

---

# MongoDB CRUD Operations

*Release 3.2.5*

**MongoDB, Inc.**

April 22, 2016





<b>1</b>	<b>MongoDB CRUD Introduction</b>	<b>3</b>
1.1	Database Operations . . . . .	3
<b>2</b>	<b>MongoDB CRUD Concepts</b>	<b>7</b>
2.1	Read Operations . . . . .	7
2.2	Write Operations . . . . .	21
2.3	Read Isolation, Consistency, and Recency . . . . .	41
<b>3</b>	<b>MongoDB CRUD Tutorials</b>	<b>45</b>
3.1	Insert Documents . . . . .	45
3.2	Query Documents . . . . .	49
3.3	Modify Documents . . . . .	57
3.4	Remove Documents . . . . .	61
3.5	Limit Fields to Return from a Query . . . . .	62
3.6	Limit Number of Elements in an Array after an Update . . . . .	65
3.7	Iterate a Cursor in the <code>mongo</code> Shell . . . . .	67
3.8	Analyze Query Performance . . . . .	68
3.9	Perform Two Phase Commits . . . . .	73
3.10	Update Document if Current . . . . .	80
3.11	Create Tailable Cursor . . . . .	81
3.12	Create an Auto-Incrementing Sequence Field . . . . .	82
3.13	Perform Quorum Reads on Replica Sets . . . . .	86
<b>4</b>	<b>MongoDB CRUD Reference</b>	<b>89</b>
4.1	Query Cursor Methods . . . . .	89
4.2	Query and Data Manipulation Collection Methods . . . . .	89
4.3	MongoDB CRUD Reference Documentation . . . . .	90



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 7)** The core documentation of query and data manipulation.

***MongoDB CRUD Tutorials* (page 45)** Examples of basic query and data modification operations.

***MongoDB CRUD Reference* (page 89)** Reference material for the query and data manipulation interfaces.



---

## MongoDB CRUD Introduction

---

### On this page

- [Database Operations](#) (page 3)

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 <https://docs.mongodb.org/manual/core/document>.

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



← field: value  
← field: value  
← field: value  
← field: value

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

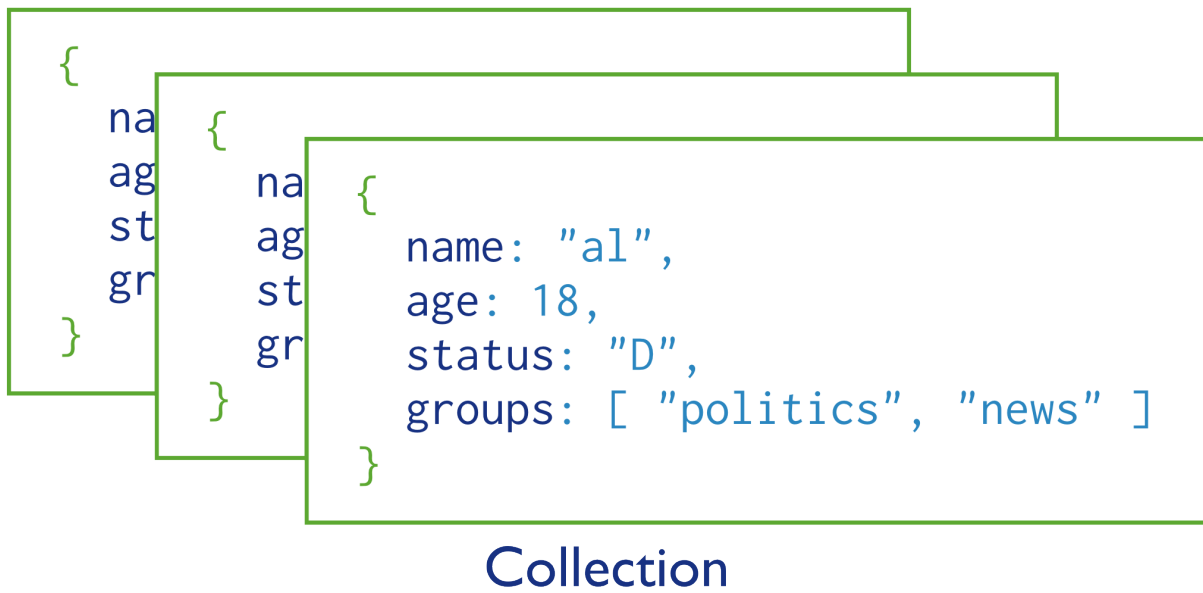
### 1.1.1 Query

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:

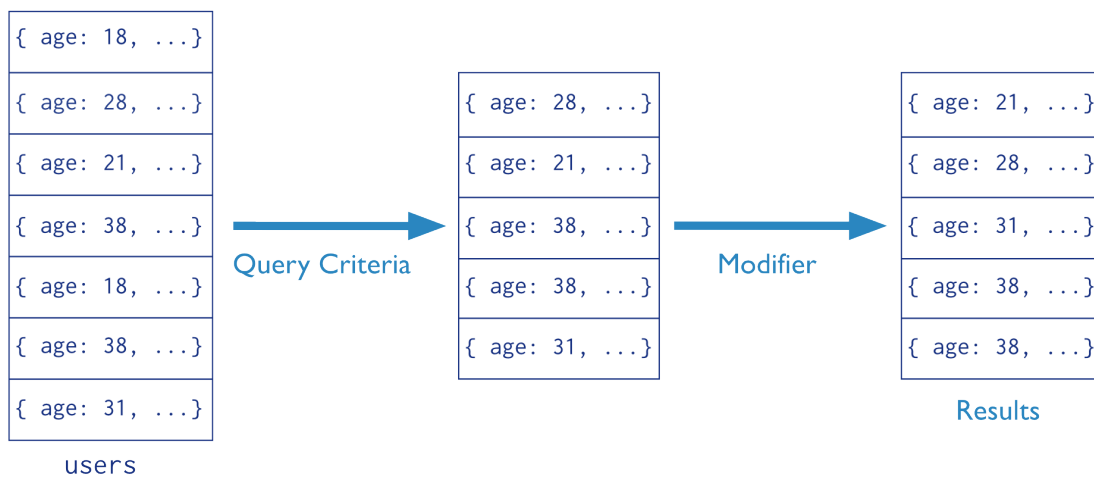
See [Read Operations Overview](#) (page 7) for more information.





Collection                      Query Criteria                      Modifier

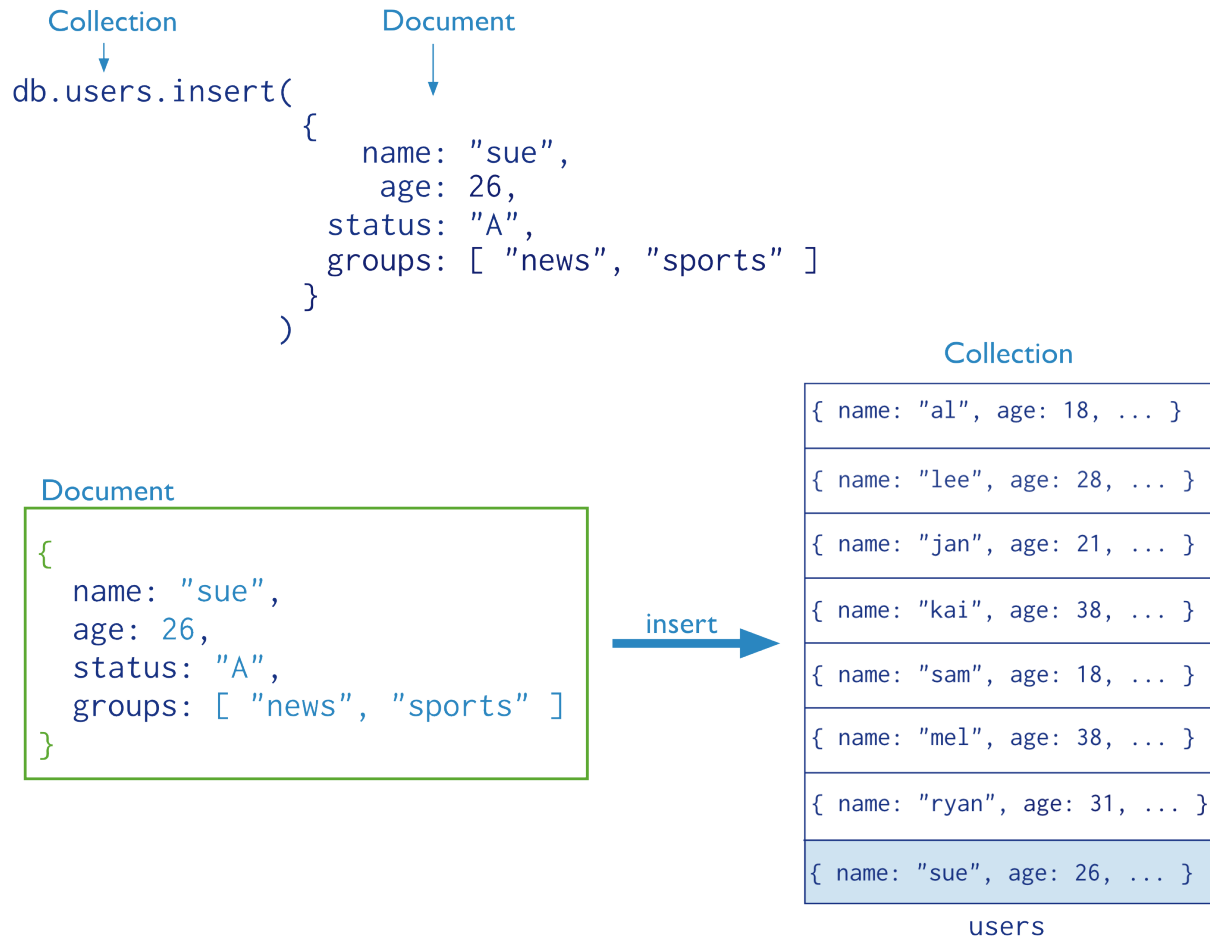
```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```



### 1.1.2 Data Modification

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

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



See *Write Operations Overview* (page 22) for more information.



---

## MongoDB CRUD Concepts

---

The *Read Operations* (page 7) and *Write Operations* (page 21) documents introduce the behavior and operations of read and write operations for MongoDB deployments.

***Read Operations* (page 7)** Queries are the core operations that return data in MongoDB. Introduces queries, their behavior, and performances.

***Cursors* (page 11)** Queries return iterable objects, called cursors, that hold the full result set.

***Query Optimization* (page 13)** Analyze and improve query performance.

***Distributed Queries* (page 16)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

***Write Operations* (page 21)** Write operations insert, update, or remove documents in MongoDB. Introduces data create and modify operations, their behavior, and performances.

***Atomicity and Transactions* (page 33)** Describes write operation atomicity in MongoDB.

***Distributed Write Operations* (page 34)** 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 21) 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 11)** Queries return iterable objects, called cursors, that hold the full result set.

***Query Optimization* (page 13)** Analyze and improve query performance.

***Query Plans* (page 15)** MongoDB executes queries using optimal *plans*.

***Distributed Queries* (page 16)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

### 2.1.1 Read Operations Overview

### On this page

- [Query Interface](#) (page 8)
- [Query Behavior](#) (page 9)
- [Query Statements](#) (page 9)
- [Projections](#) (page 10)

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

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

### Query Interface

For query operations, MongoDB provides a `db.collection.find()` method. The method accepts both the query criteria and projections and returns a *cursor* (page 11) 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:

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

← collection  
← query criteria  
← projection  
← cursor modifier

The next diagram shows the same query in SQL:

```
SELECT _id, name, address  
FROM   users  
WHERE  age > 18  
LIMIT 5
```

← projection  
← table  
← select criteria  
← cursor modifier

---

### 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 10) for details.

---

See

*SQL to MongoDB Mapping Chart* (page 94) for additional examples of MongoDB queries and the corresponding SQL statements.

## Query Behavior

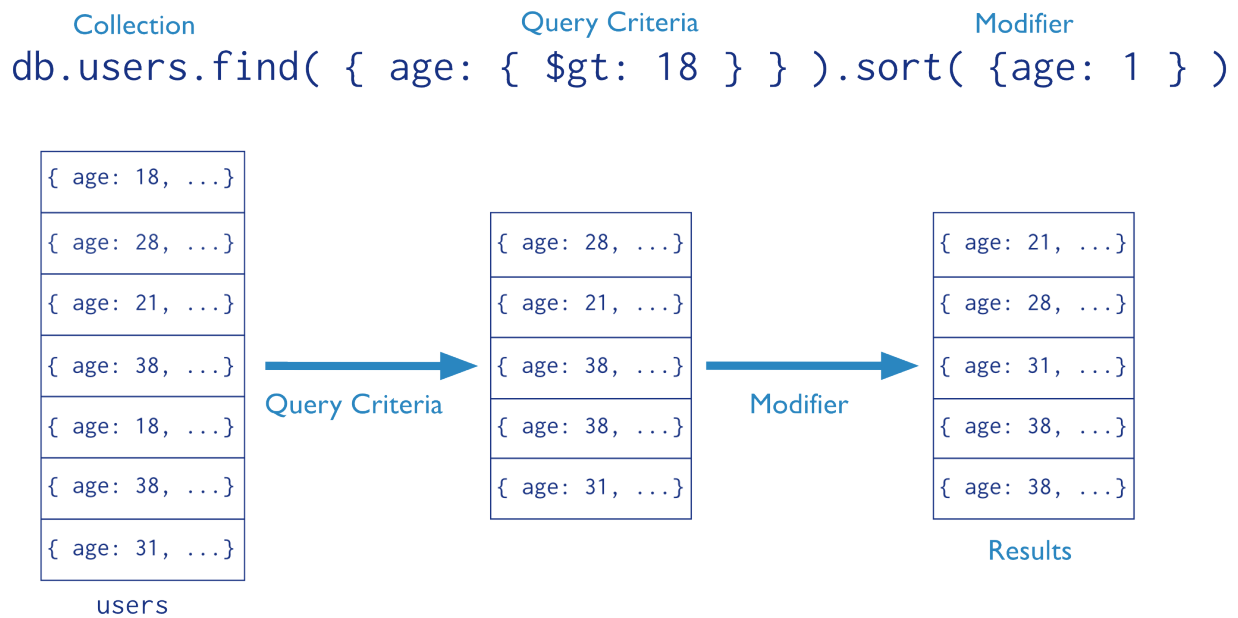
MongoDB queries exhibit the following behavior:

- 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 57) (i.e. *updates*) use the same query syntax as queries to select documents to update.
- In aggregation pipeline, the `$match` pipeline stage provides access to MongoDB queries.

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

## Query Statements

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



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

For additional examples of queries, see *Query Documents* (page 49).

## Projections

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

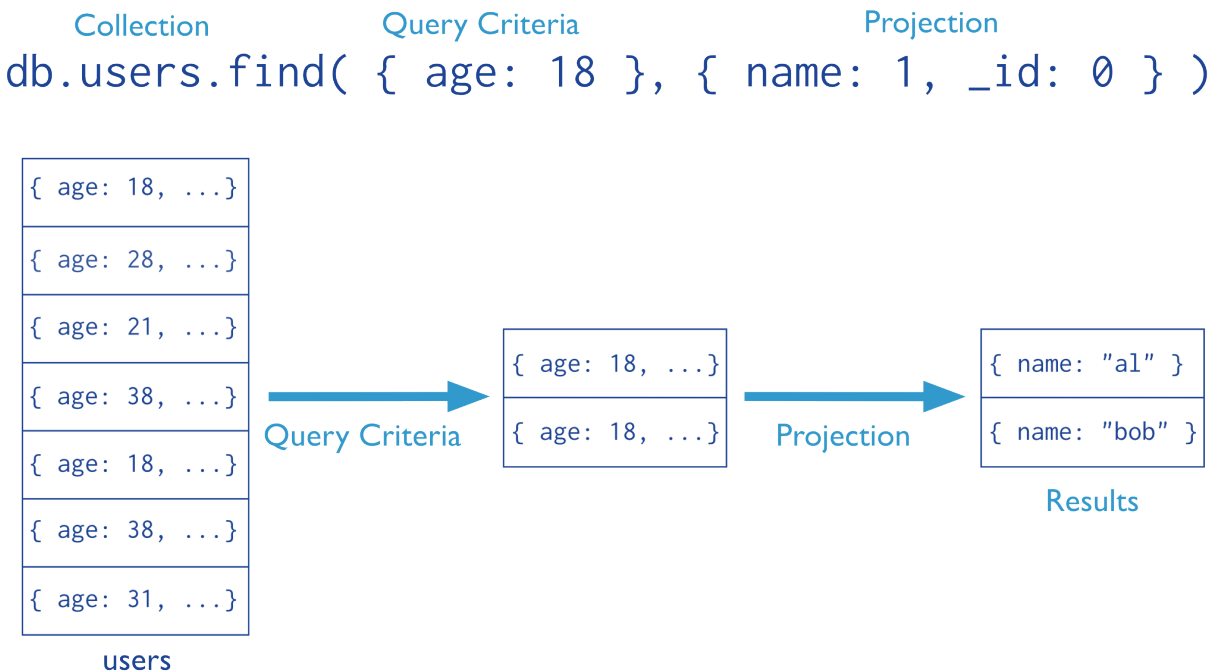
Projections, which are the *second* argument to the `find()` method, may either specify a list of fields to return *or* list fields to exclude in the result documents.

---

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

---

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



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

## Projection Examples

### Exclude One Field From a Result Set

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

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

### Return Two fields *and* the `_id` Field

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

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

### Return Two Fields and Exclude `_id`

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

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

### See

[Limit Fields to Return from a Query](#) (page 62) for more examples of queries with projection statements.

## Projection Behavior

MongoDB projections have the following properties:

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

## 2.1.2 Cursors

### On this page

- [Cursor Behaviors](#) (page 12)
- [Cursor Information](#) (page 12)

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<sup>1</sup> 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 67).

<sup>1</sup> 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.



### 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 in the mongo shell, you can use the `cursor.noCursorTimeout()` method:

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

After setting the `noCursorTimeout` option, you must either close the cursor manually with `cursor.close()` or by exhausting the cursor's results.

See your driver documentation for information on setting the `noCursorTimeout` option.

#### Cursor Isolation

As a cursor returns documents, other operations may interleave with the query. For the MMAPv1 storage engine, 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](#) (page 43).

#### Cursor Batches

The MongoDB server returns the query results in batches. Batch size will not exceed the *maximum BSON document size*. For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. To override the default size of the batch, see `batchSize()` and `limit()`.

For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort before returning any results.

As you iterate through the cursor and reach the end of the returned batch, if there are more results, `cursor.next()` will perform a `getmore` operation to retrieve the next batch. To see how many documents remain in the batch as you iterate the cursor, you can use the `objsLeftInBatch()` method, as in the following example:

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

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

myCursor.objsLeftInBatch();
```

#### Cursor Information

The `db.serverStatus()` method returns a document that includes a `metrics` field. The `metrics` field contains a `metrics.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()
```

### 2.1.3 Query Optimization

#### On this page

- [Create an Index to Support Read Operations](#) (page 13)
- [Query Selectivity](#) (page 14)
- [Covered Query](#) (page 14)

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

#### Create an Index to Support Read Operations

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

#### Example

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

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

To improve the performance of this query, add an ascending or a descending index to the `inventory` collection on the `type` field.<sup>2</sup> In the mongo shell, you can create indexes using the `db.collection.createIndex()` method:

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

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

To analyze the performance of the query with an index, see [Analyze Query Performance](#) (page 68).

<sup>2</sup> 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.

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

### Query Selectivity

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

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

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

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

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

### Covered Query

An index *covers* (page 14) a query when both of the following apply:

- all the fields in the *query* (page 49) are part of an index, **and**
- all the fields returned in the results are in the same index.

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

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

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

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

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

### Performance

Because the index contains all fields required by the query, MongoDB can both match the *query conditions* (page 49) and return the results using only the index.

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

### Limitations

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

- any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a *multi-key index* and cannot support a covered query.
- any of the indexed fields in the query predicate or returned in the projection are fields in embedded documents.<sup>3</sup> For example, consider a collection `users` with documents of the following form:

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

The collection has the following index:

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

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

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

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

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

Changed in version 3.0: In previous versions, an index cannot *cover* (page 14) a query on a *sharded* collection when run against a `mongos`.

### explain

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

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

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

## 2.1.4 Query Plans

### On this page

- [Plan Cache Flushes](#) (page 16)
- [Index Filters](#) (page 16)

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.

For each query, the query planner searches the query plan cache for an entry that fits the *query shape*. If there are no matching entries, the query planner generates candidate plans for evaluation over a trial period. The query planner chooses a winning plan, creates a cache entry containing the winning plan, and uses it to generate the result documents.

If a matching entry exists, the query planner generates a plan based on that entry and evaluates its performance through a *replanning* mechanism. This mechanism makes a *pass/fail* decision based on the plan performance

<sup>3</sup> To index fields in embedded documents, use *dot notation*.

and either keeps or evicts the cache entry. On eviction, the query planner selects a new plan using the normal planning process and caches it. The query planner executes the plan and returns the result documents for the query.

The following diagram illustrates the query planner logic:

See [Plan Cache Flushes](#) (page 16) for additional scenarios that trigger changes to the plan cache.

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

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

Changed in version 2.6: `explain()` operations no longer read from or write to the query planner cache.

### Plan Cache Flushes

Catalog operations like index or collection drops flush the plan cache.

The plan cache does not persist if a `mongod` restarts or shuts down.

New in version 2.6: MongoDB provides <https://docs.mongodb.org/manual/reference/method/js-plan-cache> to view and modify the cached query plans. The `PlanCache.clear()` method flushes the entire plan cache. Users can also clear particular plan cache entries using `PlanCache.clearPlansByQuery()`.

### Index Filters

New in version 2.6.

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

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

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

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

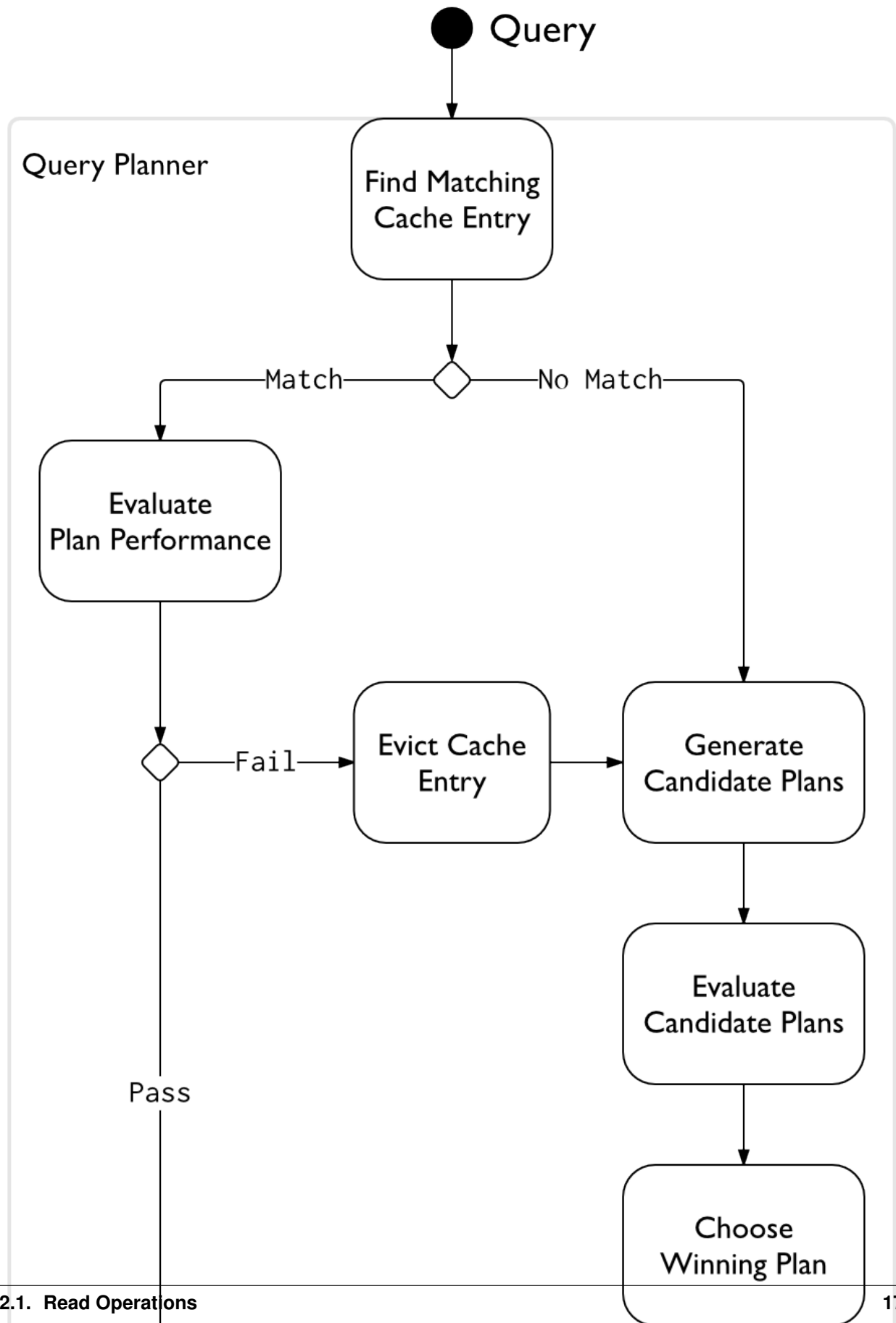
Because index filters overrides the expected behavior of the optimizer as well as the `hint()` method, use index filters sparingly.

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

## 2.1.5 Distributed Queries

### On this page

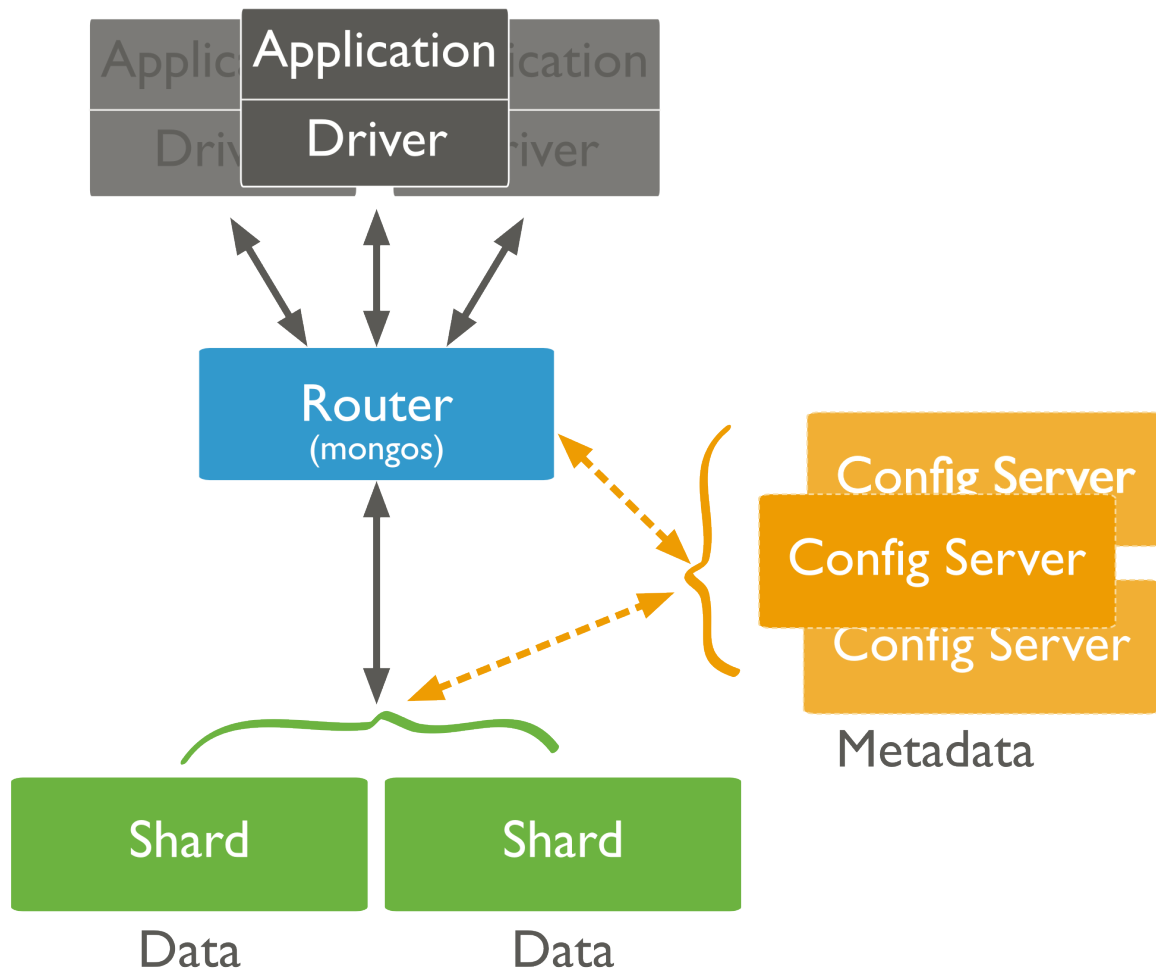
- [Read Operations to Sharded Clusters](#) (page 18)
- [Read Operations to Replica Sets](#) (page 21)



## 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 <https://docs.mongodb.org/manual/sharding> section of this manual.

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

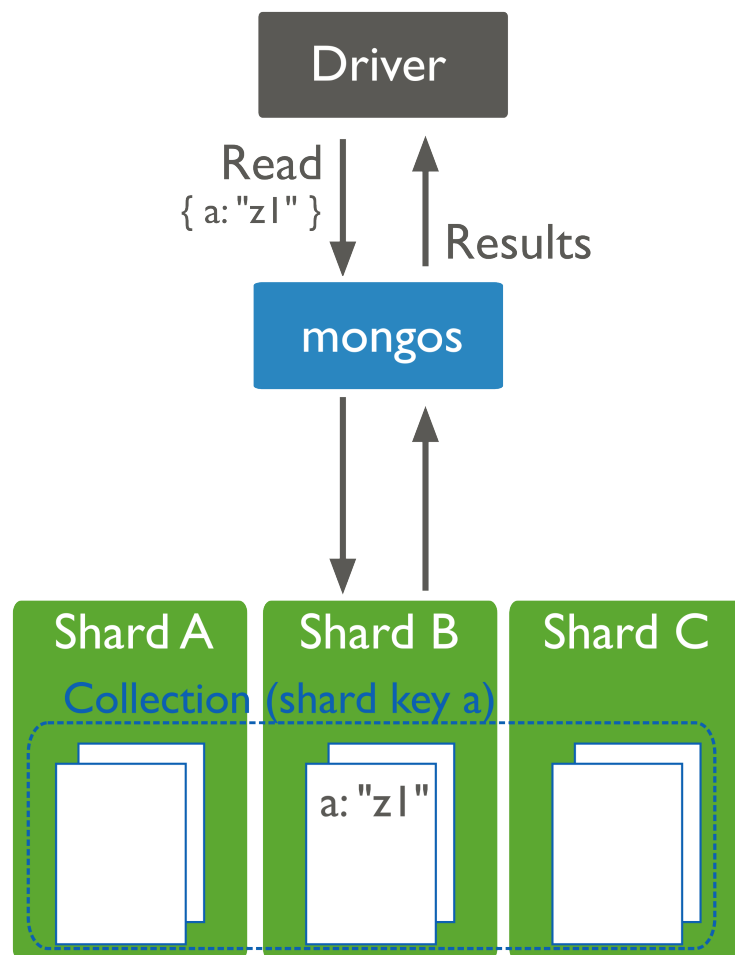


Read operations on sharded clusters are most efficient when directed to a specific shard. Queries to sharded collections should include the collection's *shard key*. When a query includes a shard key, the `mongos` can use cluster metadata from the *config database* to route the queries to shards.

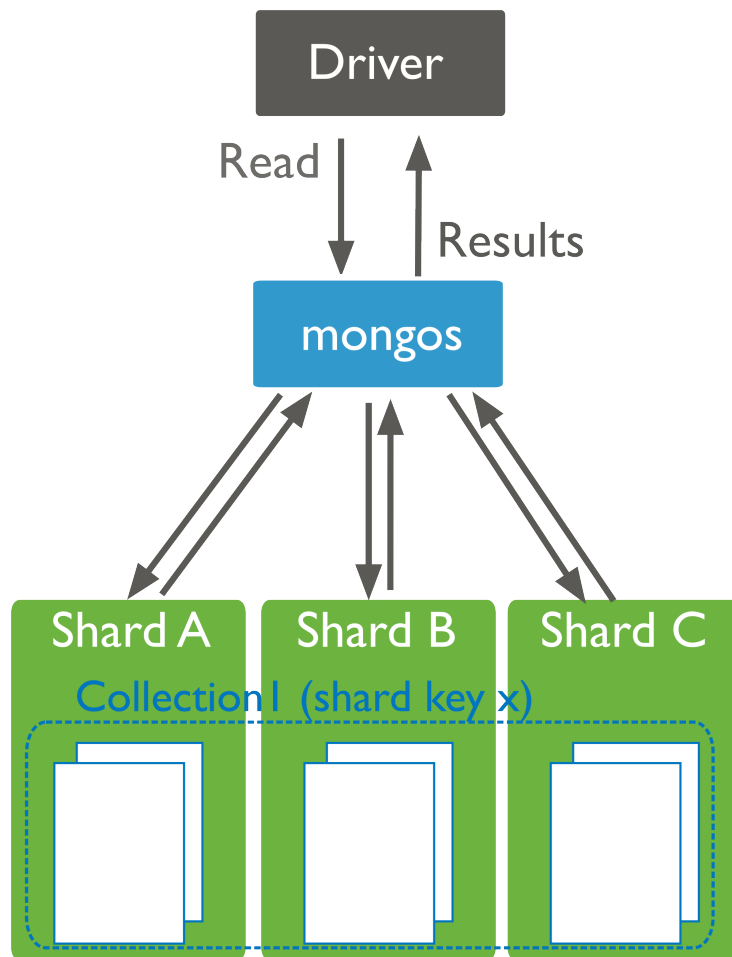
If a query does not include the shard key, the `mongos` must direct the query to *all* shards in the cluster. These *scatter gather* queries can be inefficient. On larger clusters, scatter gather queries are unfeasible for routine operations.

For replica set shards, read operations from secondary members of replica sets may not reflect the current state of the primary. Read preferences that direct read operations to different servers may result in non-monotonic reads.

For more information on read operations in sharded clusters, see the <https://docs.mongodb.org/manual/core/sharded-cluster/> and *sharding-shard-key* sections.



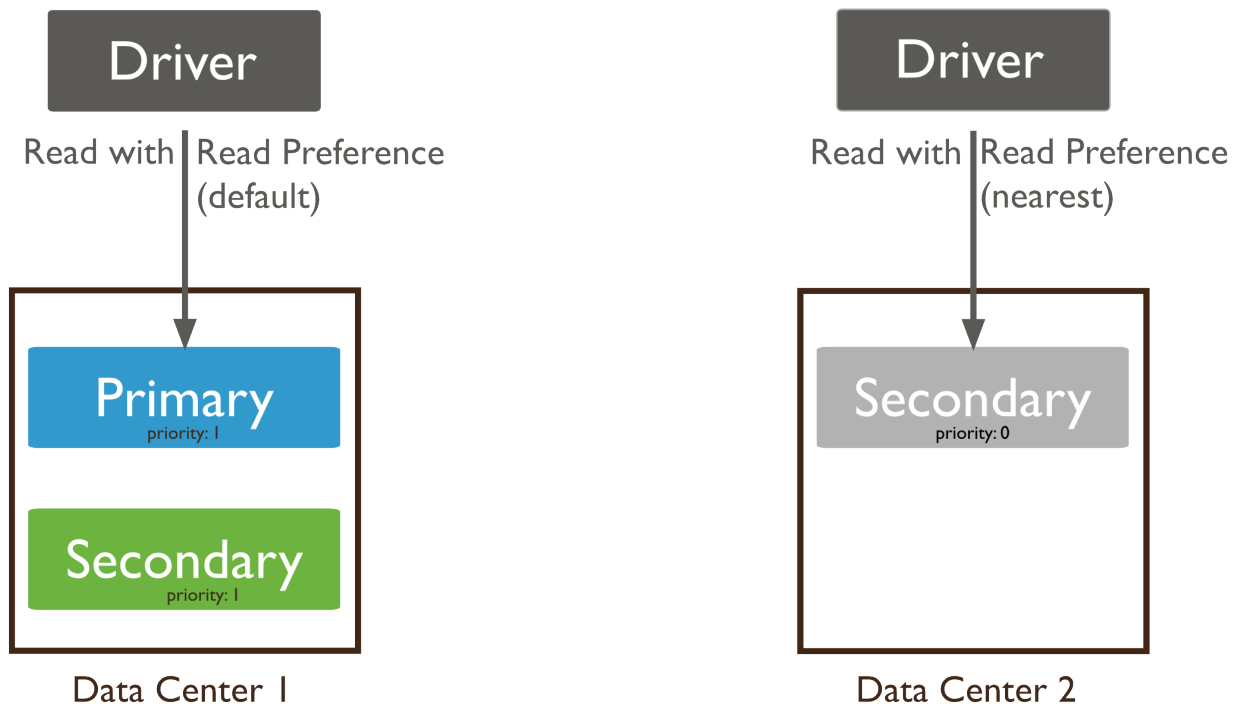




## Read Operations to Replica Sets

By default, clients reads from a replica set's *primary*; however, clients can specify a `read preference` to direct read operations to other members. For example, clients can configure read preferences to read from secondaries or from nearest member to:

- reduce latency in multi-data-center deployments,
- improve read throughput by distributing high read-volumes (relative to write volume),
- perform backup operations, and/or
- allow reads until a *new primary is elected*.



Read operations from secondary members of replica sets may not reflect the current state of the primary. Read preferences that direct read operations to different servers may result in non-monotonic reads.

You can configure the read preference on a per-connection or per-operation basis. For more information on read preference or on the read preference modes, see <https://docs.mongodb.org/manual/core/read-preference> and *replica-set-read-preference-modes*.

## 2.2 Write Operations

The following documents describe write operations:

***Write Operations Overview* (page 22)** Provides an overview of MongoDB's data insertion and modification operations, including aspects of the syntax, and behavior.

***Atomicity and Transactions* (page 33)** Describes write operation atomicity in MongoDB.

***Distributed Write Operations* (page 34)** Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

***Write Operation Performance* (page 36)** Introduces the performance constraints and factors for writing data to MongoDB deployments.

***Bulk Write Operations* (page 38)** Provides an overview of MongoDB's bulk write operations.

### 2.2.1 Write Operations Overview

#### On this page

- [Insert](#) (page 22)
- [Update](#) (page 26)
- [Delete](#) (page 30)
- [Additional Methods](#) (page 32)

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

There are three classes of write operations in MongoDB: *insert* (page 22), *update* (page 26), and *delete* (page 30). Insert operations add new documents to a collection. Update operations modify existing documents, and delete operations delete documents from a collection. No insert, update, or delete can affect more than one document atomically.

For the update and remove operations, you can specify criteria, or filters, that identify the documents to update or remove. These operations use the same query syntax to specify the criteria as *read operations* (page 7).

MongoDB allows applications to determine the acceptable level of acknowledgement required of write operations. See *Write Concern* (page 90) for more information.

#### Insert

MongoDB provides the following methods for inserting documents into a collection:

- `db.collection.insertOne()`
- `db.collection.insertMany()`
- `db.collection.insert()`

#### insertOne

New in version 3.2.

`db.collection.insertOne()` inserts a *single* document

The following diagram highlights the components of the MongoDB `insertOne()` operation:

The following diagram shows the same query in SQL:

---

#### Example

The following operation inserts a new document into the `users` collection. The new document has three fields `name`, `age`, and `status`. Since the document does not specify an `_id` field, MongoDB adds the `_id` field and a generated value to the new document. See *Insert Behavior* (page 26).

```

db.users.insertOne(  ← collection
{
  name: "sue",        ← field: value
  age: 26,            ← field: value
  status: "pending"   ← field: value
}                    } document
)

```

```

INSERT INTO users      ← table
      ( name, age, status ) ← columns
VALUES      ( "sue", 26, "pending" ) ← values/row

```

```

db.users.insertOne(
{
  name: "sue",
  age: 26,
  status: "pending"
}
)

```

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

### insertMany

New in version 3.2.

`db.collection.insertMany()` inserts *multiple* documents

The following diagram highlights the components of the MongoDB `insertMany()` operation:

The following diagram shows the same query in SQL:

#### Example

The following operation inserts three new documents into the `users` collection. Each document has three fields `name`, `age`, and `status`. Since the documents do not specify an `_id` field, MongoDB adds the `_id` field and a generated value to each document. See *Insert Behavior* (page 26).

```

db.users.insertMany(
[
  { name: "sue", age: 26, status: "pending" },
  { name: "bob", age: 25, status: "enrolled" },
  { name: "ann", age: 28, status: "enrolled" }
]
)

```

```
db.users.insertMany(      ← collection
[
  {
    name: "sue",           ← field: value
    age: 26,               ← field: value
    status: "pending"      ← field: value } document
  },
  {
    name: "bob",           ← field: value
    age: 25,               ← field: value
    status: "enrolled"     ← field: value } document
  },
  {
    name: "ann",           ← field: value
    age: 28,               ← field: value
    status: "enrolled"     ← field: value } document
  }
]
)
```

```
INSERT INTO user          ← table
      ( name, age, status) ← columns
VALUES ( "sue", 26, "pending" ),
      ( "bob", 25, "enrolled" ),
      ( "ann", 28, "enrolled" ) } values/row
```

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

## insert

In MongoDB, the `db.collection.insert()` method adds new *documents* to a collection. It can take either a single document or an array of documents to insert.

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

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

The following diagram shows the same query in SQL:

```
INSERT INTO users
( name, age, status )
VALUES
( "sue", 26, "A" )
```

## Example

The following operation inserts a new document into the `users` collection. The new document has three fields `name`, `age`, and `status`. Since the document does not specify an `_id` field, MongoDB adds the `_id` field and a generated value to the new document. See *Insert Behavior* (page 26).

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

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

### Insert Behavior

The `_id` field is required in every MongoDB *document*. The `_id` field is like the document's *primary key*.

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 pass in an `_id` value that already exists, an exception is thrown.

The `_id` field is *uniquely indexed* by default in every collection.

### Other Methods to Add Documents

The `updateOne()`, `updateMany()`, and `replaceOne()` operations accept the `upsert` parameter. When `upsert : true`, if no document in the collection matches the filter, a new document is created based on the information passed to the operation. See *Update Behavior with the upsert Option* (page 30).

### Update

MongoDB provides the following methods for updating documents in a collection:

- `db.collection.updateOne()`
- `db.collection.updateMany()`
- `db.collection.replaceOne()`
- `db.collection.update()`

#### updateOne

New in version 3.2.

`db.collection.updateOne()` updates a *single* document.

The following diagram highlights the components of the MongoDB `updateOne()` operation:

```
db.users.updateOne(           ← collection
  { age : { $lt : 18 } } ,    ← update filter
  { $set: { status : "reject" } ← update action
)
```

The following diagram shows the same query in SQL:

```
UPDATE users                ← table
SET   status = 'reject'    ← update action
WHERE age < 18              ← update filter
LIMIT 1                    ← update limit
```

**Example**

This update operation on the `users` collection sets the `status` field to `reject` for the *first* document that matches the filter of age less than 18. See *Update Behavior* (page 29).

```
db.users.updateOne(
  { age: { $lt: 18 } },
  { $set: { status: "reject" } }
)
```

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

**updateMany**

New in version 3.2.

`db.collection.updateMany()` updates *multiple* documents.

The following diagram highlights the components of the MongoDB `updateMany()` operation:

```
db.users.updateMany(
  { age: { $lt: 18 } },
  { $set: { status: "reject" } }
)
```

The following diagram shows the same query in SQL:

```
UPDATE users
SET    status = 'reject'
WHERE  age < 18
```

**Example**

This update operation on the `users` collection sets the `status` field to `reject` for *all* documents that match the filter of age less than 18. See *Update Behavior* (page 29).

```
db.users.updateMany(
  { age: { $lt: 18 } },
  { $set: { status: "reject" } }
)
```

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

**replaceOne**

New in version 3.2.



`db.collection.replaceOne()` replaces a *single* document.

The following diagram highlights the components of the MongoDB `replaceOne()` operation:

The diagram shows the MongoDB `replaceOne()` operation with green arrows pointing to its components: `collection` points to `db.users`, `replace filter` points to `{ name: "sue" }`, and `replacement document` points to the document `{ name: "amy", age: 25, status: "enrolled" }`.

```
db.users.replaceOne(  
  { name: "sue" } ,  
  {  
    name: "amy",  
    age: 25,  
    status: "enrolled"  
  }  
)
```

The following diagram shows the same query in SQL:

The diagram shows the SQL `UPDATE` statement with green arrows pointing to its components: `table` points to `users`, `update action` points to `name = 'amy'`, `update action` points to `age = '25'`, `update action` points to `status = 'enrolled'`, `update filter` points to `WHERE name = 'sue'`, and `update limit` points to `LIMIT 1`.

```
UPDATE users  
SET   name = 'amy'  
      age = '25'  
      status = 'enrolled'  
WHERE name = 'sue'  
LIMIT 1
```

---

### Example

This replace operation on the `users` collection replaces the *first* document that matches the filter of name is sue with a new document. See *Replace Behavior* (page 30).

```
db.users.replaceOne(  
  { name: "sue" },  
  { name: "amy", age : 25, score: "enrolled" }  
)
```

---

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

### update

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

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

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

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

The following diagram shows the same query in SQL:

```
UPDATE users
SET      status = 'A'
WHERE   age > 18
```

### Example

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

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

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

### Update Behavior

`updateOne()` and `updateMany()` use <https://docs.mongodb.org/manual/reference/operator/update/> such as `$set`, `$unset`, or `$rename` to modify existing documents.

`updateOne()` will update the *first* document that is returned by the filter. `db.collection.findOneAndUpdate()` offers sorting of the filter results, allowing a degree of control over which document is updated.

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

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

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

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

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

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

### Replace Behavior

`replaceOne()` cannot use <https://docs.mongodb.org/manual/reference/operator/update/> in the replacement document. The replacement document must consist of only `<field> : <value>` assignments.

`replaceOne()` will replace the *first* document that matches the filter. `db.collection.findOneAndReplace()` offers sorting of the filter results, allowing a degree of control over which document is replaced.

You cannot replace the `_id` field.

### Update Behavior with the `upsert` Option

If `update()`, `updateOne()`, `updateMany()`, or `replaceOne()` include `upsert : true` **and** no documents match the filter portion of the operation, then the operation creates a new document and inserts it. If there are matching documents, then the operation modifies the matching document or documents.

### Delete

MongoDB provides the following methods for deleting documents from a collection:

- `db.collection.deleteOne()`
- `db.collection.deleteMany()`
- `db.collection.remove()`

#### `deleteOne`

New in version 3.2.

`db.collection.deleteOne()` deletes a *single* document.

The following diagram highlights the components of the MongoDB `deleteOne()` operation:

```
db.users.deleteOne(  ← collection
    { status: "Rejected" } ← delete filter
)
```

The following diagram shows the same query in SQL:

---

#### Example

DELETE FROM	users	←	table
WHERE	status = 'reject'	←	delete filter
LIMIT	1	←	delete limit

This delete operation on the `users` collection deletes the *first* document where name is sue. See *Delete Behavior* (page 32).

```
db.users.deleteOne(
  { status: "reject" }
)
```

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

### deleteMany

New in version 3.2.

`db.collection.deleteMany()` deletes *multiple* documents.

The following diagram highlights the components of the MongoDB `deleteMany()` operation:

db.users.deleteMany(	←	collection
{ status: "Rejected" }	←	delete filter
)		

The following diagram shows the same query in SQL:

DELETE FROM	users	←	table
WHERE	status = 'reject'	←	delete filter

### Example

This delete operation on the `users` collection deletes *all* documents where status is reject. See *Delete Behavior* (page 32).

```
db.users.deleteMany(
  { status: "reject" }
)
```

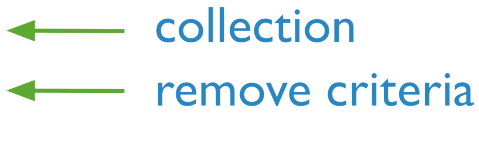
For more information and examples, see `db.collection.deleteMany()`.

### remove

In MongoDB, the `db.collection.remove()` method deletes documents from a collection. The `db.collection.remove()` method accepts query criteria to determine which documents to remove as well as an options document that affects its behavior, such as the `justOne` option to remove only a single document.

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

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



A diagram with two green arrows pointing left. The top arrow points from the text 'collection' to the `db.users` part of the code. The bottom arrow points from the text 'remove criteria' to the `{ status: "D" }` part of the code.

The following diagram shows the same query in SQL:

```
DELETE FROM users  
WHERE status = 'D'
```



A diagram with two green arrows pointing left. The top arrow points from the text 'table' to the `users` part of the SQL query. The bottom arrow points from the text 'delete criteria' to the `status = 'D'` part of the SQL query.

---

### 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 61).

### Delete Behavior

`deleteOne()` will delete the *first* document that matches the filter. `db.collection.findOneAndDelete()` offers sorting of the filter results, allowing a degree of control over which document is deleted.

### Remove Behavior

By default, `db.collection.remove()` method removes all documents that match its query. If the optional `justOne` parameter is set to `true`, `remove()` will limit the delete operation to a single document.

### Additional Methods

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

## Bulk Write

MongoDB provides the `db.collection.bulkWrite()` method for executing multiple write operations in a group. Each write operation is still atomic on the level of a single *document*.

### Example

The following `bulkWrite()` inserts several documents, performs an update, and then deletes several documents.

```

db.collection.bulkWrite(
  [
    { insertOne : { "document" : { name : "sue", age : 26 } } },
    { insertOne : { "document" : { name : "joe", age : 24 } } },
    { insertOne : { "document" : { name : "ann", age : 25 } } },
    { insertOne : { "document" : { name : "bob", age : 27 } } },
    { updateMany: {
      "filter" : { age : { $gt : 25 } },
      "update" : { $set : { "status" : "enrolled" } }
    } },
    { deleteMany : { "filter" : { "status" : { $exists : true } } } }
  ]
)

```

## 2.2.2 Atomicity and Transactions

### On this page

- [\\$isolated Operator](#) (page 33)
- [Transaction-Like Semantics](#) (page 34)
- [Concurrency Control](#) (page 34)

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

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

### \$isolated Operator

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

`$isolated` does **not** work with *sharded clusters*.

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

**Note:** `$isolated` operator causes write operations to acquire an exclusive lock on the collection, *even for document-level locking storage engines* such as WiredTiger. That is, `$isolated` operator will make WiredTiger single-threaded for the duration of the operation.

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

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

### Transaction-Like Semantics

Since a single document can contain multiple embedded documents, single-document atomicity is sufficient for many practical use cases. For cases where a sequence of write operations must operate as if in a single transaction, you can implement a *two-phase commit* (page 73) in your application.

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

For more information on two-phase commit and rollback, see *Perform Two Phase Commits* (page 73).

### Concurrency Control

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

One approach is to create a *unique index* on a field that can only have unique values. This prevents insertions or updates from creating duplicate data. Create a unique index on multiple fields to force uniqueness on that combination of field values. For examples of use cases, see *update()* and *Unique Index* and *findAndModify()* and *Unique Index*.

Another approach is to specify the expected current value of a field in the query predicate for the write operations. For an example, see *Update if Current* (page 80).

The two-phase commit pattern provides a variation where the query predicate includes the *application identifier* (page 78) as well as the expected state of the data in the write operation.

**See also:**

*Read Isolation, Consistency, and Recency* (page 41)

## 2.2.3 Distributed Write Operations

### On this page

- [Write Operations on Sharded Clusters](#) (page 34)
- [Write Operations on Replica Sets](#) (page 36)

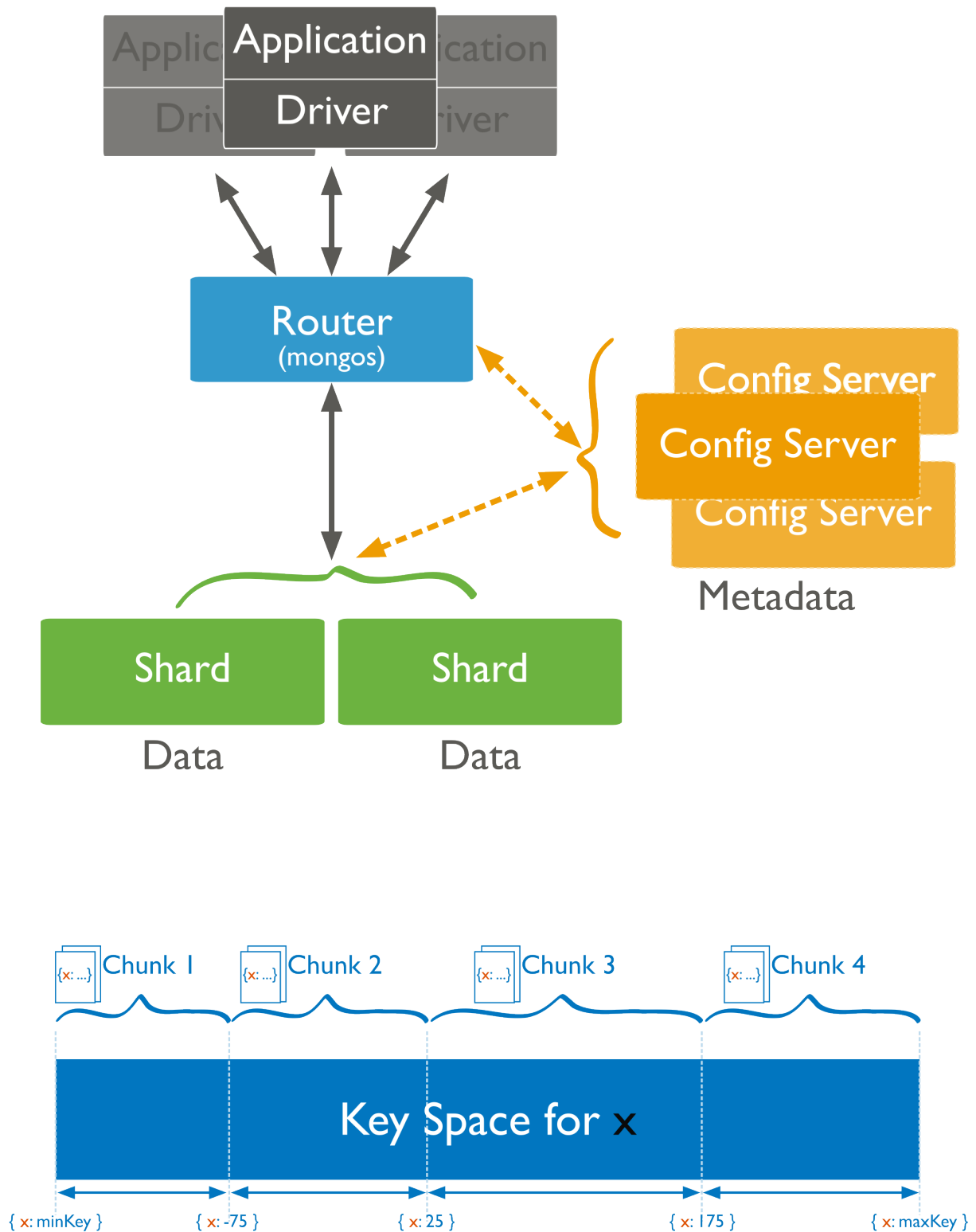
### Write Operations on Sharded Clusters

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.

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

---

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



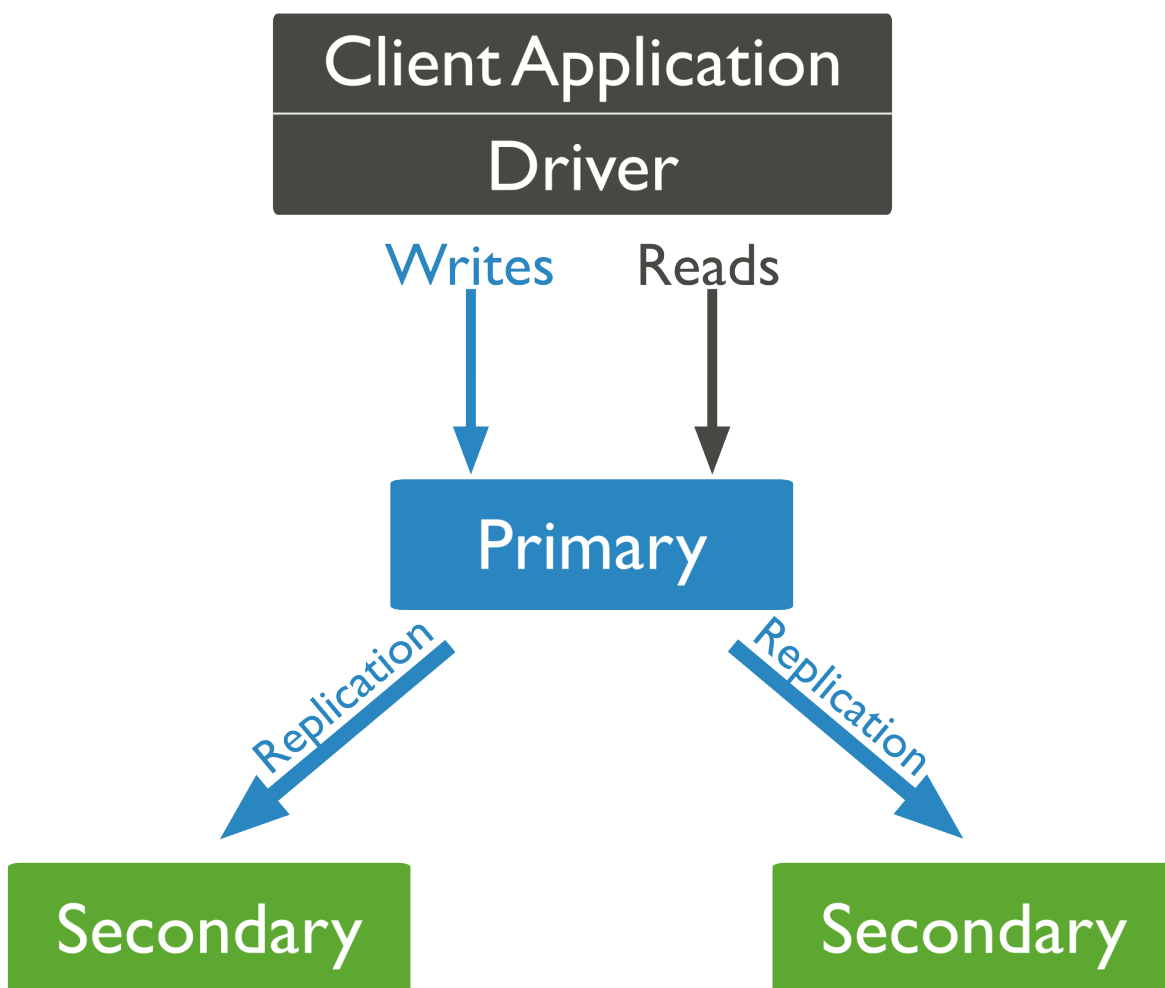


If the value of the shard key increases or decreases with every insert, all insert operations target a single shard. As a result, the capacity of a single shard becomes the limit for the insert capacity of the sharded cluster.

For more information, see <https://docs.mongodb.org/manual/administration/sharded-clusters> and *Bulk Write Operations* (page 38).

### Write Operations on Replica Sets

In *replica sets*, all write operations go to the set's *primary*. The primary applies the write operation and 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 continuously replicate the oplog and apply the operations to themselves in an asynchronous process.



For more information on replica sets and write operations, see <https://docs.mongodb.org/manual/core/replication-1> and *Write Concern* (page 90).

## 2.2.4 Write Operation Performance

**On this page**

- [Indexes](#) (page 37)
- [Document Growth and the MMAPv1 Storage Engine](#) (page 37)
- [Storage Performance](#) (page 37)
- [Additional Resources](#) (page 38)

**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.<sup>4</sup>

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 13). For more information on indexes, see <https://docs.mongodb.org/manual/indexes> and <https://docs.mongodb.org/manual/applications/indexes>.

**Document Growth and the MMAPv1 Storage Engine**

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

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

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

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

See <https://docs.mongodb.org/manual/core/mmapv1> 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.

<sup>4</sup> 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.

### See

<https://docs.mongodb.org/manual/administration/production-notes> for recommendations regarding additional hardware and configuration options.

---

## Journaling

To provide durability in the event of a crash, MongoDB uses *write ahead logging* to an on-disk *journal*. MongoDB writes the in-memory changes first to the on-disk journal files. If MongoDB should terminate or encounter an error before committing the changes to the data files, MongoDB can use the journal files to apply the write operation to the data files.

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

- If the journal and the data file reside on the same block device, the data files and the journal may have to contend for a finite number of available I/O resources. Moving the journal to a separate device may increase the capacity for write operations.
- If applications specify *write concerns* (page 90) that include the `j` option (page 92), `mongod` will decrease the duration between journal writes, which can increase the overall write load.
- The duration between journal writes 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 journal 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 <https://docs.mongodb.org/manual/core/journaling>.

## Additional Resources

- [MongoDB Performance Evaluation and Tuning Consulting Package<sup>5</sup>](#)

## 2.2.5 Bulk Write Operations

### On this page

- [Overview](#) (page 38)
- [Ordered vs Unordered Operations](#) (page 39)
- [bulkWrite\(\) Methods](#) (page 39)
- [Strategies for Bulk Inserts to a Sharded Collection](#) (page 40)

## Overview

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

New in version 3.2.

---

<sup>5</sup>[https://www.mongodb.com/products/consulting?jmp=docs#performance\\_evaluation](https://www.mongodb.com/products/consulting?jmp=docs#performance_evaluation)

The `db.collection.bulkWrite()` method provides the ability to perform bulk insert, update, and remove operations. MongoDB also supports bulk insert through the `db.collection.insertMany()`.

## Ordered vs Unordered Operations

Bulk write operations can be either *ordered* or *unordered*.

With an ordered list of operations, MongoDB executes the operations serially. If an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list. See *ordered Bulk Write*

With an unordered list of operations, MongoDB can execute the operations in parallel, but this behavior is not guaranteed. If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list. See *bulkwrite-example-unordered-bulk-write*.

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

By default, `bulkWrite()` performs *ordered* operations. To specify *unordered* write operations, set `ordered : false` in the options document.

See *bulkwrite-write-operations-executionofoperations*

## bulkWrite() Methods

`bulkWrite()` supports the following write operations:

- *bulkwrite-write-operations-insertOne*
- *updateOne*
- *updateMany*
- *bulkwrite-write-operations-replaceOne*
- *deleteOne*
- *deleteMany*

Each write operation is passed to `bulkWrite()` as a document in an array.

For example, the following performs multiple write operations:

The `characters` collection contains the following documents:

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

The following `bulkWrite()` performs multiple operations on the collection:

```
try {
  db.characters.bulkWrite(
    [
      { insertOne :
        {
          "document" :
          {
            "_id" : 4, "char" : "Dithras", "class" : "barbarian", "lvl" : 4
          }
        }
      }
    ]
  )
}
```

```
    },
    { insertOne :
      {
        "document" :
          {
            "_id" : 5, "char" : "Taeln", "class" : "fighter", "lvl" : 3
          }
      }
    },
    { updateOne :
      {
        "filter" : { "char" : "Eldon" },
        "update" : { $set : { "status" : "Critical Injury" } }
      }
    },
    { deleteOne :
      { "filter" : { "char" : "Brisbane" } }
    },
    { replaceOne :
      {
        "filter" : { "char" : "Meldane" },
        "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl" : 4 }
      }
    }
  ]
);
}
catch (e) {
  print(e);
}
```

The operation returns the following:

```
{
  "acknowledged" : true,
  "deletedCount" : 1,
  "insertedCount" : 2,
  "matchedCount" : 2,
  "upsertedCount" : 0,
  "insertedIds" : {
    "0" : 4,
    "1" : 5
  },
  "upsertedIds" : {
  }
}
```

For more examples, see *bulkWrite() Examples*

## Strategies for Bulk Inserts to a Sharded Collection

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

## Pre-Split the Collection

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

## Unordered Writes to mongos

To improve write performance to sharded clusters, use `bulkWrite()` with the optional parameter `ordered` set to `false`. `mongos` can attempt to send the writes to multiple shards simultaneously. For *empty* collections, first pre-split the collection as described in <https://docs.mongodb.org/manual/tutorial/split-chunks-in-sharded-cluster>.

## Avoid Monotonic Throttling

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

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

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

## Example

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

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

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

## See also:

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

## 2.3 Read Isolation, Consistency, and Recency

**On this page**

- [Isolation Guarantees](#) (page 42)
- [Consistency Guarantees](#) (page 43)
- [Recency](#) (page 44)

## 2.3.1 Isolation Guarantees

### Read Uncommitted

In MongoDB, clients can see the results of writes before the writes are *durable*:

- Regardless of *write concern* (page 90), other clients using `"local"` (page 93) (i.e. the default) `readConcern` can see the result of a write operation before the write operation is acknowledged to the issuing client.
- Clients using `"local"` (page 93) (i.e. the default) `readConcern` can read data which may be subsequently rolled back.

Read uncommitted is the default isolation level and applies to `mongod` standalone instances as well as to replica sets and sharded clusters.

### Read Uncommitted And Single Document Atomicity

Write operations are atomic with respect to a single document; i.e. if a write is updating multiple fields in the document, a reader will never see the document with only some of the fields updated.

With a standalone `mongod` instance, a set of read and write operations to a single document is serializable. With a replica set, a set of read and write operations to a single document is serializable *only* in the absence of a rollback.

However, although the readers may not see a *partially* updated document, read uncommitted means that concurrent readers may still see the updated document before the changes are *durable*.

### Read Uncommitted And Multiple Document Write

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

Without isolating the multi-document write operations, MongoDB exhibits the following behavior:

1. Non-point-in-time read operations. Suppose a read operation begins at time  $t_1$  and starts reading documents. A write operation then commits an update to one of the documents at some later time  $t_2$ . The reader may see the updated version of the document, and therefore does not see a point-in-time snapshot of the data.
2. Non-serializable operations. Suppose a read operation reads a document  $d_1$  at time  $t_1$  and a write operation updates  $d_1$  at some later time  $t_3$ . This introduces a read-write dependency such that, if the operations were to be serialized, the read operation must precede the write operation. But also suppose that the write operation updates document  $d_2$  at time  $t_2$  and the read operation subsequently reads  $d_2$  at some later time  $t_4$ . This introduces a write-read dependency which would instead require the read operation to come *after* the write operation in a serializable schedule. There is a dependency cycle which makes serializability impossible.
3. Reads may miss matching documents that are updated during the course of the read operation.

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

`$isolated` does **not** work with *sharded clusters*.

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

---

**Note:** `$isolated` operator causes write operations to acquire an exclusive lock on the collection, *even for document-level locking storage engines* such as WiredTiger. That is, `$isolated` operator will make WiredTiger single-threaded for the duration of the operation.

---

**See also:**

[Atomicity and Transactions](#) (page 33)

## Cursor Snapshot

MongoDB cursors can return the same document more than once in some situations. As a cursor returns documents other operations may interleave with the query. If some of these operations are *updates* (page 21) that cause the document to move (in the case of MMAPv1, caused by document growth) or that change the indexed field on the index used by the query; then the cursor will return the same document more than once.

In very specific cases, you can isolate the cursor from returning the same document more than once by using the `cursor.snapshot()` method. `snapshot()` guarantees that the query will return each document no more than once.

**Warning:**

- The `snapshot()` does not guarantee that the data returned by the query will reflect a single moment in time *nor* does it provide isolation from insert or delete operations.
- You **cannot** use `snapshot()` with *sharded collections*.
- You **cannot** use `snapshot()` with the `sort()` or `hint()` cursor methods.

As an alternative, if your collection has a field or fields that are never modified, you can use a *unique* index on this field or these fields to achieve a similar result as the `snapshot()`. Query with `hint()` to explicitly force the query to use that index.

## 2.3.2 Consistency Guarantees

### Monotonic Reads

MongoDB provides monotonic reads from a standalone `mongod` instance. Suppose an application performs a sequence of operations that consists of a read operation  $R_1$  followed later in the sequence by another read operation  $R_2$ . If the application performs the sequence on a standalone `mongod` instance, the later read  $R_2$  never returns results that reflect an earlier state than that returned from  $R_1$ ; i.e.  $R_2$  returns data that is monotonically increasing in recency from  $R_1$ .

Changed in version 3.2: For replica sets and sharded clusters, MongoDB provides monotonic reads if read operations specify *Read Concern* (page 92) "majority" and read preference `primary`.

In previous versions, MongoDB cannot make monotonic read guarantees from replica sets and sharded clusters.



### Monotonic Writes

MongoDB provides monotonic write guarantees for standalone `mongod` instances, replica sets, and sharded clusters.

Suppose an application performs a sequence of operations that consists of a write operation  $W_1$  followed later in the sequence by a write operation  $W_2$ . MongoDB guarantees that  $W_1$  operation precedes  $W_2$ .

### 2.3.3 Recency

In MongoDB, in a replica set with one primary member <sup>6</sup>,

- With `"local"` (page 93) `readConcern`, reads from the primary reflect the latest writes in absence of a failover;
- With `"majority"` (page 93) `readConcern`, read operations from the primary or the secondaries have *eventual consistency*.

---

<sup>6</sup> In *some circumstances*, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with `{ w: "majority" }` (page 91) write concern. The node that can complete `{ w: "majority" }` (page 91) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary`, and new writes to the former primary will eventually roll back.

---

## 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 1).

***Insert Documents (page 45)*** Insert new documents into a collection.

***Query Documents (page 49)*** Find documents in a collection using search criteria.

***Modify Documents (page 57)*** Modify documents in a collection

***Remove Documents (page 61)*** Remove documents from a collection.

***Limit Fields to Return from a Query (page 62)*** Limit which fields are returned by a query.

***Limit Number of Elements in an Array after an Update (page 65)*** Use `$push` with modifiers to sort and maintain an array of fixed size.

***Iterate a Cursor in the mongo Shell (page 67)*** Access documents returned by a `find` query by iterating the cursor, either manually or using the iterator index.

***Analyze Query Performance (page 68)*** Use query introspection (i.e. `explain`) to analyze the efficiency of queries and determine how a query uses available indexes.

***Perform Two Phase Commits (page 73)*** Use two-phase commits when writing data to multiple documents.

***Update Document if Current (page 80)*** Update a document only if it has not changed since it was last read.

***Create Tailable Cursor (page 81)*** Create tailable cursors for use in capped collections with high numbers of write operations for which an index would be too expensive.

***Create an Auto-Incrementing Sequence Field (page 82)*** Describes how to create an incrementing sequence number for the `_id` field using a Counters Collection or an Optimistic Loop.

***Perform Quorum Reads on Replica Sets (page 86)*** Perform quorum reads using `findAndModify`.

### 3.1 Insert Documents

#### On this page

- [Insert a Document \(page 46\)](#)
- [Insert an Array of Documents \(page 46\)](#)
- [Insert Multiple Documents with Bulk \(page 47\)](#)
- [Additional Examples and Methods \(page 49\)](#)

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

### 3.1.1 Insert a Document

#### Step 1: Insert a document into a collection.

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

```
db.inventory.insert(
  {
    item: "ABC1",
    details: {
      model: "14Q3",
      manufacturer: "XYZ Company"
    },
    stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
    category: "clothing"
  }
)
```

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

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

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

#### Step 2: Review the inserted document.

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

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

The document you inserted should return.

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

The returned document shows that MongoDB added an `_id` field to the document. If a client inserts a document that does not contain the `_id` field, MongoDB adds the field with the value set to a generated `ObjectId`<sup>1</sup>. The `ObjectId`<sup>2</sup> values in your documents will differ from the ones shown.

### 3.1.2 Insert an Array of Documents

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

#### Step 1: Create an array of documents.

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

---

<sup>1</sup><https://docs.mongodb.org/manual/reference/method/ObjectId>

<sup>2</sup><https://docs.mongodb.org/manual/reference/method/ObjectId>

```

var mydocuments =
[
  {
    item: "ABC2",
    details: { model: "14Q3", manufacturer: "M1 Corporation" },
    stock: [ { size: "M", qty: 50 } ],
    category: "clothing"
  },
  {
    item: "MNO2",
    details: { model: "14Q3", manufacturer: "ABC Company" },
    stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
    category: "clothing"
  },
  {
    item: "IJK2",
    details: { model: "14Q2", manufacturer: "M5 Corporation" },
    stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],
    category: "houseware"
  }
];

```

## Step 2: Insert the documents.

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

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

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

```

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

```

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

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

### 3.1.3 Insert Multiple Documents with Bulk

New in version 2.6.

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

### Step 1: Initialize a Bulk operations builder.

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

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

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

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

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

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

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

### Step 3: Execute the bulk operation.

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

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

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

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

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

### 3.1.4 Additional Examples and Methods

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

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

## 3.2 Query Documents

### On this page

- [Select All Documents in a Collection](#) (page 49)
- [Specify Equality Condition](#) (page 49)
- [Specify Conditions Using Query Operators](#) (page 50)
- [Specify AND Conditions](#) (page 50)
- [Specify OR Conditions](#) (page 50)
- [Specify AND as well as OR Conditions](#) (page 50)
- [Embedded Documents](#) (page 51)
- [Arrays](#) (page 51)
- [Null or Missing Fields](#) (page 56)

In MongoDB, the `db.collection.find()` method retrieves documents from a collection.<sup>3</sup> The `db.collection.find()` method returns a *cursor* (page 11) to the retrieved documents.

This tutorial provides examples of read operations using the `db.collection.find()` method in the mongo shell. In these examples, the retrieved documents contain all their fields. To restrict the fields to return in the retrieved documents, see [Limit Fields to Return from a Query](#) (page 62).

### 3.2.1 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()
```

### 3.2.2 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" } )
```

<sup>3</sup> 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.

### 3.2.3 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 <https://docs.mongodb.org/manual/reference/operator/query> document for the complete list of query operators.

### 3.2.4 Specify AND Conditions

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

In the following example, the query document specifies an equality match on the field `type` **and** a less than (`$lt`) comparison match on the field `price`:

```
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.

### 3.2.5 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 } } ]
  }
)
```

---

**Note:** Queries which use *comparison operators* are subject to *type-bracketing*.

---

### 3.2.6 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 } } ]
  }
)
```

### 3.2.7 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' } )
```

### 3.2.8 Arrays

When the field holds an array, you can query for an exact array match or for specific values in the array. If the array holds embedded documents, you can query for specific fields in the embedded documents using *dot notation*.

If you specify multiple conditions using the `$elemMatch` operator, the array must contain at least one element that satisfies all the conditions. See *Single Element Satisfies the Criteria* (page 53).



If you specify multiple conditions without using the `$elemMatch` operator, then some combination of the array elements, not necessarily a single element, must satisfy all the conditions; i.e. different elements in the array can satisfy different parts of the conditions. See *Combination of Elements Satisfies the Criteria* (page 53).

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

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

### Exact Match on an Array

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

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

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

The operation returns the following document:

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

### Match an Array Element

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

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

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

The operation returns the following documents:

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

### Match a Specific Element of an Array

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

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

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

The operation returns the following documents:

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

## Specify Multiple Criteria for Array Elements

### Single Element Satisfies the Criteria

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

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

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

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

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

### Combination of Elements Satisfies the Criteria

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

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

The operation returns the following documents:

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

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

## Array of Embedded Documents

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

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

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
```

```
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

### Match a Field in the Embedded Document Using the Array Index

If you know the array index of the embedded document, you can specify the document using the embedded document's position using the *dot notation*.

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

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

The operation returns the following document:

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

### Match a Field Without Specifying Array Index

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

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

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

The operation returns the following documents:

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

## Specify Multiple Criteria for Array of Documents

### Single Element Satisfies the Criteria

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

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

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

The operation returns the following document:

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

### Combination of Elements Satisfies the Criteria

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

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

The query returns the following documents:

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
}
```

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

### See also:

*Limit Fields to Return from a Query* (page 62)

## 3.2.9 Null or Missing Fields

Different query operators in MongoDB treat `null` values differently.

Given the collection `inventory` with the following documents:

```
{ "_id" : 900, "item" : null }
{ "_id" : 901 }
```

### Equality Filter

The `{ item : null }` query matches documents that either contain the `item` field whose value is `null` *or* that do not contain the `item` field.

Given the following query:

```
db.inventory.find( { item: null } )
```

The query returns both documents:

```
{ "_id" : 900, "item" : null }
{ "_id" : 901 }
```

If the query uses an index that is *sparse*, however, then the query will only match `null` values, not missing fields.

Changed in version 2.6: If using the sparse index results in an incomplete result, MongoDB will not use the index unless a `hint()` explicitly specifies the index. See *index-type-sparse* for more information.

### Type Check

The `{ item : { $type: 10 } }` query matches documents that contains the `item` field whose value is `null` *only*; i.e. the value of the `item` field is of BSON Type Null (i.e. 10):

```
db.inventory.find( { item : { $type: 10 } } )
```

The query returns only the document where the `item` field has a `null` value:

```
{ "_id" : 900, "item" : null }
```

## Existence Check

The `{ item : { $exists: false } }` query matches documents that do not contain the `item` field:

```
db.inventory.find( { item : { $exists: false } } )
```

The query returns only the document that does *not* contain the `item` field:

```
{ "_id" : 901 }
```

### See also:

The reference documentation for the `$type` and `$exists` operators.

## 3.3 Modify Documents

### On this page

- [Update Specific Fields in a Document](#) (page 57)
- [Replace the Document](#) (page 59)
- [upsert Option](#) (page 59)
- [Additional Examples and Methods](#) (page 61)

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

- an query filter document to determine which documents to update,
- an update document to specify the modification to perform or a replacement document that wholly replaces the matching documents except for the `_id` field, and
- an options document.

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

### 3.3.1 Update Specific Fields in a Document

To change a field value, MongoDB provides [update operators](#)<sup>4</sup>, such as `$set` to modify values.

To specify the modification to perform using update operators, use an update document of the form:

```
{
  <update operator>: { <field1>: <value1>, ... },
  <update operator>: { <field2>: <value2>, ... },
  ...
}
```

Some update operators, such as `$set`, will create the field if the field does not exist. See the individual [update operator](#)<sup>5</sup> reference.

<sup>4</sup><https://docs.mongodb.org/manual/reference/operator/update>

<sup>5</sup><https://docs.mongodb.org/manual/reference/operator/update>

### Step 1: Use update operators to change field values.

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

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

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

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

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

### Step 2: Update an embedded field.

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

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

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

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

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

### Step 3: Update multiple documents.

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

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

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

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

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

### 3.3.2 Replace the Document

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

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

#### Step 1: Replace a document.

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

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

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

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

### 3.3.3 upsert Option

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

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

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

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

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

```
db.inventory.update(
  { item: "TBD1" },
  {
```



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The information above indicates that the operation has created one new document. The `_id` field shows the generated `_id` field for the added document; you can perform a query to confirm the result:

```
db.inventory.findOne( { _id: ObjectId("53dbd7c8babeaec6342ed6c8") } )
```

The result matches the document specified in the `update()`:

```
{
  "_id" : ObjectId("56a12ec8242ae5d73c07b15e"),
  "item" : "TBD2",
  "details" : {
    "model" : "14Q3",
    "manufacturer" : "IJK Co."
  },
  "category" : "houseware"
}
```

### 3.3.4 Additional Examples and Methods

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

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

## 3.4 Remove Documents

### On this page

- [Remove All Documents](#) (page 61)
- [Remove Documents that Match a Condition](#) (page 62)
- [Remove a Single Document that Matches a Condition](#) (page 62)

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.

### 3.4.1 Remove All Documents

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

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

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

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

### 3.4.2 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.

### 3.4.3 Remove a Single Document that Matches a Condition

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

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

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

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

## 3.5 Limit Fields to Return from a Query

#### On this page

- [Return All Fields in Matching Documents](#) (page 63)
- [Return the Specified Fields and the `\_id` Field Only](#) (page 63)
- [Return Specified Fields Only](#) (page 63)
- [Return All But the Excluded Field](#) (page 63)
- [Return Specific Fields in Embedded Documents](#) (page 63)
- [Suppress Specific Fields in Embedded Documents](#) (page 64)
- [Projection for Array Fields](#) (page 65)

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

The specifications have the following forms:

Syntax	Description
<code>&lt;field&gt;: &lt;1 or true&gt;</code>	Specify the inclusion of a field.
<code>&lt;field&gt;: &lt;0 or false&gt;</code>	Specify the suppression of the field.

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

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

This tutorial offers various query examples that limit the fields to return for all matching documents. The examples in this tutorial use a collection `inventory` and use the `db.collection.find()` method in the mongo shell. The `db.collection.find()` method returns a *cursor* (page 11) to the retrieved documents. For examples on query selection criteria, see [Query Documents](#) (page 49).

### 3.5.1 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 fields.

### 3.5.2 Return the Specified Fields and the `_id` Field Only

A projection can explicitly include several fields. In the following operation, the `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 } )
```

### 3.5.3 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.

### 3.5.4 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.

### 3.5.5 Return Specific Fields in Embedded Documents

Use the *dot notation* to return specific fields inside an embedded document. For example, the `inventory` collection contains the following document:

```
{
  "_id" : 3,
  "type" : "food",
  "item" : "aaa",
  "classification": { dept: "grocery", category: "chocolate" }
}
```

The following operation returns all documents that match the query. The specified projection returns only the `category` field in the `classification` document. The returned `category` field remains inside the `classification` document.

```
db.inventory.find(
  { type: 'food', _id: 3 },
  { "classification.category": 1, _id: 0 }
)
```

The operation returns the following document:

```
{ "classification" : { "category" : "chocolate" } }
```

### 3.5.6 Suppress Specific Fields in Embedded Documents

Use *dot notation* to suppress specific fields inside an embedded document using a 0 instead of 1. For example, the `inventory` collection contains the following document:

```
{
  "_id" : 3,
  "type" : "food",
  "item" : "Super Dark Chocolate",
  "classification" : { "dept" : "grocery", "category" : "chocolate"},
  "vendor" : {
    "primary" : {
      "name" : "Marsupial Vending Co",
      "address" : "Wallaby Rd",
      "delivery" : ["M", "W", "F"]
    },
    "secondary":{
      "name" : "Intl. Chocolatiers",
      "address" : "Cocoa Plaza",
      "delivery" : ["Sa"]
    }
  }
}
```

The following operation returns all documents where the value of the `type` field is `food` and the `_id` field is 3. The projection suppresses only the `category` field in the `classification` document. The `dept` field remains inside the `classification` document.

```
db.inventory.find(
  { type: 'food', _id: 3 },
  { "classification.category": 0 }
)
```

The operation returns the following document:

```
{
  "_id" : 3,
  "type" : "food",
  "item" : "Super Dark Chocolate",
  "classification" : { "dept" : "grocery"},
  "vendor" : {
    "primary" : {
      "name" : "Bobs Vending",
      "address" : "Wallaby Rd",
      "delivery" : ["M", "W", "F"]
    },
    "secondary":{
      "name" : "Intl. Chocolatiers",
      "address" : "Cocoa Plaza",

```

```

        "delivery" : ["Sa"]
      }
    }
  }
}

```

You can suppress nested subdocuments at any depth using *dot notation*. The following specifies a projection to suppress the `delivery` array only for the secondary document.

```

db.inventory.find(
  { "type" : "food" },
  { "vendor.secondary.delivery" : 0 }
)

```

This returns all documents except the `delivery` array in the `secondary` document

```

{
  "_id" : 3,
  "type" : "food",
  "item" : "Super Dark Chocolate",
  "classification" : { "dept" : "grocery", "category" : "chocolate" },
  "vendor" : {
    "primary" : {
      "name" : "Bobs Vending",
      "address" : "Wallaby Rd",
      "delivery" : ["M", "W", "F"]
    },
    "secondary":{
      "name" : "Intl. Chocolatiers",
      "address" : "Cocoa Plaza"
    }
  }
}

```

### 3.5.7 Projection for Array Fields

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

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

```

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

```

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

```

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

```

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

**See also:**

[Query Documents](#) (page 49)

## 3.6 Limit Number of Elements in an Array after an Update

**On this page**

- [Synopsis](#) (page 66)
- [Pattern](#) (page 66)

New in version 2.4.

### 3.6.1 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.

### 3.6.2 Pattern

Consider the following document in the collection `students`:

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

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

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

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

---

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

---

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

```
{
  "_id" : 1,
  "scores" : [
```

```

    { "attempt" : 3, "score" : 7 },
    { "attempt" : 2, "score" : 8 },
    { "attempt" : 1, "score" : 10 }
  ]
}

```

**See also:**

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

## 3.7 Iterate a Cursor in the mongo Shell

### On this page

- [Manually Iterate the Cursor](#) (page 67)
- [Iterator Index](#) (page 68)

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.

### 3.7.1 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 <sup>6</sup> 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' } );

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

```

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

```

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

while (myCursor.hasNext()) {

```

<sup>6</sup> 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.



```
    printjson(myCursor.next());  
}
```

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

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

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

### 3.7.2 Iterator Index

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

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

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

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

Consider the following example:

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

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

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

## 3.8 Analyze Query Performance

### On this page

- [Evaluate the Performance of a Query](#) (page 69)
- [Compare Performance of Indexes](#) (page 71)
- [Additional Resources](#) (page 73)

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

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

### 3.8.1 Evaluate the Performance of a Query

Consider a collection `inventory` with the following documents:

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

#### Query with No Index

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

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

The query returns the following documents:

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

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

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

`explain()` returns the following results:

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

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

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

### Query with Index

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

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

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

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

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

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

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

- `executionStats.totalDocsExamined` display 3 to indicate that MongoDB scanned three documents.

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

### 3.8.2 Compare Performance of Indexes

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

Consider the following query:

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

The query returns the following documents:

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

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

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

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

Evaluate the effect of the first index on the query:

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

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

```
{
  "queryPlanner" : {
    ...
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "quantity" : 1,
          "type" : 1
        },
        ...
      }
    },
    ...
  },
  "rejectedPlans" : [ ]
},
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 2,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 5,
  "totalDocsExamined" : 2,
  "executionStages" : {
    ...
  }
}
```

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

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

Evaluate the effect of the second index on the query:

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

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

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

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

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

**See also:**

[Query Optimization](#) (page 13), [Query Plans](#) (page 15), <https://docs.mongodb.org/manual/tutorial/optimize-query/>, <https://docs.mongodb.org/manual/applications/indexes>

### 3.8.3 Additional Resources

- [MongoDB Performance Evaluation and Tuning Consulting Package](#)<sup>7</sup>

## 3.9 Perform Two Phase Commits

### On this page

- [Synopsis](#) (page 73)
- [Background](#) (page 73)
- [Pattern](#) (page 74)
- [Recovering from Failure Scenarios](#) (page 77)
- [Multiple Applications](#) (page 79)
- [Using Two-Phase Commits in Production Applications](#) (page 80)

### 3.9.1 Synopsis

This document provides a pattern for doing multi-document updates or “multi-document transactions” using a two-phase commit approach for writing data to multiple documents. Additionally, you can extend this process to provide a *rollback-like* (page 77) functionality.

### 3.9.2 Background

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

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

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

For situations that require multi-document transactions, you can implement two-phase commit in your application to provide support for these kinds of multi-document updates. Using two-phase commit ensures that data is consistent and, in case of an error, the state that preceded the transaction is *recoverable* (page 77). During the procedure, however, documents can represent pending data and states.

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

<sup>7</sup>[https://www.mongodb.com/products/consulting?jmp=docs#performance\\_evaluation](https://www.mongodb.com/products/consulting?jmp=docs#performance_evaluation)

### 3.9.3 Pattern

#### Overview

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

The examples in this tutorial use the following two collections:

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

#### Initialize Source and Destination Accounts

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

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

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

#### Initialize Transfer Record

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

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

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

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

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

## Transfer Funds Between Accounts Using Two-Phase Commit

### Step 1: Retrieve the transaction to start.

From the `transactions` collection, find a transaction in the `initial` state. Currently the `transactions` collection has only one document, namely the one added in the *Initialize Transfer Record* (page 74) step. If the collection contains additional documents, the query will return any transaction with an `initial` state unless you specify additional query conditions.

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

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

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

### Step 2: Update transaction state to pending.

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

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

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

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

### Step 3: Apply the transaction to both accounts.

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

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

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

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

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

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



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

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

### Step 4: Update transaction state to applied.

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

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

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

### Step 5: Update both accounts' list of pending transactions.

Remove the applied transaction `_id` from the `pendingTransactions` array for both accounts.

Update the source account.

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

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

Update the destination account.

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

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

### Step 6: Update transaction state to done.

Complete the transaction by setting the state of the transaction to `done` and updating the `lastModified` field:

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

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

### 3.9.4 Recovering from Failure Scenarios

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

#### Recovery Operations

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

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

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

#### Transactions in Pending State

To recover from failures that occur after step “Update transaction state to pending. (page ??)” but before “Update transaction state to applied. (page ??)” step, retrieve from the `transactions` collection a pending transaction for recovery:

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

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

And resume from step “Apply the transaction to both accounts. (page ??)”

#### Transactions in Applied State

To recover from failures that occur after step “Update transaction state to applied. (page ??)” but before “Update transaction state to done. (page ??)” step, retrieve from the `transactions` collection an applied transaction for recovery:

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

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

And resume from “Update both accounts’ list of pending transactions. (page ??)”

#### Rollback Operations

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

### Transactions in Applied State

After the “Update transaction state to applied. (page ??)” step, you should **not** roll back the transaction. Instead, complete that transaction and *create a new transaction* (page 74) to reverse the transaction by switching the values in the source and the destination fields.

### Transactions in Pending State

After the “Update transaction state to pending. (page ??)” step, but before the “Update transaction state to applied. (page ??)” step, you can rollback the transaction using the following procedure:

#### Step 1: Update transaction state to canceling.

Update the transaction state from pending to canceling.

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

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

#### Step 2: Undo the transaction on both accounts.

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

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

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

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

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

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

```

    }
)

```

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

### Step 3: Update transaction state to canceled.

To finish the rollback, update the transaction state from canceling to cancelled.

```

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

```

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

## 3.9.5 Multiple Applications

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

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

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

```

t = db.transactions.findAndModify(
  {
    query: { state: "initial", application: { $exists: false } },
    update:
      {
        $set: { state: "pending", application: "App1" },
        $currentDate: { lastModified: true }
      },
    new: true
  }
)

```

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

If the application App1 fails during transaction execution, you can use the [recovery procedures](#) (page 77), but applications should ensure that they “own” the transaction before applying the transaction. For example to find and resume the pending job, use a query that resembles the following:

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

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

### 3.9.6 Using Two-Phase Commits in Production Applications

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

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

For all transactions, ensure that you use the appropriate level of *write concern* (page 90) for your deployment.

## 3.10 Update Document if Current

### On this page

- [Overview](#) (page 80)
- [Pattern](#) (page 80)
- [Example](#) (page 80)
- [Modifications to the Pattern](#) (page 81)

### 3.10.1 Overview

The *Update if Current* pattern is an approach to *concurrency control* (page 34) when multiple applications have access to the data.

### 3.10.2 Pattern

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

### 3.10.3 Example

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

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

```

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

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

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

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

```

### 3.10.4 Modifications to the Pattern

Another approach is to add a `version` field to the documents. Applications increment this field upon each update operation to the documents. You must be able to ensure that *all* clients that connect to your database include the `version` field in the query predicate. To associate increasing numbers with documents in a collection, you can use one of the methods described in [Create an Auto-Incrementing Sequence Field](#) (page 82).

For more approaches, see [Concurrency Control](#) (page 34).

## 3.11 Create Tailable Cursor

### On this page

- [Overview](#) (page 81)

### 3.11.1 Overview

By default, MongoDB will automatically close a cursor when the client has exhausted all results in the cursor. However, for capped collections you may use a *Tailable Cursor* that remains open after the client exhausts the results in the initial cursor. Tailable cursors are conceptually equivalent to the `tail` Unix command with the `-f`

option (i.e. with “follow” mode). After clients insert new additional documents into a capped collection, the tailable cursor will continue to retrieve documents.

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

---

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

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

---

Consider the following behaviors related to tailable cursors:

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

A *dead* cursor has an id of 0.

See your `driver` documentation for the driver-specific method to specify the tailable cursor.

## 3.12 Create an Auto-Incrementing Sequence Field

### On this page

- [Synopsis](#) (page 82)
- [Considerations](#) (page 82)
- [Procedures](#) (page 83)

### 3.12.1 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 83)
- [Optimistic Loop](#) (page 84)

### 3.12.2 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`.

### 3.12.3 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.

## 3.13 Perform Quorum Reads on Replica Sets

New in version 3.2.

### 3.13.1 Overview

When reading from the primary of a replica set, it is possible to read data that is stale or not durable, depending on the read concern used<sup>8</sup>. With a read concern level of `"local"` (page 93), a client can read data before it is *durable*; that is, before they have propagated to enough replica set members to avoid a rollback. A read concern level of `"majority"` (page 93) guarantees durable reads but may return stale data that has been overwritten by another write operation.

This tutorial outlines a procedure that uses `db.collection.findAndModify()` to read data that is not stale and cannot be rolled back. To do so, the procedure uses the `findAndModify()` method with a *write concern* (page 90) to modify a dummy field in a document. Specifically, the procedure requires that:

- `db.collection.findAndModify()` use an **exact** match query, and a unique index **must exist** to satisfy the query.
- `findAndModify()` must actually modify a document; i.e. result in a change to the document.
- `findAndModify()` must use the write concern `{ w: "majority" }` (page 91).

---

**Important:** The “quorum read” procedure has a substantial cost over simply using a read concern of `"majority"` (page 93) because it incurs write latency rather than read latency. This technique should only be used if staleness is absolutely intolerable.

---

### 3.13.2 Prerequisites

This tutorial reads from a collection named `products`. Initialize the collection using the following operation.

```
db.products.insert( [
  {
    _id: 1,
    sku: "xyz123",
    description: "hats",
    available: [ { quantity: 25, size: "S" }, { quantity: 50, size: "M" } ],
    _dummy_field: 0
  },
]
```

---

<sup>8</sup> In *some circumstances*, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with `{ w: "majority" }` (page 91) write concern. The node that can complete `{ w: "majority" }` (page 91) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary`, and new writes to the former primary will eventually roll back.

```

{
  _id: 2,
  sku: "abc123",
  description: "socks",
  available: [ { quantity: 10, size: "L" } ],
  _dummy_field: 0
},
{
  _id: 3,
  sku: "ijk123",
  description: "t-shirts",
  available: [ { quantity: 30, size: "M" }, { quantity: 5, size: "L" } ],
  _dummy_field: 0
}
] )

```

The documents in this collection contain a dummy field named `_dummy_field` that will be incremented by the `db.collection.findAndModify()` in the tutorial. If the field does not exist, the `db.collection.findAndModify()` operation will add the field to the document. The purpose of the field is to ensure that the `db.collection.findAndModify()` results in a modification to the document.

### 3.13.3 Procedure

#### Step 1: Create a unique index.

Create a unique index on the fields that will be used to specify an exact match in the `db.collection.findAndModify()` operation.

This tutorial will use an exact match on the `sku` field. As such, create a unique index on the `sku` field.

```
db.products.createIndex( { sku: 1 }, { unique: true } )
```

#### Step 2: Use `findAndModify` to read committed data.

Use the `db.collection.findAndModify()` method to make a trivial update to the document you want to read and return the modified document. A write concern of `{ w: "majority" }` (page 91) is required. To specify the document to read, you must use an exact match query that is supported by a unique index.

The following `findAndModify()` operation specifies an exact match on the uniquely indexed field `sku` and increments the field named `_dummy_field` in the matching document. While not necessary, the write concern for this command also includes a *wtimeout* (page 92) value of 5000 milliseconds to prevent the operation from blocking forever if the write cannot propagate to a majority of voting members.

```

var updatedDocument = db.products.findAndModify(
{
  query: { sku: "abc123" },
  update: { $inc: { _dummy_field: 1 } },
  new: true,
  writeConcern: { w: "majority", wtimeout: 5000 }
}
);

```

Even in situations where two nodes in the replica set believe that they are the primary, only one will be able to complete the write with `w: "majority"` (page 91). As such, the `findAndModify()` method with `"majority"` (page 91) write concern will be successful only when the client has connected to the true primary to perform the operation.

Since the quorum read procedure only increments a dummy field in the document, you can safely repeat invocations of `findAndModify()`, adjusting the *wtimeout* (page 92) as necessary.

## MongoDB CRUD Reference

### On this page

- [Query Cursor Methods](#) (page 89)
- [Query and Data Manipulation Collection Methods](#) (page 89)
- [MongoDB CRUD Reference Documentation](#) (page 90)

### 4.1 Query Cursor Methods

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

### 4.2 Query and Data Manipulation Collection Methods

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

## 4.3 MongoDB CRUD Reference Documentation

**Write Concern** (page 90) Description of the write operation acknowledgements returned by MongoDB.

**Read Concern** (page 92) Description of the `readConcern` option.

**SQL to MongoDB Mapping Chart** (page 94) An overview of common database operations showing both the MongoDB operations and SQL statements.

**The bios Example Collection** (page 100) Sample data for experimenting with MongoDB. `insert()`, `update()` and `find()` pages use the data for some of their examples.

### 4.3.1 Write Concern

#### On this page

- [Write Concern Specification](#) (page 90)

Write concern describes the level of acknowledgement requested from MongoDB for write operations to a standalone `mongod` or to `replica sets` or to `sharded clusters`. In `sharded clusters`, `mongos` instances will pass the write concern on to the shards.

Changed in version 3.2: For replica sets using `protocolVersion: 1` **and** running with the *journal* enabled:

- `w: "majority"` (page 91) implies `j: true` (page 91).
- *Secondary members* acknowledge replicated write operations after the secondary members have written to their respective on-disk journals, regardless of the `j` (page 91) option used for the write on the *primary*.

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.

#### Write Concern Specification

Write concern can include the following fields:

```
{ w: <value>, j: <boolean>, wtimeout: <number> }
```

- the `w` (page 90) option to request acknowledgment that the write operation has propagated to a specified number of `mongod` instances or to `mongod` instances with specified tags.
- the `j` (page 91) option to request acknowledgement that the write operation has been written to the journal, and
- `wtimeout` (page 92) option to specify a time limit to prevent write operations from blocking indefinitely.

#### `w` Option

The `w` option requests acknowledgement that the write operation has propagated to a specified number of `mongod` instances or to `mongod` instances with specified tags.

Using the `w` option, the following `w: <value>` write concerns are available:

---

**Note:** Standalone `mongod` instances and primaries of replica sets acknowledge write operations after applying the write in memory, unless `j:true` (page 91).

Changed in version 3.2: For replica sets using `protocolVersion: 1`, secondaries acknowledge write operations after the secondary members have written to their respective on-disk journals, regardless of the *j* (page 91) option.

Value	Description
<code>&lt;number&gt;</code>	<p>Requests acknowledgement that the write operation has propagated to the specified number of <code>mongod</code> instances. For example:</p> <p><b>w: 1</b> Requests acknowledgement that the write operation has propagated to the standalone <code>mongod</code> or the primary in a replica set. <code>w: 1</code> is the default write concern for MongoDB.</p> <p><b>w: 0</b> Requests no acknowledgment of the write operation. However, <code>w: 0</code> may return information about socket exceptions and networking errors to the application.</p> <p>If you specify <code>w: 0</code> but include <code>j: true</code> (page 91), the <code>j: true</code> (page 91) prevails to request acknowledgement from the standalone <code>mongod</code> or the primary of a replica set.</p> <p>Numbers greater than 1 are valid only for replica sets to request acknowledgement from specified number of members, including the primary.</p>
<code>"majority"</code>	<p><i>Changed in version 3.2</i></p> <p>Requests acknowledgment that write operations have propagated to the majority of voting nodes <sup>1</sup>, including the primary, and have been written to the on-disk journal for these nodes.</p> <p>For replica sets using <code>protocolVersion: 1</code>, <code>w: "majority"</code> (page 91) implies <code>j: true</code> (page 91). So, unlike <code>w: &lt;number&gt;</code>, with <code>w: "majority"</code> (page 91), the primary also writes to the on-disk journal before acknowledging the write.</p> <p>After the write operation returns with a <code>w: "majority"</code> (page 91) acknowledgement to the client, the client can read the result of that write with a <code>"majority"</code> (page 93) <code>readConcern</code>.</p>
<code>&lt;tag set&gt;</code>	<p>Requests acknowledgement that the write operations have propagated to a replica set member with the specified <i>tag</i>.</p>

### j Option

The *j* (page 91) option requests acknowledgement from MongoDB that the write operation has been written to the journal.



<code>j</code>	<p>Requests acknowledgement that the <code>mongod</code> instances, as specified in the <code>w: &lt;value&gt;</code> (page 90), have written to the on-disk journal. <code>j: true</code> does not by itself guarantee that the write will not be rolled back due to replica set primary failover.</p> <p>Changed in version 3.2: With <code>j: true</code> (page 92), MongoDB returns only after the requested number of members, including the primary, have written to the journal. Previously <code>j: true</code> (page 92) write concern in a replica set only requires the <i>primary</i> to write to the journal, regardless of the <code>w: &lt;value&gt;</code> (page 90) write concern.</p> <p>For replica sets using <code>protocolVersion: 1</code>, <code>w: "majority"</code> (page 91) implies <code>j: true</code> (page 91), if journaling is enabled. Journaling is enabled by default.</p>
----------------	---

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

#### `wtimeout`

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

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

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

### 4.3.2 Read Concern

#### On this page

- [Storage Engine and Drivers Support](#) (page 92)
- [Read Concern Levels](#) (page 93)
- [readConcern Option](#) (page 93)

New in version 3.2.

MongoDB 3.2 introduces the `readConcern` query option for replica sets and replica set shards. By default, MongoDB uses a read concern of `"local"` to return the most recent data available to the MongoDB instance at the time of the query, even if the data has not been persisted to a majority of replica set members and may be rolled back.

#### Storage Engine and Drivers Support

For the `WiredTiger` storage engine, the `readConcern` option allows clients to choose a level of isolation for their reads. You can specify a read concern of `"majority"` to read data that has been written to a majority of replica set members and thus cannot be rolled back.

With the `MMAPv1` storage engine, you can only specify a `readConcern` option of `"local"`.

**Tip**

The `serverStatus` command returns the `storageEngine.supportsCommittedReads` field which indicates whether the storage engine supports "majority" read concern.

`readConcern` requires MongoDB drivers updated for 3.2.

**Read Concern Levels**

By default, MongoDB uses a `readConcern` of "local" which does not guarantee that the read data would not be rolled back.

You can specify a `readConcern` of "majority" to read data that has been written to a majority of replica set members and thus cannot be rolled back.

level	Description
"local"	Default. The query returns the instance's most recent copy of data. Provides no guarantee that the data has been written to a majority of the replica set members.
"majority"	<p>The query returns the instance's most recent copy of data confirmed as written to a majority of members in the replica set.</p> <p>To use a <i>read concern</i> level of "majority" (page 93), you must use the WiredTiger storage engine and start the <code>mongod</code> instances with the <code>--enableMajorityReadConcern</code> command line option (or the <code>replication.enableMajorityReadConcern</code> setting if using a configuration file).</p> <p>Only replica sets using protocol version 1 support "majority" (page 93) read concern. Replica sets running protocol version 0 do not support "majority" (page 93) read concern.</p> <p>To ensure that a single thread can read its own writes, use "majority" (page 93) read concern and "majority" (page 91) write concern against the primary of the replica set.</p>

Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

**readConcern Option**

Use the `readConcern` option to specify the read concern level.

```
readConcern: { level: <"majority"|"local"> }
```

For the `level` field, specify either the string "majority" or "local".

The `readConcern` option is available for the following operations:

- `find` command
- `aggregate` command and the `db.collection.aggregate()` method
- `distinct` command
- `count` command

- `parallelCollectionScan` command
- `geoNear` command
- `geoSearch` command

To specify the read concern for the mongo shell method `db.collection.find()`, use the `cursor.readConcern()` method.

### 4.3.3 SQL to MongoDB Mapping Chart

**On this page**

- [Terminology and Concepts](#) (page 94)
- [Executables](#) (page 94)
- [Examples](#) (page 95)
- [Additional Resources](#) (page 99)

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

#### Terminology and Concepts

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

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

#### Executables

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

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

## Examples

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

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

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

## Create and Alter

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

SQL Schema Statements	MongoDB Schema Statements
<pre> <b>CREATE TABLE</b> users (   id MEDIUMINT <b>NOT NULL</b>     AUTO_INCREMENT,   user_id Varchar(30),   age Number,   status char(1),   <b>PRIMARY KEY</b> (id) )  <b>ALTER TABLE</b> users <b>ADD</b> join_date DATETIME  <b>ALTER TABLE</b> users <b>DROP COLUMN</b> join_date  <b>CREATE INDEX</b> idx_user_id_asc <b>ON</b> users(user_id)  <b>CREATE INDEX</b>   idx_user_id_asc_age_desc <b>ON</b> users(user_id, age <b>DESC</b>)  <b>DROP TABLE</b> users </pre>	<p>Implicitly created on first <code>insert()</code> operation. The primary key <code>_id</code> is automatically added if <code>_id</code> field is not specified.</p> <pre> db.users.insert( {   user_id: "abc123",   age: 55,   status: "A" } ) </pre> <p>However, you can also explicitly create a collection:</p> <pre> db.createCollection("users") </pre> <p>Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.</p> <p>However, at the document level, <code>update()</code> operations can add fields to existing documents using the <code>\$set</code> operator.</p> <pre> db.users.update(   { },   { \$set: { join_date: new Date() } },   { multi: true } ) </pre> <p>Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.</p> <p>However, at the document level, <code>update()</code> operations can remove fields from documents using the <code>\$unset</code> operator.</p> <pre> db.users.update(   { },   { \$unset: { join_date: "" } },   { multi: true } ) </pre> <pre> db.users.createIndex( { user_id: 1 } )  db.users.createIndex( { user_id: 1, age: -1 } )  db.users.drop() </pre>

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

Insert

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

SQL INSERT Statements	MongoDB insert() Statements
<pre>INSERT INTO users (user_id,                     age,                     status) VALUES ("bcd001",         45,         "A")</pre>	<pre>db.users.insert (   { user_id: "bcd001", age: 45, status: "A" } )</pre>

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.

**Note:** The `find()` method always includes the `_id` field in the returned documents unless specifically excluded through *projection* (page 62). Some of the SQL queries below may include an `_id` field to reflect this, even if the field is not included in the corresponding `find()` query.

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

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
<pre>UPDATE users SET status = "C" WHERE age &gt; 25</pre>	<pre>db.users.update(   { age: { \$gt: 25 } },   { \$set: { status: "C" } },   { multi: true } )</pre>
<pre>UPDATE users SET age = age + 3 WHERE status = "A"</pre>	<pre>db.users.update(   { status: "A" },   { \$inc: { age: 3 } },   { multi: true } )</pre>

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
<pre>DELETE FROM users WHERE status = "D"</pre>	<pre>db.users.remove( { status: "D" } )</pre>
<pre>DELETE FROM users</pre>	<pre>db.users.remove({})</pre>

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

## Additional Resources

- [Transitioning from SQL to MongoDB \(Presentation\)](#)<sup>2</sup>
- [Best Practices for Migrating from RDBMS to MongoDB \(Webinar\)](#)<sup>3</sup>
- [SQL vs. MongoDB Day 1-2](#)<sup>4</sup>
- [SQL vs. MongoDB Day 3-5](#)<sup>5</sup>

<sup>2</sup><http://www.mongodb.com/presentations/webinar-transitioning-sql-mongodb?jmp=docs>

<sup>3</sup><http://www.mongodb.com/webinar/best-practices-migration?jmp=docs>

<sup>4</sup><http://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2?jmp=docs>

<sup>5</sup><http://www.mongodb.com/blog/post/mongodb-vs-sql-day-3-5?jmp=docs>



- [MongoDB vs. SQL Day 14](#)<sup>6</sup>
- [MongoDB and MySQL Compared](#)<sup>7</sup>
- [Quick Reference Cards](#)<sup>8</sup>
- [MongoDB Database Modernization Consulting Package](#)<sup>9</sup>

### 4.3.4 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"
    }
  ]
}
```

---

<sup>6</sup><http://www.mongodb.com/blog/post/mongodb-vs-sql-day-14?jmp=docs>

<sup>7</sup><http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs>

<sup>8</sup><https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs>

<sup>9</sup>[https://www.mongodb.com/products/consulting?jmp=docs#database\\_modernization](https://www.mongodb.com/products/consulting?jmp=docs#database_modernization)

```

{
  "_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"
  ]
}
```