MongoDB CRUD Operations Introduction

Release 2.2.7

MongoDB Documentation Project

April 15, 2014

Contents

CRUD stands for *create*, *read*, *update*, and *delete*, which are the four core database operations used in database driven application development. The *CRUD Operations for MongoDB* (page 43) section provides introduction to each class of operation along with complete examples of each operation. The documents in the *Read and Write Operations in MongoDB* (page 3) section provide a higher level overview of the behavior and available functionality of these operations.

Read and Write Operations in MongoDB

The *Read Operations* (page 3) and *Write Operations* (page 15) documents provide higher level introductions and description of the behavior and operations of read and write operations for MongoDB deployments. The *BSON Documents* (page 27) provides an overview of *documents* and document-orientation in MongoDB.

1.1 Read Operations

Read operations include all operations that return a cursor in response to application request data (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[1] for full API documentation.

1.1.1 Queries in MongoDB

In the `mongo` shell, the `find()` and `findOne()` methods perform read operations. The `find()` method has the following syntax: [2]

```
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 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.

[1] http://api.mongodb.org/

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

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()`).

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 51) page of the *Core MongoDB Operations (CRUD)* (page 1) section.

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 3).

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 sub-documents, you query for specific fields within the sub-documents 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/manualreference/operator` document for the complete list of query operators.

Result Projections

The *projection* specification limits the fields to return for all matching documents. 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.

1.1.2 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/manualindexes` and `http://docs.mongodb.org/manualcore/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.

Covering a Query

An index *covers* a query, a *covered query*, when:

- all the fields in the *query* (page 4) are part of that index, **and**
- all the fields returned in the documents that match the query are in the same index.

For these queries, MongoDB does not need to inspect at documents outside of the index, which is often more efficient than inspecting entire documents.

Example

Given a collection `inventory` with the following index on the `type` and `item` fields:

```
{ type: 1, item: 1 }
```

This index will cover the following query on the `type` and `item` fields, which returns only the `item` field:

```
db.inventory.find( { type: "food", item:/^c/ },
                   { item: 1, _id: 0 } )
```

However, this index will **not** cover the following query, which returns the `item` field **and** the `_id` field:

```
db.inventory.find( { type: "food", item:/^c/ },
                   { item: 1 } )
```

See *indexes-covered-queries* for more information on the behavior and use of covered queries.

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 `explain.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.

  Although the query uses an index to find the matching documents, if `indexOnly` is false then an index could not *cover* (page 8) the query: MongoDB could not both match the *query conditions* (page 4) **and** return the results using only this index. See *indexes-covered-queries* for more information.

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 candidate indexes in parallel.
2. records the matches in a common results buffer or buffers.
   - If the candidate plans include only *ordered query plans*, there is a single common results buffer.
   - If the candidate plans include only *unordered query plans*, there is a single common results buffer.
   - If the candidate plans include *both ordered query plans* and *unordered query plans*, there are two common results buffers, one for the ordered plans and the other for the unordered plans.

   If an index returns a result already returned by another index, the optimizer skips the duplicate match. In the case of the two buffers, both buffers are de-duped.
3. stops the testing of candidate plans and selects an index when one of the following events occur:
   - An *unordered query plan* has returned all the matching results; *or*
   - An *ordered query plan* has returned all the matching results; *or*
   - An *ordered query plan* has returned a threshold number of matching results:
     - Version 2.0: Threshold is the query batch size. The default batch size is 101.
     - Version 2.2: Threshold is 101.

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/manualreference/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/manualapplications/indexes`.

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.

1.1.3 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 [3] to print up to the first 20 documents that match the query, as in the following example:

```
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, consider the `printjson()` helper method to replace `print(tojson())`:

[3] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *mongo-shell-executing-queries* for more information.

```
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.

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];
```

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[4] 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 13) 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 [5] more than once. To handle this situation, see the information on *snapshot mode*.

[4] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol

[5] 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.

- 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 `batchSize()` and `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 `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:

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

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`

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 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/manualaggregation`.

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

- `count` (`count()`)
- `distinct` (`db.collection.distinct()`)
- `group` (`db.collection.group()`)
- `mapReduce`. (Also consider `mapReduce()` and `http://docs.mongodb.org/manualapplications/map-reduce`

1.1.4 Architecture

Read Operations from Sharded Clusters

Sharded clusters allow you to partition a data set among a cluster of `mongod` in a way that is nearly transparent to the application. See the `http://docs.mongodb.org/manualsharding` 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*

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*.

1.2 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.

1.2.1 Write Operators

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

- *Create* (page 43)
- *Update* (page 61)
- *Delete* (page 67)

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/manualapplications/drivers` documentation or the documentation for your client library.

1.2.2 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.

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. [6]

- *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 a window between journal commits where the write operation is not fully durable. See `journalCommitInterval` for more information on this window. 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 levels 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* document 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.

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:

[6] 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.

- `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 16).

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

- `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 16) 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 acknowledge 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.

1.2.3 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 16) 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 21), *sharding-bulk-inserts*, and `http://docs.mongodb.org/manualadministration/import-export`.

1.2.4 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. [7]

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/manualindexes` and `http://docs.mongodb.org/manualapplications/indexes`.

[7] 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.

1.2.5 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/manualtutorial/perform-two-phase-commits`.

1.2.6 Updates

Each document in a MongoDB collection has allocated *record* space which includes the entire document *and* a small amount of padding. This padding makes it possible for update operations to increase the size of a document slightly without causing the document to outgrow the allocated record size.

Documents in MongoDB can grow up to the full maximum `BSON document size`. However, when documents outgrow their allocated record size MongoDB must allocate a new record and move the document to the new record. Update operations that do not cause a document to grow, (i.e. *in-place* updates,) are significantly more efficient than those updates that cause document growth. Use *data models* (page 23) that minimize the need for document growth when possible.

For complete examples of update operations, see *Update* (page 61).

1.2.7 Padding Factor

If an update operation does not cause the document to increase in size, MongoDB can apply the update in-place. Some updates change the size of the document, for example using the `$push` operator to append a sub-document to an array can cause the top level document to grow beyond its allocated space.

When documents grow, MongoDB relocates the document on disk with enough contiguous space to hold the document. These relocations take longer than in-place updates, particularly if the collection has indexes that MongoDB must update all index entries. If collection has many indexes, the move will impact write throughput.

To minimize document movements, MongoDB employs padding. MongoDB adaptively learns if documents in a collection tend to grow, and if they do, adds a `paddingFactor` so that the documents have room to grow on subsequent writes. The `paddingFactor` indicates the padding for new inserts and moves.

New in version 2.2: You can use the `collMod` command with the `usePowerOf2Sizes` flag so that MongoDB allocates document space in sizes that are powers of 2. This helps ensure that MongoDB can efficiently reuse the space freed as a result of deletions or document relocations. As with all padding, using document space allocations with power of 2 sizes minimizes, but does not eliminate, document movements.

To check the current `paddingFactor` on a collection, you can run the `db.collection.stats()` operation in the `mongo` shell, as in the following example:

```
db.myCollection.stats()
```

Since MongoDB writes each document at a different point in time, the padding for each document will not be the same. You can calculate the padding size by subtracting 1 from the `paddingFactor`, for example:

```
padding size = (paddingFactor - 1) * <document size>.
```

For example, a `paddingFactor` of `1.0` specifies no padding whereas a paddingFactor of `1.5` specifies a padding size of `0.5` or 50 percent (50%) of the document size.

Because the `paddingFactor` is relative to the size of each document, you cannot calculate the exact amount of padding for a collection based on the average document size and padding factor.

If an update operation causes the document to *decrease* in size, for instance if you perform an `$unset` or a `$pop` update, the document remains in place and effectively has more padding. If the document remains this size, the space is not reclaimed until you perform a `compact` or a `repairDatabase` operation.

Note: The following operations remove padding:

- `compact`,
- `repairDatabase`, and
- initial replica sync operations.

However, with the `compact` command, you can run the command with a `paddingFactor` or a `paddingBytes` parameter.

Padding is also removed if you use `mongoexport` from a collection. If you use `mongoimport` into a new collection, `mongoimport` will not add padding. If you use `mongoimport` with an existing collection with padding, `mongoimport` will not affect the existing padding.

When a database operation removes padding, subsequent update that require changes in record sizes will have reduced throughput until the collection's padding factor grows. Padding does not affect in-place, and after `compact`, `repairDatabase`, and replica set initial sync the collection will require less storage.

See also:

- *faq-developers-manual-padding*
- Fast Updates with MongoDB with in-place Updates[8] (blog post)

1.2.8 Architecture

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 *read consistency*.

To help avoid this issue, you can customize the *write concern* (page 16) to return confirmation of the write operation to another member [9] 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*.

[8] http://blog.mongodb.org/post/248614779/fast-updates-with-mongodb-update-in-place

[9] 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.

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/manualadministration/sharded-clusters` and *Bulk Inserts* (page 18).

CHAPTER 2

Document Orientation Concepts

2.1 Data Modeling Considerations for MongoDB Applications

2.1.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. This means that:

- documents in the same collection do not need to have the same set of fields or structure, and
- common fields in a collection's documents may hold different types of data.

Each document only needs to contain relevant fields to the entity or object that the document represents. In practice, *most* documents in a collection share a similar structure. Schema flexibility means that you can model your documents in MongoDB so that they can closely resemble and reflect application-level objects.

As in all data modeling, when developing data models (i.e. *schema designs,*) for MongoDB you must consider the inherent properties and requirements of the application objects and the relationships between application objects. MongoDB data models must also reflect:

- how data will grow and change over time, and
- the kinds of queries your application will perform.

These considerations and requirements force developers to make a number of multi-factored decisions when modeling data, including:

- normalization and de-normalization.

  These decisions reflect degree to which the data model should store related pieces of data in a single document **or** should the data model describe relationships using *references* (page 36) between documents.

- `indexing strategy`.
- representation of data in arrays in *BSON*.

Although a number of data models may be functionally equivalent for a given application; however, different data models may have significant impacts on MongoDB and applications performance.

This document provides a high level overview of these data modeling decisions and factors. In addition, consider, the *Data Modeling Patterns and Examples* (page 27) section which provides more concrete examples of all the discussed patterns.

2.1.2 Data Modeling Decisions

Data modeling decisions involve determining how to structure the documents to model the data effectively. The primary decision is whether to *embed* (page 24) or to *use references* (page 24).

Embedding

To de-normalize data, store two related pieces of data in a single *document*.

Operations within a document are less expensive for the server than operations that involve multiple documents.

In general, use embedded data models when:

- you have "contains" relationships between entities. See *Model Embedded One-to-One Relationships Between Documents* (page 71).
- you have one-to-many relationships where the "many" objects always appear with or are viewed in the context of their parent documents. See *Model Embedded One-to-Many Relationships Between Documents* (page 72).

Embedding provides the following benefits:

- generally better performance for read operations.
- the ability to request and retrieve related data in a single database operation.

Embedding related data in documents, can lead to situations where documents grow after creation. Document growth can impact write performance and lead to data fragmentation. Furthermore, documents in MongoDB must be smaller than the `maximum BSON document size`. For larger documents, consider using *GridFS* (page 38).

See also:

- *dot notation* for information on "reaching into" embedded sub-documents.
- *Arrays* (page 6) for more examples on accessing arrays.
- *Subdocuments* (page 5) for more examples on accessing subdocuments.

Referencing

To normalize data, store *references* (page 36) between two documents to indicate a relationship between the data represented in each document.

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets. See *data-modeling-trees*.

Referencing provides more flexibility than embedding; however, to resolve the references, client-side applications must issue follow-up queries. In other words, using references requires more roundtrips to the server.

See *Model Referenced One-to-Many Relationships Between Documents* (page 73) for an example of referencing.

Atomicity

MongoDB only provides atomic operations on the level of a single document. [1] As a result needs for atomic operations influence decisions to use embedded or referenced relationships when modeling data for MongoDB.

Embed fields that need to be modified together atomically in the same document. See *Model Data for Atomic Operations* (page 75) for an example of atomic updates within a single document.

2.1.3 Operational Considerations

In addition to normalization and normalization concerns, a number of other operational factors help shape data modeling decisions in MongoDB. These factors include:

- data lifecycle management,
- number of collections and
- indexing requirements,
- sharding, and
- managing document growth.

These factors implications for database and application performance as well as future maintenance and development costs.

Data Lifecycle Management

Data modeling decisions should also take data lifecycle management into consideration.

The `Time to Live or TTL feature` of collections expires documents after a period of time. Consider using the TTL feature if your application requires some data to persist in the database for a limited period of time.

Additionally, if your application only uses recently inserted documents consider `http://docs.mongodb.org/manualcore/capped-collections`. Capped collections provide *first-in-first-out* (FIFO) management of inserted documents and optimized to support operations that insert and read documents based on insertion order.

Large Number of Collections

In certain situations, you might choose to store information in several collections rather than in a single collection.

Consider a sample collection `logs` that stores log documents for various environment and applications. The `logs` collection contains documents of the following form:

```
{ log: "dev", ts: ..., info: ... }
{ log: "debug", ts: ..., info: ...}
```

If the total number of documents is low you may group documents into collection by type. For logs, consider maintaining distinct log collections, such as `logs.dev` and `logs.debug`. The `logs.dev` collection would contain only the documents related to the dev environment.

Generally, having large number of collections has no significant performance penalty and results in very good performance. Distinct collections are very important for high-throughput batch processing.

When using models that have a large number of collections, consider the following behaviors:

[1] Document-level atomic operations include all operations within a single MongoDB document record: operations that affect multiple sub-documents within that single record are still atomic.

- Each collection has a certain minimum overhead of a few kilobytes.
- Each index, including the index on `_id`, requires at least 8KB of data space.

A single `<database>.ns` file stores all meta-data for each *database*. Each index and collection has its own entry in the namespace file, MongoDB places `limits on the size of namespace files`.

Because of `limits on namespaces`, you may wish to know the current number of namespaces in order to determine how many additional namespaces the database can support, as in the following example:

```
db.system.namespaces.count()
```

The `<database>.ns` file defaults to 16 MB. To change the size of the `<database>.ns` file, pass a new size to *`--nssize option <new size MB>`* on server start.

The *`--nssize`* sets the size for *new* `<database>.ns` files. For existing databases, after starting up the server with *`--nssize`*, run the `db.repairDatabase()` command from the `mongo` shell.

Indexes

Create indexes to support common queries. Generally, indexes and index use in MongoDB correspond to indexes and index use in relational database: build indexes on fields that appear often in queries and for all operations that return sorted results. MongoDB automatically creates a unique index on the `_id` field.

As you create indexes, consider the following behaviors of indexes:

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive as each insert must add keys to each index.
- Collections with high proportion of read operations to write operations often benefit from additional indexes. Indexes do not affect un-indexed read operations.

See `http://docs.mongodb.org/manualapplications/indexes` for more information on determining indexes. Additionally, the MongoDB `database profiler` may help identify inefficient queries.

Sharding

Sharding allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of `mongod` instances or *shards*.

The shard key determines how MongoDB distributes data among shards in a sharded collection. Selecting the proper *shard key* has significant implications for performance.

See `http://docs.mongodb.org/manualcore/sharded-clusters` for more information on sharding and the selection of the *shard key*.

Document Growth

Certain updates to documents can increase the document size, such as pushing elements to an array and adding new fields. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. This internal relocation can be both time and resource consuming.

Although MongoDB automatically provides padding to minimize the occurrence of relocations, you may still need to manually handle document growth. Refer to Pre-Aggregated Reports Use Case Study[2] for an example of the *Pre-allocation* approach to handle document growth.

[2] http://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports

2.1.4 Data Modeling Patterns and Examples

The following documents provide overviews of various data modeling patterns and common schema design considerations:

- *Model Embedded One-to-One Relationships Between Documents* (page 71)
- *Model Embedded One-to-Many Relationships Between Documents* (page 72)
- *Model Referenced One-to-Many Relationships Between Documents* (page 73)
- *Model Data for Atomic Operations* (page 75)
- *Model Tree Structures with Parent References* (page 76)
- *Model Tree Structures with Child References* (page 76)
- *Model Tree Structures with Materialized Paths* (page 78)
- *Model Tree Structures with Nested Sets* (page 79)

For more information and examples of real-world data modeling, consider the following external resources:

- Schema Design by Example[3]
- Walkthrough MongoDB Data Modeling[4]
- Document Design for MongoDB[5]
- Dynamic Schema Blog Post[6]
- MongoDB Data Modeling and Rails[7]
- Ruby Example of Materialized Paths[8]
- Sean Cribs Blog Post[9] which was the source for much of the *data-modeling-trees* content.

2.2 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 29) stored in *collections*
- the *query selectors* (page 31) that determine which records to select for read, update, and delete operations
- the *update actions* (page 31) that specify the particular field updates to perform during an update operation
- the specification of *indexes* (page 32) for collection.
- arguments to several MongoDB methods and operators, including:
  - *sort order* (page 32) for the `sort()` method.
  - *index specification* (page 32) for the `hint()` method.
- the output of a number of MongoDB commands and operations, including:

[3] http://www.mongodb.com/presentations/mongodb-melbourne-2012/schema-design-example
[4] http://blog.fiesta.cc/post/11319522700/walkthrough-mongodb-data-modeling
[5] http://oreilly.com/catalog/0636920018391
[6] http://dmerr.tumblr.com/post/6633338010/schemaless
[7] http://docs.mongodb.org/ecosystem/tutorial/model-data-for-ruby-on-rails/
[8] http://github.com/banker/newsmonger/blob/master/app/models/comment.rb
[9] http://seancribbs.com/tech/2009/09/28/modeling-a-tree-in-a-document-database

– the output of collStats command, and

– the output of the serverStatus command.

2.2.1 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 33).

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 ($).

Note: BSON documents may have more than one field with the same name; however, most MongoDB Interfaces represent MongoDB with a structure (e.g. a hash table) that does not support duplicate field names. If you need to manipulate documents that have more than one field with the same name, see your driver's documentation for more information.

Some documents created by internal MongoDB processes may have duplicate fields, but *no* MongoDB process will *ever* add duplicate keys to an existing user document.

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.

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 5) for dot notation examples with subdocuments.
- *Arrays* (page 6) for dot notation examples with arrays.

2.2.2 Document Types in MongoDB

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 27) 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* (page 34) 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/manualtutorial/create-an-auto-incrementing-field` tutorial for an implementation pattern.

- Generate a UUID in your application code. For a more efficient storage of the UUID values in the collection and in the `_id` index, store the UUID as a value of the BSON `BinData` type.

  Index keys that are of the `BinData` type are more efficiently stored in the index if:

  - the binary subtype value is in the range of 0-7 or 128-135, and
  - the length of the byte array is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, or 32.

- 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[10] for information concerning UUID interoperability.

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 4) and *Read* (page 51) for more examples on selecting documents for reads.
- *Update* (page 61) for more examples on selecting documents for updates.
- *Delete* (page 67) for more examples on selecting documents for deletes.

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 `$set` operator updates the `middle` field in the `name` subdocument. The document uses *dot notation* (page 29) to access a field in a subdocument.
- Adds an element to the field `awards` whose value is an array. Specifically, the `$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',
```

[10] http://api.mongodb.org/

```
                            by: 'IBM'
                          }
              }
    }
)
```

See also:

- *update operators* page for the available update operators and syntax.
- *update* (page 61) 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.

Index Specification Documents

Index specification documents describe the fields to index on during the `index creation`. See `indexes` for an overview of indexes. [11]

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 29) 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:

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

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:

[11] Indexes optimize a number of key *read* (page 3) and *write* (page 15) operations.

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

2.2.3 BSON Type Considerations

The following BSON types require special consideration:

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* (page 34) documentation for more information.

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. [12] In addition, MongoDB `$regex` queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular *Date* (page 34) 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.

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 34) 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. [13] 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> }
```

[12] 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.

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

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[14] refers to the BSON Date type as the *UTC datetime*.

Changed in version 2.0: BSON Date type is signed. [15] 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()
  ```

2.3 ObjectId

2.3.1 Overview

ObjectId is a 12-byte *BSON* type, constructed using:

- a 4-byte timestamp,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

In MongoDB, documents stored in a collection require a unique *_id* field that acts as a *primary key*. Because ObjectIds are small, most likely unique, and fast to generate, MongoDB uses ObjectIds as the default value for the `_id` field if the `_id` field is not specified. MongoDB clients should add an `_id` field with a unique ObjectId. However, if a client does not add an `_id` field, `mongod` will add an `_id` field that holds an ObjectId.

Using ObjectIds for the `_id` field provides the following additional benefits:

- you can access the timestamp of the ObjectId's creation, using the `getTimestamp()` method.
- sorting on an `_id` field that stores `ObjectId` values is roughly equivalent to sorting by creation time, although this relationship is not strict with `ObjectId` values generated on multiple systems within a single second.

Also consider the *BSON Documents* (page 27) section for related information on MongoDB's document orientation.

[14] http://bsonspec.org/#/specification

[15] 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.

2.3.2 ObjectId()

The `mongo` shell provides the `ObjectId()` wrapper class to generate a new ObjectId, and to provide the following helper attribute and methods:

- `str`

  The hexadecimal string value of the `ObjectId()` object.

- `getTimestamp()`

  Returns the timestamp portion of the `ObjectId()` object as a Date.

- `toString()`

  Returns the string representation of the `ObjectId()` object. The returned string literal has the format "`ObjectId(...)`".

  Changed in version 2.2: In previous versions `ObjectId.toString()` returns the value of the ObjectId as a hexadecimal string.

- `valueOf()`

  Returns the value of the `ObjectId()` object as a hexadecimal string. The returned string is the `str` attribute.

  Changed in version 2.2: In previous versions `ObjectId.valueOf()` returns the `ObjectId()` object.

2.3.3 Examples

Consider the following uses `ObjectId()` class in the `mongo` shell:

- To generate a new ObjectId, use the `ObjectId()` constructor with no argument:

  ```
  x = ObjectId()
  ```

  In this example, the value of `x` would be:

  ```
  ObjectId("507f1f77bcf86cd799439011")
  ```

- To generate a new ObjectId using the `ObjectId()` constructor with a unique hexadecimal string:

  ```
  y = ObjectId("507f191e810c19729de860ea")
  ```

  In this example, the value of `y` would be:

  ```
  ObjectId("507f191e810c19729de860ea")
  ```

- To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").getTimestamp()
  ```

  This operation will return the following Date object:

  ```
  ISODate("2012-10-17T20:46:22Z")
  ```

- Access the `str` attribute of an `ObjectId()` object, as follows:

  ```
  ObjectId("507f191e810c19729de860ea").str
  ```

  This operation will return the following hexadecimal string:

```
507f191e810c19729de860ea
```

- To return the string representation of an `ObjectId()` object, use the `toString()` method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").toString()
  ```

  This operation will return the following output:

  ```
  ObjectId("507f191e810c19729de860ea")
  ```

- To return the value of an `ObjectId()` object as a hexadecimal string, use the `valueOf()` method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").valueOf()
  ```

  This operation returns the following output:

  ```
  507f191e810c19729de860ea
  ```

2.4 Database References

MongoDB does not support joins. In MongoDB some data is *denormalized*, or stored with related data in *documents* to remove the need for joins. However, in some cases it makes sense to store related information in separate documents, typically in different collections or databases.

MongoDB applications use one of two methods for relating documents:

1. *Manual references* (page 36) where you save the `_id` field of one document in another document as a reference. Then your application can run a second query to return the embedded data. These references are simple and sufficient for most use cases.
2. *DBRefs* (page 37) are references from one document to another using the value of the first document's `_id` field collection, and optional database name. To resolve DBRefs, your application must perform additional queries to return the referenced documents. Many `drivers` have helper methods that form the query for the DBRef automatically. The drivers [16] do not *automatically* resolve DBRefs into documents.

   Use a DBRef when you need to embed documents from multiple collections in documents from one collection. DBRefs also provide a common format and type to represent these relationships among documents. The DBRef format provides common semantics for representing links between documents if your database must interact with multiple frameworks and tools.

Unless you have a compelling reason for using a DBRef, use manual references.

2.4.1 Manual References

Background

Manual references refers to the practice of including one *document's* `_id` field in another document. The application can then issue a second query to resolve the referenced fields as needed.

Process

Consider the following operation to insert two documents, using the `_id` field of the first document as a reference in the second document:

[16] Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.

```
original_id = ObjectId()

db.places.insert({
    "_id": original_id,
    "name": "Broadway Center",
    "url": "bc.example.net"
})

db.people.insert({
    "name": "Erin",
    "places_id": original_id,
    "url":  "bc.example.net/Erin"
})
```

Then, when a query returns the document from the `people` collection you can, if needed, make a second query for the document referenced by the `places_id` field in the `places` collection.

Use

For nearly every case where you want to store a relationship between two documents, use *manual references* (page 36). The references are simple to create and your application can resolve references as needed.

The only limitation of manual linking is that these references do not convey the database and collection name. If you have documents in a single collection that relate to documents in more than one collection, you may need to consider using *DBRefs* (page 37).

2.4.2 DBRefs

Background

DBRefs are a convention for representing a *document*, rather than a specific reference "type." They include the name of the collection, and in some cases the database, in addition to the value from the `_id` field.

Format

DBRefs have the following fields:

$ref
The `$ref` field holds the name of the collection where the referenced document resides.

$id
The `$id` field contains the value of the `_id` field in the referenced document.

$db
Optional.

Contains the name of the database where the referenced document resides.

Only some drivers support `$db` references.

Example

DBRef document would resemble the following:

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

Consider a document from a collection that stored a DBRef in a `creator` field:

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
                  "$ref" : "creators",
                  "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
                  "$db" : "users"
               }
}
```

The DBRef in this example, points to a document in the `creators` collection of the `users` database that has `ObjectId("5126bc054aed4daf9e2ab772")` in its `_id` field.

Note: The order of fields in the DBRef matters, and you must use the above sequence when using a DBRef.

Support

C++ The C++ driver contains no support for DBRefs. You can transverse references manually.

C# The C# driver provides access to DBRef objects with the MongoDBRef Class[17] and supplies the FetchDBRef Method[18] for accessing these objects.

Java The DBRef[19] class provides supports for DBRefs from Java.

JavaScript The `mongo` shell's `JavaScript` interface provides a DBRef.

Perl The Perl driver contains no support for DBRefs. You can transverse references manually or use the MongoDBx::AutoDeref[20] CPAN module.

PHP The PHP driver does support DBRefs, including the optional `$db` reference, through The MongoDBRef class[21].

Python The Python driver provides the DBRef class[22], and the dereference method[23] for interacting with DBRefs.

Ruby The Ruby Driver supports DBRefs using the DBRef class[24] and the deference method[25].

Use

In most cases you should use the *manual reference* (page 36) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a DBRef.

2.5 GridFS

GridFS is a specification for storing and retrieving files that exceed the *BSON*-document *size limit* of 16MB.

[17] http://api.mongodb.org/csharp/current/html/46c356d3-ed06-a6f8-42fa-e0909ab64ce2.htm
[18] http://api.mongodb.org/csharp/current/html/1b0b8f48-ba98-1367-0a7d-6e01c8df436f.htm
[19] http://api.mongodb.org/java/current/com/mongodb/DBRef.html
[20] http://search.cpan.org/dist/MongoDBx-AutoDeref/
[21] http://www.php.net/manual/en/class.mongodbref.php/
[22] http://api.mongodb.org/python/current/api/bson/dbref.html
[23] http://api.mongodb.org//python/current/api/pymongo/database.html#pymongo.database.Database.dereference
[24] http://api.mongodb.org//ruby/current/BSON/DBRef.html
[25] http://api.mongodb.org//ruby/current/Mongo/DB.html#dereference

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, [26] and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to "skip" into the middle of a video or audio file.

GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory. For more information on the indications of GridFS, see *faq-developers-when-to-use-gridfs*.

2.5.1 Implement GridFS

To store and retrieve files using *GridFS*, use either of the following:

- A MongoDB driver. See the `drivers` documentation for information on using GridFS with your driver.
- The `mongofiles` command-line tool in the `mongo` shell. See `http://docs.mongodb.org/manualreference/mongofiles`.

2.5.2 GridFS Collections

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 39).
- `files` stores the file's metadata. For details, see *The files Collection* (page 40).

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

The `chunks` Collection

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the *GridFS* store. The following is a prototype document from the `chunks` collection.:

```
{
  "_id" : <string>,
  "files_id" : <string>,
  "n" : <num>,
  "data" : <binary>
}
```

A document from the `chunks` collection contains the following fields:

`chunks._id`
: The unique *ObjectId* of the chunk.

`chunks.files_id`
: The `_id` of the "parent" document, as specified in the `files` collection.

[26] The use of the term *chunks* in the context of GridFS is not related to the use of the term *chunks* in the context of sharding.

`chunks.n`
The sequence number of the chunk. GridFS numbers all chunks, starting with 0.

`chunks.data`
The chunk's payload as a *BSON* binary type.

The `chunks` collection uses a *compound index* on `files_id` and `n`, as described in *GridFS Index* (page 41).

The `files` Collection

Each document in the `files` collection represents a file in the *GridFS* store. Consider the following prototype of a document in the `files` collection:

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>
  "uploadDate" : <timestamp>
  "md5" : <hash>

  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <dataObject>,
}
```

Documents in the `files` collection contain some or all of the following fields. Applications may create additional arbitrary fields:

`files._id`
The unique ID for this document. The `_id` is of the data type you chose for the original document. The default type for MongoDB documents is *BSON ObjectId*.

`files.length`
The size of the document in bytes.

`files.chunkSize`
The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 256 kilobytes.

`files.uploadDate`
The date the document was first stored by GridFS. This value has the `Date` type.

`files.md5`
An MD5 hash returned from the filemd5 API. This value has the `String` type.

`files.filename`
Optional. A human-readable name for the document.

`files.contentType`
Optional. A valid MIME type for the document.

`files.aliases`
Optional. An array of alias strings.

`files.metadata`
Optional. Any additional information you want to store.

2.5.3 GridFS Index

GridFS uses a *unique*, *compound* index on the `chunks` collection for `files_id` and `n`. The index allows efficient retrieval of chunks using the `files_id` and `n` values, as shown in the following example:

```
cursor = db.fs.chunks.find({files_id: myFileID}).sort({n:1});
```

See the relevant `driver` documentation for the specific behavior of your GridFS application. If your driver does not create this index, issue the following operation using the `mongo` shell:

```
db.fs.chunks.ensureIndex( { files_id: 1, n: 1 }, { unique: true } );
```

2.5.4 Example Interface

The following is an example of the GridFS interface in Java. The example is for demonstration purposes only. For API specifics, see the relevant `driver` documentation.

By default, the interface must support the default GridFS bucket, named `fs`, as in the following:

```
// returns default GridFS bucket (i.e. "fs" collection)
GridFS myFS = new GridFS(myDatabase);

// saves the file to "fs" GridFS bucket
myFS.createFile(new File("/tmp/largething.mpg"));
```

Optionally, interfaces may support other additional GridFS buckets as in the following example:

```
// returns GridFS bucket named "contracts"
GridFS myContracts = new GridFS(myDatabase, "contracts");

// retrieve GridFS object "smithco"
GridFSDBFile file = myContracts.findOne("smithco");

// saves the GridFS file to the file system
file.writeTo(new File("/tmp/smithco.pdf"));
```

CRUD Operations for MongoDB

These documents provide an overview and examples of common database operations, i.e. CRUD, in MongoDB.

3.1 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 15); for documentation of the other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 1) page.

3.1.1 Overview

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

- *insert* (page 44)
- *updates with the upsert option* (page 49)

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 16), 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 27) 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 16) in the *Write Operations* (page 15) document for more information.

3.1.2 `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.

Insert the First Document in a Collection

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,
```

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

```
                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 51) the `bios` collection:

```
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"
              }
            ]
}
```

Insert a Document without Specifying an `_id` Field

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"
               }
             ]
}
```

Bulk Insert Multiple Documents

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'
                 }
               ]
     }
   ]
)
```

Insert a Document with `save()`

The `save()` method performs an insert if the document to save does not contain the `_id` field.

The following `save()` 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'
                  }
                ]
   }
```

```
)
```

3.1.3 `update()` Operations with the `upsert` Flag

The `update()` operation in MongoDB accepts an "`upsert`" flag that modifies the behavior of `update()` from *updating existing documents* (page 61), to inserting data.

These `update()` operations with the upsert flag eliminate the need to perform an additional operation to check for existence of a record before performing either an update or an insert operation. These update operations have the use `<query>` argument to determine the write operation:

- If the query matches an existing document(s), the operation is an *update* (page 61).
- If the query matches no document in the collection, the operation is an *insert* (page 43).

An upsert operation has the following syntax [2]:

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

Insert a Document that Contains `field` and `value` Pairs

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 query does not include an `_id` field, the operation adds the `_id` field and generates a unique `ObjectId` for its value.

The following update inserts a new document into the `bios` collection [2]:

```
db.bios.update(
   { name: { first: 'Dennis', last: 'Ritchie'} },
   {
     name: { first: 'Dennis', last: 'Ritchie'},
     birth: new Date('Sep 09, 1941'),
     death: 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'
               }
             ]
```

[2] Prior to version 2.2, in the `mongo` shell, you would specify the `upsert` and the `multi` options in the `update()` method as positional boolean options. See `update()` for details.

```
    },
    { upsert: true }
)
```

Insert a Document that Contains Update Operator Expressions

If no document matches the <query> argument, the update 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 [2]:

```
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 }
)
```

Update operations with save()

The save() method is identical to an *update operation with the upsert flag* (page 49)

performs an upsert if the document to save contains the _id field. To determine whether to perform an insert or an update, save() method queries documents on the _id field.

The following operation performs an upsert that inserts a document into the bios collection since no documents in the collection contains an _id field with the value 10:

```
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'
                }
              ]
   }
)
```

3.2 Read

Of the four basic database operations (i.e. CRUD), read operations 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 3); for documentation of the other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 1) page.

3.2.1 Overview

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

- *find* (page 52)
- *findOne* (page 59)

3.2.2 `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.

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.

Return All Documents in a Collection

If there is no `<query>` argument, the :method:‘ ~db.collection.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()
```

Return Documents that Match Query Conditions

If there is a <query> argument, the `find()` method selects all documents from a collection that satisfies the query specification.

Equality Matches

The following operation returns a cursor to documents in the `bios` collection where the field `_id` equals `5`:

```
db.bios.find(
   {
      _id: 5
   }
)
```

Using Operators

The following operation returns a cursor to all documents in the `bios` collection where the field `_id` equals `5` or `ObjectId("507c35dd8fada716c89d0013")`:

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

On Arrays

Query an Element The following operation returns a cursor to all documents in the `bios` collection where the array field `contribs` contains the element `'UNIX'`:

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

Query Multiple Fields on an Array of Documents The following operation returns a cursor to 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 }
                }
      }
   }
)
```

On Subdocuments

Exact Matches The following operation returns a cursor to 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'
}
```

Fields of a Subdocument The following operation returns a cursor to 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'
}
```

Logical Operators

OR Disjunctions The following operation returns a cursor to 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') } }
          ]
   }
)
```

AND Conjunctions The following operation returns a cursor to all documents in the bios collection where the field first in the subdocument 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.

With a Projection

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.

Specify the Fields to Return

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 }
 )
```

Explicitly Exclude the _id Field

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 }
)
```

Return All but the Excluded Fields

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 }
)
```

On Arrays and Subdocuments

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 array:

```
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 6) for more examples on accessing arrays.
- *Subdocuments* (page 5) for more examples on accessing subdocuments.
- $elemMatch query operator for more information on matching array elements.
- $elemMatch projection operator for additional information on restricting array elements to return.

Iterate the Returned 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 [3] to print up to the first 20 documents that match the query, as in the following example:

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

With Variable Name

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
```

[3] You can use the DBQuery.shellBatchSize to change the number of iteration from the default value 20. See *Cursor Flags* (page 13) and *Cursor Behaviors* (page 12) for more information.

With `next()` Method

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())`:

```
if (myDocument) {
   var myName = myDocument.name;
   printjson(myName);
}
```

With `forEach()` Method

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 11)
- *JavaScript cursor methods*

Modify the 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:

Order Documents in the Result Set

The `sort()` method orders the documents in the result set.

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.

Limit the Number of Documents to Return

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.

Set the Starting Point of the Result Set

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 )
```

Combine Cursor Methods

You can chain these cursor methods, as in the following examples [4]:

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

See the *JavaScript cursor methods* reference and your `driver` documentation for additional references. See *Cursors* (page 11) for more information regarding cursors.

3.2.3 `findOne()`

The `findOne()` method selects 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.

With Empty Query Specification

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()
```

[4] Regardless of the order you chain the `limit()` and the `sort()`, the request to the server has the structure that treats the query and the :method:‘ ~cursor.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.

With a Query Specification

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') } }
          ]
   }
)
```

With a Projection

You can pass a <projection> argument to `findOne()` to control the fields included in the result set.

Specify the Fields to Return

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 }
)
```

Return All but the Excluded Fields

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 }
)
```

Access the `findOne` Result

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 example:

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

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

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

3.3 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 15); for documentation of other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 1) page.

3.3.1 Overview

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

- *update* (page 62)
- *save* (page 65)

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 16) in the *Write Operations* (page 15) document for more information.

3.3.2 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 [5]:

```
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.

Modify with Update Operators

If the `<update>` argument contains only *update operator* (page 66) 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*.

Update a Field in a Document

Use `$set` to update a value of a field.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and sets the value of the field `middle`, in the subdocument `name`, to `Warner`:

```
db.bios.update(
   { _id: 1 },
   {
     $set: { 'name.middle': 'Warner' },
   }
)
```

[5] This examples uses the interface added in MongoDB 2.2 to specify the `multi` and the `upsert` options in a document form.
Prior to version 2.2, in the `mongo` shell, you would specify the `upsert` and the `multi` options in the `update()` method as positional boolean options. See `update()` for details.

Add a New Field to a Document

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 that 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'
           }
   }
)
```

Remove a Field from a Document

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 } }
)
```

Update Arrays

Update an Element by Specifying Its Position 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' } }
)
```

Update an Element without Specifying Its Position 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' } }
    )
```

Update a Document Element without Specifying Its Position 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' } }
)
```

Add an Element to an Array The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and adds a new element to the `awards` field:

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

Update Multiple Documents

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 subdocument element with the `award` field equal to `Turing` and sets the `turing` field to `true` in the matching documents [6]:

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

Replace Existing Document with New Document

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: [
```

[6] Prior to version 2.2, in the `mongo` shell, you would specify the `upsert` and the `multi` options in the `update()` method as positional boolean options. See `update()` for details.

```
                { 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' }
              ]
   }
)
```

3.3.3 `update()` Operations with the `upsert` Flag

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. [7]

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. [8]

```
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 *Update Operations with the Upsert Flag* (page 49) in the *Create* (page 43) document.

3.3.4 Save

The `save()` method performs a special type of `update()`, depending on the `_id` field of the specified document.

[7] Prior to version 2.2, in the `mongo` shell, you would specify the `upsert` and the `multi` options in the `update()` method as positional boolean options. See `update()` for details.

[8] 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* (page 66), 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> )
```

Behavior

If you specify a document with an `_id` field, `save()` performs an `update()` with the `upsert` option set: if an existing document in the collection has the same `_id`, `save()` updates that document, and inserts the document otherwise. If you do not specify a document with an `_id` field to `save()`, performs an `insert()` operation.

That is, `save()` method is equivalent to the `update()` method with the `upsert` option and a `<query>` argument with an `_id` field.

Example

Consider the following psudocode explanation of `save()` as an illustration of its behavior:

```
function save( doc ) {
  if( doc["_id"] ) {
       update( {_id: doc["_id"] }, doc, { upsert: true } );
    }
  else {
       insert(doc);
  }
}
```

Save Performs an Update

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' ]
   }
)
```

See also:

Insert a Document with save() (page 48) and *Update operations with save()* (page 50) in the *Create* (page 43) section.

3.3.5 Update Operators

Fields

- `$inc`
- `$rename`
- `$set`

- `$unset`

Array

- `$`
- `$addToSet`
- `$pop`
- `$pullAll`
- `$pull`
- `$pushAll`
- `$push`

Bitwise

- `$bit`

Isolation

- `$isolated`

3.4 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 15); for documentation of other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 1) page.

3.4.1 Overview

The *remove()* (page 67) 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 16) in the *Write Operations* (page 15) document for more information.

Use the `remove()` method to delete documents from a collection. 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 takes a Boolean and has the same effect as `LIMIT 1`.

`remove()` deletes documents from the collection. If you do not specify a query, `remove()` removes all documents from a collection, but does not remove the indexes. [9]

Note: For large deletion operations, it may be more efficient to copy the documents that you want to keep to a new collection and then use `drop()` on the original collection.

3.4.2 Remove All Documents that Match a Condition

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/ } )
```

3.4.3 Remove a Single Document that Matches a Condition

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 )
```

3.4.4 Remove All Documents from a Collection

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.

---

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

3.4.5 Capped Collection

You cannot use the `remove()` method with a *capped collection*.

3.4.6 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 `$isolated` isolation operator, effectively isolating the delete operation from other write operations. To isolate the operation, include `$isolated: 1` in the `<query>` parameter as in the following example:

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

CHAPTER 4

Data Modeling Patterns

4.1 Model Embedded One-to-One Relationships Between Documents

4.1.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 24) documents to describe relationships between connected data.

4.1.2 Pattern

Consider the following example that maps patron and address relationships. The example illustrates the advantage of embedding over referencing if you need to view one data entity in context of the other. In this one-to-one relationship between `patron` and `address` data, the `address` belongs to the `patron`.

In the normalized data model, the `address` contains a reference to the `parent`.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA"
   zip: 12345
}
```

If the `address` data is frequently retrieved with the `name` information, then with referencing, your application needs to issue multiple queries to resolve the reference. The better data model would be to embed the `address` data in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
```

```
               street: "123 Fake Street",
               city: "Faketon",
               state: "MA"
               zip: 12345
             }
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

4.2 Model Embedded One-to-Many Relationships Between Documents

4.2.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 24) documents to describe relationships between connected data.

4.2.2 Pattern

Consider the following example that maps patron and multiple address relationships. The example illustrates the advantage of embedding over referencing if you need to view many data entities in context of another. In this one-to-many relationship between `patron` and `address` data, the `patron` has multiple `address` entities.

In the normalized data model, the `address` contains a reference to the `parent`.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: 12345
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: 12345
}
```

If your application frequently retrieves the `address` data with the `name` information, then your application needs to issue multiple queries to resolve the references. A more optimal schema would be to embed the `address` data entities in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: 12345
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: 12345
                }
              ]
 }
```

With the embedded data model, your application can retrieve the complete patron information with one query.

4.3 Model Referenced One-to-Many Relationships Between Documents

4.3.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *references* (page 24) between documents to describe relationships between connected data.

4.3.2 Pattern

Consider the following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

```
{
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

```
{
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

```
{
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA",
   books: [12346789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{
   _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}

{
   _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
```

```
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher_id: "oreilly"
}
```

4.4 Model Data for Atomic Operations

4.4.1 Pattern

Consider the following example that keeps a library book and its checkout information. The example illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.

Consider the following `book` document that stores the number of available copies for checkout and the current checkout information:

```
book = {
          _id: 123456789,
          title: "MongoDB: The Definitive Guide",
          author: [ "Kristina Chodorow", "Mike Dirolf" ],
          published_date: ISODate("2010-09-24"),
          pages: 216,
          language: "English",
          publisher_id: "oreilly",
          available: 3,
          checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
        }
```

You can use the `db.collection.findAndModify()` method to atomically determine if a book is available for checkout and update with the new checkout information. Embedding the `available` field and the `checkout` field within the same document ensures that the updates to these fields are in sync:

```
db.books.findAndModify ( {
   query: {
            _id: 123456789,
            available: { $gt: 0 }
          },
   update: {
             $inc: { available: -1 },
             $push: { checkout: { by: "abc", date: new Date() } }
           }
} )
```

4.5 Model Tree Structures with Parent References

4.5.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 24) to "parent" nodes in children nodes.

4.5.2 Pattern

The *Parent References* pattern stores each tree node in a document; in addition to the tree node, the document stores the id of the node's parent.

Consider the following example that models a tree of categories using *Parent References*:

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "Postgres", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

- The query to retrieve the parent of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).parent
  ```

- You can create an index on the field `parent` to enable fast search by the parent node:

  ```
  db.categories.ensureIndex( { parent: 1 } )
  ```

- You can query by the `parent` field to find its immediate children nodes:

  ```
  db.categories.find( { parent: "Databases" } )
  ```

The *Parent Links* pattern provides a simple solution to tree storage, but requires multiple queries to retrieve subtrees.

4.6 Model Tree Structures with Child References

4.6.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 24) in the parent-nodes to children nodes.

4.6.2 Pattern

The *Child References* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's children.

Consider the following example that models a tree of categories using *Child References*:

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "Postgres", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "Postgres" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

- The query to retrieve the immediate children of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "Databases" } ).children
  ```

- You can create an index on the field `children` to enable fast search by the child nodes:

  ```
  db.categories.ensureIndex( { children: 1 } )
  ```

- You can query for a node in the `children` field to find its parent node as well as its siblings:

  ```
  db.categories.find( { children: "MongoDB" } )
  ```

The *Child References* pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary. This pattern may also provide a suitable solution for storing graphs where a node may have multiple parents.

4.7 Model Tree Structures with an Array of Ancestors

4.7.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents using *references* (page 24) to parent nodes and an array that stores all ancestors.

4.7.2 Pattern

The *Array of Ancestors* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's ancestors or path.

Consider the following example that models a tree of categories using *Array of Ancestors*:

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "
db.categories.insert( { _id: "Postgres", ancestors: [ "Books", "Programming", "Databases" ], parent:
db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programmin
db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programmin
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

- The query to retrieve the ancestors or path of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).ancestors
  ```

- You can create an index on the field `ancestors` to enable fast search by the ancestors nodes:

```
db.categories.ensureIndex( { ancestors: 1 } )
```

- You can query by the `ancestors` to find all its descendants:

  ```
  db.categories.find( { ancestors: "Programming" } )
  ```

The *Array of Ancestors* pattern provides a fast and efficient solution to find the descendants and the ancestors of a node by creating an index on the elements of the ancestors field. This makes *Array of Ancestors* a good choice for working with subtrees.

The *Array of Ancestors* pattern is slightly slower than the *Materialized Paths* pattern but is more straightforward to use.

4.8 Model Tree Structures with Materialized Paths

4.8.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing full relationship paths between documents.

4.8.2 Pattern

The *Materialized Paths* pattern stores each tree node in a document; in addition to the tree node, document stores as a string the id(s) of the node's ancestors or path. Although the *Materialized Paths* pattern requires additional steps of working with strings and regular expressions, the pattern also provides more flexibility in working with the path, such as finding nodes by partial paths.

Consider the following example that models a tree of categories using *Materialized Paths* ; the path string uses the comma `,` as a delimiter:

```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "Postgres", path: ",Books,Programming,Databases," } )
```

- You can query to retrieve the whole tree, sorting by the `path`:

  ```
  db.categories.find().sort( { path: 1 } )
  ```

- You can use regular expressions on the `path` field to find the descendants of `Programming`:

  ```
  db.categories.find( { path: /,Programming,/ } )
  ```

- You can also retrieve the descendants of `Books` where the `Books` is also at the topmost level of the hierarchy:

  ```
  db.categories.find( { path: /^,Books,/ } )
  ```

- To create an index on the field `path` use the following invocation:

```
db.categories.ensureIndex( { path: 1 } )
```

This index may improve performance, depending on the query:

– For queries of the `Books` sub-tree (e.g. `http://docs.mongodb.org/manual^,Books,/`) an index on the `path` field improves the query performance significantly.

– For queries of the `Programming` sub-tree (e.g. `http://docs.mongodb.org/manual,Programming,/`), or similar queries of sub-tress, where the node might be in the middle of the indexed string, the query must inspect the entire index.

  For these queries an index *may* provide some performance improvement *if* the index is significantly smaller than the entire collection.

4.9 Model Tree Structures with Nested Sets

4.9.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 23) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree like structure that optimizes discovering subtrees at the expense of tree mutability.

4.9.2 Pattern

The *Nested Sets* pattern identifies each node in the tree as stops in a round-trip traversal of the tree. The application visits each node in the tree twice; first during the initial trip, and second during the return trip. The *Nested Sets* pattern stores each tree node in a document; in addition to the tree node, document stores the id of node's parent, the node's initial stop in the `left` field, and its return stop in the `right` field.

Consider the following example that models a tree of categories using *Nested Sets*:

```
db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right: 10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "Postgres", parent: "Databases", left: 8, right: 9 } )
```

You can query to retrieve the descendants of a node:

```
var databaseCategory = db.v.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt: databaseCategory.right } }
```

The *Nested Sets* pattern provides a fast and efficient solution for finding subtrees but is inefficient for modifying the tree structure. As such, this pattern is best for static trees that do not change.

4.10 Model Data to Support Keyword Search

If your application needs to perform queries on the content of a field that holds text you can perform exact matches on the text or use `$regex` to use regular expression pattern matches. However, for many operations on text, these methods do not satisfy application requirements.

This pattern describes one method for supporting keyword search using MongoDB to support application search functionality, that uses keywords stored in an array in the same document as the text field. Combined with a *multi-key index*, this pattern can support application's keyword search operations.

Note: Keyword search is *not* the same as text search or full text search, and does not provide stemming or other text-processing features. See the *Limitations of Keyword Indexes* (page 80) section for more information.

4.10.1 Pattern

To add structures to your document to support keyword-based queries, create an array field in your documents and add the keywords as strings in the array. You can then create a *multi-key index* on the array and create queries that select values from the array.

Example

Suppose you have a collection of library volumes that you want to make searchable by topics. For each volume, you add the array `topics`, and you add as many keywords as needed for a given volume.

For the `Moby-Dick` volume you might have the following document:

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
  ISBN : 0451526996 ,
  topics : [ "whaling" , "allegory" , "revenge" , "American" ,
    "novel" , "nautical" , "voyage" , "Cape Cod" ]
}
```

You then create a multi-key index on the `topics` array:

```
db.volumes.ensureIndex( { topics: 1 } )
```

The multi-key index creates separate index entries for each keyword in the `topics` array. For example the index contains one entry for `whaling` and another for `allegory`.

You then query based on the keywords. For example:

```
db.volumes.findOne( { topics : "voyage" }, { title: 1 } )
```

Note: An array with a large number of elements, such as one with several hundreds or thousands of keywords will incur greater indexing costs on insertion.

4.10.2 Limitations of Keyword Indexes

MongoDB can support keyword searches using specific data models and *multi-key indexes*; however, these keyword indexes are not sufficient or comparable to full-text products in the following respects:

- *Stemming*. Keyword queries in MongoDB can not parse keywords for root or related words.
- *Synonyms*. Keyword-based search features must provide support for synonym or related queries in the application layer.
- *Ranking*. The keyword look ups described in this document do not provide a way to weight results.

- *Asynchronous Indexing*. MongoDB builds indexes synchronously, which means that the indexes used for keyword indexes are always current and can operate in real-time. However, asynchronous bulk indexes may be more efficient for some kinds of content and workloads.