MongoDB CRUD Operations

Release 2.6.1

MongoDB Documentation Project

June 05, 2014

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.

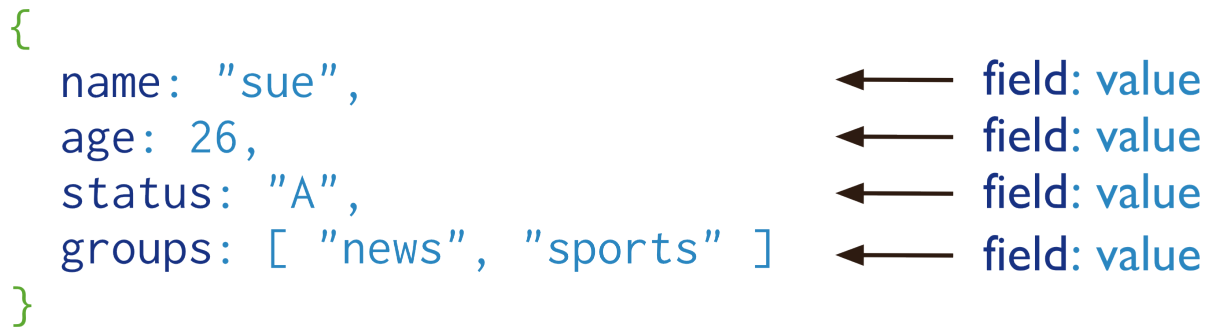
MongoDB CRUD Concepts **(page 6)** The core documentation of query and data manipulation.

MongoDB CRUD Tutorials **(page 35)** Examples of basic query and data modification operations.

MongoDB CRUD Reference **(page 61)** 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/manualcore/document`.



```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

Figure 1: A MongoDB 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.

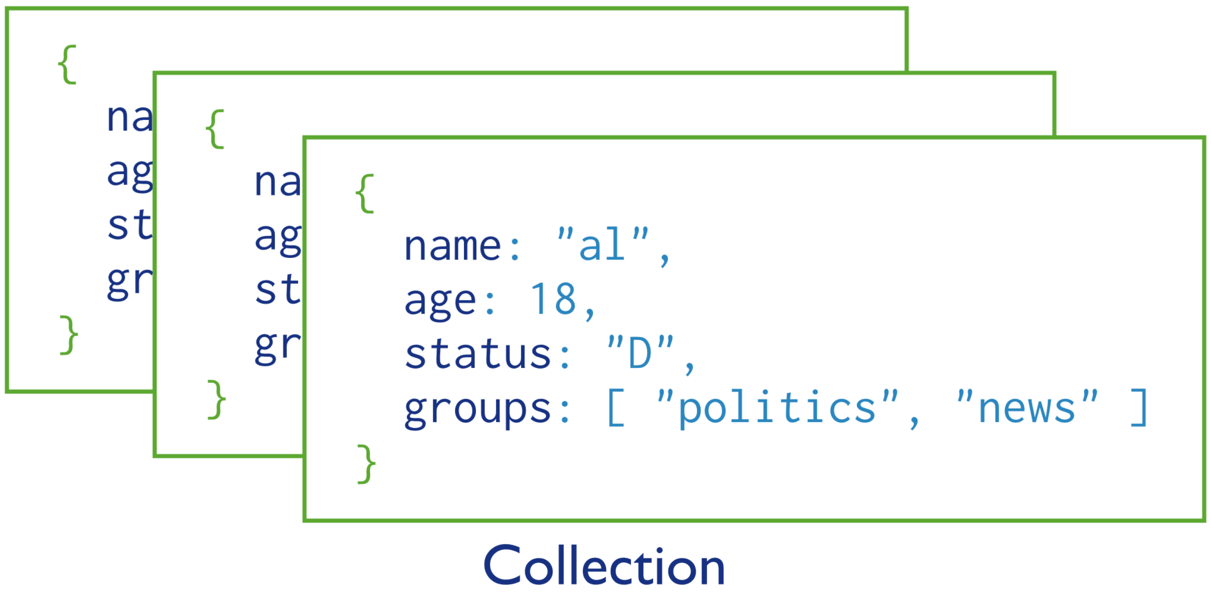


Figure 2: A collection of MongoDB documents.

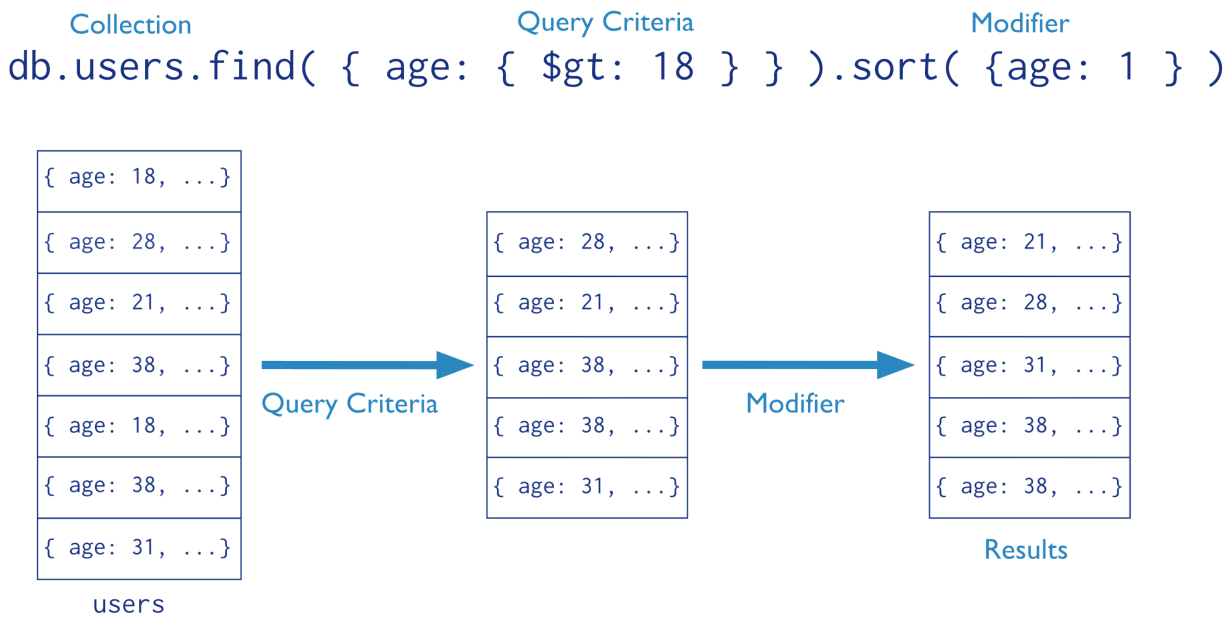


Figure 3: The stages of a MongoDB query with a query criteria and a sort modifier.

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.

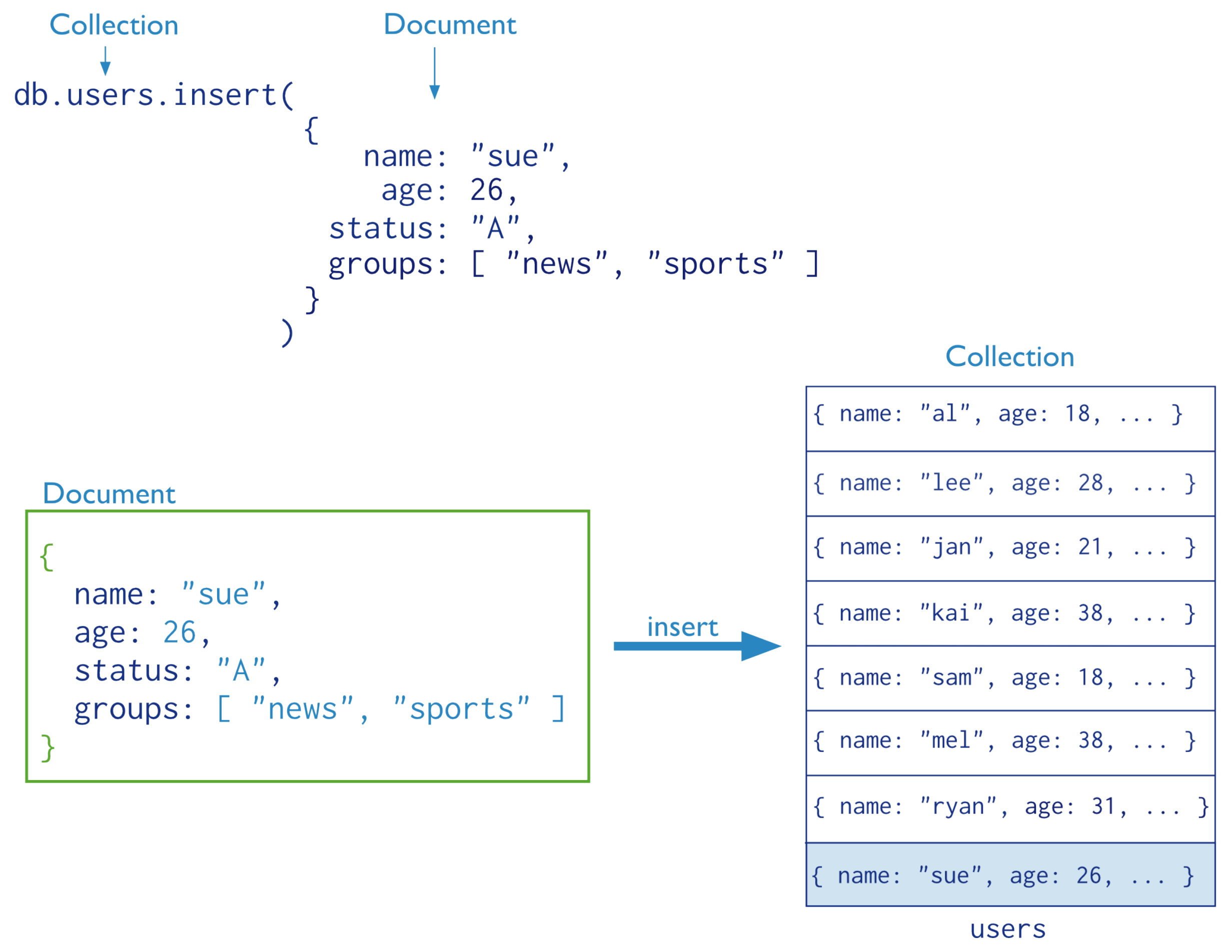

Collection
Document
db.users.insert(
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
)
Collection
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
users
Document
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
insert

Figure 4: The stages of a MongoDB insert operation.

See *Write Operations Overview* (page 19) 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`.

`/core/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 18) documents introduce the behavior and operations of read and write operations for MongoDB deployments.

Read Operations **(page 6)** Introduces all operations that select and return documents to clients, including the query specifications.

> ***Cursors*** **(page 10)** Queries return iterable objects, called cursors, that hold the full result set.
>
> ***Query Optimization*** **(page 12)** Analyze and improve query performance.
>
> ***Distributed Queries*** **(page 14)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Write Operations **(page 18)** 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 27)** 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 18) 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 12)** Analyze and improve query performance.

Query Plans **(page 13)** MongoDB executes queries using optimal *plans*.

Distributed Queries **(page 14)** 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 provide 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:

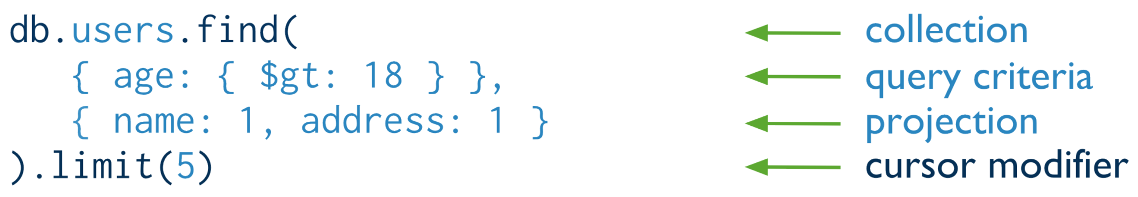


Figure 5: The components of a MongoDB find operation.

The next diagram shows the same query in SQL:

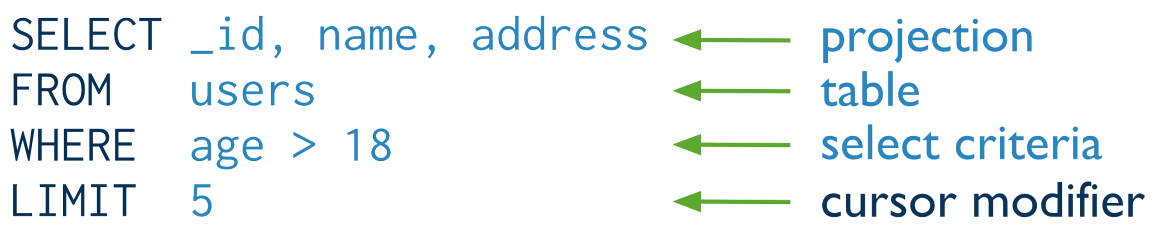


Figure 6: The components of a SQL SELECT statement.

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 9) for details.

See

SQL to MongoDB Mapping Chart (page 63) 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 45) (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:

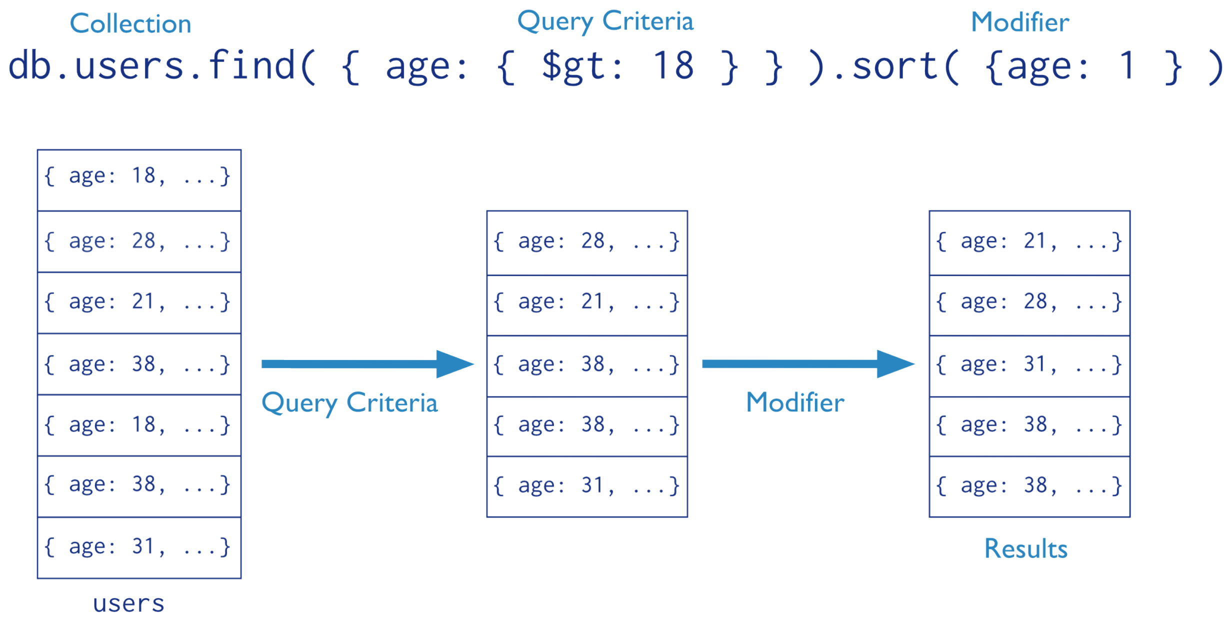


Figure 7: The stages of a MongoDB query with a query criteria and a sort modifier.

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 36).

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:

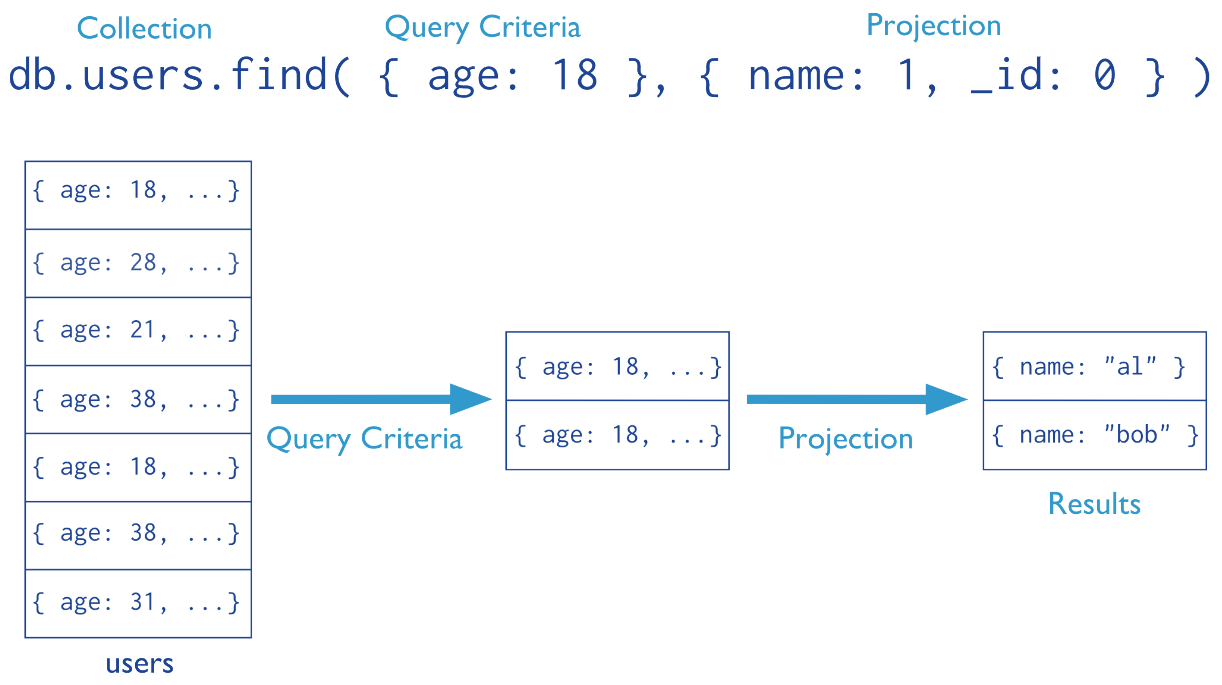


Figure 8: The stages of a MongoDB query with a query criteria and projection. MongoDB only transmits the projected data to the clients.

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 a number of documents in the `records` collection that match the query `{ "user_id": { $lt: 42} }`, but excludes the `history` field.

Return Two fields *and* the `_id` Field

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

This query selects a number of documents in the `records` collection that match the query `{ "user_id": { $lt: 42} }`, but returns documents that have the `_id` field (implicitly included) as well as the `name` and `email` fields.

Return Two Fields *and* Exclude `_id`

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

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

See

Limit Fields to Return from a Query (page 42) for more examples of queries with projection statements.

Projection Behavior MongoDB projections have the following properties:

- In MongoDB, the `_id` field is always included in results unless explicitly excluded.
- For fields that contain arrays, MongoDB provides the following projection operators: `$elemMatch`, `$slice`, `$`.
- For related projection functionality in the `aggregation framework` pipeline, use the `$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 43).

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);
```

[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

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*.

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 and will return all documents in the first batch.

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 fields, then an index on the queried field or 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.ensureIndex()` method:

```
db.inventory.ensureIndex( { 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 44).

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

Query Selectivity

Some query operations are not selective. These operations cannot use indexes effectively or cannot use indexes at all.

The inequality operators `$nin` and `$ne` are not very selective, as they often match a large portion of the index. As a result, in most cases, a `$nin` or `$ne` query with an index may perform no better than a `$nin` or `$ne` query that must scan all documents in a collection.

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

Covering a Query

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

- all the fields in the *query* (page 36) are part of that index, **and**
- all the fields returned in the documents that match the query 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 these queries, MongoDB does not need to inspect documents outside of the index. This is often more efficient than inspecting entire documents.

Example

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

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

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

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

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

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

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

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 14).

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

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.

Cached Query Plan Interface

New in version 2.6.

MongoDB provides `http://docs.mongodb.org/manualreference/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, check the `explain.filterSet` field of the `explain()` output.

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/manualsharding` section of this manual.

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

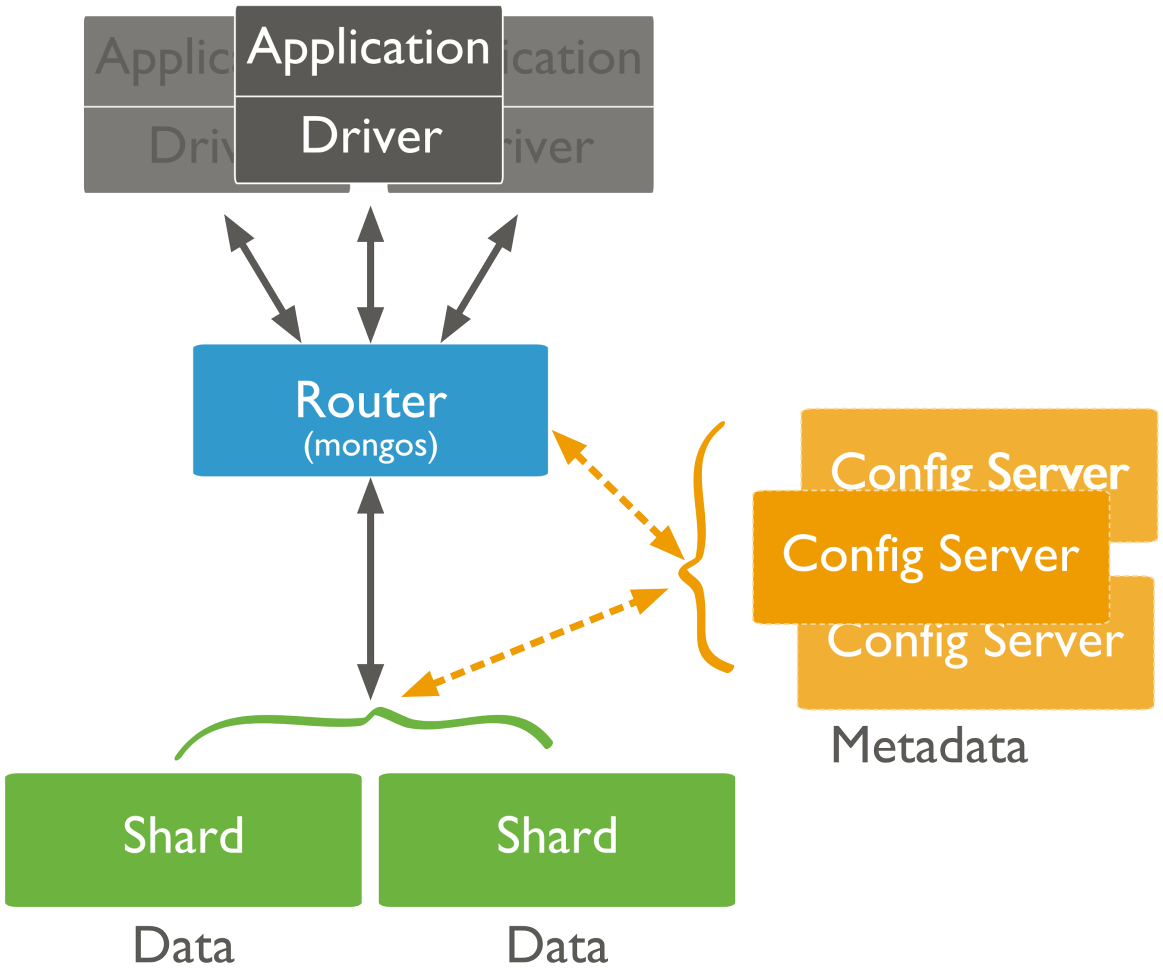


Figure 9: Diagram of a sharded cluster.

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.

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/manualcore/sharded-clu` 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*.

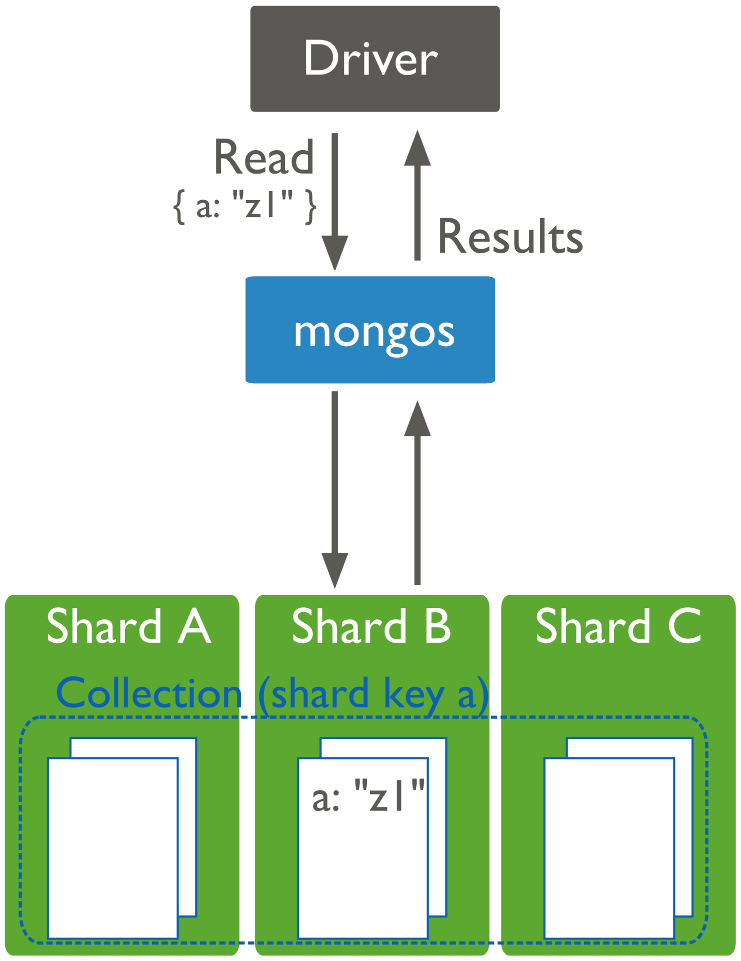


Figure 10: Read operations to a sharded cluster. Query criteria includes the shard key. The query router `mongos` can target the query to the appropriate shard or shards.

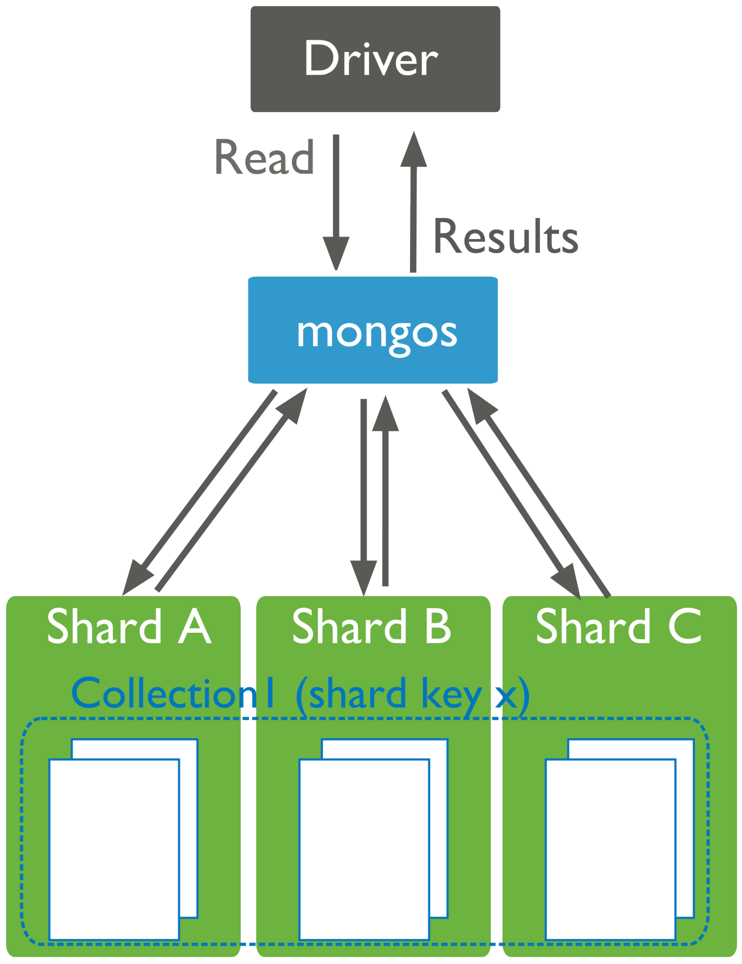


Figure 11: Read operations to a sharded cluster. Query criteria does not include the shard key. The query router `mongos` must broadcast query to all shards for the collection.

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.

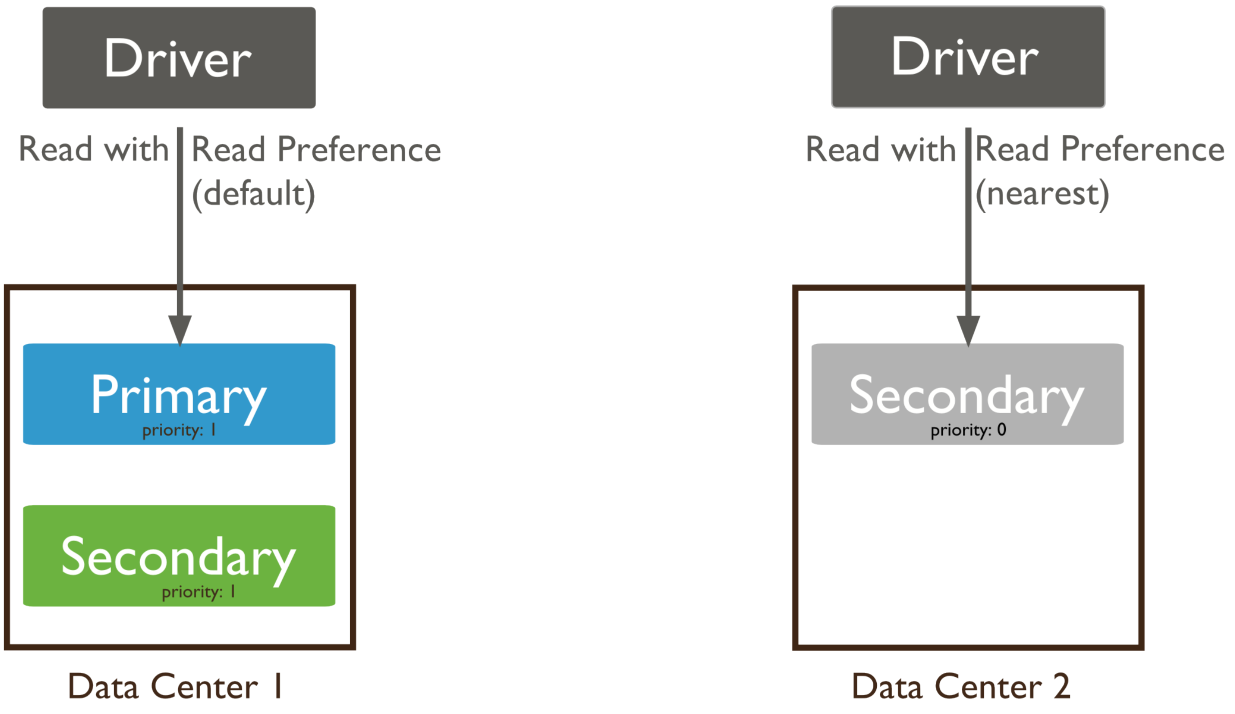


Figure 12: Read operations to a replica set. Default read preference routes the read to the primary. Read preference of `nearest` routes the read to the nearest member.

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/manualcore/read-preference` and *replica-set-read-preference-modes*.

2.2 Write Operations

The following documents describe write operations:

Write Operations Overview **(page 19)** 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.

***Distributed Write Operations* (page 27)** Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

***Write Operation Performance* (page 31)** Introduces the performance constraints and factors for writing data to MongoDB deployments.

***Bulk Inserts in MongoDB* (page 32)** Describe behaviors associated with inserting an array of documents.

***Storage* (page 33)** 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, update, and remove. 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.

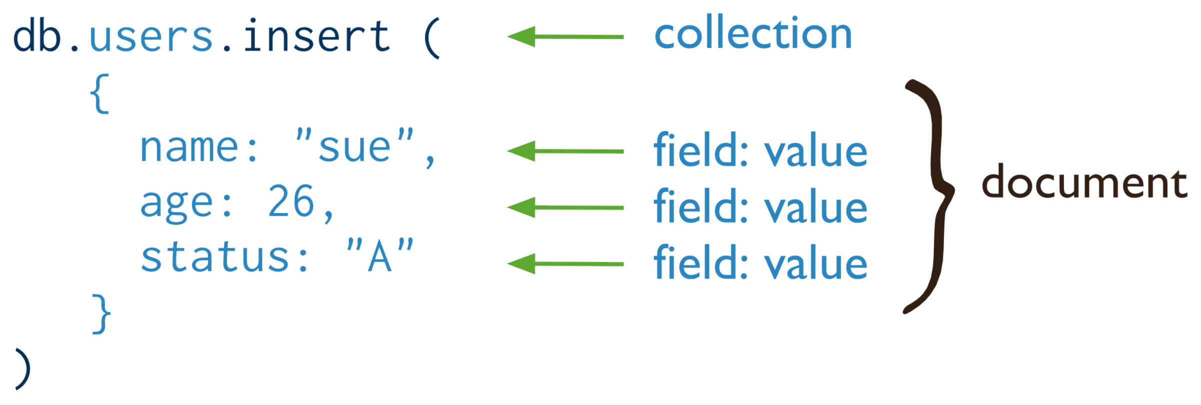
For the update and remove operations, you can specify criteria, or conditions, that identify the documents to update or remove. These operations use the same query syntax to specify the criteria as *read operations* (page 6).

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

Create

Create operations add new *documents* to a collection. In MongoDB, the `db.collection.insert()` method perform create operations.

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



```
db.users.insert (
   {
     name: "sue",
     age: 26,
     status: "A"
   }
)
```

Figure 13: The components of a MongoDB insert operations.

The following diagram shows the same query in SQL:

Example

The following operation inserts a new documents 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.

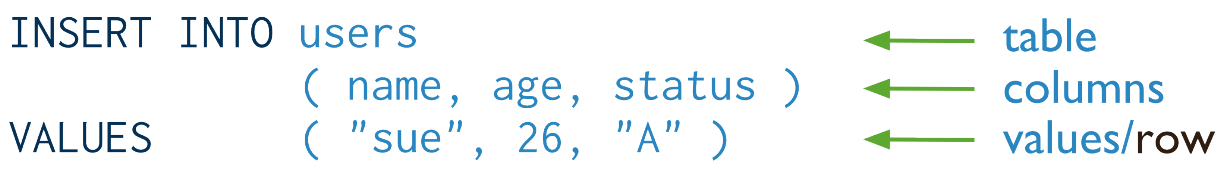


Figure 14: The components of a SQL INSERT statement.

```
db.users.insert(
   {
      name: "sue",
      age: 26,
      status: "A"
   }
)
```

For more information, see `db.collection.insert()` and *Insert Documents* (page 35).

Some updates also create records. If an update operation specifies the *upsert* flag *and* there are no documents that match the query portion of the update operation, then the update operation creates a new document. If there are matching documents, then an update operation with the *upsert* flag modifies the matching document or documents.

With an *upsert*, applications can decide between performing an update or an insert operation using just a single call. Both the `update()` method and the `save()` method can perform an *upsert*. See `update()` and `save()` for details on performing an *upsert* with these methods.

See

SQL to MongoDB Mapping Chart (page 63) for additional examples of MongoDB write operations and the corresponding SQL statements.

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.

Update

Update operations modify existing *documents* in a *collection*. In MongoDB, `db.collection.update()` and the `db.collection.save()` methods perform update operations. The `db.collection.update()` method can accept query criteria to determine which documents to update as well as an option to update multiple rows. The method can also accept options that affect its behavior such as the `multi` option to update multiple documents.

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

The following diagram shows the same query in SQL:

Example

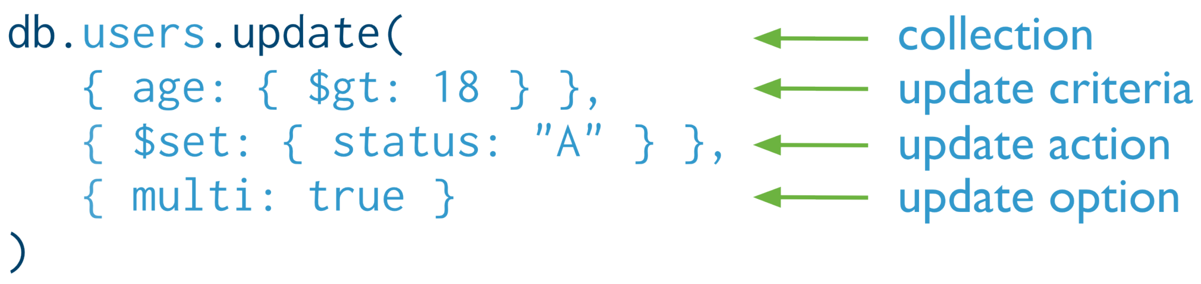


Figure 15: The components of a MongoDB update operation.

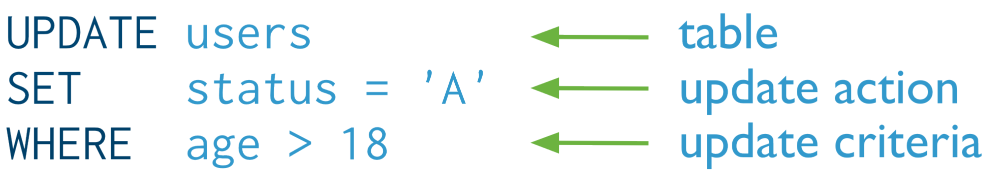


Figure 16: The components of a SQL UPDATE statement.

```
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 `db.collection.save()`, and *Modify Documents* (page 45) for examples.

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.

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.

The `db.collection.save()` method replaces a document and can only update a single document. See `db.collection.save()` and *Insert Documents* (page 35) for more information

Delete

Delete operations remove documents from a collection. In MongoDB, `db.collection.remove()` method performs delete operations. 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:

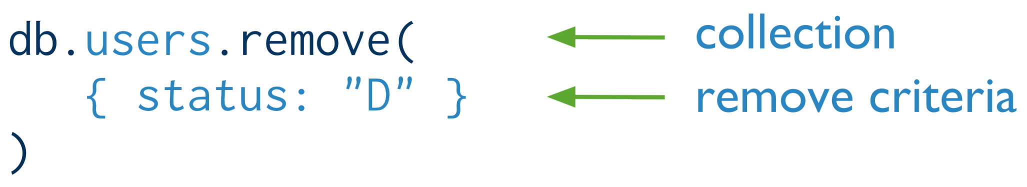


Figure 17: The components of a MongoDB remove operation.

The following diagram shows the same query in SQL:

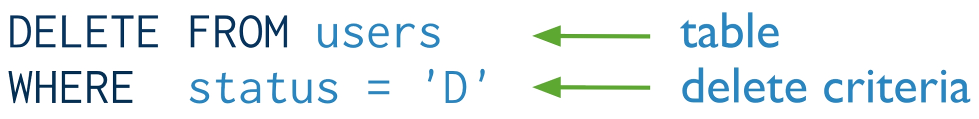


Figure 18: The components of a SQL DELETE statement.

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 46).

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 sub-documents *within* that document. For write operations that modify multiple documents, the operation as a whole is not atomic, and other operations may interleave.

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

To isolate a sequence of write operations from other read and write operations, see *Perform Two Phase Commits* (page 47).

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 62).

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 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.

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.

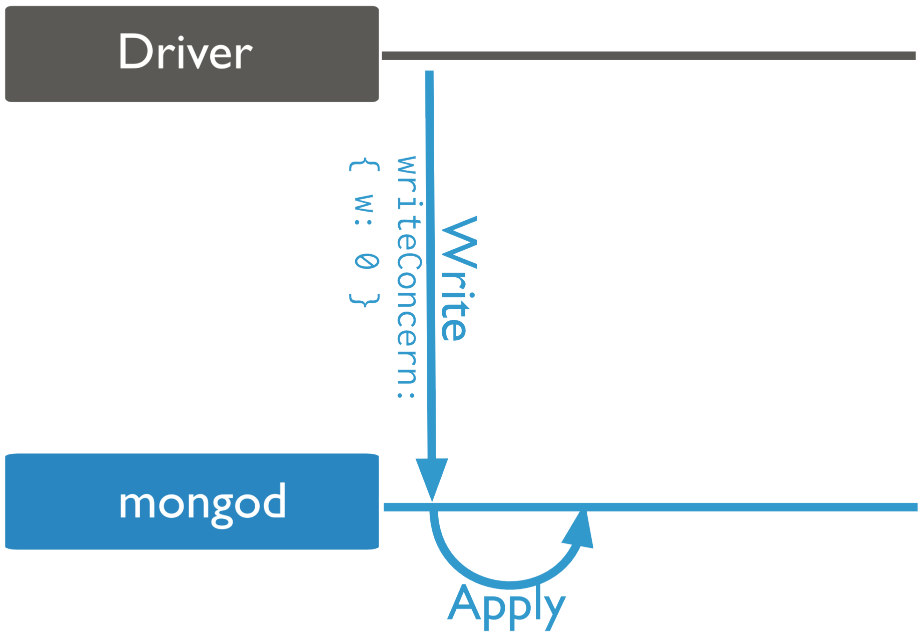


Figure 19: Write operation to a `mongod` instance with write concern of `unacknowledged`. The client does not wait for any acknowledgment.

Acknowledged With a receipt *acknowledged* write concern, the `mongod` confirms the receipt of the write operation. *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.

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.

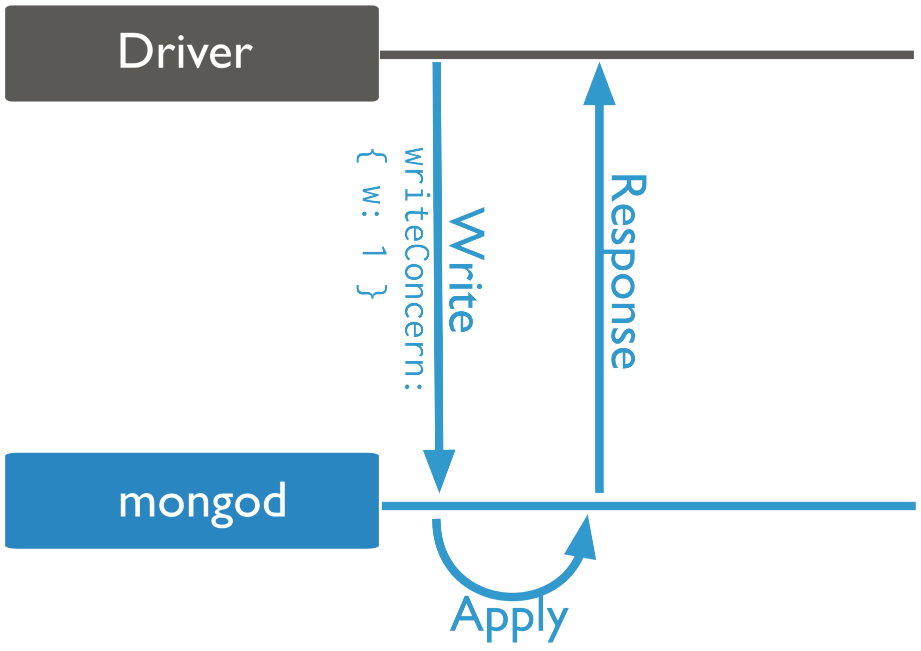


Figure 20: Write operation to a `mongod` instance with write concern of `acknowledged`. The client waits for acknowledgment of success or exception.

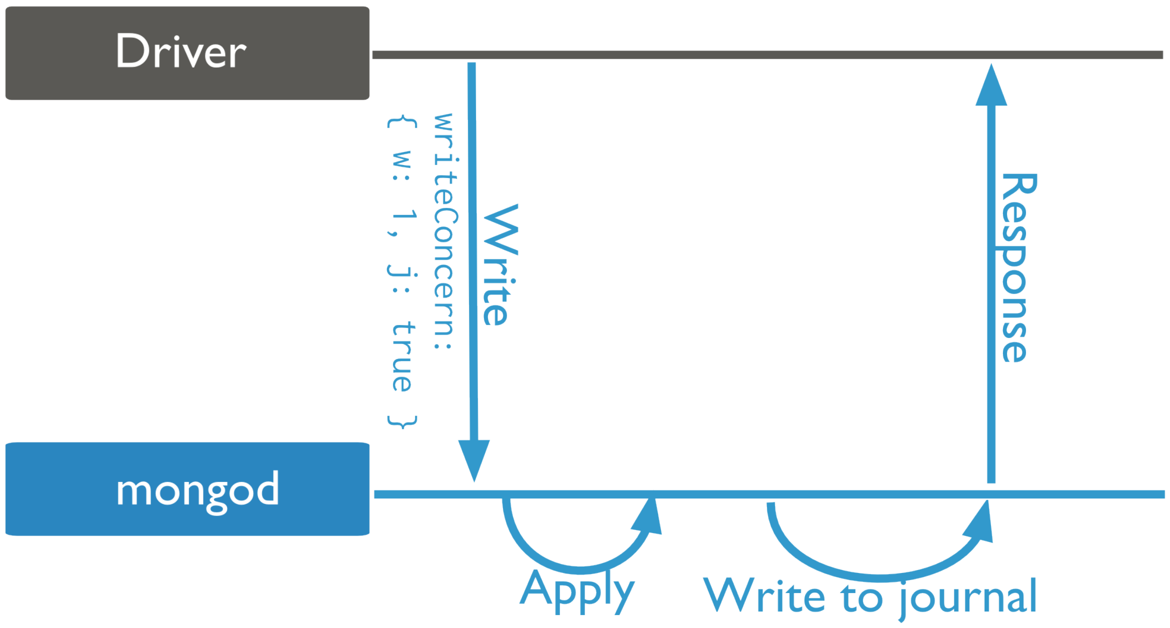


Figure 21: Write operation to a `mongod` instance with write concern of `journaled`. The `mongod` sends acknowledgment after it commits the write operation to the journal.

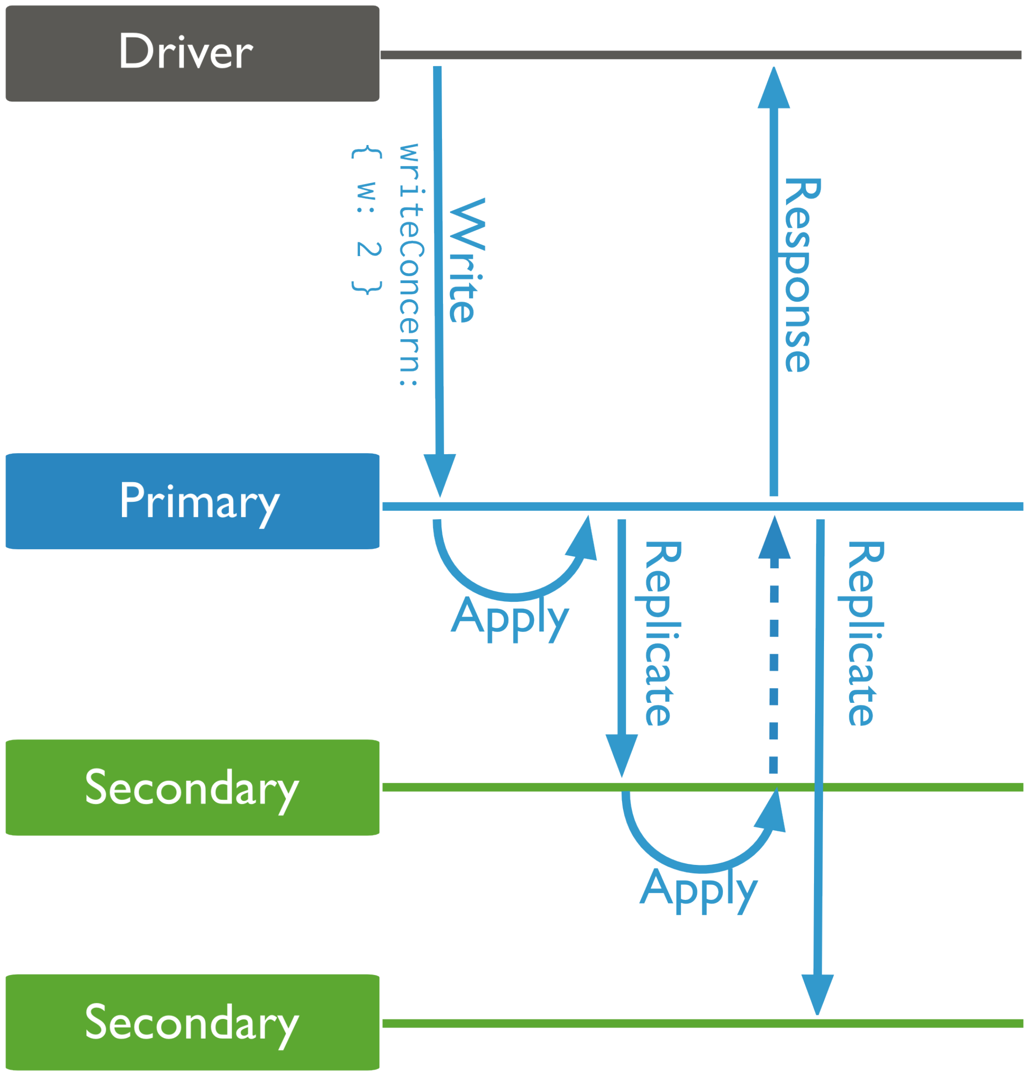


Figure 22: Write operation to a replica set with write concern level of `w:2` or write to the primary and at least one secondary.

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 62)

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.

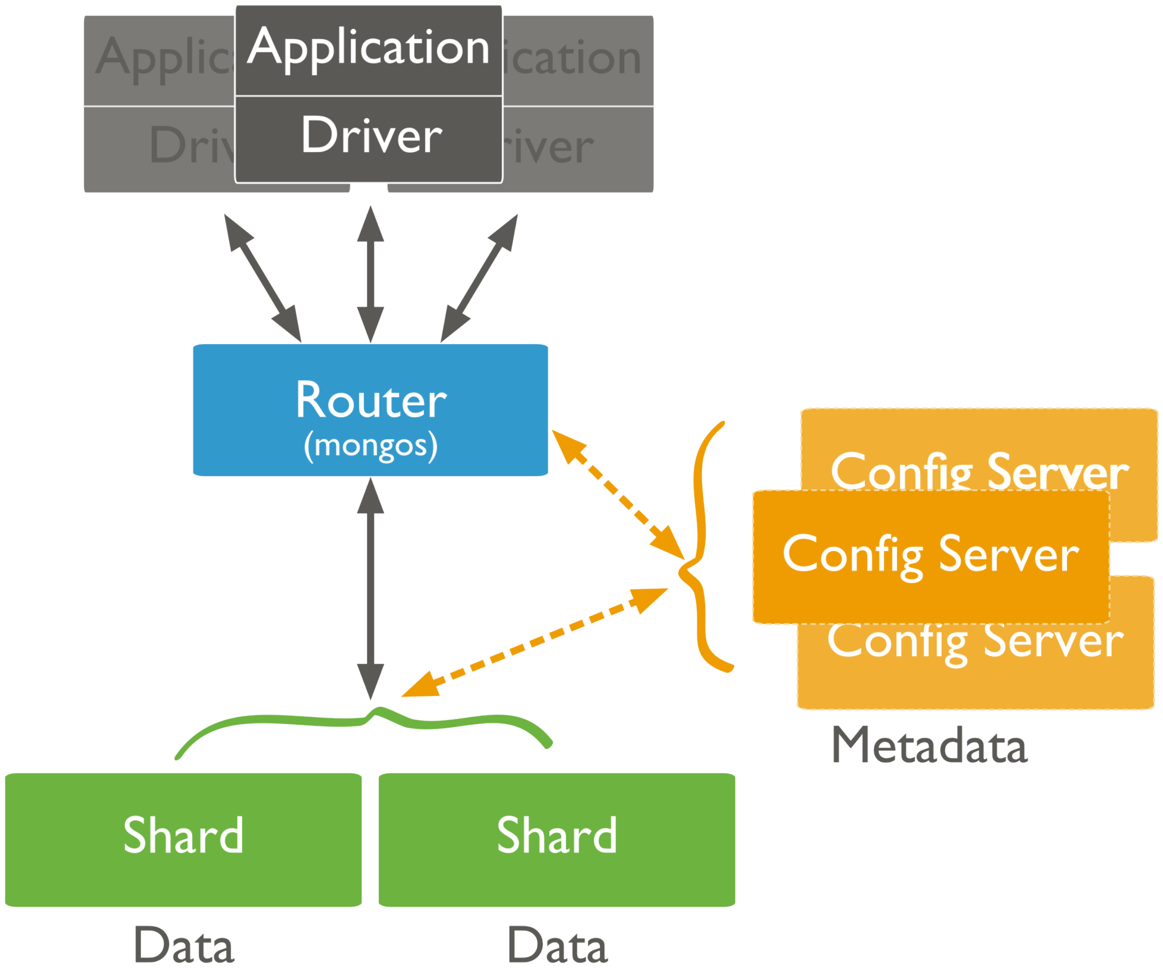


Figure 23: Diagram of a sharded cluster.

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.

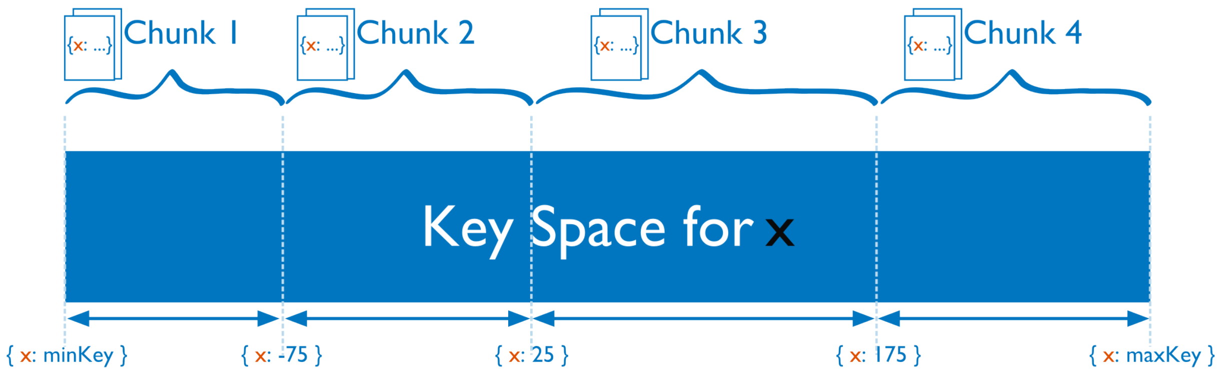


Figure 24: Diagram of the shard key value space segmented into smaller ranges or chunks.

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/manualadministration/sharded-clusters` and *Bulk Inserts in MongoDB* (page 32).

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 [4] 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 24), *replica-set-oplog-sizing*, and `http://docs.mongodb.org/manualtutorial/change-oplog-size`.

[4] 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.

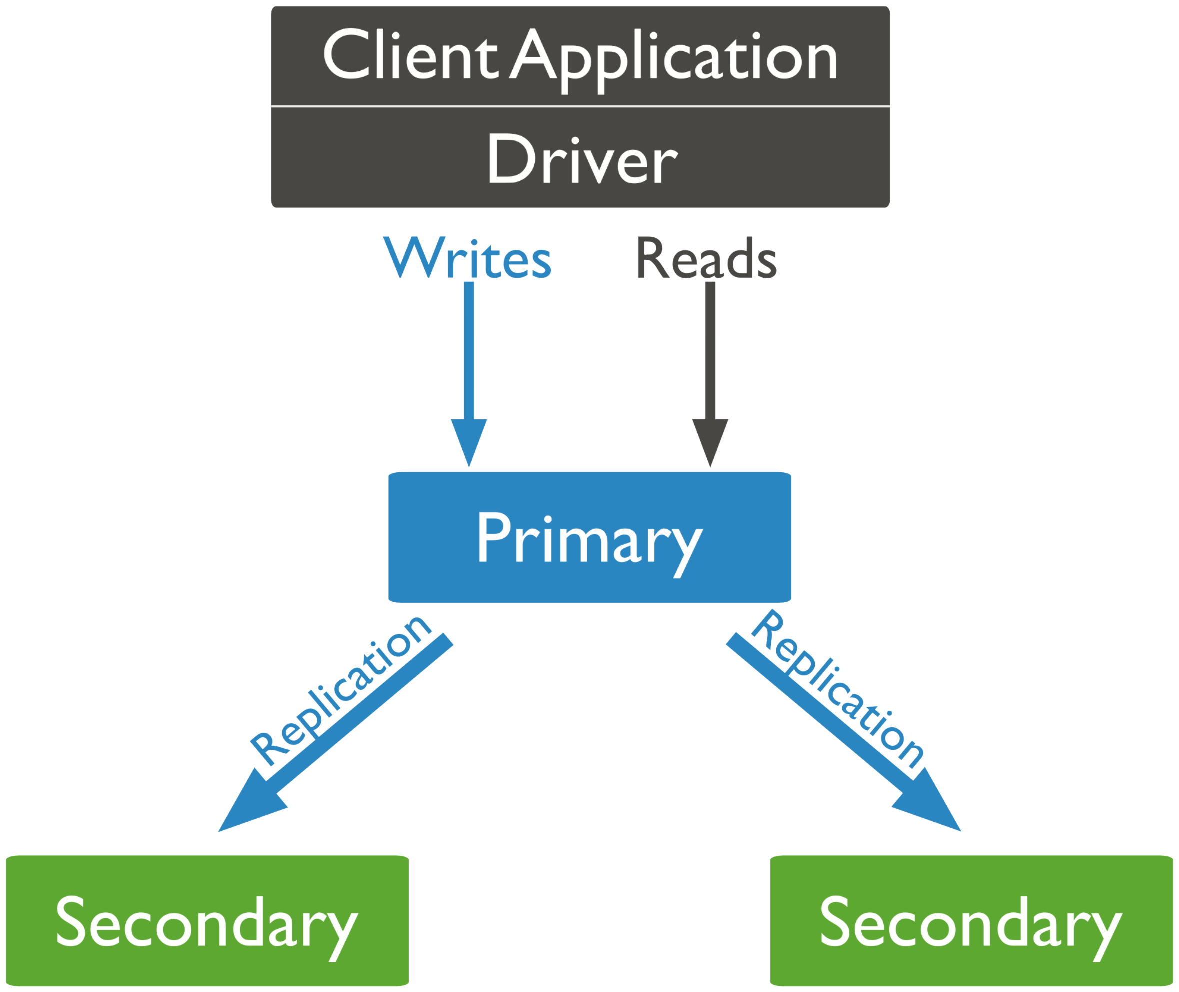


Figure 25: Diagram of default routing of reads and writes to the primary.

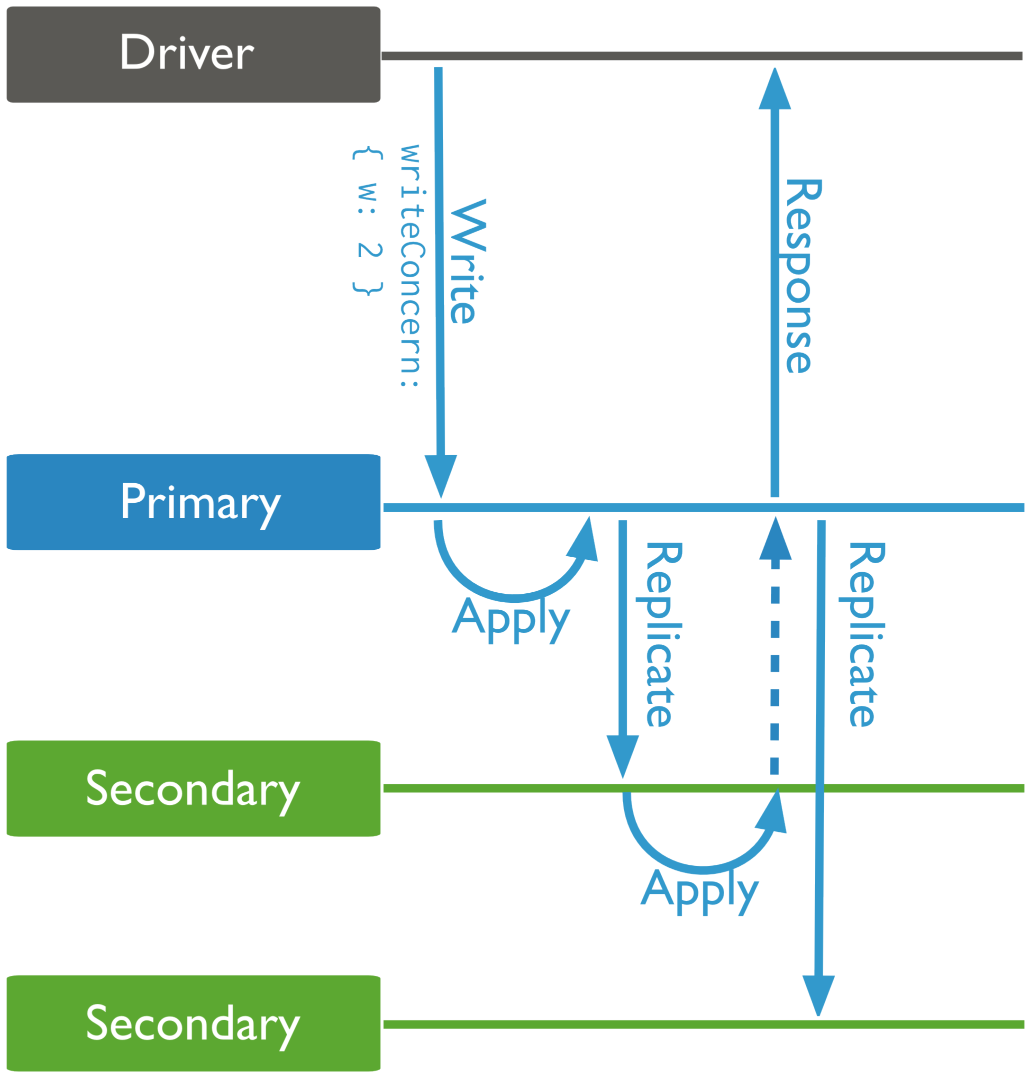


Figure 26: Write operation to a replica set with write concern level of `w:2` or write to the primary and at least one secondary.

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. [5]

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 12). For more information on indexes, see `http://docs.mongodb.org/manualindexes` and `http://docs.mongodb.org/manualapplications/indexes`.

Document Growth

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. These relocations take longer than in-place updates, 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.

Some update operations, such as the `$inc` operation, do not cause an increase in document size. For these update operations, MongoDB can apply the updates in-place. Other update operations, such as the `$push` operation, change the size of the document.

In-place-updates are significantly more efficient than updates that cause document growth. When possible, use `data models` that minimize the need for document growth.

See *Storage* (page 33) 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/manualadministration/production-notes` for recommendations regarding additional hardware and configuration options.

Journaling MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* (page 18) 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:

[5] 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.

- 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 24), `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/manualcore/journaling`.

Bulk Inserts in MongoDB

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

Use the `insert()` Method

The `insert()` method, when passed an array of documents, performs a bulk insert, and inserts each document atomically. Bulk inserts can significantly increase performance by amortizing *write concern* (page 23) costs.

New in version 2.2: `insert()` in the `mongo` shell gained support for bulk inserts in version 2.2.

In the `drivers`, you can configure write concern for batches rather than on a per-document level.

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

Note: If multiple errors occur during a bulk insert, clients only receive the last error generated.

See also:

`Driver documentation` for details on performing bulk inserts in your application. Also see `http://docs.mongodb.org/manualcore/import-export`.

Bulk Inserts on Sharded Clusters

While `ContinueOnError` is optional on unsharded clusters, all bulk operations to a *sharded collection* run with `ContinueOnError`, which cannot be disabled.

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/manualtutorial/split-chunks-in-sharded-cluster`.

Insert to Multiple `mongos` To parallelize import processes, send insert operations to more than one `mongos` instance. Pre-split empty collections first as described in `http://docs.mongodb.org/manualtutorial/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 that 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

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`.

Every document in MongoDB is stored in a *record* which contains the document itself and extra space, or *padding*, which allows the document to grow as the result of updates.

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.

All records are part of a *collection*, which is a logical grouping 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. Also, each database has one distinct write lock, that blocks operations to the database during write operations. A single MongoDB deployment may have many databases.

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/manualcore/journaling` for more information about the journal in MongoDB.

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 34) are more efficient for insert/update/delete workloads; while *exact fit allocations* (page 34) is ideal for collections *without* update and delete workloads.

Power of 2 Sized Allocations Changed in version 2.6: For all new collections, `usePowerOf2Sizes` became the default allocation strategy. To change the default allocation strategy, use the `newCollectionsUsePowerOf2Sizes` parameter.

`mongod` uses an allocation strategy called `usePowerOf2Sizes` where 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 two key properties:

- there are a limited number of record allocation sizes, which makes it easier for `mongod` to reuse existing allocations, which will reduce fragmentation in some cases.
- in many cases, the record allocations are significantly larger than the documents they hold. This allows documents to grow while minimizing or eliminating the chance that the `mongod` will need to allocate a new record if the document grows.

The `usePowerOf2Sizes` strategy does not *eliminate* document reallocation as a result of document growth, but it minimizes its occurrence in many common operations.

Exact Fit Allocation The exact fit allocation strategy allocates record sizes based on the size of the document and an additional *padding factor*. Each collection has its own padding factor, which defaults to `0` when you insert the first document in a collection, and may grow incrementally to `2` as as documents grow as a result of updates.

Multiply the size of a document by the padding factor to determine the total record size. That is:

```
record size = paddingFactor * <document size>.
```

The size of each record in a collection reflects the size of the padding factor at the time of allocation. See the `paddingFactor` field in the output of `db.collection.stats()` to see the current padding factor for a collection.

On average, this exact fit allocation strategy uses less storage space than the `usePowerOf2Sizes` strategy but will result in higher levels of storage fragmentation if documents grow beyond the size of their initial allocation.

The `compact` and `repairDatabase` operations remove padding by default, as do the `mongodump` and `mongorestore`. `compact` does allow you to specify a padding for records during compaction.

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/manualcore/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 2).

***Insert Documents* (page 35)** Insert new documents into a collection.

***Query Documents* (page 36)** Find documents in a collection using search criteria.

***Limit Fields to Return from a Query* (page 42)** Limit which fields are returned by a query.

***Iterate a Cursor in the mongo Shell* (page 43)** Access documents returned by a `find` query by iterating the cursor, either manually or using the iterator index.

***Analyze Query Performance* (page 44)** Analyze the efficiency of queries and determine how a query uses available indexes.

***Modify Documents* (page 45)** Modify documents in a collection

***Remove Documents* (page 46)** Remove documents from a collection.

***Perform Two Phase Commits* (page 47)** Use two-phase commits when writing data to multiple documents.

***Create Tailable Cursor* (page 52)** Create tailable cursors for use in capped collections with high numbers of write operations for which an index would be too expensive.

***Isolate Sequence of Operations* (page 55)** Use the `<isolation> isolated` operator to *isolate* a single write operation that affects multiple documents, preventing other operations from interrupting the sequence of write operations.

***Create an Auto-Incrementing Sequence Field* (page 56)** Describes how to create an incrementing sequence number for the `_id` field using a Counters Collection or an Optimistic Loop.

***Limit Number of Elements in an Array after an Update* (page 60)** Use `$push` with various modifiers to sort and maintain an array of fixed size after update

3.1 Insert Documents

In MongoDB, the `db.collection.insert()` method adds new documents into a collection. In addition, both the `db.collection.update()` method and the `db.collection.save()` method can also add new documents through an operation called an *upsert*. An *upsert* is an operation that performs either an update of an existing document or an insert of a new document if the document to modify does not exist.

This tutorial provides examples of insert operations using each of the three methods in the `mongo` shell.

Insert a Document with `insert()` Method

The following statement inserts a document with three fields into the collection `inventory`:

```
db.inventory.insert( { _id: 10, type: "misc", item: "card", qty: 15 } )
```

In the example, the document has a user-specified `_id` field value of `10`. The value must be unique within the `inventory` collection.

For more examples, see `insert()`.

Insert a Document with `update()` Method

Call the `update()` method with the `upsert` flag to create a new document if no document matches the update's query criteria. [6]

The following example creates a new document if no document in the `inventory` collection contains `{ type: "books", item : "journal" }`:

```
db.inventory.update(
                     { type: "book", item : "journal" },
                     { $set : { qty: 10 } },
                     { upsert : true }
                   )
```

MongoDB adds the `_id` field and assigns as its value a unique ObjectId. The new document includes the `item` and `type` fields from the `<query>` criteria and the `qty` field from the `<update>` parameter.

```
{ "_id" : ObjectId("51e8636953dbe31d5f34a38a"), "item" : "journal", "qty" : 10, "type" : "book" }
```

For more examples, see `update()`.

Insert a Document with `save()` Method

To insert a document with the `save()` method, pass the method a document that does not contain the `_id` field or a document that contains an `_id` field that does not exist in the collection.

The following example creates a new document in the `inventory` collection:

```
db.inventory.save( { type: "book", item: "notebook", qty: 40 } )
```

MongoDB adds the `_id` field and assigns as its value a unique ObjectId.

```
{ "_id" : ObjectId("51e866e48737f72b32ae4fbc"), "type" : "book", "item" : "notebook", "qty" : 40 }
```

For more examples, see `save()`.

3.2 Query Documents

In MongoDB, the `db.collection.find()` method retrieves documents from a collection. [7] The `db.collection.find()` method returns a *cursor* (page 10) to the retrieved documents.

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

[7] 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.

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 42).

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/manualreference/operator` 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 `food` **and** a less than (`$lt`) comparison match on the field `price`:

```
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*.

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.

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

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

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.

In the following example, the query matches all documents where the value of the field `tags` is an array that contains `'fruit'` as one of its elements:

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

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 value of the `tags` field is an array whose first element equals `'fruit'`:

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

Array of Embedded Documents

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

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

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  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 subdocument'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,
   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 subdocument.

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

Match Multiple Fields To match by multiple fields in the embedded document, you can use either dot notation or the `$elemMatch` operator:

The following example uses dot notation to find documents where whose `memos` field is an array that contains at least one document that contains the field `memo` equal to `'on time'` and at least one document that contains the field `by` equal to `'shipping'`.

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

The embedded documents that satisfy the two conditions can be either the same document or separate documents; i.e. a single embedded document can, but does not need to, satisfy both conditions. The query returns the following documents:

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

The following example uses `$elemMatch` to query for documents where `memos` field is an array that 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,
   memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
```

3.3 Limit Fields to Return from a Query

The *projection* specification limits the fields to return for all matching documents. The projection takes the form of a *document* with a list of fields for inclusion or exclusion from the result set. You can either specify the fields to include (e.g. `{ field: 1 }`) or specify the fields to exclude (e.g. `{ field: 0 }`).

Important: The `_id` field is, by default, included in the result set. To exclude the `_id` field from the result set, you need to specify in the projection document the exclusion of the `_id` field (i.e. `{ _id: 0 }`).

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 36).

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

The `$elemMatch` and `$slice` projection operators are the *only* way to project *portions* of an array.

Tip

MongoDB does not support projections of portions of arrays *except* when using the `$elemMatch` and `$slice` projection operators.

3.4 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 [8] and print the matching documents, as in the following example:

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

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

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

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

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

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

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

myCursor.forEach(printjson);
```

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

Iterator Index

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

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

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

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

Consider the following example:

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

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

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

3.5 Analyze Query Performance

The `explain()` cursor method allows you to inspect the operation of the query system. This method is useful for analyzing the efficiency of queries, and for determining how the query uses the index. The `explain()` method tests the query operation, and *not* the timing of query performance. Because `explain()` attempts multiple query plans, it does not reflect an accurate timing of query performance.

Evaluate the Performance of a Query

To use the `explain()` method, call the method on a cursor returned by `find()`.

Example

Evaluate a query on the `type` field on the collection `inventory` that has an index on the `type` field.

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

Consider the results:

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

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

This query returned 5 documents, as indicated by the `n` field.

To return these 5 documents, the query scanned 5 documents from the index, as indicated by the `nscanned` field, and then read 5 full documents from the collection, as indicated by the `nscannedObjects` field.

Without the index, the query would have scanned the whole collection to return the 5 documents.

See *explain-results* method for full details on the output.

Compare Performance of Indexes

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

Example

Evaluate a query using different indexes:

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

These return the statistics regarding the execution of the query using the respective index.

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

For more detail on the explain output, see *explain-results*.

3.6 Modify Documents

In MongoDB, both `db.collection.update()` and `db.collection.save()` modify existing documents in a collection. `db.collection.update()` provides additional control over the modification. For example, you can modify existing data or modify a group of documents that match a query with `db.collection.update()`. Alternately, `db.collection.save()` replaces an existing document with the same `_id` field.

This document provides examples of the update operations using each of the two methods in the `mongo` shell.

Modify Multiple Documents with `update()` Method

By default, the `update()` method updates a single document that matches its selection criteria. Call the method with the `multi` option set to `true` to update multiple documents. [9]

The following example finds all documents with `type` equal to `"book"` and modifies their `qty` field by `-1`. The example uses `$inc`, which is one of the *update operators* available.

```
db.inventory.update(
   { type : "book" },
   { $inc : { qty : -1 } },
   { multi: true }
)
```

For more examples, see `update()`.

Modify a Document with `save()` Method

The `save()` method can replace an existing document. To replace a document with the `save()` method, pass the method a document with an `_id` field that matches an existing document.

The following example completely replaces the document with the `_id` equal to `10` in the `inventory` collection:

```
db.inventory.save(
   {
     _id: 10,
     type: "misc",
     item: "placard"
   }
)
```

For further examples, see `save()`.

3.7 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:

[9] This shows the syntax for MongoDB 2.2 and later. For syntax for versions prior to 2.2, see `update()`.

```
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.

3.8 Perform Two Phase Commits

Synopsis

This document provides a pattern for doing multi-document updates or “transactions” using a two-phase commit approach for writing data to multiple documents. Additionally, you can extend this process to provide a *rollback* (page 50) like 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 “transactions,” are not atomic. Since documents can be fairly complex and contain multiple “nested” documents, single-document atomicity provides necessary support for many practical use cases.

Thus, without precautions, success or failure of the database operation cannot be “all or nothing,” and without support for multi-document transactions it’s possible for an operation to succeed for some operations and fail with others. When executing a transaction composed of several sequential operations the following issues arise:

- 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.”)
- Isolation: operations that run concurrently with the transaction operation set must “see” a consistent view of the data throughout the transaction process.
- Consistency: if a major failure (i.e. network, hardware) interrupts the transaction, the database must be able to recover a consistent state.

Despite the power of single-document atomic operations, there are cases that require multi-document transactions. For these situations, you can use a two-phase commit, to provide support for these kinds of multi-document updates.

Because documents can represent both pending data and states, you can use a two-phase commit to ensure that data is consistent, and that in the case of an error, the state that preceded the transaction is *recoverable* (page 50).

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

Pattern

Overview

The most common example of transaction is to transfer funds from account A to B in a reliable way, and this pattern uses this operation as an example. In a relational database system, this operation would encapsulate subtracting funds from the source (`A`) account and adding them to the destination (`B`) within a single atomic transaction. For MongoDB, you can use a two-phase commit in these situations to achieve a compatible response.

All of the examples in this document use the `mongo` shell to interact with the database, and assume that you have two collections: First, a collection named `accounts` that will store data about accounts with one account per document, and a collection named `transactions` which will store the transactions themselves.

Begin by creating two accounts named `A` and `B`, with the following command:

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

To verify that these operations succeeded, use `find()`:

```
db.accounts.find()
```

`mongo` will return two *documents* that resemble the following:

```
{ "_id" : ObjectId("4d7bc66cb8a04f512696151f"), "name" : "A", "balance" : 1000, "pendingTransactions"
{ "_id" : ObjectId("4d7bc67bb8a04f5126961520"), "name" : "B", "balance" : 1000, "pendingTransactions"
```

Transaction Description

Set Transaction State to Initial Create the `transaction` collection by inserting the following document. The transaction document holds the `source` and `destination`, which refer to the `name` fields of the `accounts` collection, as well as the `value` field that represents the amount of data change to the `balance` field. Finally, the `state` field reflects the current state of the transaction.

```
db.transactions.save({source: "A", destination: "B", value: 100, state: "initial"})
```

To verify that these operations succeeded, use `find()`:

```
db.transactions.find()
```

This will return a document similar to the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "source" : "A", "destination" : "B", "value" : 100,
```

Switch Transaction State to Pending Before modifying either records in the accounts collection, set the transaction state to pending from initial.

Set the local variable t in your shell session, to the transaction document using findOne():

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

After assigning this variable t, the shell will return the value of t, you will see the following output:

```
{
     "_id" : ObjectId("4d7bc7a8b8a04f5126961522"),
     "source" : "A",
     "destination" : "B",
     "value" : 100,
     "state" : "initial"
}
```

Use update() to change the value of state to pending:

```
db.transactions.update({_id: t._id}, {$set: {state: "pending"}})
db.transactions.find()
```

The find() operation will return the contents of the transactions collection, which should resemble the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "source" : "A", "destination" : "B", "value" : 100, "
```

Apply Transaction to Both Accounts Continue by applying the transaction to both accounts using the update() method.

In the query for the update(), the condition pendingTransactions: {$ne: t._id} prevents the update from applying the transaction t to an account if the pendingTransactions field for the account contains the _id of the transaction t:

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

db.accounts.find()
```

The find() operation will return the contents of the accounts collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 900, "name" : "A", "pendingTransactions"
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions"
```

Set Transaction State to Committed Use the following update() operation to set the transaction's state to committed:

```
db.transactions.update({_id: t._id}, {$set: {state: "committed"}})
db.transactions.find()
```

The find() operation will return the contents of the transactions collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "destination" : "B", "source" : "A", "state" : "comm
```

Remove Pending Transaction Use the following `update()` operation to set remove the pending transaction from the *documents* in the `accounts` collection:

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

The `find()` operation will return the contents of the `accounts` collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 900, "name" : "A", "pendingTransactions"
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions'
```

Set Transaction State to Done Complete the transaction by setting the `state` of the transaction *document* to `done`:

```
db.transactions.update({_id: t._id}, {$set: {state: "done"}})
db.transactions.find()
```

The `find()` operation will return the contents of the `transactions` collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "destination" : "B", "source" : "A", "state" : "done'
```

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 as intended. This section will provide an overview of possible failures and provide methods to recover from these kinds of events.

There are two classes of failures:

- all failures that occur after the first step (i.e. *setting the transaction set to initial* (page 48)) but before the third step (i.e. *applying the transaction to both accounts* (page 49).)

  To recover, applications should get a list of transactions in the `pending` state and resume from the second step (i.e. *switching the transaction state to pending* (page 49).)

- all failures that occur after the third step (i.e. *applying the transaction to both accounts* (page 49)) but before the fifth step (i.e. *setting the transaction state to done* (page 50).)

  To recover, application should get a list of transactions in the `committed` state and resume from the fourth step (i.e. *remove the pending transaction* (page 50).)

Thus, the application will always be able to resume the transaction and eventually arrive at a consistent state. Run the following recovery operations every time the application starts to catch any unfinished transactions. You may also wish run the recovery operation at regular intervals to ensure that your data remains in a consistent state.

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

Rollback In some cases you may need to "rollback" or undo a transaction when the application needs to "cancel" the transaction, or because it can never recover as in cases where one of the accounts doesn't exist, or stops existing during the transaction.

There are two possible rollback operations:

1. After you *apply the transaction* (page 49) (i.e. the third step), you have fully committed the transaction and you should not roll back the transaction. Instead, create a new transaction and switch the values in the source and destination fields.
2. After you *create the transaction* (page 48) (i.e. the first step), but before you *apply the transaction* (page 49) (i.e the third step), use the following process:

Set Transaction State to Canceling Begin by setting the transaction's state to `canceling` using the following `update()` operation:

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

Undo the Transaction Use the following sequence of operations to undo the transaction operation from both accounts:

```
db.accounts.update({name: t.source, pendingTransactions: t._id}, {$inc: {balance: t.value}, $pull: {p
db.accounts.update({name: t.destination, pendingTransactions: t._id}, {$inc: {balance: -t.value}, $pu
db.accounts.find()
```

The `find()` operation will return the contents of the `accounts` collection, which should resemble the following:

```
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 1000, "name" : "A", "pendingTransactions'
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1000, "name" : "B", "pendingTransactions'
```

Set Transaction State to Canceled Finally, use the following `update()` operation to set the transaction's state to `canceled`:

```
db.transactions.update({_id: t._id}, {$set: {state: "canceled"}})
```

Multiple Applications Transactions exist, in part, so that several applications can create and run operations concurrently without causing data inconsistency or conflicts. As a result, it is crucial that only one 1 application can handle a given transaction at any point in time.

Consider the following example, with a single transaction (i.e. `T1`) and two applications (i.e. `A1` and `A2`). If both applications begin processing the transaction which is still in the `initial` state (i.e. *step 1* (page 48)), then:

- `A1` can apply the entire whole transaction before `A2` starts.
- `A2` will then apply `T1` for the second time, because the transaction does not appear as pending in the `accounts` documents.

To handle multiple applications, create a marker in the transaction document itself to identify the application that is handling the transaction. Use `findAndModify()` method to modify the transaction:

```
t = db.transactions.findAndModify({query: {state: "initial", application: {$exists: 0}},
                                   update: {$set: {state: "pending", application: "A1"}},
                                   new: true})
```

When you modify and reassign the local shell variable `t`, the `mongo` shell will return the `t` object, which should resemble the following:

```
{
     "_id" : ObjectId("4d7be8af2c10315c0847fc85"),
     "application" : "A1",
     "destination" : "B",
     "source" : "A",
     "state" : "pending",
     "value" : 150
}
```

Amend the transaction operations to ensure that only applications that match the identifier in the value of the `application` field before applying the transaction.

If the application `A1` fails during transaction execution, you can use the *recovery procedures* (page 50), but applications should ensure that they "own" the transaction before applying the transaction. For example to resume pending jobs, use a query that resembles the following:

```
db.transactions.find({application: "A1", state: "pending"})
```

This will (or may) return a document from the `transactions` document that resembles the following:

```
{ "_id" : ObjectId("4d7be8af2c10315c0847fc85"), "application" : "A1", "destination" : "B", "source"
```

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 an account.
- account balances can hold negative values.

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

- when your application *switches the transaction state to pending* (page 49) (i.e. step 2) it would also make sure that the account has sufficient funds for the transaction. During this update operation, the application would also modify the values of the credits and debits as well as adding the transaction as pending.
- when your application *removes the pending transaction* (page 49) (i.e. step 4) the application would apply the transaction on balance, modify the credits and debits as well as removing the transaction from the `pending` field., all in one update.

Because all of the changes in the above two operations occur within a single `update()` operation, these changes are all atomic.

Additionally, for most important transactions, ensure that:

- the database interface (i.e. client library or *driver*) has a reasonable *write concern* configured to ensure that operations return a response on the success or failure of a write operation.
- your `mongod` instance has *journaling* enabled to ensure that your data is always in a recoverable state, in the event of an unclean `mongod` shutdown.

3.9 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 those 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[10] 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() ) {
```

[10] 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[11]

3.10 Isolate Sequence of Operations

Overview

Write operations are atomic on the level of a single document: no single write operation can atomically affect more than one document or more than one collection.

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

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

This document describes one method of updating documents *only* if the local copy of the document reflects the current state of the document in the database. In addition the following methods provide a way to manage isolated sequences of operations:

- the `findAndModify()` provides an isolated query and modify operation.
- *Perform Two Phase Commits* (page 47)
- Create a *unique index*, to ensure that a key doesn't exist when you insert it.

Update if Current

In this pattern, you will:

- query for a document,
- modify the fields in that document
- and update the fields of a document *only if* the fields have not changed in the collection since the query.

Consider the following example in JavaScript which attempts to update the `qty` field of a document in the `products` collection:

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 myCollection = db.products;
var myDocument = myCollection.findOne( { sku: 'abc123' } );

if (myDocument) {

   var oldQty = myDocument.qty;

   if (myDocument.qty < 10) {
       myDocument.qty *= 4;
   } else if ( myDocument.qty < 20 ) {
       myDocument.qty *= 3;
   } else {
       myDocument.qty *= 2;
   }
```

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

```
    var results = myCollection.update(
       {
         _id: myDocument._id,
         qty: oldQty
       },
       {
         $set: { qty: myDocument.qty }
       }
    );

    if ( results.hasWriteError() ) {
        print("unexpected error updating document: " + tojson( results ));
    } else if ( results.nMatched == 0 ) {
        print("No update: no matching document for { _id: " + myDocument._id + ", qty: " + oldQty + "
    }

}
```

Your application may require some modifications of this pattern, such as:

- Use the entire document as the query in the `update()` operation, to generalize the operation and guarantee that the original document was not modified, rather than ensuring that as single field was not changed.
- Add a version variable to the document that applications increment upon each update operation to the documents. Use this version variable in the query expression. You must be able to ensure that *all* clients that connect to your database obey this constraint.
- Use `$set` in the update expression to modify only your fields and prevent overriding other fields.
- Use one of the methods described in *Create an Auto-Incrementing Sequence Field* (page 56).

3.11 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 56)
- *Optimistic Loop* (page 58)

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;
   }
   ```

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.

3.12 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.

Important: The array elements must be documents in order to use the `$sort` modifier.

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 subdocument 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.

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 62)** Configuration options associated with the guarantee MongoDB provides when reporting on the success of a write operation.

SQL to MongoDB Mapping Chart **(page 63)** An overview of common database operations showing both the MongoDB operations and SQL statements.

The bios Example Collection **(page 68)** 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 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 62) option to specify the required number of acknowledgments before returning, the *j* (page 63) option to require writes to the journal before returning, and *wtimeout* (page 63) 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 configured replica set: a majority of the set's configured 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 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. This option causes write operations to return upon reaching this limit, even if the required write concern has yet to be fulfilled.

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), *Write Concern for Replica Sets* (page 24)

SQL to MongoDB Mapping Chart

In addition to the charts that follow, you might want to consider the `http://docs.mongodb.org/manualfaq` 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/manualreference/sql-aggregation-com`

Executables

The following table presents the MySQL/Oracle executables and the corresponding MongoDB executables.

	MySQL/Oracle	MongoDB
Database Server	`mysqld/oracle`	`mongod`
Database Client	`mysql/sqlplus`	`mongo`

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.ensureIndex( { user_id: 1 } )`
`CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)`	`db.users.ensureIndex( { 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.ensureIndex()`, `indexes`, `db.collection.drop()`, and `http://docs.mongodb.org/manualcore/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()`.

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"),
    "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"
    ]
}
```

Index