie'},
       birth: new Date('Sep 09, 1941'),
       died: new Date('Oct 12, 2011'),
       contribs: [ 'UNIX', 'C' ],
       awards: [
                 {
                   award: 'Turing Award',
                   year: 1983,
                   by: 'ACM'
                 },
                 {
                   award: 'National Medal of Technology',
                   year: 1998,
                   by: 'United States'
                 },
  ```

```
                {
                  award: 'Japan Prize',
                  year: 2011,
                  by: 'The Japan Prize Foundation'
                }
    ]
  },
  { upsert: true }
)
```

- If no document matches the <query> argument, the upsert operation inserts a new document. If the <update> argument includes only *update operators*, the new document contains the fields and values from <query> argument with the operations from the <update> argument applied.

  The following operation inserts a new document into the bios collection:

```
db.bios.update(
  {
    _id: 7,
    name: { first: 'Ken', last: 'Thompson' }
  },
  {
    $set: {
             birth: new Date('Feb 04, 1943'),
             contribs: [ 'UNIX', 'C', 'B', 'UTF-8' ],
             awards: [
                       {
                         award: 'Turing Award',
                         year: 1983,
                         by: 'ACM'
                       },
                       {
                         award: 'IEEE Richard W. Hamming Medal',
                         year: 1990,
                         by: 'IEEE'
                       },
                       {
                         award: 'National Medal of Technology',
                         year: 1998,
                         by: 'United States'
                       },
                       {
                         award: 'Tsutomu Kanai Award',
                         year: 1999,
                         by: 'IEEE'
                       },
                       {
                         award: 'Japan Prize',
                         year: 2011,
                         by: 'The Japan Prize Foundation'
                       }
           ]
         }
  },
  { upsert: true }
)
```

CHAPTER

FIVE

READ

Of the four basic database operations (i.e. CRUD), read operation are those that retrieve records or *documents* from a *collection* in MongoDB. For general information about read operations and the factors that affect their performance, see *Read Operations* (page 7); for documentation of the other CRUD operations, see the *CRUD* (page 1) page.

5.1 Overview

You can retrieve documents from MongoDB using either of the following methods:

- *find* (page 45)
- *findOne* (page 51)

5.2 Find

The `find()` method is the primary method to select documents from a collection. The `find()` method returns a cursor that contains a number of documents. Most `drivers` provide application developers with a native iterable interface for handling cursors and accessing documents. The `find()` method has the following syntax:

```
db.collection.find( <query>, <projection> )
```

Corresponding Operation in SQL

The `find()` method is analogous to the `SELECT` statement, while:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<projection>` argument corresponds to the list of fields to select from the result set.

Consider the following examples that illustrate the use of the `find()` method:

The examples refer to a collection named `bios` that contains documents with the following prototype:

```
{
  "_id" : 1,
  "name" : {
             "first" : "John",
             "last" :"Backus"
           },
  "birth" : ISODate("1924-12-03T05:00:00Z"),
  "death" : ISODate("2007-03-17T04:00:00Z"),
```

```
    "contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
    "awards" : [
                {
                  "award" : "W.W. McDowellAward",
                  "year" : 1967,
                  "by" : "IEEE Computer Society"
                },
                {
                  "award" : "National Medal of Science",
                  "year" : 1975,
                  "by" : "National Science Foundation"
                },
                {
                  "award" : "Turing Award",
                  "year" : 1977,
                  "by" : "ACM"
                },
                {
                  "award" : "Draper Prize",
                  "year" : 1993,
                  "by" : "National Academy of Engineering"
                }
    ]
}
```

Note: In the `mongo` shell, you can format the output by adding `.pretty()` to the `find()` method call.

- If there is no <query> argument, the `find()` method selects all documents from a collection.
  - The following operation returns all documents (or more precisely, a cursor to all documents) in the `bios` collection:

    ```
    db.bios.find()
    ```

- If there is a <query> argument, the `find()` method selects all documents from a collection that satisfies the criteria of the query:
  - The following operation returns all documents in the `bios` collection where the field `_id` equals `5` or `ObjectId("507c35dd8fada716c89d0013")`:

    ```
    db.bios.find(
       {
         _id: { $in: [ 5,  ObjectId("507c35dd8fada716c89d0013") ] }
       }
    )
    ```

  - The following operation returns all documents in the `bios` collection where the array field `contribs` contains the element `'UNIX'`:

    ```
    db.bios.find(
       {
         contribs: 'UNIX'
       }
    )
    ```

  - The following operation returns all documents in the `bios` collection where `awards` array contains a subdocument element that contains the `award` field equal to `'Turing Award'` and the `year` field greater than 1980:

```
db.bios.find(
   {
     awards: {
               $elemMatch: {
                    award: 'Turing Award',
                    year: { $gt: 1980 }
               }
     }
   }
)
```

– The following operation returns all documents in the `bios` collection where the subdocument `name` contains a field `first` with the value `'Yukihiro'` and a field `last` with the value `'Matsumoto'`; the query uses *dot notation* to access fields in a subdocument:

```
db.bios.find(
   {
     'name.first': 'Yukihiro',
     'name.last': 'Matsumoto'
   }
)
```

The query matches the document where the `name` field contains a subdocument with the field `first` with the value `'Yukihiro'` and a field `last` with the value `'Matsumoto'`. For instance, the query would match documents with `name` fields that held either of the following values:

```
{
  first: 'Yukihiro',
  aka: 'Matz',
  last: 'Matsumoto'
}

{
  last: 'Matsumoto',
  first: 'Yukihiro'
}
```

– The following operation returns all documents in the `bios` collection where the subdocument `name` is *exactly* `{ first: 'Yukihiro', last: 'Matsumoto' }`, including the order:

```
db.bios.find(
   {
     name: {
              first: 'Yukihiro',
              last: 'Matsumoto'
     }
   }
)
```

The `name` field must match the sub-document exactly, including order. For instance, the query would **not** match documents with `name` fields that held either of the following values:

```
{
  first: 'Yukihiro',
  aka: 'Matz',
  last: 'Matsumoto'
}

{
```

```
  last: 'Matsumoto',
  first: 'Yukihiro'
}
```

– The following operation returns all documents in the `bios` collection where either the field `first` in the sub-document `name` starts with the letter `G` **or** where the field `birth` is less than `new Date('01/01/1945')`:

```
db.bios.find(
   { $or: [
            { 'name.first' : /^G/ },
            { birth: { $lt: new Date('01/01/1945') } }
          ]
   }
)
```

– The following operation returns all documents in the `bios` collection where the field `first` in the sub-document `name` starts with the letter `K` **and** the array field `contribs` contains the element `UNIX`:

```
db.bios.find(
   {
     'name.first': /^K/,
     contribs: 'UNIX'
   }
)
```

In this query, the parameters (i.e. the selections of both fields) combine using an implicit logical AND for criteria on different fields `contribs` and `name.first`. For multiple AND criteria on the same field, use the `$and` operator.

- If there is a `<projection>` argument, the `find()` method returns only those fields as specified in the `<projection>` argument to include or exclude:

Note: The `_id` field is implicitly included in the `<projection>` argument. In projections that explicitly include fields, `_id` is the only field that you can explicitly exclude. Otherwise, you cannot mix include field and exclude field specifications.

– The following operation finds all documents in the `bios` collection and returns only the `name` field, the `contribs` field, and the `_id` field:

```
db.bios.find(
   { },
   { name: 1, contribs: 1 }
)
```

– The following operation finds all documents in the `bios` collection and returns only the `name` field and the `contribs` field:

```
db.bios.find(
   { },
   { name: 1, contribs: 1, _id: 0 }
)
```

– The following operation finds the documents in the `bios` collection where the `contribs` field contains the element `'OOP'` and returns all fields *except* the `_id` field, the `first` field in the `name` subdocument, and the `birth` field from the matching documents:

```
db.bios.find(
   { contribs: 'OOP' },
   { _id: 0, 'name.first': 0, birth: 0 }
)
```

– The following operation finds all documents in the `bios` collection and returns the the `last` field in the `name` subdocument and the first two elements in the `contribs` field:

```
db.bios.find(
   { },
   {
     _id: 0,
     'name.last': 1,
     contribs: { $slice: 2 }
   }
)
```

See Also:

- *dot notation* for information on "reaching into" embedded sub-documents.
- *Arrays* (page 10) for more examples on accessing arrays
- *Subdocuments* (page 9) for more examples on accessing subdocuments

5.2.1 Cursor

The `find()` method returns a *cursor* to the results; however, in the `mongo` shell, if the returned cursor is not assigned to a variable, then the cursor is automatically iterated up to 20 times [1] to print up to the first 20 documents that match the query, as in the following example:

```
db.bios.find( { _id: 1 } );
```

When you assign the `find()` to a variable:

- you can type the name of the cursor variable to iterate up to 20 times [1] and print the matching documents, as in the following example:

```
var myCursor = db.bios.find( { _id: 1 } );

myCursor
```

- you can use the cursor method `next()` to access the documents, as in the following example:

```
var myCursor = db.bios.find( { _id: 1 } );

var myDocument = myCursor.hasNext() ? myCursor.next() : null;

if (myDocument) {

   var myName = myDocument.name;

   print (tojson(myName));
}
```

To print, you can also use the `printjson()` method instead of `print(tojson())`:

[1] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *Cursor Flags* (page 17) and *Cursor Behaviors* (page 16) for more information.

```
if (myDocument) {

    var myName = myDocument.name;

    printjson(myName);
}
```

- you can use the cursor method `forEach()` to iterate the cursor and access the documents, as in the following example:

```
var myCursor = db.bios.find( { _id: 1 } );

myCursor.forEach(printjson);
```

For more information on cursor handling, see:

- `cursor.hasNext()`
- `cursor.next()`
- `cursor.forEach()`
- *cursors* (page 14)
- *JavaScript cursor methods*

Modify Cursor Behavior

In addition to the `<query>` and the `<projection>` arguments, the `mongo` shell and the `drivers` provide several cursor methods that you can call on the *cursor* returned by `find()` method to modify its behavior, such as:

- `sort`, which orders the documents in the result set according to the field or fields specified to the method.

  The following operation returns all documents (or more precisely, a cursor to all documents) in the `bios` collection ordered by the `name` field ascending:

  ```
  db.bios.find().sort( { name: 1 } )
  ```

  `sort()` corresponds to the `ORDER BY` statement in SQL.

- The `limit()` method limits the number of documents in the result set.

  The following operation returns at most `5` documents (or more precisely, a cursor to at most 5 documents) in the `bios` collection:

  ```
  db.bios.find().limit( 5 )
  ```

  `limit()` corresponds to the `LIMIT` statement in SQL.

- The `skip()` method controls the starting point of the results set.

  The following operation returns all documents, skipping the first `5` documents in the `bios` collection:

  ```
  db.bios.find().skip( 5 )
  ```

You can chain these cursor methods, as in the following examples [2]:

```
db.bios.find().sort( { name: 1 } ).limit( 5 )
db.bios.find().limit( 5 ).sort( { name: 1 } )
```

[2] Regardless of the order you chain the `limit()` and the `sort()`, the request to the server has the following structure that treats the query and the `sort()` modifier as a single object. Therefore, the `limit()` operation method is always applied after the `sort()` regardless of the specified order of the operations in the chain. See the `meta query operators` for more information.

See the *JavaScript cursor methods* reference and your `driver` documentation for additional references. See *Cursors* (page 14) for more information regarding cursors.

5.3 Find One

The `findOne()` method selects and returns a single document from a collection and returns that document. `findOne()` does *not* return a cursor.

The `findOne()` method has the following syntax:

```
db.collection.findOne( <query>, <projection> )
```

Except for the return value, `findOne()` method is quite similar to the `find()` method; in fact, internally, the `findOne()` method is the `find()` method with a limit of 1.

Consider the following examples that illustrate the use of the `findOne()` method:

- If there is no `<query>` argument, the `findOne()` method selects just one document from a collection.
  - The following operation returns a single document from the `bios` collection:

    ```
    db.bios.findOne()
    ```

- If there is a `<query>` argument, the `findOne()` method selects the first document from a collection that meets the `<query>` argument:
  - The following operation returns the first matching document from the `bios` collection where either the field `first` in the subdocument `name` starts with the letter `G` **or** where the field `birth` is less than `new Date('01/01/1945')`:

    ```
    db.bios.findOne(
       {
         $or: [
                { 'name.first' : /^G/ },
                { birth: { $lt: new Date('01/01/1945') } }
              ]
       }
    )
    ```

- You can pass a `<projection>` argument to `findOne()` to control the fields included in the result set:
  - The following operation finds a document in the `bios` collection and returns only the `name` field, the `contribs` field, and the `_id` field:

    ```
    db.bios.findOne(
       { },
       { name: 1, contribs: 1 }
    )
    ```

  - The following operation returns a document in the `bios` collection where the `contribs` field contains the element `OOP` and returns all fields *except* the `_id` field, the `first` field in the `name` subdocument, and the `birth` field from the matching documents:

    ```
    db.bios.findOne(
       { contribs: 'OOP' },
       { _id: 0, 'name.first': 0, birth: 0 }
    )
    ```

Although similar to the `find()` method, because the `findOne()` method returns a document rather than a cursor, you cannot apply the cursor methods such as `limit()`, `sort()`, and `skip()` to the result of the `findOne()` method. However, you can access the document directly, as in the following example:

```
var myDocument = db.bios.findOne();

if (myDocument) {
   var myName = myDocument.name;

   print (tojson(myName));
}
```

CHAPTER

SIX

UPDATE

Of the four basic database operations (i.e. CRUD), *update* operations are those that modify existing records or *documents* in a MongoDB *collection*. For general information about write operations and the factors that affect their performance, see *Write Operations* (page 19); for documentation of other CRUD operations, see the *CRUD* (page 1) page.

6.1 Overview

Update operation modifies an existing *document* or documents in a *collection*. MongoDB provides the following methods to perform update operations:

- *update* (page 53)
- *save* (page 56)

Note: Consider the following behaviors of MongoDB's update operations.

- When performing update operations that increase the document size beyond the allocated space for that document, the update operation relocates the document on disk and may reorder the document fields depending on the type of update.
- As of these *driver versions*, all write operations will issue a `getLastError` command to confirm the result of the write operation:

  ```
  { getLastError: 1 }
  ```

  Refer to the documentation on *write concern* (page 19) in the *Write Operations* (page 19) document for more information.

6.2 Update

The `update()` method is the primary method used to modify documents in a MongoDB collection. By default, the `update()` method updates a **single** document, but by using the `multi` option, `update()` can update all documents that match the query criteria in the collection. The `update()` method can either replace the existing document with the new document or update specific fields in the existing document.

The `update()` has the following syntax:

```
db.collection.update( <query>, <update>, <options> )
```

Corresponding operation in SQL

The `update()` method corresponds to the `UPDATE` operation in SQL, and:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<update>` corresponds to the `SET ...` statement.

The default behavior of the `update()` method updates a **single** document and would correspond to the SQL `UPDATE` statement with the `LIMIT 1`. With the `multi` option, `update()` method would correspond to the SQL `UPDATE` statement without the `LIMIT` clause.

Consider the following examples that illustrate the use of the `update()` method:

- If the `<update>` argument contains only *update operator* expressions such as the `$set` operator expression, the `update()` method updates the corresponding fields in the document. To update fields in subdocuments, MongoDB uses *dot notation*.

  The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and sets the field named `middle` to the value `Warner` in the `name` subdocument and adds a new element to the `awards` field:

  ```
  db.bios.update(
     { _id: 1 },
     {
       $set: { 'name.middle': 'Warner' },
       $push: { awards: { award: 'IBM Fellow', year: 1963, by: 'IBM' } }
     }
  )
  ```

- If the `<update>` argument contains `$unset` operator, the `update()` method removes the field from the document.

  The following operation queries the `bios` collection for the first document that has an `_id` field equal to `3` and removes the `birth` field from the document:

  ```
  db.bios.update(
     { _id: 3 },
     { $unset: { birth: 1 } }
  )
  ```

- If the `<update>` argument contains fields not currently in the document, the `update()` method adds the new fields to the document.

  The following operation queries the `bios` collection for the first document that has an `_id` field equal to `3` and adds to the document a new `mbranch` field and a new `aka` field in the subdocument `name`:

  ```
  db.bios.update(
     { _id: 3 },
     { $set: {
               mbranch: 'Navy',
               'name.aka': 'Amazing Grace'
             }
     }
  )
  ```

- If the `<update>` argument contains only field and value pairs, the `update()` method *replaces* the existing document with the document in the `<update>` argument, except for the `_id` field.

The following operation queries the `bios` collection for the first document that has a `name` field equal to `{ first: 'John', last: 'McCarthy' }` and replaces all but the `_id` field in the document with the fields in the `<update>` argument:

```
db.bios.update(
   { name: { first: 'John', last: 'McCarthy' } },
   { name: { first: 'Ken', last: 'Iverson' },
     born: new Date('Dec 17, 1941'),
     died: new Date('Oct 19, 2004'),
     contribs: [ 'APL', 'J' ],
     awards: [
               { award: 'Turing Award',
                 year: 1979,
                 by: 'ACM' },
               { award: 'Harry H. Goode Memorial Award',
                 year: 1975,
                 by: 'IEEE Computer Society' },
               { award: 'IBM Fellow',
                 year: 1970,
                 by: 'IBM' }
     ]
   }
)
```

- If the update operation requires an update of an element in an array field:
  - The `update()` method can perform the update using the position of the element and *dot notation*. Arrays in MongoDB are zero-based.

    The following operation queries the `bios` collection for the first document with `_id` field equal to `1` and updates the second element in the `contribs` array:

    ```
    db.bios.update(
       { _id: 1 },
       { $set: { 'contribs.1': 'ALGOL 58' } }
    )
    ```

  - The `update()` method can perform the update using the `$` positional operator if the position is not known. The array field must appear in the `query` argument in order to determine which array element to update.

    The following operation queries the `bios` collection for the first document where the `_id` field equals `3` and the `contribs` array contains an element equal to `compiler`. If found, the `update()` method updates the first matching element in the array to `A compiler` in the document:

    ```
    db.bios.update(
       { _id: 3, 'contribs': 'compiler' },
       { $set: { 'contribs.$': 'A compiler' } }
    )
    ```

  - The `update()` method can perform the update of an array that contains subdocuments by using the positional operator (i.e. `$`) and the *dot notation*.

    The following operation queries the `bios` collection for the first document where the `_id` field equals `6` and the `awards` array contains a subdocument element with the `by` field equal to `ACM`. If found, the `update()` method updates the `by` field in the first matching subdocument:

    ```
    db.bios.update(
       { _id: 6, 'awards.by': 'ACM'  } ,
    ```

```
        { $set: { 'awards.$.by': 'Association for Computing Machinery' } }
    )
```

- If the <options> argument contains the multi option set to true or 1, the update() method updates all documents that match the query.

  The following operation queries the bios collection for all documents where the awards field contains a sub-document element with the award field equal to Turing and sets the turing field to true in the matching documents:

```
db.bios.update(
   { 'awards.award': 'Turing' },
   { $set: { turing: true } },
   { multi: true }
)
```

- If you set the upsert option in the <options> argument to true or 1 and no existing document match the <query> argument, the update() method can insert a new document into the collection.

  The following operation queries the bios collection for a document with the _id field equal to 11 and the name field equal to { first: 'James', last: 'Gosling'}. If the query selects a document, the operation performs an update operation. If a document is not found, update() inserts a new document containing the fields and values from <query> argument with the operations from the <update> argument applied. [1]

```
db.bios.update(
   { _id:11, name: { first: 'James', last: 'Gosling' } },
   {
     $set: {
              born: new Date('May 19, 1955'),
              contribs: [ 'Java' ],
              awards: [
                        { award: 'The Economist Innovation Award',
                          year: 2002,
                          by: 'The Economist' },
                        { award: 'Officer of the Order of Canada',
                          year: 2007,
                          by: 'Canada' }
            ]
          }
   },
   { upsert: true }
)
```

  See also *Create with Upsert* (page 43).

6.3 Save

The save() method updates an existing document or inserts a document depending on the _id field of the document. The save() method is analogous to the update() method with the upsert option and a <query> argument on the _id field.

[1] If the <update> argument includes only field and value pairs, the new document contains the fields and values specified in the <update> argument. If the <update> argument includes only *update operators*, the new document contains the fields and values from <query> argument with the operations from the <update> argument applied.

The `save()` method has the following syntax:

```
db.collection.save( <document> )
```

Consider the following examples of the `save()` method:

- If the `<document>` argument contains the `_id` field that exists in the collection, the `save()` method performs an update that replaces the existing document with the `<document>` argument.

  The following operation queries the `bios` collection for a document where the `_id` equals `ObjectId("507c4e138fada716c89d0014")` and replaces the document with the `<document>` argument:

  ```
  db.bios.save(
     {
       _id: ObjectId("507c4e138fada716c89d0014"),
       name: { first: 'Martin', last: 'Odersky' },
       contribs: [ 'Scala' ]
     }
  )
  ```

- If no `_id` field exists or if the `_id` field exists but does not match any document in the collection, the `save()` method performs an insert.

  The following operation adds the `_id` field to the document, assigns to the field a unique *ObjectId*, and inserts the document into the `bios` collection:

  ```
  db.bios.save(
     {
       name: { first: 'Larry', last: 'Wall' },
       contribs: [ 'Perl' ]
     }
  )
  ```

  See Also:

  Create with Save (page 42).

CHAPTER

SEVEN

DELETE

Of the four basic database operations (i.e. CRUD), *delete* operations are those that remove documents from a *collection* in MongoDB.

For general information about write operations and the factors that affect their performance, see *Write Operations* (page 19); for documentation of other CRUD operations, see the *CRUD* (page 1) page.

7.1 Overview

The *remove()* (page 59) method in the `mongo` shell provides this operation, as do corresponding methods in the `drivers`.

Note: As of these *driver versions*, all write operations will issue a `getLastError` command to confirm the result of the write operation:

```
{ getLastError: 1 }
```

Refer to the documentation on *write concern* (page 19) in the *Write Operations* (page 19) document for more information.

7.2 Remove

Use the `remove()` method to delete documents from a collection; this action does not remove the indexes. [1]

The `remove()` method has the following syntax:

```
db.collection.remove( <query>, <justOne> )
```

Corresponding operation in SQL

The `remove()` method is analogous to the `DELETE` statement, and:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<justOne>` argument has the same affect as `LIMIT 1`.

Consider the following examples that illustrate the use of the `remove()`:

[1] To remove all documents from a collection, it may be faster to use the `drop()` method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

- If there is a <query> argument, the remove() method deletes from the collection all documents that match the argument.

  The following operation deletes all documents from the bios collection where the subdocument name contains a field first whose value starts with G:

  ```
  db.bios.remove( { 'name.first' : /^G/ } )
  ```

- If there is a <query> argument and you specify the <justOne> argument as true or 1, remove() only deletes a single document from the collection that matches the query.

  The following operation deletes a single document from the bios collection where the turing field equals true:

  ```
  db.bios.remove( { turing: true }, 1 )
  ```

- If there is no <query> argument, the remove() method deletes all documents from a collection. The following operation deletes all documents from the bios collection:

  ```
  db.bios.remove()
  ```

  Note: This operation is not equivalent to the drop() method.

7.2.1 Capped Collection

You cannot apply the remove() method to a *capped collection*.

7.2.2 Isolation

If the <query> argument to the remove() method matches multiple documents in the collection, the delete operation may interleave with other write operations to that collection. For an unsharded collection, you have the option to override this behavior with the $atomic isolation operator, effectively isolating the delete operation from other write operations. To isolate the operation, include $atomic: 1 in the <query> parameter as in the following example:

```
db.bios.remove( { turing: true, $atomic: 1 } )
```

INDEX

C

connection pooling
    read operations, 17

Q

query optimizer, 13

R

read operation
    architecture, 17
    connection pooling, 17
read operations
    query, 7

W

write concern, 19
write operators, 19