CRUD Operations

On this page

CRUD operations are those which deal with creating, reading, updating, and deleting documents.

Key-value Pair Notation

Key-value pairs appear in many different contexts in the MongoDB Ruby driver, and there are some quirks of syntax with regard to how they can be notated which depend on which version of Ruby you're using.

When constructing a document, the following syntax is acceptable and correct for Ruby version 1.9 and later:

```
document = { name: "Harriet", age: 36 }
```

If you're using Ruby version 2.2 or greater, you can optionally enclose your keys in quotes.

```
document = { "name": "Harriet", "age": 36 }
```

If you need to use any MongoDB operator which begins with `$`, such as `$set`, `$gte`, or `$near`, you must enclose it in quotes. If you're using Ruby version 2.2 or greater, you can notate it as follows:

```
collection.update_one({ name: "Harriet" }, { "$set": { age: 42 } })
```

If you're using an earlier version of Ruby, use the hashrocket symbol:

```
collection.update_one({ name: "Harriet" }, { "$set" => { age: 42 } })
```

Quoted strings and hashrockets for key-value pairs will work with any version of Ruby:

```
collection.update_one({ "name" => "Harriet" }, { "$set" => { age: 42 } })
```

Creating Documents

To insert documents into a collection, select a collection on the client and call `insert_one` or `insert_many`.

Insert operations return a `Mongo::Operation::Result` object which gives you information about the insert itself.

On MongoDB 2.6 and later, if the insert fails, an exception is raised, because write commands are used.

On MongoDB 2.4, an exception is only raised if the insert fails and the write concern is 1 or higher.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')

result = client[:artists].insert_one( { :name => 'FKA Twigs' } )
result.n # returns 1, because 1 document was inserted.

result = client[:artists].insert_many([
  { :name => 'Flying Lotus' },
  { :name => 'Aphex Twin' }
])
result.inserted_count # returns 2, because 2 documents were inserted.
```

Specify a `Decimal128` number

New in version 3.4.

Decimal128 is a BSON datatype that employs 128-bit decimal-based floating-point values capable of emulating decimal rounding with exact precision. This functionality is intended for applications that handle monetary data, such as financial and tax computations.

The following example inserts a value of type `Decimal128` into the `price` field of a collection named `inventory`:

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test')

price = BSON::Decimal128.new("428.79")
client[:inventory].insert_one({ "_id" => 1,
                                "item" => "26 inch monitor",
                                "price" => price })
```

The above operation produces the following document:

```
{ "_id" : 1, "item" : "26 inch monitor", "price" : NumberDecimal("428.79") }
```

You can also create a `Decimal128` object from a Ruby `BigDecimal` object, or with `Decimal128.from_string()`.

```
big_decimal = BigDecimal.new(428.79, 5)
price = BSON::Decimal128.new(big_decimal)
# => BSON::Decimal128('428.79')

price = BSON::Decimal128.from_string("428.79")
# => BSON::Decimal128('428.79')
```

Reading

The Ruby driver provides a fluent interface for queries using the `find` method on the collection. Various options are available to the `find` method.

The query is lazily executed against the server only when iterating the results - at that point the query is dispatched and a `Mongo::Cursor` is returned.

To find all documents for a given filter, call `find` with the query:

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')

client[:artists].find(:name => 'Flying Lotus').each do |document|
  #=> Yields a BSON::Document.
end
```

Query Options

To add options to a query, chain the appropriate methods after the `find` method. Note that the underlying object, the `Mongo::Collection::View`, is immutable and a new object will be returned after each method call.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')

documents = client[:artists].find(:name => 'Flying Lotus').skip(10).limit(10)
documents.each do |document|
  #=> Yields a BSON::Document.
end
```

The following is a full list of the available options that can be added when querying and their corresponding methods as examples.

Option	Description
`allow_partial_results`	For use with sharded clusters. If a shard is down, allows the query to return results from the shards that are up, potentially only getting a portion of the results.
`batch_size(Integer)`	Specifies the size of each batch of documents the cursor will return on each `GETMORE` operation.
`comment(String)`	Adds a comment to the query.

Option	Description
`hint(Hash)`	Provides the query with an index hint to use.
`limit(Integer)`	Limits the number of returned documents to the provided value.
`max_scan(Integer)`	Sets the maximum number of documents to scan if a full collection scan would be performed.
`no_cursor_timeout`	MongoDB automatically closes inactive cursors after a period of 10 minutes. Call this for cursors to remain open indefinitely on the server.
`projection(Hash)`	Specifies the fields to include or exclude from the results. `client[:artists].find.projection(:name => 1)`
`read(Hash)`	Changes the read preference for this query only. `client[:artists].find.read(:mode => :secondary_preferred)`
`show_disk_loc(Boolean)`	Tells the results to also include the location of the documents on disk.
`skip(Integer)`	Skip the provided number of documents in the results.
`snapshot`	Execute the query in snapshot mode.
`sort(Hash)`	Specifies sort criteria for the query. `client[:artists].find.sort(:name => -1)`

Additional Query Operations

`count`

Get the total number of documents an operation returns.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')

client[:artists].find(:name => 'Flying Lotus').count
```

`distinct`

Filters out documents with duplicate values. Equivalent to the SQL `distinct` clause.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')

client[:artists].find.distinct(:name )
```

Tailable Cursors

For capped collections you may use a tailable cursor that remains open after the client exhausts the results in the initial cursor. The following code example shows how a tailable cursor might be used:

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
client[:artists].drop
client[:artists, capped: true, size: 512].create

result = client[:artists].insert_many([
  { :name => 'Flying Lotus' },
  { :name => 'Aphex Twin' }
])

enum = client[:artists].find({}, cursor_type: :tailable_await).to_enum

while true
  doc = enum.next
  # do something
  sleep(1)
end
```

Read Preference

Read preference determines the candidate replica set members to which a query or command can be sent. They consist of a **mode** specified as a symbol, an array of hashes known as **tag_sets**, and two timing options: **local_threshold** and **server_selection_timeout**.

`local_threshold`

Defines the upper limit in seconds of the latency window between the nearest server and suitable servers to which an operation may be sent. The default is 15 milliseconds, or 0.015 seconds.

`server_selection_timeout`
Defines how long to block for server selection before throwing an exception. The default is 30,000 milliseconds, or 30 seconds.

For more information on the algorithm used to select a server, please refer to the Server Selection documentation, available on GitHub.

The read preference is set as an option on the client or passed an option when a command is run on a database.

```
# Set read preference on a client, used for all operations
client = Mongo::Client.new([ '127.0.0.1:27017' ],
                           read: { mode: :secondary,
                                   tag_sets: [ { 'dc' => 'nyc' } ]
                                  } )

# Set read preference for a given command
client.database.command( { collstats: 'test' }, read: { mode: secondary,
                                                        tag_sets: [ { 'dc' => 'nyc' } ] } )
```

Mode

There are five possible read preference modes. They are `:primary`, `:secondary`, `:primary_preferred`, `:secondary_preferred`, `:nearest`. Please see the read preference documentation in the MongoDB Manual for an explanation of the modes and tag sets.

Tag sets

The `tag_sets` parameter is an ordered list of tag sets used to restrict the eligibility of servers for selection, such as for data center awareness.

A read preference tag set (T) matches a server tag set (S) – or equivalently a server tag set (S) matches a read preference tag set (T) — if T is a subset of S.

For example, the read preference tag set `{ dc: 'ny', rack: 2 }` matches a secondary server with tag set `{ dc: 'ny', rack: 2, size: 'large' }`.

A tag set that is an empty document matches any server, because the empty tag set is a subset of any tag set. This means the default `tag_sets` parameter `[{}]` matches all servers.

Updating

Updating documents is possible by executing a single or multiple update, or by using the `$findAndModify` command.

update_one

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
artists = client[:artists]

result = artists.find(:name => 'Goldie').update_one("$inc" => { :plays =>  1 } )
result.n # Returns 1.

result = artists.update_one( { :name => 'Goldie' }, { "$inc" => { :plays =>  1 } } )
result.n # Returns 1.
```

update_many

```
result = artists.find(:label => 'Hospital').update_many( "$inc" => { :plays =>  1 } )
result.modified_count # Returns the number of documents that were updated.

result = artists.update_many( { :label => 'Hospital' }, { "$inc" => { :plays =>  1 } } )
result.modified_count # Returns the number of documents that were updated.
```

replace_one

```
result = artists.find(:name => 'Aphex Twin').replace_one(:name => 'Richard James')
result.modified_count # Returns 1.

result = artists.replace_one( { :name => 'Aphex Twin' }, { :name => 'Richard James' } )
result.modified_count # Returns 1.
```

To update documents and return a document via `$findAndModify`, use one of the three provided helpers: `find_one_and_delete`, `find_one_and_replace`, or `find_one_and_update`. You can opt to return the document before or after the modification occurs.

`find_one_and_delete`

```
client = Mongo::Client.new( [ '127.0.0.1:27017' ], :database => 'music')
artists = client[:artists]

artists.find(:name => 'José James').find_one_and_delete # Returns the document.
```

`find_one_and_replace`

```
doc = artists.find(:name => 'José James').find_one_and_replace(:name => 'José')
doc # Return the document before the update.

doc = artists.find_one_and_replace({ :name => 'José James' }, { :name => 'José' })
doc # Return the document before the update.

doc = artists.find(:name => 'José James').
  find_one_and_replace( { :name => 'José' }, :return_document => :after )
doc # Return the document after the update.
```

`find_one_and_update`

```
doc = artists.find(:name => 'José James').
  find_one_and_update( '$set' => { :name => 'José' } )
doc # Return the document before the update.

doc = artists.find_one_and_update( { :name => 'José James' }, { '$set' => { :name => 'José' }
doc # Return the document before the update.
```

`find_one_and_replace`

```
doc = artists.find(:name => 'José James').
  find_one_and_replace( { '$set' => { :name => 'José' } }, :return_document => :after )
doc # Return the document after the update.

doc = artists.find_one_and_replace(
  { :name => 'José James' },
  { '$set' => { :name => 'José' } },
  :return_document => :after
)
doc # Return the document after the update.
```

Deleting

delete_one

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
artists = client[:artists]

result = artists.find(:name => 'Björk').delete_one
result.deleted_count # Returns 1.

result = artists.delete_one(:name => 'Björk')
result.deleted_count # Returns 1.
```

delete_many

```
result = artists.find(:label => 'Mute').delete_many
result.deleted_count # Returns the number deleted.

result = artists.delete_many(:label => 'Mute')
result.deleted_count # Returns the number deleted.
```