MongoDB CRUD Operations

Release 3.0.0-rc6

MongoDB Documentation Project

February 25, 2015

Contents

MongoDB provides rich semantics for reading and manipulating data. CRUD stands for *create*, *read*, *update*, and *delete*. These terms are the foundation for all interactions with the database.

MongoDB CRUD Introduction **(page 3)** An introduction to the MongoDB data model as well as queries and data manipulations.

MongoDB CRUD Concepts **(page 6)** The core documentation of query and data manipulation.

MongoDB CRUD Tutorials **(page 39)** Examples of basic query and data modification operations.

MongoDB CRUD Reference **(page 75)** Reference material for the query and data manipulation interfaces.

1 MongoDB CRUD Introduction

MongoDB stores data in the form of *documents*, which are JSON-like field and value pairs. Documents are analogous to structures in programming languages that associate keys with values (e.g. dictionaries, hashes, maps, and associative arrays). Formally, MongoDB documents are *BSON* documents. BSON is a binary representation of *JSON* with additional type information. In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For more information, see `http://docs.mongodb.org/manual/core/document`.

MongoDB stores all documents in *collections*. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases.

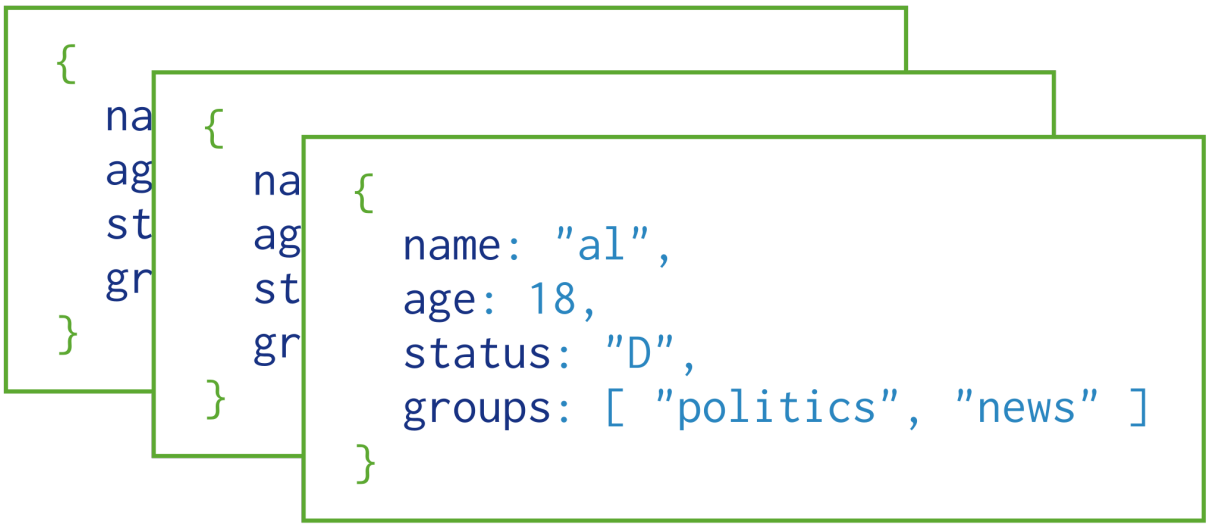
1.1 Database Operations

Query

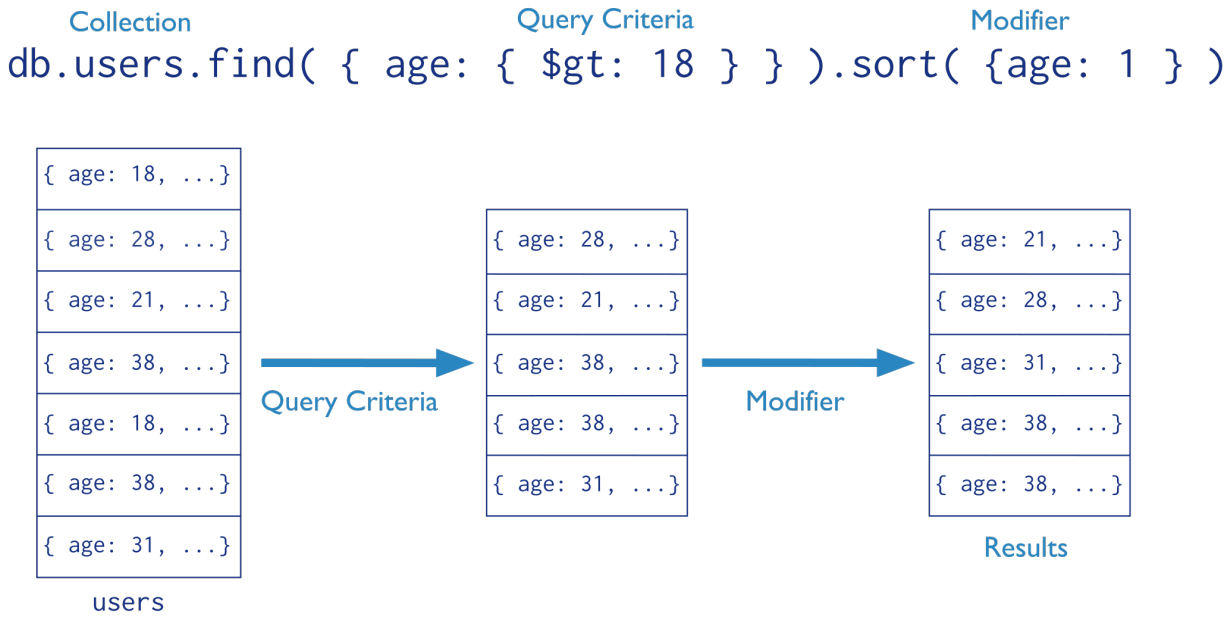
In MongoDB a query targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a *projection* that specifies the fields from the matching documents to return. You can optionally modify queries to impose limits, skips, and sort orders.

In the following diagram, the query process specifies a query criteria and a sort modifier:

See *Read Operations Overview* (page 7) for more information.


{
name: "al",
age: 18,
status: "D",
groups: ["politics", "news"]
}

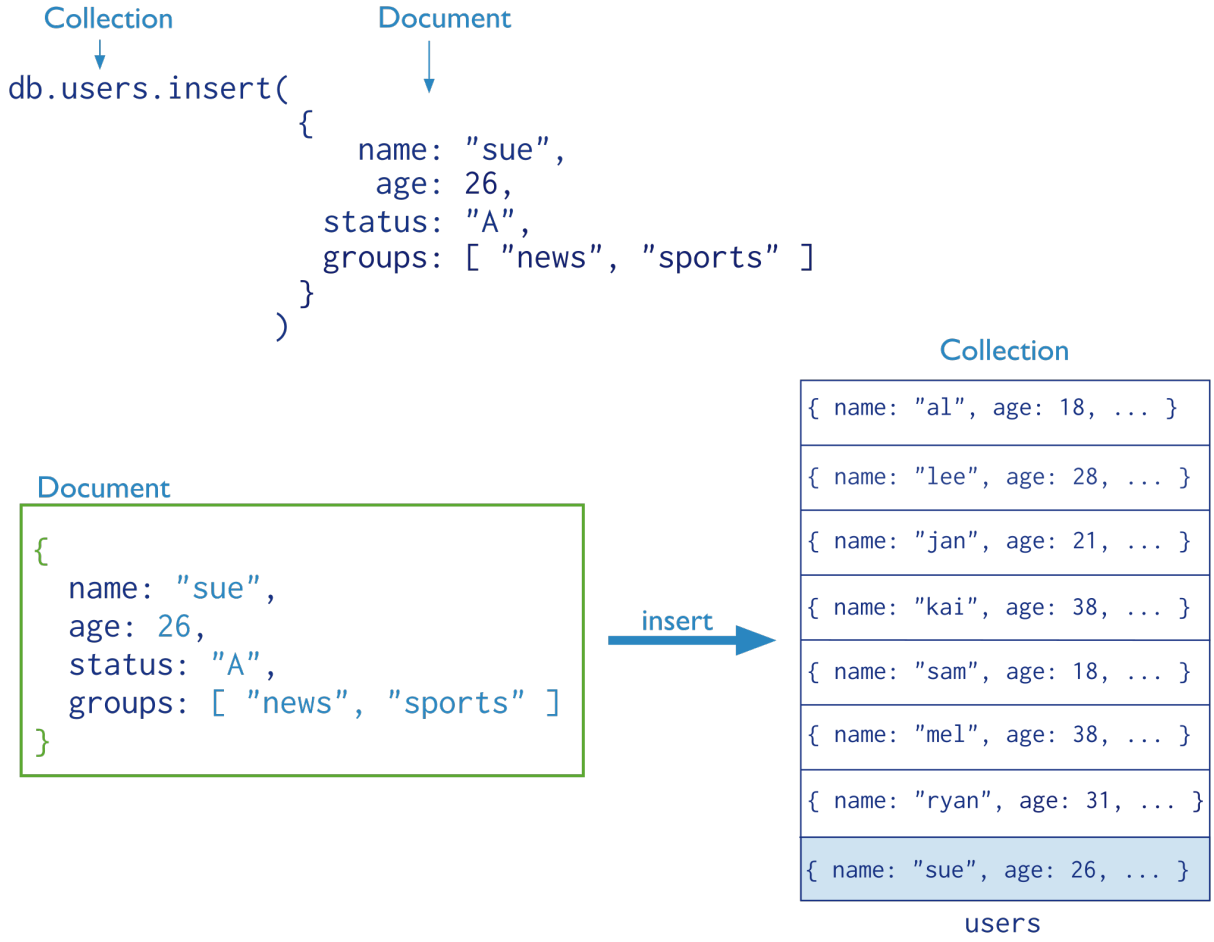
Collection


Collection
Query Criteria
Modifier
db.users.find({ age: { $gt: 18 } }).sort({age: 1 })
{ age: 18, ...}
{ age: 28, ...}
{ age: 21, ...}
{ age: 38, ...}
{ age: 18, ...}
{ age: 38, ...}
{ age: 31, ...}
users
Query Criteria
{ age: 28, ...}
{ age: 21, ...}
{ age: 38, ...}
{ age: 38, ...}
{ age: 31, ...}
Modifier
{ age: 21, ...}
{ age: 28, ...}
{ age: 31, ...}
{ age: 38, ...}
{ age: 38, ...}
Results

Data Modification

Data modification refers to operations that create, update, or delete data. In MongoDB, these operations modify the data of a single *collection*. For the update and delete operations, you can specify the criteria to select the documents to update or remove.

In the following diagram, the insert operation adds a new document to the `users` collection.



See *Write Operations Overview* (page 20) for more information.

1.2 Related Features

Indexes

To enhance the performance of common queries and updates, MongoDB has full support for secondary indexes. These indexes allow applications to store a *view* of a portion of the collection in an efficient data structure. Most indexes store an ordered representation of all values of a field or a group of fields. Indexes may also *enforce uniqueness*, store objects in a `geospatial representation`, and facilitate `text search`.

Replica Set Read Preference

For replica sets and sharded clusters with replica set components, applications specify *read preferences*. A read preference determines how the client direct read operations to the set.

Write Concern

Applications can also control the behavior of write operations using *write concern* (page 23). Particularly useful for deployments with replica sets, the write concern semantics allow clients to specify the assurance that MongoDB provides when reporting on the success of a write operation.

Aggregation

In addition to the basic queries, MongoDB provides several data aggregation features. For example, MongoDB can return counts of the number of documents that match a query, or return the number of distinct values for a field, or process a collection of documents using a versatile stage-based data processing pipeline or map-reduce operations.

2 MongoDB CRUD Concepts

The *Read Operations* (page 6) and *Write Operations* (page 19) documents introduce the behavior and operations of read and write operations for MongoDB deployments.

Read Operations **(page 6)** Queries are the core operations that return data in MongoDB. Introduces queries, their behavior, and performances.

> ***Cursors*** **(page 10)** Queries return iterable objects, called cursors, that hold the full result set.
>
> ***Query Optimization*** **(page 11)** Analyze and improve query performance.
>
> ***Distributed Queries*** **(page 15)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Write Operations **(page 19)** Write operations insert, update, or remove documents in MongoDB. Introduces data create and modify operations, their behavior, and performances.

> ***Write Concern*** **(page 23)** Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.
>
> ***Distributed Write Operations*** **(page 28)** Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.
>
> Continue reading from *Write Operations* (page 19) for additional background on the behavior of data modification operations in MongoDB.

2.1 Read Operations

The following documents describe read operations:

Read Operations Overview **(page 7)** A high level overview of queries and projections in MongoDB, including a discussion of syntax and behavior.

Cursors **(page 10)** Queries return iterable objects, called cursors, that hold the full result set.

Query Optimization **(page 11)** Analyze and improve query performance.

Query Plans **(page 14)** MongoDB executes queries using optimal *plans*.

Distributed Queries **(page 15)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Read Operations Overview

Read operations, or *queries*, retrieve data stored in the database. In MongoDB, queries select *documents* from a single *collection*.

Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a *projection* that specifies the fields from the matching documents to return. The projection limits the amount of data that MongoDB returns to the client over the network.

Query Interface

For query operations, MongoDB provides a `db.collection.find()` method. The method accepts both the query criteria and projections and returns a *cursor* (page 10) to the matching documents. You can optionally modify the query to impose limits, skips, and sort orders.

The following diagram highlights the components of a MongoDB query operation:

The next diagram shows the same query in SQL:

Example

```
db.users.find( { age: { $gt: 18 } }, { name: 1, address: 1 } ).limit(5)
```

This query selects the documents in the `users` collection that match the condition `age` is greater than `18`. To specify the greater than condition, query criteria uses the greater than (i.e. `$gt`) *query selection operator*. The query returns at most `5` matching documents (or more precisely, a cursor to those documents). The matching documents will return with only the `_id`, `name` and `address` fields. See *Projections* (page 8) for details.

See

SQL to MongoDB Mapping Chart (page 78) for additional examples of MongoDB queries and the corresponding SQL statements.

Query Behavior

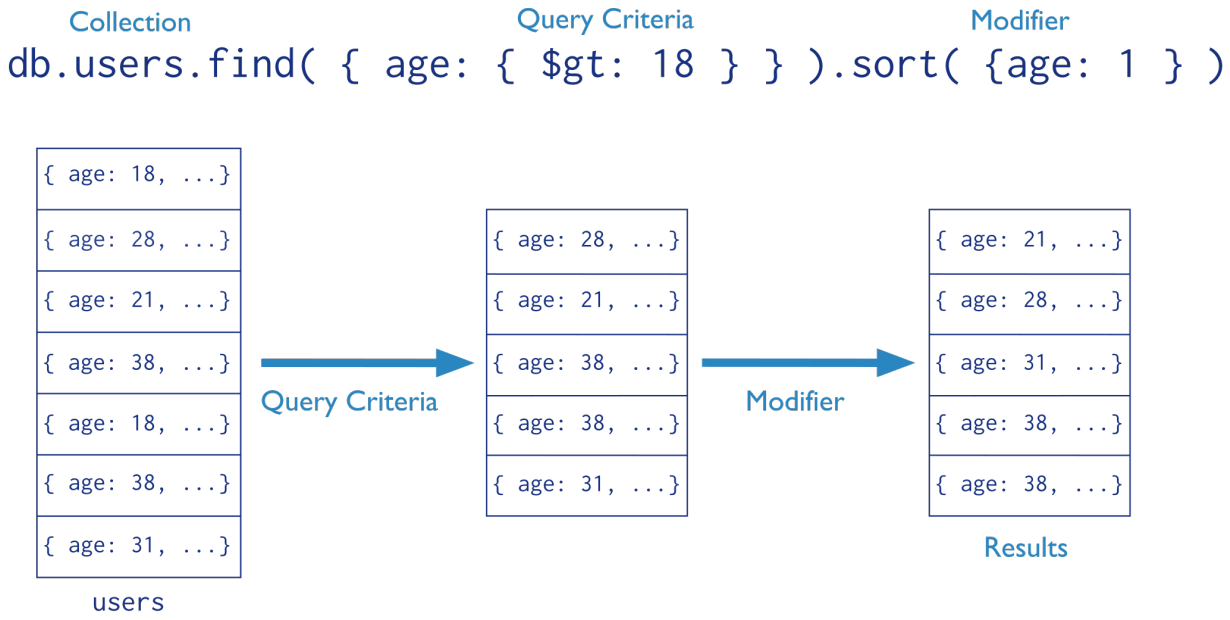
MongoDB queries exhibit the following behavior:

- All queries in MongoDB address a *single* collection.
- You can modify the query to impose `limits`, `skips`, and `sort orders`.
- The order of documents returned by a query is not defined unless you specify a `sort()`.
- Operations that *modify existing documents* (page 49) (i.e. *updates*) use the same query syntax as queries to select documents to update.
- In `aggregation` pipeline, the `$match` pipeline stage provides access to MongoDB queries.

MongoDB provides a `db.collection.findOne()` method as a special case of `find()` that returns a single document.

Query Statements

Consider the following diagram of the query process that specifies a query criteria and a sort modifier:



In the diagram, the query selects documents from the `users` collection. Using a `query selection operator` to define the conditions for matching documents, the query selects documents that have `age` greater than (i.e. `$gt`) `18`. Then the `sort()` modifier sorts the results by `age` in ascending order.

For additional examples of queries, see *Query Documents* (page 43).

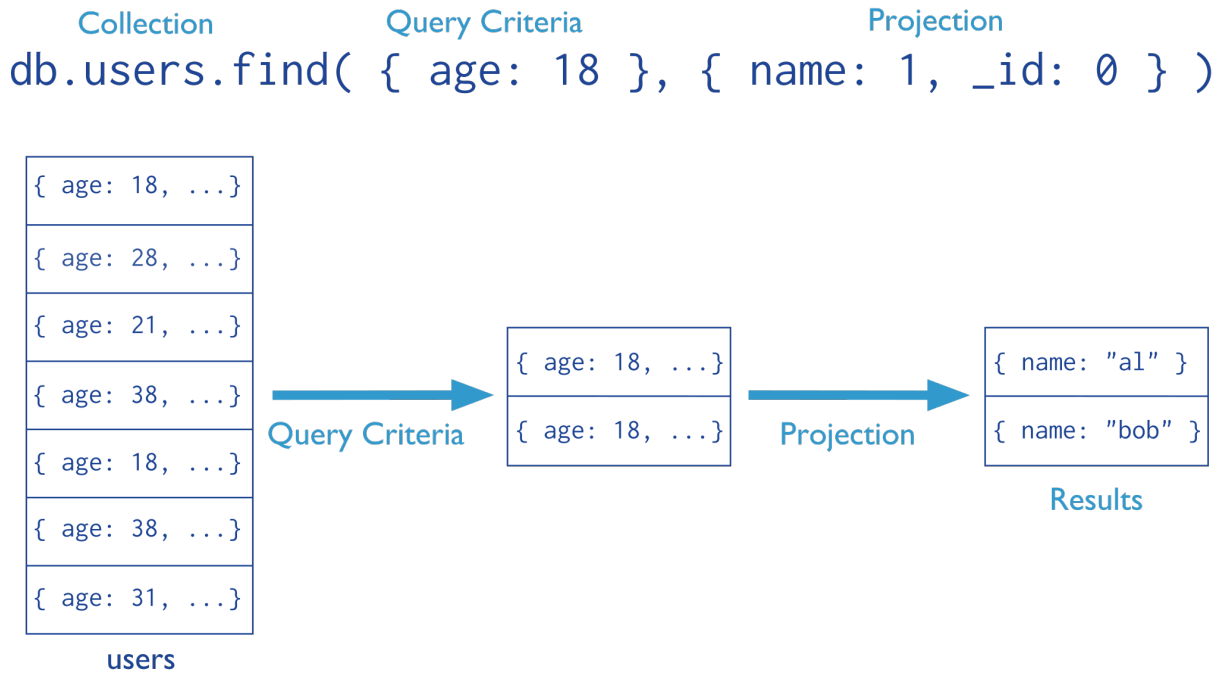
Projections

Queries in MongoDB return all fields in all matching documents by default. To limit the amount of data that MongoDB sends to applications, include a *projection* in the queries. By projecting results with a subset of fields, applications reduce their network overhead and processing requirements.

Projections, which are the *second* argument to the `find()` method, may either specify a list of fields to return *or* list fields to exclude in the result documents.

Important: Except for excluding the `_id` field in inclusive projections, you cannot mix exclusive and inclusive projections.

Consider the following diagram of the query process that specifies a query criteria and a projection:



In the diagram, the query selects from the `users` collection. The criteria matches the documents that have `age` equal to `18`. Then the projection specifies that only the `name` field should return in the matching documents.

Projection Examples

Exclude One Field From a Result Set

```
db.records.find( { "user_id": { $lt: 42 } }, { "history": 0 } )
```

This query selects documents in the `records` collection that match the condition `{ "user_id": { $lt: 42 } }`, and uses the projection `{ "history": 0 }` to exclude the `history` field from the documents in the result set.

Return Two fields *and* the `_id` Field

```
db.records.find( { "user_id": { $lt: 42 } }, { "name": 1, "email": 1 } )
```

This query selects documents in the `records` collection that match the query `{ "user_id": { $lt: 42 } }` and uses the projection `{ "name": 1, "email": 1 }` to return just the `_id` field (implicitly included), `name` field, and the `email` field in the documents in the result set.

Return Two Fields *and* Exclude `_id`

```
db.records.find( { "user_id": { $lt: 42} }, { "_id": 0, "name": 1 , "email": 1 } )
```

This query selects documents in the `records` collection that match the query `{ "user_id": { $lt: 42} }`, and only returns the `name` and `email` fields in the documents in the result set.

See

Limit Fields to Return from a Query (page 54) for more examples of queries with projection statements.

Projection Behavior MongoDB projections have the following properties:

- By default, the `_id` field is included in the results. To suppress the `_id` field from the result set, specify `_id: 0` in the projection document.
- For fields that contain arrays, MongoDB provides the following projection operators: `$elemMatch`, `$slice`, and `$`.
- For related projection functionality in the `aggregation framework` pipeline, use the `$project` pipeline stage.

Cursors

In the `mongo` shell, the primary method for the read operation is the `db.collection.find()` method. This method queries a collection and returns a *cursor* to the returning documents.

To access the documents, you need to iterate the cursor. However, in the `mongo` shell, if the returned cursor is not assigned to a variable using the `var` keyword, then the cursor is automatically iterated up to 20 times [1] to print up to the first 20 documents in the results.

For example, in the `mongo` shell, the following read operation queries the `inventory` collection for documents that have `type` equal to `'food'` and automatically print up to the first 20 matching documents:

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

To manually iterate the cursor to access the documents, see *Iterate a Cursor in the mongo Shell* (page 56).

Cursor Behaviors

Closure of Inactive 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[2] 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. For the `mongo` shell, see `cursor.addOption()` for a complete list of available cursor flags.

Cursor Isolation Because the cursor is not isolated during its lifetime, intervening write operations on a document may result in a cursor that returns a document more than once if that document has changed. To handle this situation, see the information on *snapshot mode*.

[1] 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.

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

Cursor Batches The MongoDB server returns the query results in batches. Batch size will not exceed the *maximum BSON document size*. 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 before returning any results.

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();
```

Cursor Information

The `db.serverStatus()` method returns a document that includes a `metrics` field. The `metrics` field contains a `cursor` field with the following information:

- number of timed out cursors since the last server restart
- number of open cursors with the option `DBQuery.Option.noTimeout` set to prevent timeout after a period of inactivity
- number of "pinned" open cursors
- total number of open cursors

Consider the following example which calls the `db.serverStatus()` method and accesses the `metrics` field from the results and then the `cursor` field from the `metrics` field:

```
db.serverStatus().metrics.cursor
```

The result is the following document:

```
{
   "timedOut" : <number>
   "open" : {
      "noTimeout" : <number>,
      "pinned" : <number>,
      "total" : <number>
   }
}
```

See also:

`db.serverStatus()`

Query Optimization

Indexes improve the efficiency of read operations by reducing the amount of data that query operations need to process. This simplifies the work associated with fulfilling queries within MongoDB.

Create an Index to Support Read Operations

If your application queries a collection on a particular field or set of fields, then an index on the queried field or a compound index on the set of fields can prevent the query from scanning the whole collection to find and return the query results. For more information about indexes, see the complete documentation of indexes in MongoDB.

Example

An application queries the inventory collection on the type field. The value of the type field is user-driven.

```
var typeValue = <someUserInput>;
db.inventory.find( { type: typeValue } );
```

To improve the performance of this query, add an ascending, or a descending, index to the inventory collection on the type field. [3] In the mongo shell, you can create indexes using the db.collection.createIndex() method:

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

This index can prevent the above query on type from scanning the whole collection to return the results.

To analyze the performance of the query with an index, see *Analyze Query Performance* (page 57).

In addition to optimizing read operations, indexes can support sort operations and allow for a more efficient storage utilization. See db.collection.createIndex() and http://docs.mongodb.org/manual/administration/indexes for more information about index creation.

Query Selectivity

Query selectivity refers to how well the query predicate excludes or filters out documents in a collection. Query selectivity can determine whether or not queries can use indexes effectively or even use indexes at all.

More selective queries match a smaller percentage of documents. For instance, an equality match on the unique _id field is highly selective as it can match at most one document.

Less selective queries match a larger percentage of documents. Less selective queries cannot use indexes effectively or even at all.

For instance, the inequality operators $nin and $ne are *not* very selective since they often match a large portion of the index. As a result, in many 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.

The selectivity of regular expressions depends on the expressions themselves. For details, see *regular expression and index use*.

Covering a Query

An index *covers* (page 12) a query when both of the following apply:

- all the fields in the *query* (page 43) are part of an index, **and**
- all the fields returned in the results are in the same index.

[3] For single-field indexes, the selection between ascending and descending order is immaterial. For compound indexes, the selection is important. See *indexing order* for more details.

For example, a collection `inventory` has the following index on the `type` and `item` fields:

```
db.inventory.createIndex( { type: 1, item: 1 } )
```

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

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

For the specified index to cover the query, the projection document must explicitly specify `_id: 0` to exclude the `_id` field from the result since the index does not include the `_id` field.

Performance Because the index contains all fields required by the query, MongoDB can both match the *query conditions* (page 43) and return the results using only the index.

Querying *only* the index can be much faster than querying documents outside of the index. Index keys are typically smaller than the documents they catalog, and indexes are typically available in RAM or located sequentially on disk.

Limitations

Restrictions on Indexed Fields An index **cannot** cover a query if:

- any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a *multi-key index* index and cannot support a covered query.
- any of the indexed field in the query predicate or returned in the projection are fields in embedded documents. [4] For example, consider a collection `users` with documents of the following form:

  ```
  { _id: 1, user: { login: "tester" } }
  ```

  The collection has the following index:

  ```
  { "user.login": 1 }
  ```

  The `{ "user.login": 1 }` index does **not** cover the following query:

  ```
  db.users.find( { "user.login": "tester" }, { "user.login": 1, _id: 0 } )
  ```

  However, the query can use the `{ "user.login": 1 }` index to find matching documents.

Restrictions on Sharded Collection An index cannot cover a query on a *sharded* collection when run against a `mongos` if the index does not contain the shard key, with the following exception for the `_id` index: If a query on a sharded collection only specifies a condition on the `_id` field and returns only the `_id` field, the `_id` index can cover the query when run against a `mongos` even if the `_id` field is not the shard key.

Changed in version 3.0.

In previous versions, an index cannot *cover* (page 12) a query on a *sharded* collection when run against a `mongos`.

[4] To index fields in embedded documents, use *dot notation*.

explain To determine whether a query is a covered query, use the `db.collection.explain()` or the `explain()` method and review the *results*.

`db.collection.explain()` provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

For more information see *indexes-measuring-use*.

Query Plans

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 only caches the plans for those query shapes that can have more than one viable plan.

The query optimizer occasionally reevaluates query plans as the content of the collection changes to ensure optimal query plans. You can also specify which indexes the optimizer evaluates with *Index Filters* (page 15).

You can use the `db.collection.explain()` or the `cursor.explain()` method to view statistics about the query plan for a given query. This information can help as you develop `indexing strategies`.

`db.collection.explain()` provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

Query Optimization

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' } )
```

Query Plan Revision

As collections change over time, the query optimizer deletes the query plan and re-evaluates 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.
- You run `db.collection.explain()` or `cursor.explain()`.

Cached Query Plan Interface

New in version 2.6.

MongoDB provides `http://docs.mongodb.org/manual/reference/method/js-plan-cache` to view and modify the cached query plans.

Index Filters

New in version 2.6.

Index filters determine which indexes the optimizer evaluates for a *query shape*. A query shape consists of a combination of query, sort, and projection specifications. If an index filter exists for a given query shape, the optimizer only considers those indexes specified in the filter.

When an index filter exists for the query shape, MongoDB ignores the `hint()`. To see whether MongoDB applied an index filter for a query shape, check the `indexFilterSet` field of either the `db.collection.explain()` or the `cursor.explain()` method.

Index filters only affects which indexes the optimizer evaluates; the optimizer may still select the collection scan as the winning plan for a given query shape.

Index filters exist for the duration of the server process and do not persist after shutdown. MongoDB also provides a command to manually remove filters.

Because index filters overrides the expected behavior of the optimizer as well as the `hint()` method, use index filters sparingly.

See `planCacheListFilters`, `planCacheClearFilters`, and `planCacheSetFilter`.

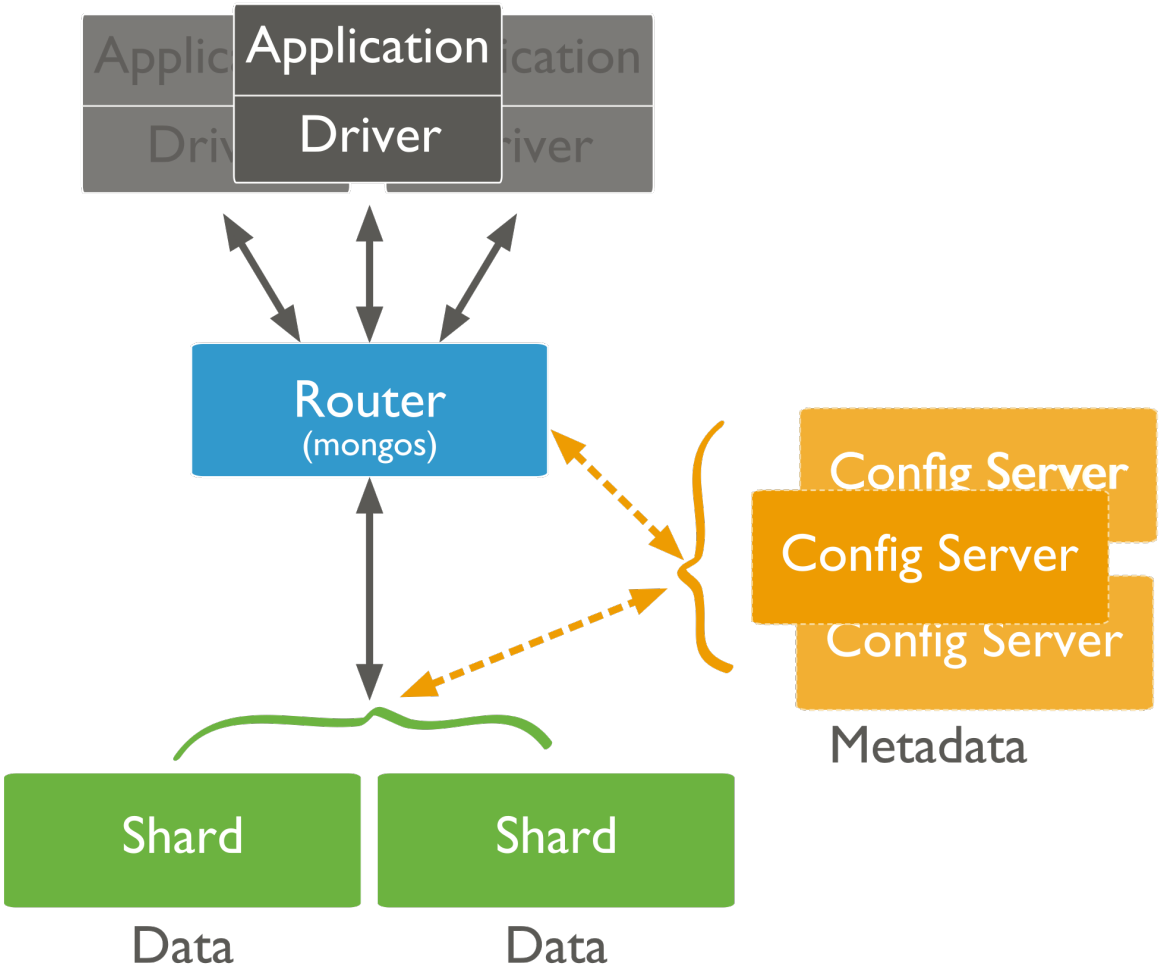
Distributed Queries

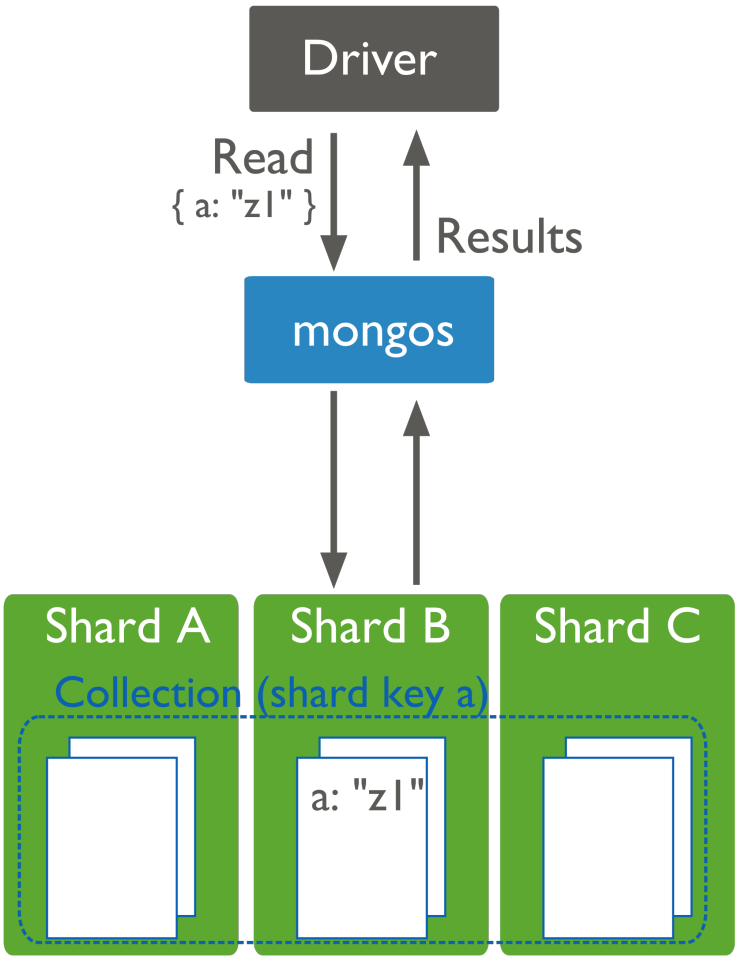
Read Operations to Sharded Clusters

Sharded clusters allow you to partition a data set among a cluster of `mongod` instances in a way that is nearly transparent to the application. For an overview of sharded clusters, see the `http://docs.mongodb.org/manual/sharding` section of this manual.

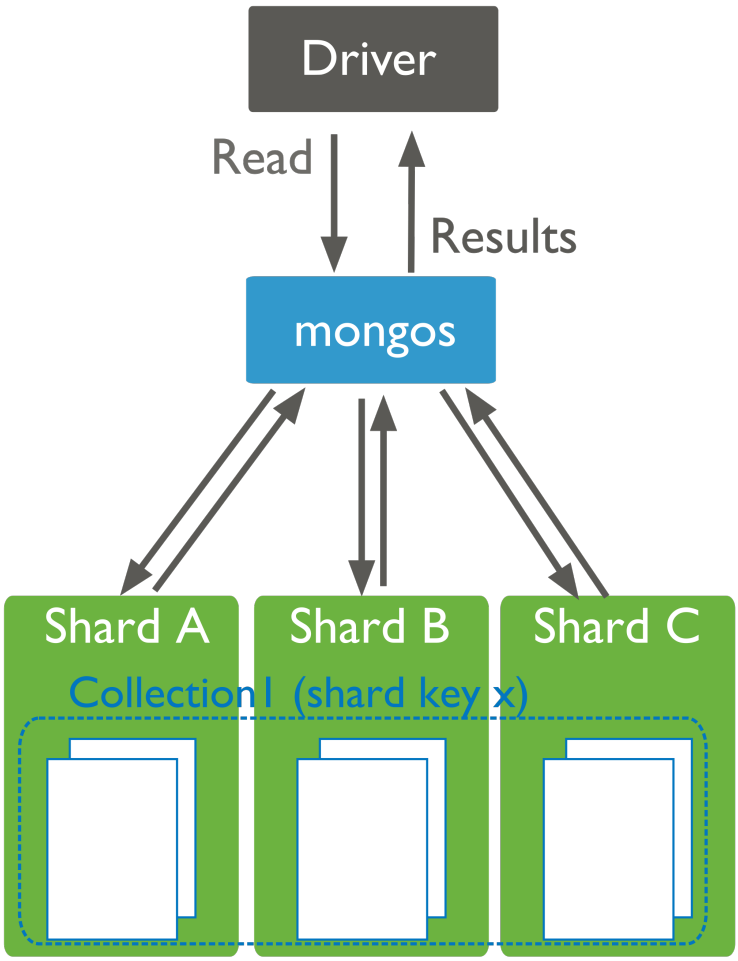
For a sharded cluster, applications issue operations to one of the `mongos` instances associated with the cluster.

Read operations on sharded clusters are most efficient when directed to a specific shard. Queries to sharded collections should include the collection's *shard key*. When a query includes a shard key, the `mongos` can use cluster metadata from the *config database* to route the queries to shards.


Application
Driver
Router
(mongos)
Config Server
Config Server
Config Server
Metadata
Shard
Shard
Data
Data


Driver
Read
{ a: "z1" }
Results
mongos
Shard A
Shard B
Shard C
Collection (shard key a)
a: "z1"

If a query does not include the shard key, the `mongos` must direct the query to *all* shards in the cluster. These *scatter gather* queries can be inefficient. On larger clusters, scatter gather queries are unfeasible for routine operations.



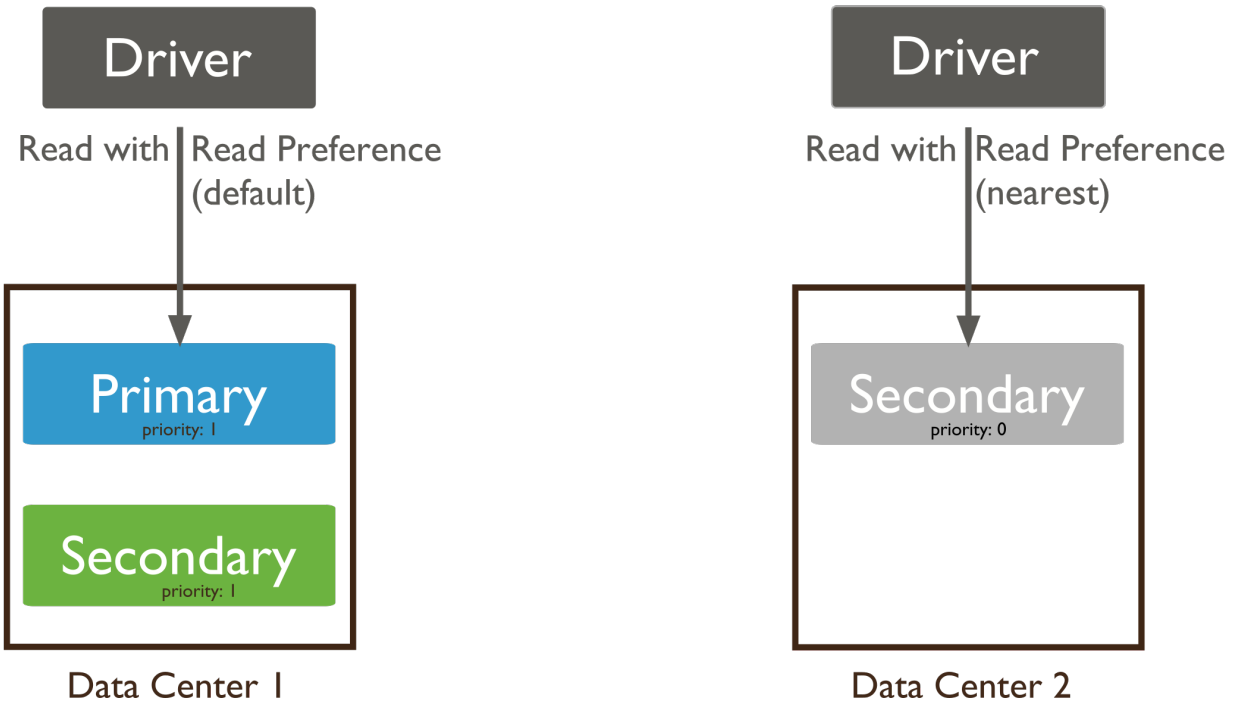
For more information on read operations in sharded clusters, see the `http://docs.mongodb.org/manual/core/sharded-cl` and *sharding-shard-key* sections.

Read Operations to 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 preference or on the read preference modes, see `http://docs.mongodb.org/manual/core/read-preference` and *replica-set-read-preference-modes*.

2.2 Write Operations

The following documents describe write operations:

Write Operations Overview **(page 20)** Provides an overview of MongoDB's data insertion and modification operations, including aspects of the syntax, and behavior.

Write Concern **(page 23)** Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.

Atomicity and Transactions **(page 28)** Describes write operation atomicity in MongoDB.

Distributed Write Operations **(page 28)** Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

Write Operation Performance **(page 33)** Introduces the performance constraints and factors for writing data to MongoDB deployments.

Bulk Write Operations **(page 34)** Provides an overview of MongoDB's bulk write operations.

Storage **(page 36)** Introduces the storage allocation strategies available for MongoDB collections.

Write Operations Overview

A write operation is any operation that creates or modifies data in the MongoDB instance. In MongoDB, write operations target a single *collection*. All write operations in MongoDB are atomic on the level of a single *document*.

There are three classes of write operations in MongoDB: *insert* (page 20), *update* (page 21), and *remove* (page 22). Insert operations add new data to a collection. Update operations modify existing data, and remove operations delete data from a collection. No insert, update, or remove can affect more than one document atomically.

For the update and remove operations, you can specify criteria, or conditions, that identify the documents to update or remove. These operations use the same query syntax to specify the criteria as *read operations* (page 6).

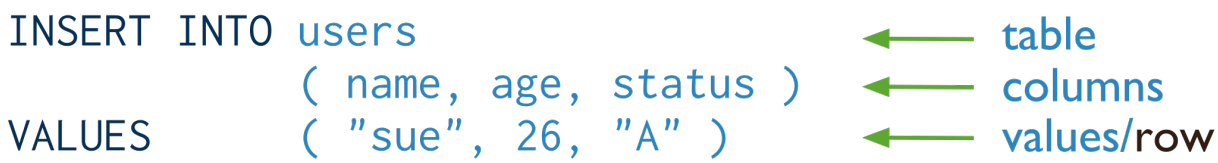
MongoDB allows applications to determine the acceptable level of acknowledgement required of write operations. See *Write Concern* (page 23) for more information.

Insert

In MongoDB, the `db.collection.insert()` method adds new *documents* to a collection.

The following diagram highlights the components of a MongoDB insert operation:

The following diagram shows the same query in SQL:



Example

The following operation inserts a new document into the `users` collection. The new document has four fields `name`, `age`, and `status`, and an `_id` field. MongoDB always adds the `_id` field to the new document if that field does not exist.

```
db.users.insert(
   {
      name: "sue",
```

```
      age: 26,
      status: "A"
   }
)
```

For more information and examples, see `db.collection.insert()`.

Insert Behavior If you add a new document *without* the *_id* field, the client library or the `mongod` instance adds an `_id` field and populates the field with a unique *ObjectId*.

If you specify the `_id` field, the value must be unique within the collection. For operations with *write concern* (page 23), if you try to create a document with a duplicate `_id` value, `mongod` returns a duplicate key exception.

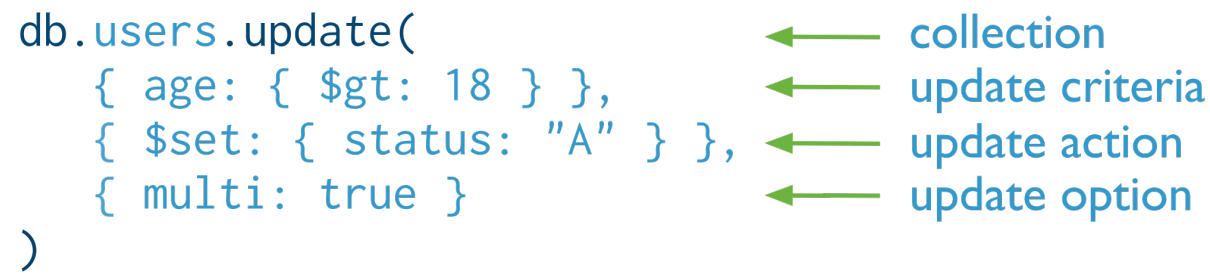
Other Methods to Add Documents You can also add new documents to a collection using methods that have an *upsert* (page 22) option. If the option is set to `true`, these methods will either modify existing documents or add a new document when no matching documents exist for the query. For more information, see *Update Behavior with the upsert Option* (page 22).

Update

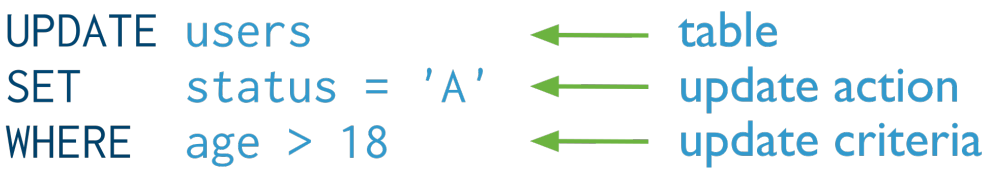
In MongoDB, the `db.collection.update()` method modifies existing *documents* in a *collection*. The `db.collection.update()` method can accept query criteria to determine which documents to update as well as an options document that affects its behavior, such as the `multi` option to update multiple documents.

Operations performed by an update are atomic within a single document. For example, you can safely use the `$inc` and `$mul` operators to modify frequently-changed fields in concurrent applications.

The following diagram highlights the components of a MongoDB update operation:



The following diagram shows the same query in SQL:



Example

```
db.users.update(
   { age: { $gt: 18 } },
   { $set: { status: "A" } },
   { multi: true }
)
```

This update operation on the `users` collection sets the `status` field to `A` for the documents that match the criteria of `age` greater than `18`.

For more information, see `db.collection.update()` and *update() Examples*.

Default Update Behavior By default, the `db.collection.update()` method updates a **single** document. However, with the `multi` option, `update()` can update all documents in a collection that match a query.

The `db.collection.update()` method either updates specific fields in the existing document or replaces the document. See `db.collection.update()` for details as well as examples.

When performing update operations that increase the document size beyond the allocated space for that document, the update operation relocates the document on disk.

MongoDB preserves the order of the document fields following write operations *except* for the following cases:

- The `_id` field is always the first field in the document.
- Updates that include `renaming` of field names may result in the reordering of fields in the document.

Changed in version 2.6: Starting in version 2.6, MongoDB actively attempts to preserve the field order in a document. Before version 2.6, MongoDB did not actively preserve the order of the fields in a document.

Update Behavior with the `upsert` Option If the `update()` method includes *upsert: true and* no documents match the query portion of the update operation, then the update operation creates a new document. If there are matching documents, then the update operation with the *upsert: true* modifies the matching document or documents.

By specifying *upsert: true*, applications can indicate, in a *single* operation, that if no matching documents are found for the update, an insert should be performed. See `update()` for details on performing an *upsert*.

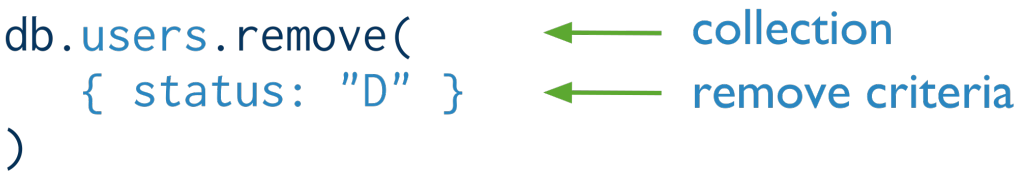
Changed in version 2.6: In 2.6, the new `Bulk()` methods and the underlying `update` command allow you to perform many updates with `upsert: true` operations in a single call.

If you create documents using the `upsert` option to `update()` consider using a *a unique index* to prevent duplicated operations.

Remove

In MongoDB, the `db.collection.remove()` method deletes documents from a collection. The `db.collection.remove()` method accepts a query criteria to determine which documents to remove.

The following diagram highlights the components of a MongoDB remove operation:



The following diagram shows the same query in SQL:

Example

```
db.users.remove(
   { status: "D" }
)
```

This delete operation on the `users` collection removes all documents that match the criteria of `status` equal to `D`.

For more information, see `db.collection.remove()` method and *Remove Documents* (page 53).

Remove Behavior By default, `db.collection.remove()` method removes all documents that match its query. However, the method can accept a flag to limit the delete operation to a single document.

Isolation of Write Operations

The modification of a single document is always atomic, even if the write operation modifies multiple embedded documents *within* that document. No other operations are atomic.

If a write operation modifies multiple documents, the operation as a whole is not atomic, and other operations may interleave. You can, however, attempt to isolate a write operation that affects multiple documents using the `isolation operator`.

For more information *Atomicity and Transactions* (page 28).

Additional Methods

The `db.collection.save()` method can either update an existing document or insert a document if the document cannot be found by the `_id` field. See `db.collection.save()` for more information and examples.

MongoDB also provides methods to perform write operations in bulk. See `Bulk()` for more information.

Write Concern

Write concern describes the guarantee that MongoDB provides when reporting on the success of a write operation. The strength of the write concerns determine the level of guarantee. When inserts, updates and deletes have a *weak* write concern, write operations return quickly. In some failure cases, write operations issued with weak write concerns may not persist. With *stronger* write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations.

MongoDB provides different levels of write concern to better address the specific needs of applications. Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment. For other less critical operations, clients can adjust the write concern to ensure faster performance rather than ensure persistence to the entire deployment.

Changed in version 2.6: A new protocol for *write operations* integrates write concern with the write operations.

For details on write concern configurations, see *Write Concern Reference* (page 76).

Considerations

Default Write Concern The `mongo` shell and the MongoDB drivers use *Acknowledged* (page 24) as the default write concern.

See *Acknowledged* (page 24) for more information, including when this write concern became the default.

Read Isolation MongoDB allows clients to read documents inserted or modified before it commits these modifications to disk, regardless of write concern level or journaling configuration. As a result, applications may observe two classes of behaviors:

- For systems with multiple concurrent readers and writers, MongoDB will allow clients to read the results of a write operation before the write operation returns.
- If the `mongod` terminates before the journal commits, even if a write returns successfully, queries may have read data that will not exist after the `mongod` restarts.

Other database systems refer to these isolation semantics as *read uncommitted*. For all inserts and updates, MongoDB modifies each document in isolation: clients never see documents in intermediate states. For multi-document operations, MongoDB does not provide any multi-document transactions or isolation.

When `mongod` returns a successful *journaled write concern*, the data is fully committed to disk and will be available after `mongod` restarts.

For replica sets, write operations are durable only after a write replicates and commits to the journal of a majority of the voting members of the set. MongoDB regularly commits data to the journal regardless of journaled write concern: use the `commitIntervalMs` to control how often a `mongod` commits the journal.

Timeouts Clients can set a *wtimeout* (page 77) value as part of a *replica acknowledged* (page 25) write concern. If the write concern is not satisfied in the specified interval, the operation returns an error, even if the write concern will eventually succeed.

MongoDB does not "rollback" or undo modifications made before the `wtimeout` interval expired.

Write Concern Levels

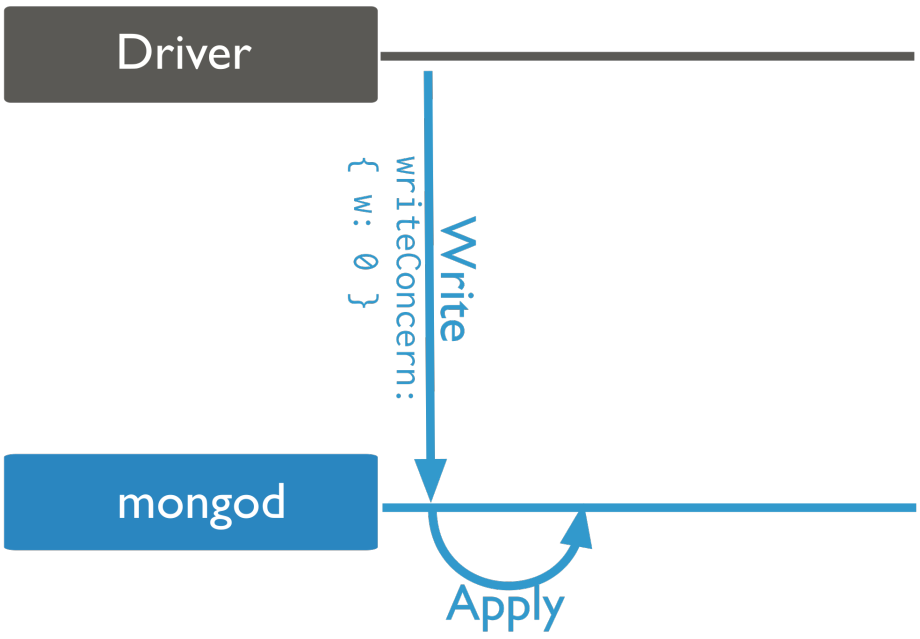
MongoDB has the following levels of conceptual write concern, listed from weakest to strongest:

Unacknowledged With an *unacknowledged* write concern, MongoDB does not acknowledge the receipt of write operations. *Unacknowledged* is similar to *errors ignored*; however, drivers will attempt to receive and handle network errors when possible. The driver's ability to detect network errors depends on the system's networking configuration.

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

Acknowledged With a receipt *acknowledged* write concern, the `mongod` confirms that it received the write operation and applied the change to the in-memory view of data. *Acknowledged* write concern allows clients to catch network, duplicate key, and other errors.

MongoDB uses the *acknowledged* write concern by default starting in the driver releases outlined in *write-concern-change-releases*.



Changed in version 2.6: The `mongo` shell write methods now incorporates the *write concern* (page 23) in the write methods and provide the default write concern whether run interactively or in a script. See *write-methods-incompatibility* for details.

Acknowledged write concern does *not* confirm that the write operation has persisted to the disk system.

Journaled With a *journaled* write concern, the MongoDB acknowledges the write operation only after committing the data to the *journal*. This write concern ensures that MongoDB can recover the data following a shutdown or power interruption.

You must have journaling enabled to use this write concern.

With a *journaled* write concern, write operations must wait for the next journal commit. To reduce latency for these operations, MongoDB also increases the frequency that it commits operations to the journal. See `commitIntervalMs` 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.

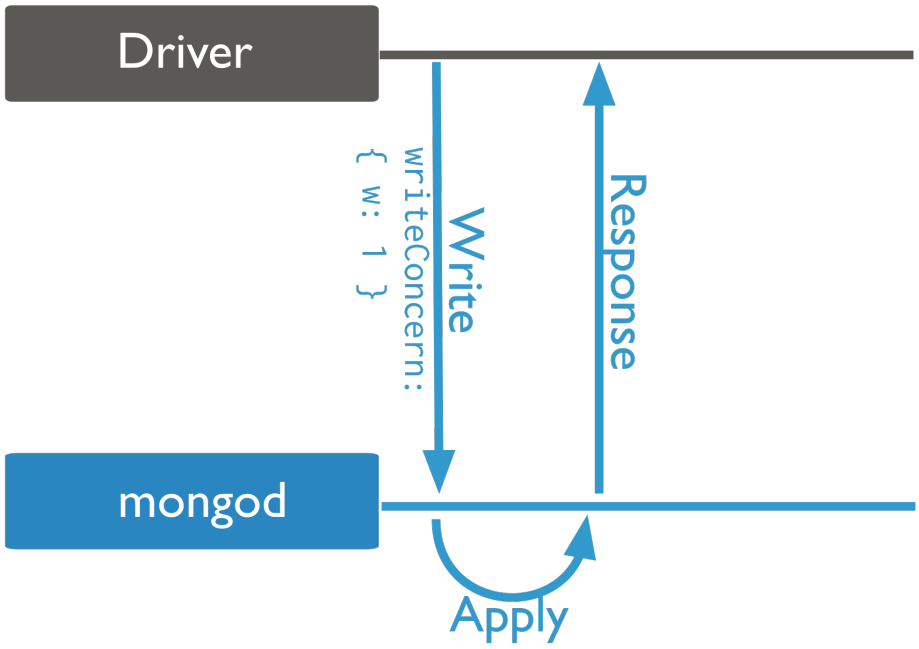
Replica Acknowledged *Replica sets* present additional considerations with regards to write concern. The default write concern only requires acknowledgement from the primary.

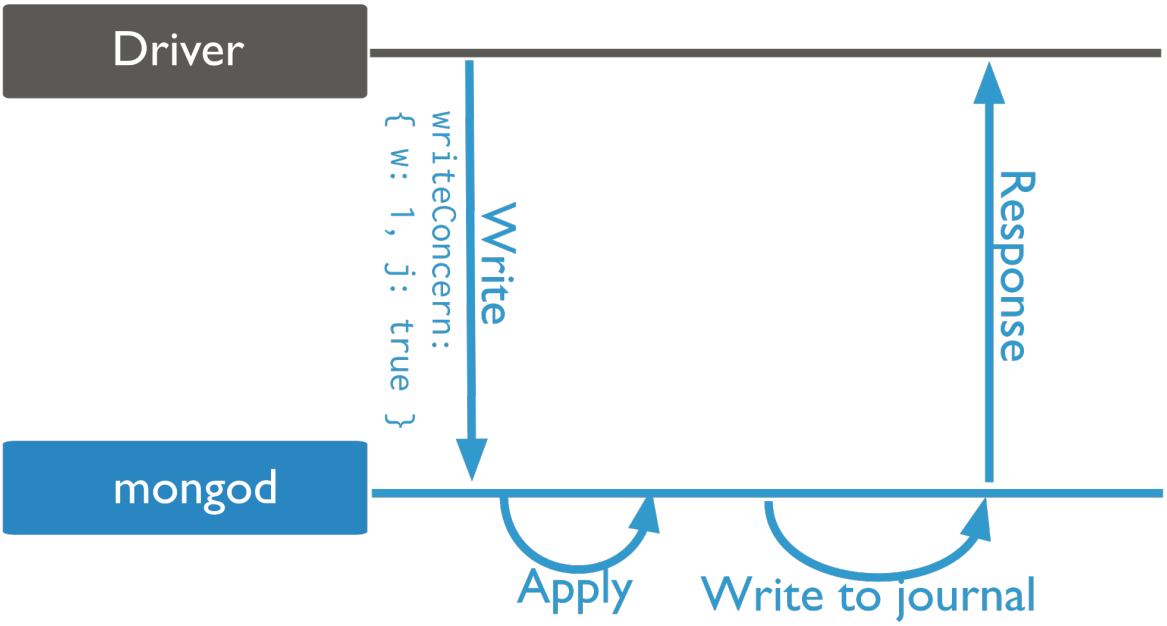
With *replica acknowledged* write concern, you can guarantee that the write operation propagates to additional members of the replica set. See `Write Concern for Replica Sets` for more information.

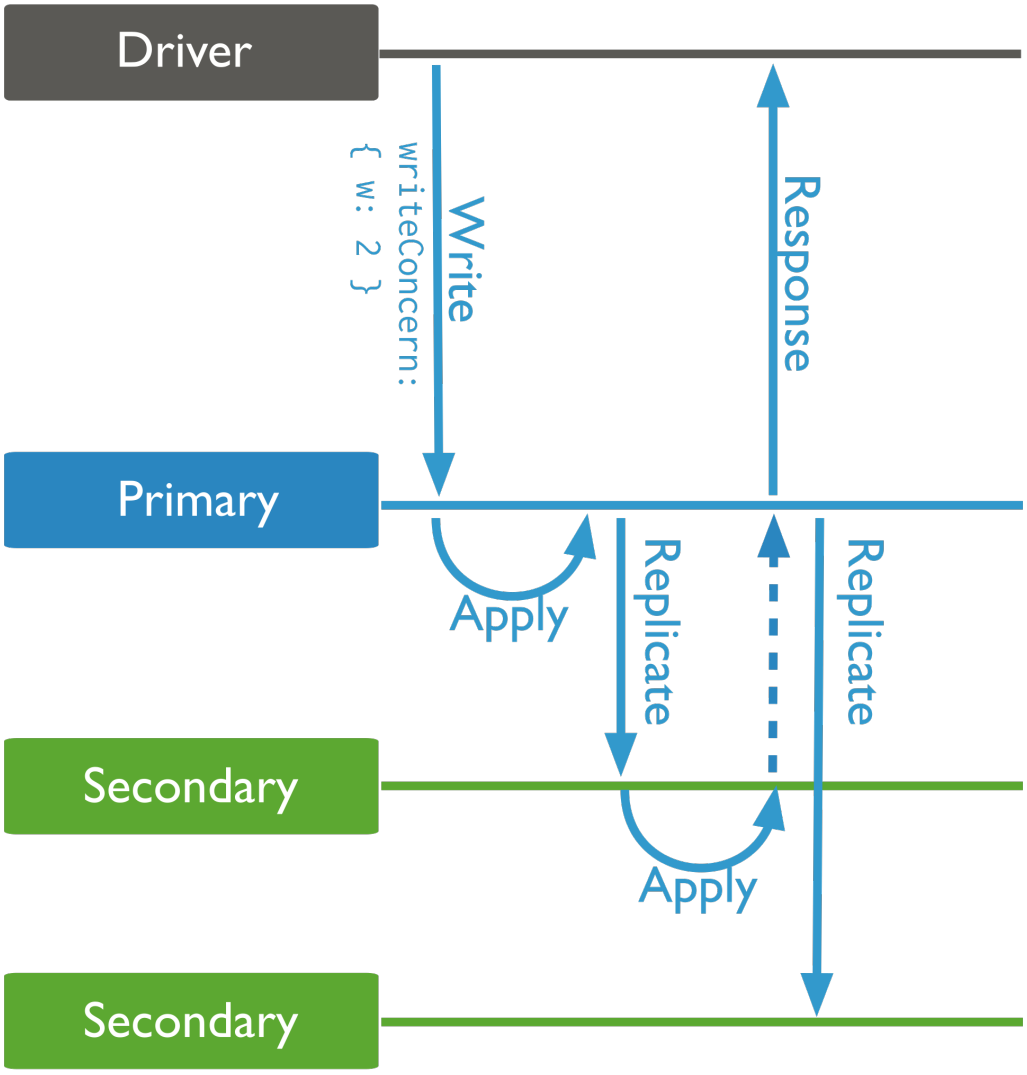
Note: Requiring *journaled* write concern in a replica set only requires a journal commit of the write operation to the *primary* of the set regardless of the level of *replica acknowledged* write concern.

See also:

Write Concern Reference (page 76)


Driver
writeConcern:
{ w: 1 }
Write
Response
mongod
Apply


Driver
writeConcern:
{ w: 1, j: true }
Write
Response
mongod
Apply
Write to journal


Driver
Write
writeConcern:
{ w: 2 }
Response
Primary
Apply
Replicate
Replicate
Secondary
Apply
Secondary

Atomicity and Transactions

In MongoDB, a write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents *within* a single document.

When a single write operation modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic and other operations may interleave. However, you can *isolate* a single write operation that affects multiple documents using the `$isolated` operator.

`$isolated` Operator

Using the `$isolated` operator, a write operation that affect multiple documents can prevent other processes from interleaving once the write operation modifies the first document. This ensures that no client sees the changes until the write operation completes or errors out.

Isolated write operation does not provide "all-or-nothing" atomicity. That is, an error during the write operation does not roll back all its changes that preceded the error.

The `$isolated` does **not** work on sharded clusters.

For an example of an update operation that uses the `$isolated` operator, see `$isolated`. For an example of a remove operation that uses the `$isolated` operator, see *isolate-remove-operations*.

Transaction-Like Semantics

Since a single document can contain multiple embedded documents, single-document atomicity is sufficient for many practical use cases. For cases where a sequence of write operations must operate as if in a single transaction, you can implement a *two-phase commit* (page 62) in your application.

However, two-phase commits can only offer transaction-*like* semantics. Using two-phase commit ensures data consistency, but it is possible for applications to return intermediate data during the two-phase commit or rollback.

For more information on two-phase commit and rollback, see *Perform Two Phase Commits* (page 62).

Concurrency Control

Concurrency control allows multiple applications to run concurrently without causing data inconsistency or conflicts.

An approach may be to create a *unique index* on a field (or fields) that should have only unique values (or unique combination of values) prevents duplicate insertions or updates that result in duplicate values. For examples of use cases, see *update() and Unique Index* and *findAndModify() and Unique Index*.

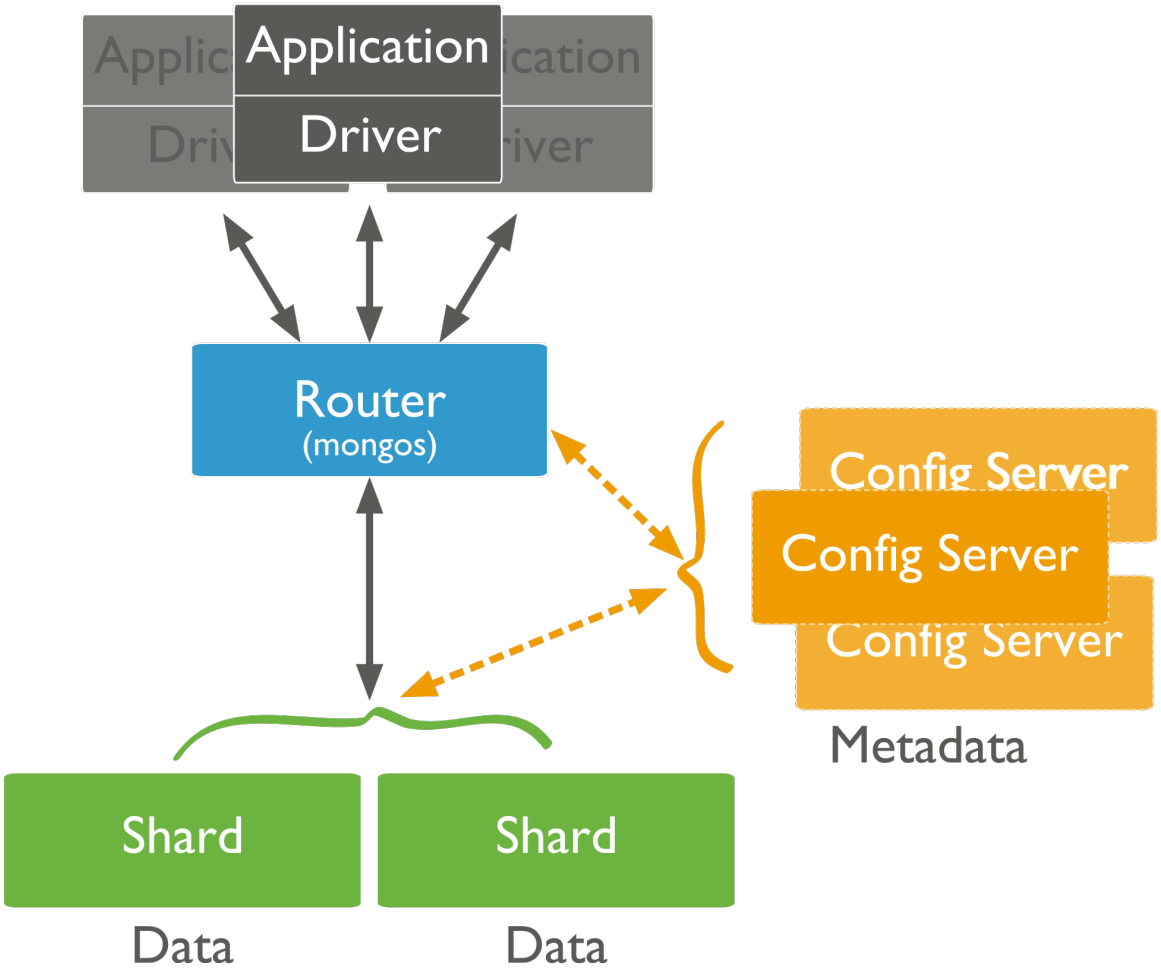
Another approach is to specify the expected current value of a field in the query predicate for the write operations. For an example, see *Update if Current* (page 68).

The two-phase commit pattern provides a variation where the query predicate includes the *application identifier* (page 66) as well as the expected state of the data in the write operation.

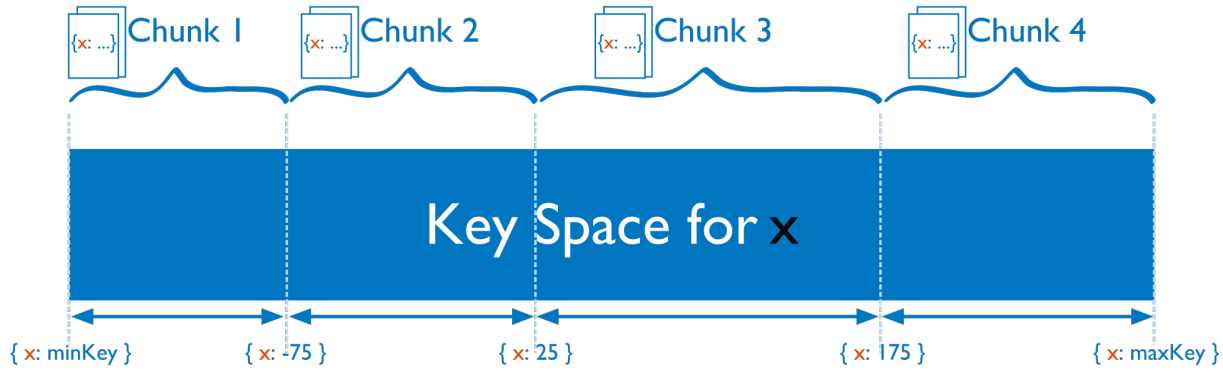
Distributed Write Operations

Write Operations on Sharded Clusters

For sharded collections in a *sharded cluster*, the `mongos` directs write operations from applications to the shards that are responsible for the specific *portion* of the data set. The `mongos` uses the cluster metadata from the *config database* to route the write operation to the appropriate shards.


Application
Driver
Router
(mongos)
Config Server
Config Server
Config Server
Metadata
Shard
Shard
Data
Data

MongoDB partitions data in a sharded collection into *ranges* based on the values of the *shard key*. Then, MongoDB distributes these chunks to shards. The shard key determines the distribution of chunks to shards. This can affect the performance of write operations in the cluster.



Important: Update operations that affect a *single* document **must** include the *shard key* or the `_id` field. Updates that affect multiple documents are more efficient in some situations if they have the *shard key*, but can be broadcast to all shards.

If the value of the shard key increases or decreases with every insert, all insert operations target a single shard. As a result, the capacity of a single shard becomes the limit for the insert capacity of the sharded cluster.

For more information, see `http://docs.mongodb.org/manual/administration/sharded-clusters` and *Bulk Write Operations* (page 34).

Write Operations on 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.

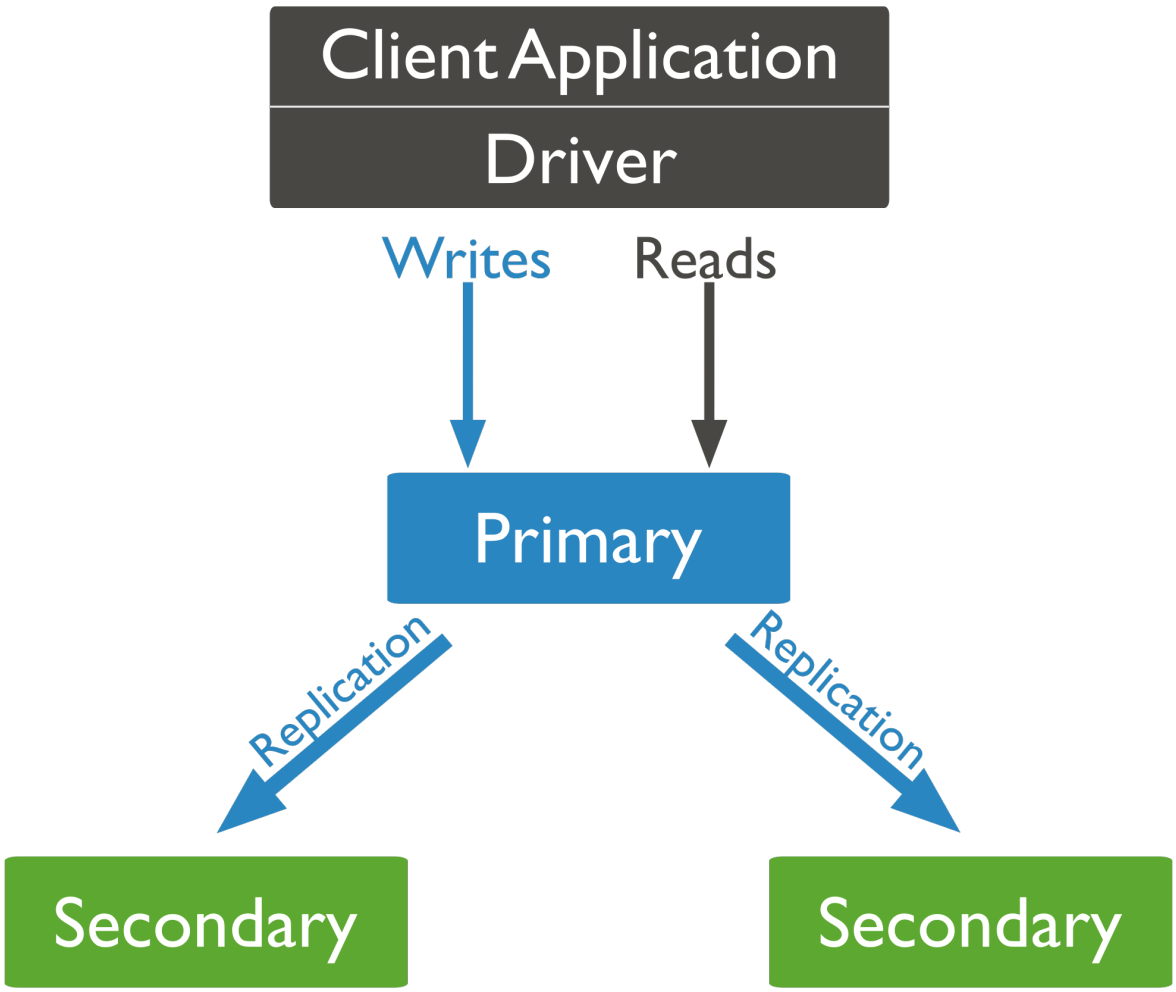
Large volumes of write operations, particularly bulk operations, may create situations where the secondary members have difficulty applying the replicating operations from the primary at a sufficient rate: this can cause the secondary's state to fall behind that of the primary. Secondaries that are significantly 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`.

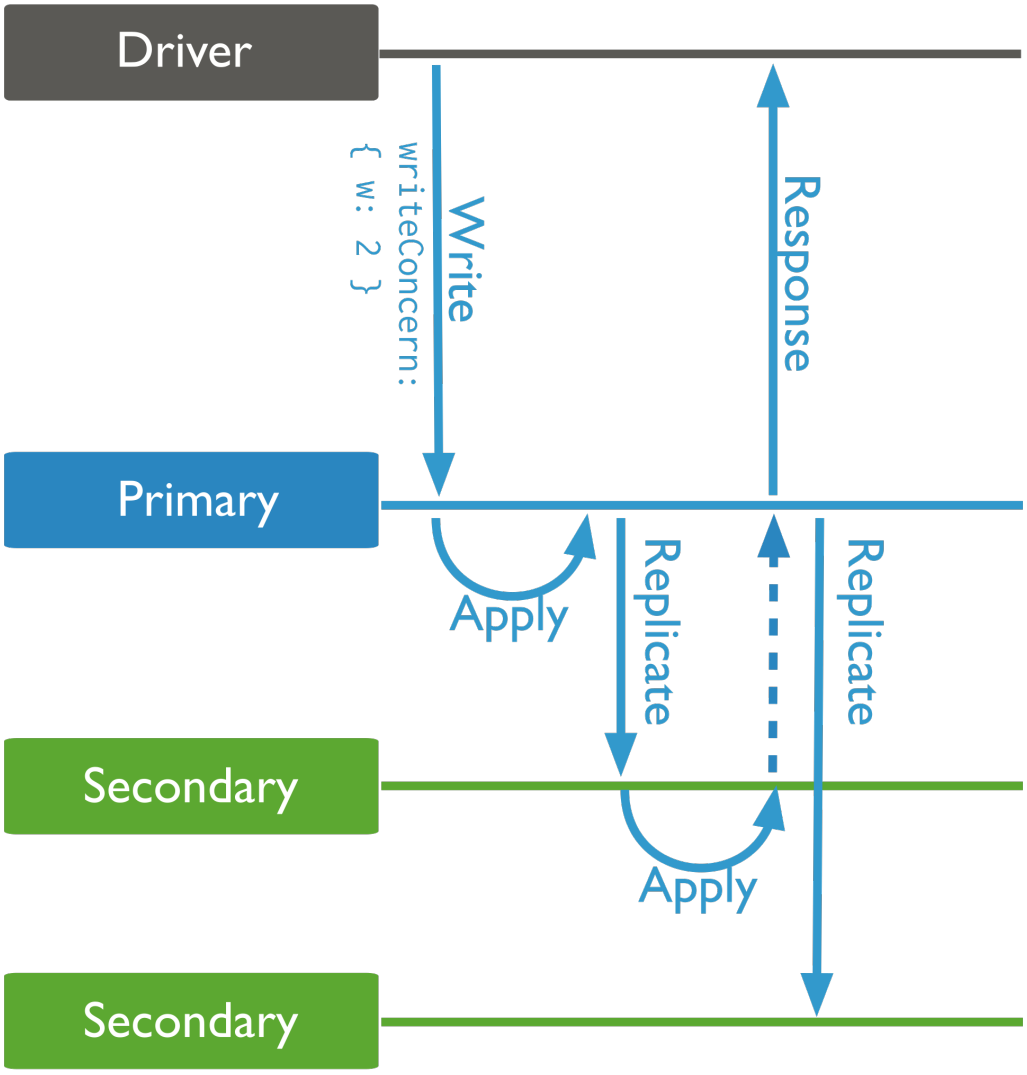
To help avoid this issue, you can customize the *write concern* (page 23) to return confirmation of the write operation to another member [5] of the replica set every 100 or 1,000 operations. This provides an opportunity for secondaries to catch up with the primary. Write concern can slow the overall 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 Acknowledged* (page 25), *replica-set-oplog-sizing*, and `http://docs.mongodb.org/manual/tutorial/change-oplog-size`.

[5] Intermittently issuing a write concern 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.

Changed in version 2.6: In `Master/Slave` deployments, MongoDB treats `w: "majority"` as equivalent to `w: 1`. In earlier versions of MongoDB, `w: "majority"` produces an error in `master/slave` deployments.


Client Application
Driver
Writes
Reads
Primary
Replication
Replication
Secondary
Secondary


Driver
Write
writeConcern:
{ w: 2 }
Response
Primary
Apply
Replicate
Replicate
Secondary
Apply
Secondary

Write Operation Performance

Indexes

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. [6]

In general, the performance gains that indexes provide for *read operations* are worth the insertion penalty. However, in order to optimize write performance when possible, be careful when creating new indexes and evaluate the existing indexes to ensure that your queries actually use these indexes.

For indexes and queries, see *Query Optimization* (page 11). For more information on indexes, see `http://docs.mongodb.org/manual/indexes` and `http://docs.mongodb.org/manual/applications/indexes`.

Document Growth and the MMAPv1 Storage Engine

Some update operations can increase the size of the document; for instance, if an update adds a new field to the document.

For the MMAPv1 storage engine, if an update operation causes a document to exceed the currently allocated *record size*, MongoDB relocates the document on disk with enough contiguous space to hold the document. Updates that require relocations take longer than updates that do not, particularly if the collection has indexes. If a collection has indexes, MongoDB must update all index entries. Thus, for a collection with many indexes, the move will impact the write throughput.

Changed in version 3.0.0: By default, MongoDB uses *Power of 2 Sized Allocations* (page 38) to add *padding automatically* (page 38) for the MMAPv1 storage engine. The *Power of 2 Sized Allocations* (page 38) ensures that MongoDB allocates document space in sizes that are powers of 2, which helps ensure that MongoDB can efficiently reuse free space created by document deletion or relocation as well as reduce the occurrences of reallocations in many cases.

Although ref:*power-of-2-allocation* minimizes the minimize the occurrence of reallocation, it does not eliminate document re-allocation.

See *Storage* (page 36) for more information.

Storage Performance

Hardware The capability of the storage system creates some important physical limits for the performance of MongoDB's write operations. Many unique factors related to the storage system of the drive affect write performance, including random access patterns, disk caches, disk readahead and RAID configurations.

Solid state drives (SSDs) can outperform spinning hard disks (HDDs) by 100 times or more for random workloads.

See

`http://docs.mongodb.org/manual/administration/production-notes` for recommendations regarding additional hardware and configuration options.

---

[6] For inserts and updates to un-indexed fields, the overhead for *sparse indexes* is less than for non-sparse indexes. Also for non-sparse indexes, updates that do not change the record size have less indexing overhead.

Journaling MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* (page 19) durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal.

While the durability assurance provided by the journal typically outweigh the performance costs of the additional write operations, consider the following interactions between the journal and performance:

- if the journal and the data file reside on the same block device, the data files and the journal may have to contend for a finite number of available write operations. Moving the journal to a separate device may increase the capacity for write operations.
- if applications specify *write concern* (page 23) that includes *journaled* (page 25), `mongod` will decrease the duration between journal commits, which can increases the overall write load.
- the duration between journal commits is configurable using the `commitIntervalMs` run-time option. Decreasing the period between journal commits will increase the number of write operations, which can limit MongoDB's capacity for write operations. Increasing the amount of time between commits may decrease the total number of write operation, but also increases the chance that the journal will not record a write operation in the event of a failure.

For additional information on journaling, see `http://docs.mongodb.org/manual/core/journaling`.

Bulk Write Operations

Overview

MongoDB provides clients the ability to perform write operations in bulk. Bulk write operations affect a *single* collection. MongoDB allows applications to determine the acceptable level of acknowledgement required for bulk write operations.

New `Bulk` methods provide the ability to perform bulk insert, update, and remove operations. MongoDB also supports bulk insert through passing an array of documents to the `db.collection.insert()` method.

Changed in version 2.6: Previous versions of MongoDB provided the ability for bulk inserts only. With previous versions, clients could perform bulk inserts by passing an array of documents to the db.collection.insert()[7] method. To see the documentation for earlier versions, see Bulk Inserts[8].

Ordered vs Unordered Operations

Bulk write operations can be either *ordered* or *unordered*. With an ordered list of operations, MongoDB executes the operations serially. If an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list.

With an unordered list of operations, MongoDB can execute the operations in parallel. If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

Executing an ordered list of operations on a sharded collection will generally be slower than executing an unordered list since with an ordered list, each operation must wait for the previous operation to finish.

`Bulk` Methods

To use the `Bulk()` methods:

[7] http://docs.mongodb.org/v2.4/core/bulk-inserts
[8] http://docs.mongodb.org/v2.4/core/bulk-inserts

1. Initialize a list of operations using either `db.collection.initializeUnorderedBulkOp()` or `db.collection.initializeOrderedBulkOp()`.
2. Add write operations to the list using the following methods:
   - `Bulk.insert()`
   - `Bulk.find()`
   - `Bulk.find.upsert()`
   - `Bulk.find.update()`
   - `Bulk.find.updateOne()`
   - `Bulk.find.replaceOne()`
   - `Bulk.find.remove()`
   - `Bulk.find.removeOne()`
3. To execute the list of operations, use the `Bulk.execute()` method. You can specify the write concern for the list in the `Bulk.execute()` method.

   Once executed, you cannot re-execute the list without reinitializing.

For example,

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { _id: 1, item: "abc123", status: "A", soldQty: 5000 } );
bulk.insert( { _id: 2, item: "abc456", status: "A", soldQty: 150 } );
bulk.insert( { _id: 3, item: "abc789", status: "P", soldQty: 0 } );
bulk.execute( { w: "majority", wtimeout: 5000 } );
```

For more examples, refer to the reference page for each `http://docs.mongodb.org/manual/reference/method/js-bul` method. For information and examples on performing bulk insert using the `db.collection.insert()`, see `db.collection.insert()`.

See also:

rel-notes-write-operations

`Bulk` Execution Mechanics

When executing an `ordered` list of operations, MongoDB groups adjacent operations by the `operation type`. When executing an `unordered` list of operations, MongoDB groups and may also reorder the operations to increase performance. As such, when performing *unordered* bulk operations, applications should not depend on the ordering.

Each group of operations can have at most `1000 operations`. If a group exceeds this `limit`, MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call `Bulk.getOperations()` *after* the execution.

For more information, see `Bulk.execute()`.

Strategies for Bulk Inserts to a Sharded Collection

Large bulk insert operations, including initial data inserts or routine data import, can affect *sharded cluster* performance. For bulk inserts, consider the following strategies:

Pre-Split the Collection If the sharded collection is empty, then the collection has only one initial *chunk*, which resides on a single shard. MongoDB must then take time to receive data, create splits, and distribute the split chunks to the available shards. To avoid this performance cost, you can pre-split the collection, as described in `http://docs.mongodb.org/manual/tutorial/split-chunks-in-sharded-cluster`.

Insert to Multiple `mongos` To parallelize import processes, send bulk insert or insert operations to more than one `mongos` instance. For *empty* collections, first pre-split the collection as described in `http://docs.mongodb.org/manual/tutorial/split-chunks-in-sharded-cluster`.

Avoid Monotonic Throttling If your shard key increases monotonically during an insert, then all inserted data goes to the last chunk in the collection, which will always end up on a single shard. Therefore, the insert capacity of the cluster will never exceed the insert capacity of that single shard.

If your insert volume is larger than what a single shard can process, and if you cannot avoid a monotonically increasing shard key, then consider the following modifications to your application:

- Reverse the binary bits of the shard key. This preserves the information and avoids correlating insertion order with increasing sequence of values.
- Swap the first and last 16-bit words to "shuffle" the inserts.

Example

The following example, in C++, swaps the leading and trailing 16-bit word of *BSON ObjectIds* generated so they are no longer monotonically increasing.

```
using namespace mongo;
OID make_an_id() {
  OID x = OID::gen();
  const unsigned char *p = x.getData();
  swap( (unsigned short&) p[0], (unsigned short&) p[10] );
  return x;
}

void foo() {
  // create an object
  BSONObj o = BSON( "_id" << make_an_id() << "x" << 3 << "name" << "jane" );
  // now we may insert o into a sharded collection
}
```

See also:

sharding-shard-key for information on choosing a sharded key. Also see *Shard Key Internals* (in particular, *sharding-internals-operations-and-reliability*).

Storage

New in version 3.0: MongoDB adds support for additional storage engines. MongoDB's original storage engine, known as `mmapv1` remains the default in 3.0, but the new `wiredTiger` engine is available and can offer additional flexibility and improved throughput for many workloads.

Data Model

MongoDB stores data in the form of *BSON* documents, which are rich mappings of keys, or field names, to values. BSON supports a rich collection of types, and fields in BSON documents may hold arrays of values or embedded documents. All documents in MongoDB must be less than 16MB, which is the `BSON document size`.

All documents are part of a *collection*, which are a logical groupings of documents in a MongoDB database. The documents in a collection share a set of indexes, and typically these documents share common fields and structure.

In MongoDB the *database* construct is a group of related collections. Each database has a distinct set of data files and can contain a large number of collections. A single MongoDB deployment may have many databases.

WiredTiger Storage Engine

New in version 3.0.

WiredTiger is a storage engine that is optionally available in the 64-bit build of MongoDB 3.0. It excels at read and insert workloads as well as more complex update workloads.

Document Level Locking With WiredTiger, all write operations happen within the context of a *document* level lock. As a result, multiple clients can modify more than one document in a single collection at the same time. With this very granular concurrency control, MongoDB can more effectively support workloads with read, write and updates as well as high-throughput concurrent workloads.

Journal WiredTiger uses a write-ahead transaction log in combination with checkpoints to ensure data persistence. With WiredTiger, by default MongoDB will commit a checkpoint to disk every 60 seconds, or when there are 2 gigabytes of data to write. The checkpoint thresholds are configurable. Between and during checkpoints the data files are *always* valid.

The WiredTiger journal persists all data modifications between checkpoints. If MongoDB exits between checkpoints, it uses the journal to replay all data modified since the last checkpoint. By default the WiredTiger journal is compressed using the *snappy* algorithm.

You can disable journaling by setting `storage.journal.enabled` to `false`, which can reduce the overhead of maintaining the journal. For *standalone* instances, not using the journal means that you will lose some data modifications when MongoDB exits unexpectedly between checkpoints. For members of *replica sets*, the replication process may provide sufficient durability guarantees.

Compression MongoDB supports compression for all collections and indexes using both block and *prefix compression*. Compression minimizes storage use at the expense of additional CPU.

By default, all indexes with the WiredTiger engine use *prefix compression*. Also, by default all collections with WiredTiger use block compression with the *snappy* algorithm. Compression with *zlib* is also available.

You can modify the default compression settings for all collections and indexes. Compression is also configurable on a per-collection and per-index basis during collection and index creation.

For most workloads, the default compression settings balance storage efficiency and processing requirements.

MMAPv1 Storage Engine

MMAPv1 is MongoDB's original storage engine based on memory mapped files. It excels at workloads with high volume inserts, reads, and in-place updates. MMAPv1 is the default storage engine in MongoDB 3.0 and all previous versions.

Journal In order to ensure that all modifications to a MongoDB data set are durably written to disk, MongoDB records all modifications to a journal that it writes to disk more frequently than it writes the data files. The journal allows MongoDB to successfully recover data from data files after a `mongod` instance exits without flushing all changes.

See `http://docs.mongodb.org/manual/core/journaling` for more information about the journal in MongoDB.

Record Storage Characteristics All records are contiguously located on disk, and when a document becomes larger than the allocated record, MongoDB must allocate a new record. New allocations require MongoDB to move a document and update all indexes that refer to the document, which takes more time than in-place updates and leads to storage fragmentation.

Changed in version 3.0.0.

By default, MongoDB uses *Power of 2 Sized Allocations* (page 38) so that every document in MongoDB is stored in a *record* which contains the document itself and extra space, or *padding*. Padding allows the document to grow as the result of updates while minimizing the likelihood of reallocations.

Record Allocation Strategies MongoDB supports multiple record allocation strategies that determine how `mongod` adds padding to a document when creating a record. Because documents in MongoDB may grow after insertion and all records are contiguous on disk, the padding can reduce the need to relocate documents on disk following updates. Relocations are less efficient than in-place updates and can lead to storage fragmentation. As a result, all padding strategies trade additional space for increased efficiency and decreased fragmentation.

Different allocation strategies support different kinds of workloads: the *power of 2 allocations* (page 38) are more efficient for insert/update/delete workloads; while *exact fit allocations* (page 38) is ideal for collections *without* update and delete workloads.

Power of 2 Sized Allocations Changed in version 3.0.0.

MongoDB 3.0 uses the power of 2 sizes allocation as the default record allocation strategy for MMAPv1. With the power of 2 sizes allocation strategy, each record has a size in bytes that is a power of 2 (e.g. 32, 64, 128, 256, 512 ... 16777216). The smallest allocation for a document is 32 bytes.

The power of 2 sizes allocation strategy has the following key properties:

- Can efficiently reuse freed records to reduce fragmentation. The limited number of record allocation sizes help ensure that MongoDB can efficiently reuse free space created by document deletion or relocation.
- Can minimize the occurrence of re-allocation. In many cases, the record allocations are significantly larger than the documents they hold. This significantly larger size minimizes the chance that the `mongod` will need to allocate a new record if the document grows. Although the power of 2 sizes strategy can minimize the occurrence of reallocation, it does not eliminate document re-allocation.

Exact Fit Allocation with No Padding Changed in version 3.0.0.

For collections whose workloads do not change the document sizes, such as workloads that consist of insert-only operations, insert-and-remove operations on documents of uniform size, or update operations that do not increase document size, you can disable the *power of 2 allocation* (page 38) using the `collMod` command with the `noPadding` flag or the `db.createCollection()` method with the `noPadding` option.

Prior to version 3.0.0, MongoDB used an exact fit allocation strategy that included a dynamically calculated `padding` as a factor of the document size.

Capped Collections

Capped collections are fixed-size collections that support high-throughput operations that store records in insertion order. Capped collections work like circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

See `http://docs.mongodb.org/manual/core/capped-collections` for more information.

3 MongoDB CRUD Tutorials

The following tutorials provide instructions for querying and modifying data. For a higher-level overview of these operations, see *MongoDB CRUD Operations* (page 3).

Insert Documents **(page 39)** Insert new documents into a collection.

Query Documents **(page 43)** Find documents in a collection using search criteria.

Modify Documents **(page 49)** Modify documents in a collection

Remove Documents **(page 53)** Remove documents from a collection.

Limit Fields to Return from a Query **(page 54)** Limit which fields are returned by a query.

Limit Number of Elements in an Array after an Update **(page 55)** Use `$push` with modifiers to sort and maintain an array of fixed size.

Iterate a Cursor in the mongo Shell **(page 56)** Access documents returned by a `find` query by iterating the cursor, either manually or using the iterator index.

Analyze Query Performance **(page 57)** Use query introspection (i.e. `explain`) to analyze the efficiency of queries and determine how a query uses available indexes.

Perform Two Phase Commits **(page 62)** Use two-phase commits when writing data to multiple documents.

Update Document if Current **(page 68)** Update a document only if it has not changed since it was last read.

Create Tailable Cursor **(page 69)** Create tailable cursors for use in capped collections with high numbers of write operations for which an index would be too expensive.

Create an Auto-Incrementing Sequence Field **(page 72)** Describes how to create an incrementing sequence number for the `_id` field using a Counters Collection or an Optimistic Loop.

3.1 Insert Documents

In MongoDB, the `db.collection.insert()` method adds new documents into a collection.

Insert a Document

Step 1: Insert a document into a collection.

Insert a document into a collection named `inventory`. The operation will create the collection if the collection does not currently exist.

```
db.inventory.insert(
   {
     item: "ABC1",
     details: {
```

```
        model: "14Q3",
        manufacturer: "XYZ Company"
     },
     stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
     category: "clothing"
   }
)
```

The operation returns a `WriteResult` object with the status of the operation. A successful insert of the document returns the following object:

```
WriteResult({ "nInserted" : 1 })
```

The `nInserted` field specifies the number of documents inserted. If the operation encounters an error, the `WriteResult` object will contain the error information.

Step 2: Review the inserted document.

If the insert operation is successful, verify the insertion by querying the collection.

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

The document you inserted should return.

```
{ "_id" : ObjectId("53d98f133bb604791249ca99"), "item" : "ABC1", "details" : { "model" : "14Q3", "mar
```

The returned document shows that MongoDB added an `_id` field to the document. If a client inserts a document that does not contain the `_id` field, MongoDB adds the field with the value set to a generated ObjectId[9]. The ObjectId[10] values in your documents will differ from the ones shown.

Insert an Array of Documents

You can pass an array of documents to the `db.collection.insert()` method to insert multiple documents.

Step 1: Create an array of documents.

Define a variable `mydocuments` that holds an array of documents to insert.

```
var mydocuments =
    [
      {
        item: "ABC2",
        details: { model: "14Q3", manufacturer: "M1 Corporation" },
        stock: [ { size: "M", qty: 50 } ],
        category: "clothing"
      },
      {
        item: "MNO2",
        details: { model: "14Q3", manufacturer: "ABC Company" },
        stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
        category: "clothing"
      },
      {
```

[9] http://docs.mongodb.org/manual/reference/object-id
[10] http://docs.mongodb.org/manual/reference/object-id

```
        item: "IJK2",
        details: { model: "14Q2", manufacturer: "M5 Corporation" },
        stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],
        category: "houseware"
      }
    ];
```

Step 2: Insert the documents.

Pass the `mydocuments` array to the `db.collection.insert()` to perform a bulk insert.

```
db.inventory.insert( mydocuments );
```

The method returns a `BulkWriteResult` object with the status of the operation. A successful insert of the documents returns the following object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 3,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

The `nInserted` field specifies the number of documents inserted. If the operation encounters an error, the `BulkWriteResult` object will contain information regarding the error.

The inserted documents will each have an `_id` field added by MongoDB.

Insert Multiple Documents with `Bulk`

New in version 2.6.

MongoDB provides a `Bulk()` API that you can use to perform multiple write operations in bulk. The following sequence of operations describes how you would use the `Bulk()` API to insert a group of documents into a MongoDB collection.

Step 1: Initialize a `Bulk` operations builder.

Initialize a `Bulk` operations builder for the collection `inventory`.

```
var bulk = db.inventory.initializeUnorderedBulkOp();
```

The operation returns an unordered operations builder which maintains a list of operations to perform. Unordered operations means that MongoDB can execute in parallel as well as in nondeterministic order. If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

You can also initialize an ordered operations builder; see `db.collection.initializeOrderedBulkOp()` for details.

Step 2: Add insert operations to the `bulk` object.

Add two insert operations to the `bulk` object using the `Bulk.insert()` method.

```
bulk.insert(
   {
     item: "BE10",
     details: { model: "14Q2", manufacturer: "XYZ Company" },
     stock: [ { size: "L", qty: 5 } ],
     category: "clothing"
   }
);
bulk.insert(
   {
     item: "ZYT1",
     details: { model: "14Q1", manufacturer: "ABC Company"  },
     stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 } ],
     category: "houseware"
   }
);
```

Step 3: Execute the bulk operation.

Call the `execute()` method on the `bulk` object to execute the operations in its list.

```
bulk.execute();
```

The method returns a `BulkWriteResult` object with the status of the operation. A successful insert of the documents returns the following object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 2,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

The `nInserted` field specifies the number of documents inserted. If the operation encounters an error, the `BulkWriteResult` object will contain information regarding the error.

Additional Examples and Methods

For more examples, see `db.collection.insert()`.

The `db.collection.update()` method, the `db.collection.findAndModify()`, and the `db.collection.save()` method can also add new documents. See the individual reference pages for the methods for more information and examples.

3.2 Query Documents

In MongoDB, the `db.collection.find()` method retrieves documents from a collection. [11] The `db.collection.find()` method returns a *cursor* (page 10) to the retrieved documents.

This tutorial provides examples of read operations using the `db.collection.find()` method in the `mongo` shell. In these examples, the retrieved documents contain all their fields. To restrict the fields to return in the retrieved documents, see *Limit Fields to Return from a Query* (page 54).

Select All Documents in a Collection

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()
```

Specify Equality Condition

To specify equality condition, use the query document `{ <field>: <value> }` to select all documents that contain the `<field>` with the specified `<value>`.

The following example retrieves from the `inventory` collection all documents where the `type` field has the value `snacks`:

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

Specify Conditions Using Query Operators

A query document can use the *query operators* to specify conditions in a MongoDB query.

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

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

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

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

Specify AND Conditions

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 the field `type` **and** a less than (`$lt`) comparison match on the field `price`:

[11] The `db.collection.findOne()` method also performs a read operation to return a single document. Internally, the `db.collection.findOne()` method is the `db.collection.find()` method with a limit of 1.

```
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 `9.95`. See *comparison operators* for other comparison operators.

Specify OR Conditions

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 } } ]
   }
)
```

Specify AND as well as OR Conditions

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 } } ]
   }
)
```

Embedded Documents

When the field holds an embedded document, a query can either specify an exact match on the embedded document or specify a match by individual fields in the embedded document using the *dot notation*.

Exact Match on the Embedded Document

To specify an equality match on the whole embedded document, use the query document `{ <field>: <value> }` where `<value>` is the document to match. Equality matches on an embedded document require an *exact* match of the specified `<value>`, including the field order.

In the following example, the query matches all documents where the value of the field `producer` is an embedded document 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 Match on Fields within an Embedded Document

Use the *dot notation* to match by specific fields in an embedded document. Equality matches for specific fields in an embedded document will select documents in the collection where the embedded document contains the specified fields with the specified values. The embedded document can contain additional fields.

In the following example, the query uses the *dot notation* to match all documents where the value of the field `producer` is an embedded document 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 an exact array match or for specific values in the array. If the array holds embedded documents, you can query for specific fields in the embedded documents using *dot notation*.

If you specify multiple conditions using the `$elemMatch` operator, the array must contain at least one element that satisfies all the conditions. See *Single Element Satisfies the Criteria* (page 46).

If you specify multiple conditions without using the `$elemMatch` operator, then some combination of the array elements, not necessarily a single element, must satisfy all the conditions; i.e. different elements in the array can satisfy different parts of the conditions. See *Combination of Elements Satisfies the Criteria* (page 46).

Consider an `inventory` collection that contains the following documents:

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

Exact Match on an Array

To specify equality match on an array, use the query document `{ <field>: <value> }` where `<value>` is the array to match. Equality matches on the array require that the array field match *exactly* the specified `<value>`, including the element order.

The following example queries for all documents where the field `ratings` is an array that holds exactly three elements, `5`, `8`, and `9`, in this order:

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

The operation returns the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Match an Array Element

Equality matches can specify a single element in the array to match. These specifications match if the array contains at least *one* element with the specified value.

The following example queries for all documents where `ratings` is an array that contains `5` as one of its elements:

```
db.inventory.find( { ratings: 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Match a Specific Element of an Array

Equality matches can specify equality matches for an element at a particular index or position of the array using the *dot notation*.

In the following example, the query uses the *dot notation* to match all documents where the `ratings` array contains `5` as the first element:

```
db.inventory.find( { 'ratings.0': 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
```

Specify Multiple Criteria for Array Elements

Single Element Satisfies the Criteria Use `$elemMatch` operator to specify multiple criteria on the elements of an array such that at least one array element satisfies all the specified criteria.

The following example queries for documents where the `ratings` array contains at least one element that is greater than (`$gt`) `5` and less than (`$lt`) `9`:

```
db.inventory.find( { ratings: { $elemMatch: { $gt: 5, $lt: 9 } } } )
```

The operation returns the following documents, whose `ratings` array contains the element `8` which meets the criteria:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Combination of Elements Satisfies the Criteria The following example queries for documents where the `ratings` array contains elements that in some combination satisfy the query conditions; e.g., one element can satisfy the greater than `5` condition and another element can satisfy the less than `9` condition, or a single element can satisfy both:

```
db.inventory.find( { ratings: { $gt: 5, $lt: 9 } } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

The document with the `"ratings" : [ 5, 9 ]` matches the query since the element `9` is greater than `5` (the first condition) and the element `5` is less than `9` (the second condition).

Array of Embedded Documents

Consider that the `inventory` collection includes the following documents:

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Match a Field in the Embedded Document Using the Array Index If you know the array index of the embedded document, you can specify the document using the embedded document's position using the *dot notation*.

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

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

The operation returns the following document:

```
{
   _id: 100,
   type: "food",
   item: "xyz",
   qty: 25,
   price: 2.5,
   ratings: [ 5, 8, 9 ],
   memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
```

Match a Field Without Specifying Array Index If you do not know the index position of the document in the array, concatenate the name of the field that contains the array, with a dot (`.`) and the name of the field in the embedded document.

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

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

The operation returns the following documents:

```
{
  _id: 100,
```

```
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Specify Multiple Criteria for Array of Documents

Single Element Satisfies the Criteria Use `$elemMatch` operator to specify multiple criteria on an array of embedded documents such that at least one embedded document satisfies all the specified criteria.

The following example queries for documents where the `memos` array has at least one embedded document that contains both the field `memo` equal to `'on time'` and the field `by` equal to `'shipping'`:

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

The operation returns the following document:

```
{
   _id: 100,
   type: "food",
   item: "xyz",
   qty: 25,
   price: 2.5,
   ratings: [ 5, 8, 9 ],
   memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
```

Combination of Elements Satisfies the Criteria The following example queries for documents where the `memos` array contains elements that in some combination satisfy the query conditions; e.g. one element satisfies the field `memo` equal to `'on time'` condition and another element satisfies the field `by` equal to `'shipping'` condition, or a single element can satisfy both criteria:

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

The query returns the following documents:

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]

}
```

3.3 Modify Documents

MongoDB provides the `update()` method to update the documents of a collection. The method accepts as its parameters:

- an update conditions document to match the documents to update,
- an update operations document to specify the modification to perform, and
- an options document.

To specify the update condition, use the same structure and syntax as the query conditions.

By default, `update()` updates a single document. To update multiple documents, use the *multi* option.

Update Specific Fields in a Document

To change a field value, MongoDB provides update operators[12], such as `$set` to modify values.

Some update operators, such as `$set`, will create the field if the field does not exist. See the individual update operator[13] reference.

[12] http://docs.mongodb.org/manual/reference/operator/update
[13] http://docs.mongodb.org/manual/reference/operator/update

Step 1: Use update operators to change field values.

For the document with `item` equal to `"MNO2"`, use the `$set` operator to update the `category` field and the `details` field to the specified values and the `$currentDate` operator to update the field `lastModified` with the current date.

```
db.inventory.update(
    { item: "MNO2" },
    {
      $set: {
        category: "apparel",
        details: { model: "14Q3", manufacturer: "XYZ Company" }
      },
      $currentDate: { lastModified: true }
    }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The `nMatched` field specifies the number of existing documents matched for the update, and `nModified` specifies the number of existing documents modified.

Step 2: Update an embedded field.

To update a field within an embedded document, use the *dot notation*. When using the dot notation, enclose the whole dotted field name in quotes.

The following updates the `model` field within the embedded `details` document.

```
db.inventory.update(
  { item: "ABC1" },
  { $set: { "details.model": "14Q2" } }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Step 3: Update multiple documents.

By default, the `update()` method updates a single document. To update multiple documents, use the `multi` option in the `update()` method.

Update the `category` field to `"apparel"` and update the `lastModified` field to the current date for *all* documents that have `category` field equal to `"clothing"`.

```
db.inventory.update(
   { category: "clothing" },
   {
     $set: { category: "apparel" },
     $currentDate: { lastModified: true }
   },
```

```
   { multi: true }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 3 })
```

Replace the Document

To replace the entire content of a document except for the `_id` field, pass an entirely new document as the second argument to `update()`.

The replacement document can have different fields from the original document. In the replacement document, you can omit the `_id` field since the `_id` field is immutable. If you do include the `_id` field, it must be the same value as the existing value.

Step 1: Replace a document.

The following operation replaces the document with `item` equal to `"BE10"`. The newly replaced document will only contain the the `_id` field and the fields in the replacement document.

```
db.inventory.update(
   { item: "BE10" },
   {
     item: "BE05",
     stock: [ { size: "S", qty: 20 }, { size: "M", qty: 5 } ],
     category: "apparel"
   }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

`upsert` Option

By default, if no document matches the update query, the `update()` method does nothing.

However, by specifying *upsert: true*, the `update()` method either updates matching document or documents, or inserts a new document using the update specification if no matching document exists.

Step 1: Specify `upsert: true` for the update replacement operation.

When you specify `upsert: true` for an update operation to replace a document and no matching documents are found, MongoDB creates a new document using the equality conditions in the update conditions document, and replaces this document, except for the `_id` field if specified, with the update document.

The following operation either updates a matching document by replacing it with a new document or adds a new document if no matching document exists.

```
db.inventory.update(
   { item: "TBD1" },
   {
     item: "TBD1",
     details: { "model" : "14Q4", "manufacturer" : "ABC Company" },
     stock: [ { "size" : "S", "qty" : 25 } ],
     category: "houseware"
   },
   { upsert: true }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation, including whether the `db.collection.update()` method modified an existing document or added a new document.

```
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 1,
    "nModified" : 0,
    "_id" : ObjectId("53dbd684babeaec6342ed6c7")
})
```

The `nMatched` field shows that the operation matched `0` documents.

The `nUpserted` of `1` shows that the update added a document.

The `nModified` of `0` specifies that no existing documents were updated.

The `_id` field shows the generated `_id` field for the added document.

Step 2: Specify an `upsert: true` for the update specific fields operation.

When you specify an `upsert: true` for an update operation that modifies specific fields and no matching documents are found, MongoDB creates a new document using the equality conditions in the update conditions document, and applies the modification as specified in the update document.

The following update operation either updates specific fields of a matching document or adds a new document if no matching document exists.

```
db.inventory.update(
   { item: "TBD2" },
   {
     $set: {
        details: { "model" : "14Q3", "manufacturer" : "IJK Co." },
        category: "houseware"
     }
   },
   { upsert: true }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation, including whether the `db.collection.update()` method modified an existing document or added a new document.

```
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 1,
    "nModified" : 0,
    "_id" : ObjectId("53dbd7c8babeaec6342ed6c8")
})
```

The `nMatched` field shows that the operation matched `0` documents.

The `nUpserted` of `1` shows that the update added a document.

The `nModified` of `0` specifies that no existing documents were updated.

The `_id` field shows the generated `_id` field for the added document.

Additional Examples and Methods

For more examples, see *Update examples* in the `db.collection.update()` reference page.

The `db.collection.findAndModify()` and the `db.collection.save()` method can also modify existing documents or insert a new one. See the individual reference pages for the methods for more information and examples.

3.4 Remove Documents

In MongoDB, the `db.collection.remove()` method removes documents from a collection. You can remove all documents from a collection, remove all documents that match a condition, or limit the operation to remove just a single document.

This tutorial provides examples of remove operations using the `db.collection.remove()` method in the `mongo` shell.

Remove All Documents

To remove all documents from a collection, pass an empty query document `{}` to the `remove()` method. The `remove()` method does not remove the indexes.

The following example removes all documents from the `inventory` collection:

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

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.

Remove Documents that Match a Condition

To remove the documents that match a deletion criteria, call the `remove()` method with the `<query>` parameter.

The following example removes all documents from the `inventory` collection where the `type` field equals `food`:

```
db.inventory.remove( { type : "food" } )
```

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.

Remove a Single Document that Matches a Condition

To remove a single document, call the `remove()` method with the `justOne` parameter set to `true` or `1`.

The following example removes one document from the `inventory` collection where the `type` field equals `food`:

```
db.inventory.remove( { type : "food" }, 1 )
```

To delete a single document sorted by some specified order, use the *findAndModify()* method.

3.5 Limit Fields to Return from a Query

The *projection* document limits the fields to return for all matching documents. The projection document can specify the inclusion of fields or the exclusion of fields.

The specifications have the following forms:

Syntax	Description
`<field>: <1 or true>`	Specify the inclusion of a field.
`<field>: <0 or false>`	Specify the suppression of the field.

Important: The `_id` field is, by default, included in the result set. To suppress the `_id` field from the result set, specify `_id: 0` in the projection document.

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

This tutorial offers various query examples that limit the fields to return for all matching documents. The examples in this tutorial use a collection `inventory` and use the `db.collection.find()` method in the `mongo` shell. The `db.collection.find()` method returns a *cursor* (page 10) to the retrieved documents. For examples on query selection criteria, see *Query Documents* (page 43).

Return All Fields in Matching Documents

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'`. The returned documents contain all its fields.

Return the Specified Fields and the `_id` Field Only

A projection can explicitly include several fields. In the following operation, `find()` method returns all documents that match the query. In the result set, only the `item` and `qty` fields and, by default, the `_id` field return in the matching documents.

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

Return Specified Fields Only

You can remove the `_id` field from the results by specifying its exclusion in 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. In the result set, *only* the `item` and `qty` fields return in the matching documents.

Return All But the Excluded Field

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`. In the result set, the `type` field does not return in the matching documents.

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

Projection for Array Fields

For fields that contain arrays, MongoDB provides the following projection operators: `$elemMatch`, `$slice`, and `$`.

For example, the `inventory` collection contains the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Then the following operation uses the `$slice` projection operator to return just the first two elements in the `ratings` array.

```
db.inventory.find( { _id: 5 }, { ratings: { $slice: 2 } } )
```

`$elemMatch`, `$slice`, and `$` are the *only* way to project *portions* of an array. For instance, you *cannot* project a portion of an array using the array index; e.g. `{ "ratings.0": 1 }` projection will *not* project the array with the first element.

3.6 Limit Number of Elements in an Array after an Update

New in version 2.4.

Synopsis

Consider an application where users may submit many scores (e.g. for a test), but the application only needs to track the top three test scores.

This pattern uses the `$push` operator with the `$each`, `$sort`, and `$slice` modifiers to sort and maintain an array of fixed size.

Pattern

Consider the following document in the collection `students`:

```
{
  _id: 1,
  scores: [
    { attempt: 1, score: 10 },
    { attempt: 2 , score:8 }
  ]
}
```

The following update uses the `$push` operator with:

- the `$each` modifier to append to the array 2 new elements,
- the `$sort` modifier to order the elements by ascending (`1`) score, and
- the `$slice` modifier to keep the last `3` elements of the ordered array.

```
db.students.update(
   { _id: 1 },
   {
     $push: {
        scores: {
           $each: [ { attempt: 3, score: 7 }, { attempt: 4, score: 4 } ],
           $sort: { score: 1 },
           $slice: -3
        }
     }
   }
)
```

Note: When using the `$sort` modifier on the array element, access the field in the embedded document element directly instead of using the *dot notation* on the array field.

After the operation, the document contains only the top 3 scores in the `scores` array:

```
{
   "_id" : 1,
   "scores" : [
     { "attempt" : 3, "score" : 7 },
     { "attempt" : 2, "score" : 8 },
     { "attempt" : 1, "score" : 10 }
   ]
}
```

See also:

- `$push` operator,
- `$each` modifier,
- `$sort` modifier, and
- `$slice` modifier.

3.7 Iterate a Cursor in the `mongo` Shell

The `db.collection.find()` method returns a cursor. To access the documents, you need to iterate the cursor. However, in the `mongo` shell, if the returned cursor is not assigned to a variable using the `var` keyword, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents in the results. The following describes ways to manually iterate the cursor to access the documents or to use the iterator index.

Manually Iterate the Cursor

In the `mongo` shell, when you assign the cursor returned from the `find()` method to a variable using the `var` keyword, the cursor does not automatically iterate.

You can call the cursor variable in the shell to iterate up to 20 times [14] and print the matching documents, as in the following example:

[14] 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.

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

myCursor
```

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

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

while (myCursor.hasNext()) {
   print(tojson(myCursor.next()));
}
```

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

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

while (myCursor.hasNext()) {
   printjson(myCursor.next());
}
```

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];
```

3.8 Analyze Query Performance

The `cursor.explain("executionStats")` and the `db.collection.explain("executionStats")` methods provide statistics about the performance of a query. This data output can be useful in measuring if and how a

query uses an index.

`db.collection.explain()` provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

Evaluate the Performance of a Query

Consider a collection `inventory` with the following documents:

```
{ "_id" : 1, "item" : "f1", type: "food", quantity: 500 }
{ "_id" : 2, "item" : "f2", type: "food", quantity: 100 }
{ "_id" : 3, "item" : "p1", type: "paper", quantity: 200 }
{ "_id" : 4, "item" : "p2", type: "paper", quantity: 150 }
{ "_id" : 5, "item" : "f3", type: "food", quantity: 300 }
{ "_id" : 6, "item" : "t1", type: "toys", quantity: 500 }
{ "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 }
{ "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 }
{ "_id" : 9, "item" : "t2", type: "toys", quantity: 50 }
{ "_id" : 10, "item" : "f4", type: "food", quantity: 75 }
```

Query with No Index

The following query retrieves documents where the `quantity` field has a value between `100` and `200`, inclusive:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } )
```

The query returns the following documents:

```
{ "_id" : 2, "item" : "f2", "type" : "food", "quantity" : 100 }
{ "_id" : 3, "item" : "p1", "type" : "paper", "quantity" : 200 }
{ "_id" : 4, "item" : "p2", "type" : "paper", "quantity" : 150 }
```

To view the query plan selected, use the `explain("executionStats")` method:

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 200 } }
).explain("executionStats")
```

`explain()` returns the following results:

```
{
   "queryPlanner" : {
         "plannerVersion" : 1,
         ...
         "winningPlan" : {
            "stage" : "COLLSCAN",
            ...
         }
   },
   "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 3,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 0,
      "totalDocsExamined" : 10,
      "executionStages" : {
         "stage" : "COLLSCAN",
```

```
            ...
         },
         ...
      },
      ...
}
```

- `winningPlan.stage` displays `COLLSCAN` to indicate a collection scan.
- `executionStats.nReturned` displays `3` to indicate that the query matches and returns three documents.
- `executionStats.totalDocsExamined` display `10` to indicate that MongoDB had to scan ten documents (i.e. all documents in the collection) to find the three matching documents.

The difference between the number of matching documents and the number of examined documents may suggest that, to improve efficiency, the query might benefit from the use of an index.

Query with Index

To support the query on the `quantity` field, add an index on the `quantity` field:

```
db.inventory.createIndex( { quantity: 1 } )
```

To view the query plan statistics, use the `explain("executionStats")` method:

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 200 } }
).explain("executionStats")
```

The `explain()` method returns the following results:

```
{
   "queryPlanner" : {
      "plannerVersion" : 1,
      ...
      "winningPlan" : {
         "stage" : "FETCH",
         "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
               "quantity" : 1
            },
            ...
            }
         }
      },
      "rejectedPlans" : [ ]
   },
   "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 3,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 3,
      "totalDocsExamined" : 3,
      "executionStages" : {
         ...
      },
      ...
   },
```

```
   ...
}
```

- `winningPlan.stage.inputStage.stage` displays `IXSCAN` to indicate index use.
- `executionStats.nReturned` displays `3` to indicate that the query matches and returns three documents.
- `executionStats.totalKeysExamined` display `3` to indicate that MongoDB scanned three index entries.
- `executionStats.totalDocsExamined` display `3` to indicate that MongoDB scanned three documents.

When run with an index, the query scanned `3` index entries and `3` documents to return `3` matching documents. Without the index, to return the `3` matching documents, the query had to scan the whole collection, scanning `10` documents.

Compare Performance of Indexes

To manually compare the performance of a query using more than one index, you can use the `hint()` method in conjunction with the `explain()` method.

Consider the following query:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 300 }, type: "food" } )
```

The query returns the following documents:

```
{ "_id" : 2, "item" : "f2", "type" : "food", "quantity" : 100 }
{ "_id" : 5, "item" : "f3", "type" : "food", "quantity" : 300 }
```

To support the query, add a `compound index`. With `compound indexes`, the order of the fields matter.

For example, add the following two compound indexes. The first index orders by `quantity` field first, and then the `type` field. The second index orders by `type` first, and then the `quantity` field.

```
db.inventory.createIndex( { quantity: 1, type: 1 } )
db.inventory.createIndex( { type: 1, quantity: 1 } )
```

Evaluate the effect of the first index on the query:

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 300 }, type: "food" }
).hint({ quantity: 1, type: 1 }).explain("executionStats")
```

The `explain()` method returns the following output:

```
{
   "queryPlanner" : {
      ...
      "winningPlan" : {
         "stage" : "FETCH",
         "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
               "quantity" : 1,
               "type" : 1
            },
            ...
            }
         }
      },
      "rejectedPlans" : [ ]
```

```
   },
   "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 2,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 5,
      "totalDocsExamined" : 2,
      "executionStages" : {
      ...
      }
   },
   ...
}
```

MongoDB scanned `5` index keys (`executionStats.totalKeysExamined`) to return `2` matching documents (`executionStats.nReturned`).

Evaluate the effect of the second index on the query:

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 300 }, type: "food" }
).hint({ type: 1, quantity: 1 }).explain("executionStats")
```

The `explain()` method returns the following output:

```
{
   "queryPlanner" : {
      ...
      "winningPlan" : {
         "stage" : "FETCH",
         "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
               "type" : 1,
               "quantity" : 1
            },
            ...
         }
      },
      "rejectedPlans" : [ ]
   },
   "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 2,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 2,
      "totalDocsExamined" : 2,
      "executionStages" : {
         ...
      }
   },
   ...
}
```

MongoDB scanned `2` index keys (`executionStats.totalKeysExamined`) to return `2` matching documents (`executionStats.nReturned`).

For this example query, the compound index `{ type: 1, quantity: 1 }` is more efficient than the compound index `{ quantity: 1, type: 1 }`.

See also:

Query Optimization (page 11), *Query Plans* (page 14), `http://docs.mongodb.org/manual/tutorial/optimize-query-`
`http://docs.mongodb.org/manual/applications/indexes`

3.9 Perform Two Phase Commits

Synopsis

This document provides a pattern for doing multi-document updates or "multi-document transactions" using a two-phase commit approach for writing data to multiple documents. Additionally, you can extend this process to provide a *rollback-like* (page 66) functionality.

Background

Operations on a single *document* are always atomic with MongoDB databases; however, operations that involve multiple documents, which are often referred to as "multi-document transactions", are not atomic. Since documents can be fairly complex and contain multiple "nested" documents, single-document atomicity provides the necessary support for many practical use cases.

Despite the power of single-document atomic operations, there are cases that require multi-document transactions. When executing a transaction composed of sequential operations, certain issues arise, such as:

- Atomicity: if one operation fails, the previous operation within the transaction must "rollback" to the previous state (i.e. the "nothing," in "all or nothing").
- Consistency: if a major failure (i.e. network, hardware) interrupts the transaction, the database must be able to recover a consistent state.

For situations that require multi-document transactions, you can implement two-phase commit in your application to provide support for these kinds of multi-document updates. Using two-phase commit ensures that data is consistent and, in case of an error, the state that preceded the transaction is *recoverable* (page 66). During the procedure, however, documents can represent pending data and states.

Note: Because only single-document operations are atomic with MongoDB, two-phase commits can only offer transaction-*like* semantics. It is possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback.

Pattern

Overview

Consider a scenario where you want to transfer funds from account `A` to account `B`. In a relational database system, you can subtract the funds from `A` and add the funds to `B` in a single multi-statement transaction. In MongoDB, you can emulate a two-phase commit to achieve a comparable result.

The examples in this tutorial use the following two collections:

1. A collection named `accounts` to store account information.
2. A collection named `transactions` to store information on the fund transfer transactions.

Initialize Source and Destination Accounts

Insert into the `accounts` collection a document for account `A` and a document for account `B`.

```
db.accounts.insert(
   [
     { _id: "A", balance: 1000, pendingTransactions: [] },
     { _id: "B", balance: 1000, pendingTransactions: [] }
   ]
)
```

The operation returns a `BulkWriteResult()` object with the status of the operation. Upon successful insert, the `BulkWriteResult()` has `nInserted` set to `2` .

Initialize Transfer Record

For each fund transfer to perform, insert into the `transactions` collection a document with the transfer information. The document contains the following fields:

- `source` and `destination` fields, which refer to the `_id` fields from the `accounts` collection,
- `value` field, which specifies the amount of transfer affecting the `balance` of the `source` and `destination` accounts,
- `state` field, which reflects the current state of the transfer. The `state` field can have the value of `initial`, `pending`, `applied`, `done`, `canceling`, and `canceled`.
- `lastModified` field, which reflects last modification date.

To initialize the transfer of `100` from account `A` to account `B`, insert into the `transactions` collection a document with the transfer information, the transaction `state` of `"initial"`, and the `lastModified` field set to the current date:

```
db.transactions.insert(
    { _id: 1, source: "A", destination: "B", value: 100, state: "initial", lastModified: new Date() }
)
```

The operation returns a `WriteResult()` object with the status of the operation. Upon successful insert, the `WriteResult()` object has `nInserted` set to `1`.

Transfer Funds Between Accounts Using Two-Phase Commit

Step 1: Retrieve the transaction to start. From the `transactions` collection, find a transaction in the `initial` state. Currently the `transactions` collection has only one document, namely the one added in the *Initialize Transfer Record* (page 63) step. If the collection contains additional documents, the query will return any transaction with an `initial` state unless you specify additional query conditions.

```
var t = db.transactions.findOne( { state: "initial" } )
```

Type the variable `t` in the `mongo` shell to print the contents of the variable. The operation should print a document similar to the following except the `lastModified` field should reflect date of your insert operation:

```
{ "_id" : 1, "source" : "A", "destination" : "B", "value" : 100, "state" : "initial", "lastModified"
```

Step 2: Update transaction state to pending. Set the transaction `state` from `initial` to `pending` and use the `$currentDate` operator to set the `lastModified` field to the current date.

```
db.transactions.update(
    { _id: t._id, state: "initial" },
    {
      $set: { state: "pending" },
      $currentDate: { lastModified: true }
    }
)
```

The operation returns a `WriteResult()` object with the status of the operation. Upon successful update, the `nMatched` and `nModified` displays `1`.

In the update statement, the `state: "initial"` condition ensures that no other process has already updated this record. If `nMatched` and `nModified` is `0`, go back to the first step to get a different transaction and restart the procedure.

Step 3: Apply the transaction to both accounts. Apply the transaction `t` to both accounts using the `update()` method *if* the transaction has not been applied to the accounts. In the update condition, include the condition `pendingTransactions: { $ne: t._id }` in order to avoid re-applying the transaction if the step is run more than once.

To apply the transaction to the account, update both the `balance` field and the `pendingTransactions` field.

Update the source account, subtracting from its `balance` the transaction `value` and adding to its `pendingTransactions` array the transaction `_id`.

```
db.accounts.update(
   { _id: t.source, pendingTransactions: { $ne: t._id } },
   { $inc: { balance: -t.value }, $push: { pendingTransactions: t._id } }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Update the destination account, adding to its `balance` the transaction `value` and adding to its `pendingTransactions` array the transaction `_id`.

```
db.accounts.update(
   { _id: t.destination, pendingTransactions: { $ne: t._id } },
   { $inc: { balance: t.value }, $push: { pendingTransactions: t._id } }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 4: Update transaction state to applied. Use the following `update()` operation to set the transaction's `state` to `applied` and update the `lastModified` field:

```
db.transactions.update(
   { _id: t._id, state: "pending" },
   {
     $set: { state: "applied" },
     $currentDate: { lastModified: true }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 5: Update both accounts' list of pending transactions. Remove the applied transaction `_id` from the `pendingTransactions` array for both accounts.

Update the source account.

```
db.accounts.update(
   { _id: t.source, pendingTransactions: t._id },
   { $pull: { pendingTransactions: t._id } }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Update the destination account.

```
db.accounts.update(
   { _id: t.destination, pendingTransactions: t._id },
   { $pull: { pendingTransactions: t._id } }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 6: Update transaction state to done. Complete the transaction by setting the `state` of the transaction to `done` and updating the `lastModified` field:

```
db.transactions.update(
   { _id: t._id, state: "applied" },
   {
     $set: { state: "done" },
     $currentDate: { lastModified: true }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Recovering from Failure Scenarios

The most important part of the transaction procedure is not the prototypical example above, but rather the possibility for recovering from the various failure scenarios when transactions do not complete successfully. This section presents an overview of possible failures and provides steps to recover from these kinds of events.

Recovery Operations

The two-phase commit pattern allows applications running the sequence to resume the transaction and arrive at a consistent state. Run the recovery operations at application startup, and possibly at regular intervals, to catch any unfinished transactions.

The time required to reach a consistent state depends on how long the application needs to recover each transaction.

The following recovery procedures uses the `lastModified` date as an indicator of whether the pending transaction requires recovery; specifically, if the pending or applied transaction has not been updated in the last 30 minutes, the procedures determine that these transactions require recovery. You can use different conditions to make this determination.

Transactions in Pending State To recover from failures that occur after step "Update transaction state to pending. (page **??**)" but before "Update transaction state to applied. (page **??**)" step, retrieve from the `transactions` collection a pending transaction for recovery:

```
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

var t = db.transactions.findOne( { state: "pending", lastModified: { $lt: dateThreshold } } );
```

And resume from step "Apply the transaction to both accounts. (page **??**)"

Transactions in Applied State To recover from failures that occur after step "Update transaction state to applied. (page **??**)" but before "Update transaction state to done. (page **??**)" step, retrieve from the `transactions` collection an applied transaction for recovery:

```
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

var t = db.transactions.findOne( { state: "applied", lastModified: { $lt: dateThreshold } } );
```

And resume from "Update both accounts' list of pending transactions. (page **??**)"

Rollback Operations

In some cases, you may need to "roll back" or undo a transaction; e.g., if the application needs to "cancel" the transaction or if one of the accounts does not exist or stops existing during the transaction.

Transactions in Applied State After the "Update transaction state to applied. (page **??**)" step, you should **not** roll back the transaction. Instead, complete that transaction and *create a new transaction* (page 63) to reverse the transaction by switching the values in the source and the destination fields.

Transactions in Pending State After the "Update transaction state to pending. (page **??**)" step, but before the "Update transaction state to applied. (page **??**)" step, you can rollback the transaction using the following procedure:

Step 1: Update transaction state to canceling. Update the transaction `state` from `pending` to `canceling`.

```
db.transactions.update(
   { _id: t._id, state: "pending" },
   {
     $set: { state: "canceling" },
     $currentDate: { lastModified: true }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 2: Undo the transaction on both accounts. To undo the transaction on both accounts, reverse the transaction `t` if the transaction has been applied. In the update condition, include the condition `pendingTransactions: t._id` in order to update the account only if the pending transaction has been applied.

Update the destination account, subtracting from its `balance` the transaction `value` and removing the transaction `_id` from the `pendingTransactions` array.

```
db.accounts.update(
   { _id: t.destination, pendingTransactions: t._id },
   {
     $inc: { balance: -t.value },
```

```
      $pull: { pendingTransactions: t._id }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`. If the pending transaction has not been previously applied to this account, no document will match the update condition and `nMatched` and `nModified` will be `0`.

Update the source account, adding to its `balance` the transaction `value` and removing the transaction `_id` from the `pendingTransactions` array.

```
db.accounts.update(
   { _id: t.source, pendingTransactions: t._id },
   {
     $inc: { balance: t.value},
     $pull: { pendingTransactions: t._id }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`. If the pending transaction has not been previously applied to this account, no document will match the update condition and `nMatched` and `nModified` will be `0`.

Step 3: Update transaction state to canceled. To finish the rollback, update the transaction `state` from `canceling` to `cancelled`.

```
db.transactions.update(
   { _id: t._id, state: "canceling" },
   {
     $set: { state: "cancelled" },
     $currentDate: { lastModified: true }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Multiple Applications

Transactions exist, in part, so that multiple applications can create and run operations concurrently without causing data inconsistency or conflicts. In our procedure, to update or retrieve the transaction document, the update conditions include a condition on the `state` field to prevent reapplication of the transaction by multiple applications.

For example, applications `App1` and `App2` both grab the same transaction, which is in the `initial` state. `App1` applies the whole transaction before `App2` starts. When `App2` attempts to perform the "Update transaction state to pending. (page **??**)" step, the update condition, which includes the `state: "initial"` criterion, will not match any document, and the `nMatched` and `nModified` will be `0`. This should signal to `App2` to go back to the first step to restart the procedure with a different transaction.

When multiple applications are running, it is crucial that only one application can handle a given transaction at any point in time. As such, in addition including the expected state of the transaction in the update condition, you can also create a marker in the transaction document itself to identify the application that is handling the transaction. Use `findAndModify()` method to modify the transaction and get it back in one step:

```
t = db.transactions.findAndModify(
       {
         query: { state: "initial", application: { $exists: false } },
         update:
```

```
            {
              $set: { state: "pending", application: "App1" },
              $currentDate: { lastModified: true }
            },
          new: true
        }
    )
```

Amend the transaction operations to ensure that only applications that match the identifier in the `application` field apply the transaction.

If the application `App1` fails during transaction execution, you can use the *recovery procedures* (page 65), but applications should ensure that they "own" the transaction before applying the transaction. For example to find and resume the pending job, use a query that resembles the following:

```
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

db.transactions.find(
   {
     application: "App1",
     state: "pending",
     lastModified: { $lt: dateThreshold }
   }
)
```

Using Two-Phase Commits in Production Applications

The example transaction above is intentionally simple. For example, it assumes that it is always possible to roll back operations to an account and that account balances can hold negative values.

Production implementations would likely be more complex. Typically, accounts need information about current balance, pending credits, and pending debits.

For all transactions, ensure that you use the appropriate level of *write concern* (page 23) for your deployment.

3.10 Update Document if Current

Overview

The *Update if Current* pattern is an approach to *concurrency control* (page 28) when multiple applications have access to the data.

Pattern

The pattern queries for the document to update. Then, for each field to modify, the pattern includes the field and its value in the returned document in the query predicate for the update operation. This way, the update only modifies the document fields *if* the fields have not changed since the query.

Example

Consider the following example in the `mongo` shell. The example updates the `quantity` and the `reordered` fields of a document *only* if the fields have not changed since the query.

Changed in version 2.6: The `db.collection.update()` method now returns a `WriteResult()` object that contains the status of the operation. Previous versions required an extra `db.getLastErrorObj()` method call.

```
var myDocument = db.products.findOne( { sku: "abc123" } );

if ( myDocument ) {
   var oldQuantity = myDocument.quantity;
   var oldReordered = myDocument.reordered;

   var results = db.products.update(
      {
        _id: myDocument._id,
        quantity: oldQuantity,
        reordered: oldReordered
      },
      {
        $inc: { quantity: 50 },
        $set: { reordered: true }
      }
   )

   if ( results.hasWriteError() ) {
      print( "unexpected error updating document: " + tojson(results) );
   }
   else if ( results.nMatched === 0 ) {
      print( "No matching document for " +
             "{ _id: "+ myDocument._id.toString() +
             ", quantity: " + oldQuantity +
             ", reordered: " + oldReordered
             + " } "
      );
   }
}
```

Modifications to the Pattern

Another approach is to add a `version` field to the documents. Applications increment this field upon each update operation to the documents. You must be able to ensure that *all* clients that connect to your database include the `version` field in the query predicate. To associate increasing numbers with documents in a collection, you can use one of the methods described in *Create an Auto-Incrementing Sequence Field* (page 72).

For more approaches, see *Concurrency Control* (page 28).

3.11 Create Tailable Cursor

Overview

By default, MongoDB will automatically close a cursor when the client has exhausted all results in the cursor. However, for `capped collections` you may use a *Tailable Cursor* that remains open after the client exhausts the results in the initial cursor. Tailable cursors are conceptually equivalent to the `tail` Unix command with the `-f` option (i.e. with "follow" mode). After clients insert new additional documents into a capped collection, the tailable cursor will continue to retrieve documents.

Use tailable cursors on capped collections that have high write volumes where indexes aren't practical. For instance, MongoDB `replication` uses tailable cursors to tail the primary's *oplog*.

Note: If your query is on an indexed field, do not use tailable cursors, but instead, use a regular cursor. Keep track of the last value of the indexed field returned by the query. To retrieve the newly added documents, query the collection again using the last value of the indexed field in the query criteria, as in the following example:

```
db.<collection>.find( { indexedField: { $gt: <lastvalue> } } )
```

Consider the following behaviors related to tailable cursors:

- Tailable cursors do not use indexes and return documents in *natural order*.
- Because tailable cursors do not use indexes, the initial scan for the query may be expensive; but, after initially exhausting the cursor, subsequent retrievals of the newly added documents are inexpensive.
- Tailable cursors may become *dead*, or invalid, if either:
  - the query returns no match.
  - the cursor returns the document at the "end" of the collection and then the application deletes that document.

  A *dead* cursor has an id of `0`.

See your `driver documentation` for the driver-specific method to specify the tailable cursor. For more information on the details of specifying a tailable cursor, see MongoDB wire protocol[15] documentation.

C++ Example

The `tail` function uses a tailable cursor to output the results from a query to a capped collection:

- The function handles the case of the dead cursor by having the query be inside a loop.
- To periodically check for new data, the `cursor->more()` statement is also inside a loop.

```
#include "client/dbclient.h"

using namespace mongo;

/*
 * Example of a tailable cursor.
 * The function "tails" the capped collection (ns) and output elements as they are added.
 * The function also handles the possibility of a dead cursor by tracking the field 'insertDate'.
 * New documents are added with increasing values of 'insertDate'.
 */

void tail(DBClientBase& conn, const char *ns) {

    BSONElement lastValue = minKey.firstElement();

    Query query = Query().hint( BSON( "$natural" << 1 ) );

    while ( 1 ) {
        auto_ptr<DBClientCursor> c =
            conn.query(ns, query, 0, 0, 0,
                       QueryOption_CursorTailable | QueryOption_AwaitData );

        while ( 1 ) {
            if ( !c->more() ) {
```

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

```
                if ( c->isDead() ) {
                    break;
                }

                continue;
            }

            BSONObj o = c->next();
            lastValue = o["insertDate"];
            cout << o.toString() << endl;
        }

        query = QUERY( "insertDate" << GT << lastValue ).hint( BSON( "$natural" << 1 ) );
    }
}
```

The `tail` function performs the following actions:

- Initialize the `lastValue` variable, which tracks the last accessed value. The function will use the `lastValue` if the cursor becomes *invalid* and `tail` needs to restart the query. Use `hint()` to ensure that the query uses the `$natural` order.
- In an outer `while(1)` loop,
  - Query the capped collection and return a tailable cursor that blocks for several seconds waiting for new documents

    ```
    auto_ptr<DBClientCursor> c =
         conn.query(ns, query, 0, 0, 0,
                    QueryOption_CursorTailable | QueryOption_AwaitData );
    ```

    * Specify the capped collection using `ns` as an argument to the function.
    * Set the `QueryOption_CursorTailable` option to create a tailable cursor.
    * Set the `QueryOption_AwaitData` option so that the returned cursor blocks for a few seconds to wait for data.
  - In an inner `while (1)` loop, read the documents from the cursor:
    * If the cursor has no more documents and is not invalid, loop the inner `while` loop to recheck for more documents.
    * If the cursor has no more documents and is dead, break the inner `while` loop.
    * If the cursor has documents:
      · output the document,
      · update the `lastValue` value,
      · and loop the inner `while (1)` loop to recheck for more documents.
  - If the logic breaks out of the inner `while (1)` loop and the cursor is invalid:
    * Use the `lastValue` value to create a new query condition that matches documents added after the `lastValue`. Explicitly ensure `$natural` order with the `hint()` method:

      ```
      query = QUERY( "insertDate" << GT << lastValue ).hint( BSON( "$natural" << 1 ) );
      ```

    * Loop through the outer `while (1)` loop to re-query with the new query condition and repeat.

See also:

Detailed blog post on tailable cursor[16]

3.12 Create an Auto-Incrementing Sequence Field

Synopsis

MongoDB reserves the `_id` field in the top level of all documents as a primary key. `_id` must be unique, and always has an index with a *unique constraint*. However, except for the unique constraint you can use any value for the `_id` field in your collections. This tutorial describes two methods for creating an incrementing sequence number for the `_id` field using the following:

- *Use Counters Collection* (page 72)
- *Optimistic Loop* (page 74)

Considerations

Generally in MongoDB, you would not use an auto-increment pattern for the `_id` field, or any field, because it does not scale for databases with large numbers of documents. Typically the default value *ObjectId* is more ideal for the `_id`.

Procedures

Use Counters Collection

Counter Collection Implementation Use a separate `counters` collection to track the *last* number sequence used. The `_id` field contains the sequence name and the `seq` field contains the last value of the sequence.

1. Insert into the `counters` collection, the initial value for the `userid`:

```
db.counters.insert(
   {
      _id: "userid",
      seq: 0
   }
)
```

2. Create a `getNextSequence` function that accepts a `name` of the sequence. The function uses the `findAndModify()` method to atomically increment the `seq` value and return this new value:

```
function getNextSequence(name) {
   var ret = db.counters.findAndModify(
          {
            query: { _id: name },
            update: { $inc: { seq: 1 } },
            new: true
          }
   );

   return ret.seq;
}
```

[16] http://shtylman.com/post/the-tail-of-mongodb

3. Use this `getNextSequence()` function during `insert()`.

```
db.users.insert(
   {
     _id: getNextSequence("userid"),
     name: "Sarah C."
   }
)

db.users.insert(
   {
     _id: getNextSequence("userid"),
     name: "Bob D."
   }
)
```

You can verify the results with `find()`:

```
db.users.find()
```

The `_id` fields contain incrementing sequence values:

```
{
  _id : 1,
  name : "Sarah C."
}
{
  _id : 2,
  name : "Bob D."
}
```

`findAndModify` Behavior When `findAndModify()` includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances. For instance, if multiple clients each invoke the method with the same query condition and these methods complete the find phase before any of methods perform the modify phase, these methods could insert the same document.

In the `counters` collection example, the query field is the `_id` field, which always has a unique index. Consider that the `findAndModify()` includes the `upsert: true` option, as in the following modified example:

```
function getNextSequence(name) {
   var ret = db.counters.findAndModify(
          {
            query: { _id: name },
            update: { $inc: { seq: 1 } },
            new: true,
            upsert: true
          }
   );

   return ret.seq;
}
```

If multiple clients were to invoke the `getNextSequence()` method with the same `name` parameter, then the methods would observe one of the following behaviors:

- Exactly one `findAndModify()` would successfully insert a new document.
- Zero or more `findAndModify()` methods would update the newly inserted document.
- Zero or more `findAndModify()` methods would fail when they attempted to insert a duplicate.

If the method fails due to a unique index constraint violation, retry the method. Absent a delete of the document, the retry should not fail.

Optimistic Loop

In this pattern, an *Optimistic Loop* calculates the incremented `_id` value and attempts to insert a document with the calculated `_id` value. If the insert is successful, the loop ends. Otherwise, the loop will iterate through possible `_id` values until the insert is successful.

1. Create a function named `insertDocument` that performs the "insert if not present" loop. The function wraps the `insert()` method and takes a `doc` and a `targetCollection` arguments.

   Changed in version 2.6: The `db.collection.insert()` method now returns a *writeresults-insert* object that contains the status of the operation. Previous versions required an extra `db.getLastErrorObj()` method call.

   ```
   function insertDocument(doc, targetCollection) {

       while (1) {

           var cursor = targetCollection.find( {}, { _id: 1 } ).sort( { _id: -1 } ).limit(1);

           var seq = cursor.hasNext() ? cursor.next()._id + 1 : 1;

           doc._id = seq;

           var results = targetCollection.insert(doc);

           if( results.hasWriteError() ) {
               if( results.writeError.code == 11000 /* dup key */ )
                   continue;
               else
                   print( "unexpected error inserting data: " + tojson( results ) );
           }

           break;
       }
   }
   ```

   The `while (1)` loop performs the following actions:

   - Queries the `targetCollection` for the document with the maximum `_id` value.
   - Determines the next sequence value for `_id` by:
     - adding `1` to the returned `_id` value if the returned cursor points to a document.
     - otherwise: it sets the next sequence value to `1` if the returned cursor points to no document.
   - For the `doc` to insert, set its `_id` field to the calculated sequence value `seq`.
   - Insert the `doc` into the `targetCollection`.
   - If the insert operation errors with duplicate key, repeat the loop. Otherwise, if the insert operation encounters some other error or if the operation succeeds, break out of the loop.

2. Use the `insertDocument()` function to perform an insert:

   ```
   var myCollection = db.users2;

   insertDocument(
   ```

```
    {
      name: "Grace H."
    },
    myCollection
);

insertDocument(
    {
      name: "Ted R."
    },
    myCollection
)
```

You can verify the results with `find()`:

```
db.users2.find()
```

The `_id` fields contain incrementing sequence values:

```
{
  _id: 1,
  name: "Grace H."
}
{
  _id : 2,
  "name" : "Ted R."
}
```

The `while` loop may iterate many times in collections with larger insert volumes.

4 MongoDB CRUD Reference

4.1 Query Cursor Methods

Name	Description
`cursor.count()`	Returns a count of the documents in a cursor.
`cursor.explain()`	Reports on the query execution plan, including index use, for a cursor.
`cursor.hint()`	Forces MongoDB to use a specific index for a query.
`cursor.limit()`	Constrains the size of a cursor's result set.
`cursor.next()`	Returns the next document in a cursor.
`cursor.skip()`	Returns a cursor that begins returning results only after passing or skipping a number of documents.
`cursor.sort()`	Returns results ordered according to a sort specification.
`cursor.toArray()`	Returns an array that contains all documents returned by the cursor.

4.2 Query and Data Manipulation Collection Methods

Name	Description
`db.collection.count()`	Wraps `count` to return a count of the number of documents in a collection or matching a query.
`db.collection.distinct()`	Returns an array of documents that have distinct values for the specified field.
`db.collection.find()`	Performs a query on a collection and returns a cursor object.
`db.collection.findOne()`	Performs a query and returns a single document.
`db.collection.insert()`	Creates a new document in a collection.
`db.collection.remove()`	Deletes documents from a collection.
`db.collection.save()`	Provides a wrapper around an `insert()` and `update()` to insert new documents.
`db.collection.update()`	Modifies a document in a collection.

4.3 MongoDB CRUD Reference Documentation

Write Concern Reference **(page 76)** Configuration options associated with the guarantee MongoDB provides when reporting on the success of a write operation.

SQL to MongoDB Mapping Chart **(page 78)** An overview of common database operations showing both the MongoDB operations and SQL statements.

The bios Example Collection **(page 84)** Sample data for experimenting with MongoDB. `insert()`, `update()` and `find()` pages use the data for some of their examples.

Write Concern Reference

Write concern (page 23) describes the guarantee that MongoDB provides when reporting on the success of a write operation.

Changed in version 2.6: A new protocol for *write operations* integrates write concerns with the write operations and eliminates the need to call the `getLastError` command. Previous versions required a `getLastError` command immediately after a write operation to specify the write concern.

Read Isolation Behavior

MongoDB allows clients to read documents inserted or modified before it commits these modifications to disk, regardless of write concern level or journaling configuration. As a result, applications may observe two classes of behaviors:

- For systems with multiple concurrent readers and writers, MongoDB will allow clients to read the results of a write operation before the write operation returns.
- If the `mongod` terminates before the journal commits, even if a write returns successfully, queries may have read data that will not exist after the `mongod` restarts.

Other database systems refer to these isolation semantics as *read uncommitted*. For all inserts and updates, MongoDB modifies each document in isolation: clients never see documents in intermediate states. For multi-document operations, MongoDB does not provide any multi-document transactions or isolation.

When `mongod` returns a successful *journaled write concern*, the data is fully committed to disk and will be available after `mongod` restarts.

For replica sets, write operations are durable only after a write replicates and commits to the journal of a majority of the voting members of the set. MongoDB regularly commits data to the journal regardless of journaled write concern: use the `commitIntervalMs` to control how often a `mongod` commits the journal.

Available Write Concern

Write concern can include the *w* (page 77) option to specify the required number of acknowledgments before returning, the *j* (page 77) option to require writes to the journal before returning, and *wtimeout* (page 77) option to specify a time limit to prevent write operations from blocking indefinitely.

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

`w` Option The `w` option provides the ability to disable write concern entirely *as well as* specify the write concern for *replica sets*.

MongoDB uses `w: 1` as the default write concern. `w: 1` provides basic receipt acknowledgment.

The `w` option accepts the following values:

Value	Description
`1`	Provides acknowledgment of write operations on a standalone `mongod` or the *primary* in a replica set. This is the default write concern for MongoDB.
`0`	Disables basic acknowledgment of write operations, but returns information about socket exceptions and networking errors to the application. If you disable basic write operation acknowledgment but require journal commit acknowledgment, the journal commit prevails, and the server will require that `mongod` acknowledge the write operation.
<Number greater than 1>	Guarantees that write operations have propagated successfully to the specified number of replica set members including the primary. For example, `w: 2` indicates acknowledgements from the primary and at least one secondary. 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 voting nodes: a majority of the replica set's voting members must acknowledge the write operation before it succeeds. This allows you to avoid hard coding assumptions about the size of your replica set into your application. Changed in version 3.0: In previous versions, `w: "majority"` refers to the majority of the replica set's members. Changed in version 2.6: In `Master/Slave` deployments, MongoDB treats `w: "majority"` as equivalent to `w: 1`. In earlier versions of MongoDB, `w: "majority"` produces an error in `master/slave` deployments.
<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.

`j` Option The `j` option confirms that the `mongod` instance has written the data to the on-disk journal. This ensures that data is not lost if the `mongod` instance shuts down unexpectedly. Set to `true` to enable.

Changed in version 2.6: Specifying a write concern that includes `j: true` to a `mongod` or `mongos` running with `--nojournal` option now errors. Previous versions would ignore the `j: true`.

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.

`wtimeout` This option specifies a time limit, in milliseconds, for the write concern. `wtimeout` is only applicable for `w` values greater than `1`.

`wtimeout` causes write operations to return with an error after the specified limit, even if the required write concern will eventually succeed. When these write operations return, MongoDB **does not** undo successful data modifications performed before the write concern exceeded the `wtimeout` time limit.

If you do not specify the `wtimeout` option and the level of write concern is unachievable, the write operation will block indefinitely. Specifying a `wtimeout` value of `0` is equivalent to a write concern without the `wtimeout` option.

See also:

Write Concern Introduction (page 23) and *Write Concern for Replica Sets* (page 25).

SQL to MongoDB Mapping Chart

In addition to the charts that follow, you might want to consider the `http://docs.mongodb.org/manual/faq` section for a selection of common questions about MongoDB.

Terminology and Concepts

The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	*database*
table	*collection*
row	*document* or *BSON* document
column	*field*
index	*index*
table joins	embedded documents and linking
primary key	*primary key*
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the *_id* field.
aggregation (e.g. group by)	aggregation pipeline See the `http://docs.mongodb.org/manual/reference/sql-aggregation-co`

Executables

The following table presents some database executables and the corresponding MongoDB executables. This table is *not* meant to be exhaustive.

	MongoDB	MySQL	Oracle	Informix	DB2
Database Server	`mongod`	`mysqld`	`oracle`	`IDS`	`DB2 Server`
Database Client	`mongo`	`mysql`	`sqlplus`	`DB-Access`	`DB2 Client`

Examples

The following table presents the various SQL statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume a table named `users`.
- The MongoDB examples assume a collection named `users` that contain documents of the following prototype:

```
{
  _id: ObjectId("509a8fb2f3f4948bd2f983a0"),
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

Create and Alter The following table presents the various SQL statements related to table-level actions and the corresponding MongoDB statements.

SQL Schema Statements	MongoDB Schema Statements

`CREATE TABLE users (`
`  id MEDIUMINT NOT NULL`
`  AUTO_INCREMENT,`
`  user_id Varchar(30),`
`  age Number,`
`  status char(1),`
`  PRIMARY KEY (id)`
`)`
	Implicitly created on first `insert()` operation. The primary key `_id` is automatically added if `_id` field is not specified.
	`db.users.insert( {`
	`  user_id: "abc123",`
	`  age: 55,`
	`  status: "A"`
	`} )`
	However, you can also explicitly create a collection:
	`db.createCollection("users")`

`ALTER TABLE users`
`ADD join_date DATETIME`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, `update()` operations can add fields to existing documents using the `$set` operator.
	`db.users.update(`
	`  { },`
	`  { $set: { join_date: new Date() } },`
	`  { multi: true }`
	`)`

`ALTER TABLE users`
`DROP COLUMN join_date`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, `update()` operations can remove fields from documents using the `$unset` operator.
	`db.users.update(`
	`  { },`
	`  { $unset: { join_date: "" } },`
	`  { multi: true }`
	`)`

`CREATE INDEX idx_user_id_asc` `ON users(user_id)`	`db.users.createIndex( { user_id: 1 } )`

`CREATE INDEX`
`  idx_user_id_asc_age_desc`
`ON users(user_id, age DESC)`
	`db.users.createIndex( { user_id: 1, age: -1 } )`

`DROP TABLE users`	`db.users.drop()`

For more information, see `db.collection.insert()`, `db.createCollection()`, `db.collection.update()`, `$set`, `$unset`, `db.collection.createIndex()`, `indexes`, `db.collection.drop()`, and `http://docs.mongodb.org/manual/core/data-models`.

Insert The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL INSERT Statements	MongoDB insert() Statements

`INSERT INTO users(user_id,`
`  age,`
`  status)`
`VALUES ("bcd001",`
`  45,`
`  "A")`
	`db.users.insert(`
	`  { user_id: "bcd001", age: 45, status: "A" }`
	`)`

For more information, see `db.collection.insert()`.

Select The following table presents the various SQL statements related to reading records from tables and the corresponding MongoDB statements.

SQL SELECT Statements	MongoDB find() Statements
`SELECT * FROM users`	`db.users.find()`
`SELECT id, user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1 } )`
`SELECT user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } )`
`SELECT user_id, status FROM users WHERE status = "A"`	`db.users.find( { status: "A" }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status != "A"`	`db.users.find( { status: { $ne: "A" } } )`
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`
`SELECT * FROM users WHERE status = "A" OR age = 50`	`db.users.find( { $or: [ { status: "A" } , { age: 50 } ] } )`
`SELECT * FROM users WHERE age > 25`	`db.users.find( { age: { $gt: 25 } } )`
`SELECT * FROM users WHERE age < 25`	`db.users.find( { age: { $lt: 25 } } )`
`SELECT * FROM users WHERE age > 25 AND age <= 50`	`db.users.find( { age: { $gt: 25, $lte: 50 } } )`
`SELECT * FROM users WHERE user_id like "%bc%"`	`db.users.find( { user_id: /bc/ } )`

For more information, see `db.collection.find()`, `db.collection.distinct()`, `db.collection.findOne()`, `$ne` `$and`, `$or`, `$gt`, `$lt`, `$exists`, `$lte`, `$regex`, `limit()`, `skip()`, `explain()`, `sort()`, and `count()`.

Update Records The following table presents the various SQL statements related to updating existing records in tables and the corresponding MongoDB statements.

SQL Update Statements	MongoDB update() Statements

`UPDATE users`
`SET status = "C"`
`WHERE age > 25`
	`db.users.update(`
	`  { age: { $gt: 25 } },`
	`  { $set: { status: "C" } },`
	`  { multi: true }`
	`)`

`UPDATE users`
`SET age = age + 3`
`WHERE status = "A"`
	`db.users.update(`
	`  { status: "A" } ,`
	`  { $inc: { age: 3 } },`
	`  { multi: true }`
	`)`

For more information, see `db.collection.update()`, `$set`, `$inc`, and `$gt`.

Delete Records The following table presents the various SQL statements related to deleting records from tables and the corresponding MongoDB statements.

SQL Delete Statements	MongoDB remove() Statements
`DELETE FROM users` `WHERE status = "D"`	`db.users.remove( { status: "D" } )`
`DELETE FROM users`	`db.users.remove({})`

For more information, see `db.collection.remove()`.

Additional Resources

- Transitioning from SQL to MongoDB (Presentation)[17]
- Best Practices for Migrating from RDBMS to MongoDB (Webinar)[18]
- RDBMS to MongoDB Migration Guide[19]
- SQL vs. MongoDB Day 1-2[20]
- SQL vs. MongoDB Day 3-5[21]
- MongoDB vs. SQL Day 18[22]

[17] http://www.mongodb.com/presentations/webinar-transitioning-sql-mongodb
[18] http://www.mongodb.com/webinar/best-practices-migration
[19] http://www.mongodb.com/lp/white-paper/migration-rdbms-nosql-mongodb
[20] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2
[21] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-3-5
[22] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-14

- MongoDB and MySQL Compared[23]

The `bios` Example Collection

The `bios` collection provides example data for experimenting with MongoDB. Many of this guide's examples on `insert`, `update` and `read` operations create or query data from the `bios` collection.

The following documents comprise the `bios` collection. In the examples, the data might be different, as the examples themselves make changes to the data.

```
{
    "_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"
        }
    ]
}

{
    "_id" : ObjectId("51df07b094c6acd67e492f41"),
    "name" : {
        "first" : "John",
        "last" : "McCarthy"
    },
    "birth" : ISODate("1927-09-04T04:00:00Z"),
    "death" : ISODate("2011-12-24T05:00:00Z"),
```

[23] http://www.mongodb.com/mongodb-and-mysql-compared

```
    "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"
        }
    ]
}

{
    "_id" : 3,
    "name" : {
        "first" : "Grace",
        "last" : "Hopper"
    },
    "title" : "Rear Admiral",
    "birth" : ISODate("1906-12-09T05:00:00Z"),
    "death" : ISODate("1992-01-01T05:00:00Z"),
    "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" : ISODate("1926-08-27T04:00:00Z"),
    "death" : ISODate("2002-08-10T04:00:00Z"),
    "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" : ISODate("1931-10-12T04:00:00Z"),
    "death" : ISODate("2002-06-29T04:00:00Z"),
    "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" : 6,
    "name" : {
        "first" : "Guido",
        "last" : "van Rossum"
    },
    "birth" : ISODate("1956-01-31T05:00:00Z"),
    "contribs" : [
        "Python"
    ],
    "awards" : [
        {
            "award" : "Award for the Advancement of Free Software",
            "year" : 2001,
            "by" : "Free Software Foundation"
        },
        {
            "award" : "NLUUG Award",
            "year" : 2003,
            "by" : "NLUUG"
        }
    ]
}

{
    "_id" : ObjectId("51e062189c6ae665454e301d"),
    "name" : {
        "first" : "Dennis",
        "last" : "Ritchie"
    },
    "birth" : ISODate("1941-09-09T04:00:00Z"),
    "death" : ISODate("2011-10-12T04:00:00Z"),
    "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"
        }
    ]
}

{
    "_id" : 8,
    "name" : {
        "first" : "Yukihiro",
        "aka" : "Matz",
        "last" : "Matsumoto"
    },
    "birth" : ISODate("1965-04-14T04:00:00Z"),
    "contribs" : [
        "Ruby"
    ],
    "awards" : [
        {
            "award" : "Award for the Advancement of Free Software",
            "year" : "2011",
            "by" : "Free Software Foundation"
        }
    ]
}

{
    "_id" : 9,
    "name" : {
        "first" : "James",
        "last" : "Gosling"
    },
    "birth" : ISODate("1955-05-19T04:00:00Z"),
    "contribs" : [
        "Java"
    ],
    "awards" : [
        {
            "award" : "The Economist Innovation Award",
            "year" : 2002,
            "by" : "The Economist"
        },
        {
            "award" : "Officer of the Order of Canada",
            "year" : 2007,
            "by" : "Canada"
        }
    ]
}

{
    "_id" : 10,
    "name" : {
        "first" : "Martin",
        "last" : "Odersky"
    },
    "contribs" : [
        "Scala"
```

```
    ]
}
```