Query

- Query.prototype.findOneAndRemove()
- Query.prototype.findOneAndReplace()
- Query.prototype.findOneAndUpdate()
- Query.prototype.geometry()
- Query.prototype.get()
- Query.prototype.getFilter()
- Query.prototype.getOptions()
- Query.prototype.getPopulatedPaths()
- Query.prototype.getQuery()
- Query.prototype.getUpdate()
- Query.prototype.gt()
- Query.prototype.gte()
- Query.prototype.hint()
- Query.prototype.in()
- Query.prototype.intersects()
- Query.prototype.j()
- Query.prototype.lean()
- Query.prototype.limit()
- Query.prototype.lt()
- Query.prototype.lte()
- Query.prototype.map()
- Query.prototype.maxDistance()
- Query.prototype.maxScan()
- Query.prototype.maxTimeMS()
- Query.prototype.maxscan()
- Query.prototype.merge()
- Query.prototype.mod()
- Query.prototype.model
- Query.prototype.mongooseOptions()
- Query.prototype.ne()
- Query.prototype.near()
- Query.prototype.nearSphere()
- Query.prototype.nin()
- Query.prototype.nor()
- Query.prototype.or()
- Query.prototype.orFail()
- Query.prototype.polygon()

- Query.prototype.populate()
- Query.prototype.post()
- Query.prototype.pre()
- Query.prototype.projection()
- Query.prototype.read()
- Query.prototype.readConcern()
- Query.prototype.regex()
- Query.prototype.remove()
- Query.prototype.replaceOne()
- Query.prototype.select()
- Query.prototype.selected()
- Query.prototype.selectedExclusively()
- Query.prototype.selectedInclusively()
- Query.prototype.session()
- Query.prototype.set()
- Query.prototype.setOptions()
- Query.prototype.setQuery()
- Query.prototype.setUpdate()
- Query.prototype.size()
- Query.prototype.skip()
- Query.prototype.slaveOk()
- Query.prototype.slice()
- Query.prototype.snapshot()
- Query.prototype.sort()
- Query.prototype.tailable()
- Query.prototype.then()
- Query.prototype.toConstructor()
- Query.prototype.update()
- Query.prototype.updateMany()
- Query.prototype.updateOne()
- Query.prototype.use$geoWithin
- Query.prototype.w()
- Query.prototype.where()
- Query.prototype.within()
- Query.prototype.writeConcern()
- Query.prototype.wtimeout()

Query()

Parameters

- [options] «Object»
- [model] «Object»
- [conditions] «Object»
- [collection] «Object» Mongoose collection

Query constructor used for building queries. You do not need to instantiate a `Query` directly. Instead use Model functions like `Model.find()`.

Example:

```
const query = MyModel.find(); // `query` is an instance of `Query`
query.setOptions({ lean : true });
query.collection(MyModel.collection);
query.where('age').gte(21).exec(callback);

// You can instantiate a query directly. There is no need to do
// this unless you're an advanced user with a very good reason to.
const query = new mongoose.Query();
```

Query.prototype.$where()

Parameters

- js «String|Function» javascript string or function

Returns:

- «Query» this

Specifies a javascript function or expression to pass to MongoDBs query system.

Example

```
query.$where('this.comments.length === 10 || this.name.length === 5')

// or

query.$where(function () {
  return this.comments.length === 10 || this.name.length === 5;
})
```

NOTE:

Only use `$where` when you have a condition that cannot be met using other MongoDB operators like `$lt`. **Be sure to read about all of its caveats before using.**

Query.prototype.Symbol.asyncIterator()

Returns an asyncIterator for use with `for/await/of` loops This function *only* works for `find()` queries. You do not need to call this function explicitly, the JavaScript runtime will call it for you.

Example

```
for await (const doc of Model.aggregate([{ $sort: { name: 1 } }])) {
  console.log(doc.name);
}
```

Node.js 10.x supports async iterators natively without any flags. You can enable async iterators in Node.js 8.x using the `--harmony_async_iteration` flag.

Note: This function is not if `Symbol.asyncIterator` is undefined. If `Symbol.asyncIterator` is undefined, that means your Node.js version does not support async iterators.

Query.prototype.all()

Parameters

- [path] «String»
- val «Array»

Specifies an `$all` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Example:

```
MyModel.find().where('pets').all(['dog', 'cat', 'ferret']);
// Equivalent:
MyModel.find().all('pets', ['dog', 'cat', 'ferret']);
```

Query.prototype.allowDiskUse()

Parameters

- [v] «Boolean» Enable/disable `allowDiskUse`. If called with 0 arguments, sets `allowDiskUse: true`

Returns:

- «Query» this

Sets the `allowDiskUse` option, which allows the MongoDB server to use more than 100 MB for this query's `sort()`. This option can let you work around `QueryExceededMemoryLimitNoDiskUseAllowed` errors from the MongoDB server.

Note that this option requires MongoDB server >= 4.4. Setting this option is a no-op for MongoDB 4.2 and earlier.

Calling `query.allowDiskUse(v)` is equivalent to `query.setOptions({ allowDiskUse: v })`

Example:

```
await query.find().sort({ name: 1 }).allowDiskUse(true);
// Equivalent:
await query.find().sort({ name: 1 }).allowDiskUse();
```

Query.prototype.and()

Parameters

- array «Array» array of conditions

Returns:

- «Query» this

Specifies arguments for a `$and` condition.

Example

```
query.and([{ color: 'green' }, { status: 'ok' }])
```

Query.prototype.batchSize()

Parameters

- val «Number»

Specifies the batchSize option.

Example

```
query.batchSize(100)
```

Note

Cannot be used with `distinct()`

Query.prototype.box()

Parameters

- val «Object»
- Upper «[Array]» Right Coords

Returns:

- «Query» this

Specifies a `$box` condition

Example

```
const lowerLeft = [40.73083, -73.99756]
const upperRight= [40.741404,  -73.988135]

query.where('loc').within().box(lowerLeft, upperRight)
query.box({ ll : lowerLeft, ur : upperRight })
```

Query.prototype.cast()

Parameters

- [model] «Model» the model to cast to. If not set, defaults to `this.model`
- [obj] «Object»

Returns:

- «Object»

Casts this query to the schema of `model`

Note

If `obj` is present, it is cast instead of this query.

Query.prototype.catch()

Parameters

- [reject] «Function»

Returns:

- «Promise»

Executes the query returning a `Promise` which will be resolved with either the doc(s) or rejected with the error. Like `.then()`, but only takes a rejection handler.

More about Promise `catch()` in JavaScript.

Query.prototype.center()

DEPRECATED Alias for circle

Deprecated. Use circle instead.

Query.prototype.centerSphere()

Parameters

- [path] «String»
- val «Object»

Returns:

- «Query» this

DEPRECATED Specifies a `$centerSphere` condition

Deprecated. Use circle instead.

Example

```
const area = { center: [50, 50], radius: 10 };
query.where('loc').within().centerSphere(area);
```

Query.prototype.circle()

Parameters

- [path] «String»
- area «Object»

Returns:

- «Query» this

Specifies a `$center` or `$centerSphere` condition.

Example

```
const area = { center: [50, 50], radius: 10, unique: true }
query.where('loc').within().circle(area)
// alternatively
query.circle('loc', area);

// spherical calculations
const area = { center: [50, 50], radius: 10, unique: true, spherical: true }
query.where('loc').within().circle(area)
// alternatively
query.circle('loc', area);
```

Query.prototype.clone()

Returns:

- «Query» copy

Make a copy of this query so you can re-execute it.

Example:

```
const q = Book.findOne({ title: 'Casino Royale' });
await q.exec();
await q.exec(); // Throws an error because you can't execute a query twice

await q.clone().exec(); // Works
```

Query.prototype.collation()

Parameters

- value «Object»

Returns:

- «Query» this

Adds a collation to this op (MongoDB 3.4 and up)

Query.prototype.comment()

Parameters

- val «String»

Specifies the `comment` option.

Example

```
query.comment('login query')
```

Note

Cannot be used with `distinct()`

Query.prototype.count()

Parameters

- [filter] «Object» count documents that match this object
- [callback] «Function» optional params are (error, count)

Returns:

- «Query» this

Specifies this query as a `count` query.

This method is deprecated. If you want to count the number of documents in a collection, e.g. `count({})`, use the `estimatedDocumentCount()` function instead. Otherwise, use the `countDocuments()` function instead.

Passing a `callback` executes the query.

This function triggers the following middleware.

- `count()`

Example:

```
const countQuery = model.where({ 'color': 'black' }).count();

query.count({ color: 'black' }).count(callback)

query.count({ color: 'black' }, callback)

query.where('color', 'black').count(function (err, count) {
  if (err) return handleError(err);
  console.log('there are %d kittens', count);
})
```

Query.prototype.countDocuments()

Parameters

- [filter] «Object» mongodb selector
- [callback] «Function» optional params are (error, count)

Returns:

- «Query» this

Specifies this query as a `countDocuments()` query. Behaves like `count()`, except it always does a full collection scan when passed an empty filter `{}`.

There are also minor differences in how `countDocuments()` handles `$where` and a couple geospatial operators. versus `count()`.

Passing a `callback` executes the query.

This function triggers the following middleware.

- `countDocuments()`

Example:

```
const countQuery = model.where({ 'color': 'black' }).countDocuments();

query.countDocuments({ color: 'black' }).count(callback);

query.countDocuments({ color: 'black' }, callback);

query.where('color', 'black').countDocuments(function(err, count) {
  if (err) return handleError(err);
  console.log('there are %d kittens', count);
});
```

The `countDocuments()` function is similar to `count()`, but there are a few operators that `countDocuments()` does not support. Below are the operators that `count()` supports but `countDocuments()` does not, and the suggested replacement:

- `$where`: `$expr`
- `$near`: `$geoWithin` with `$center`
- `$nearSphere`: `$geoWithin` with `$centerSphere`

Query.prototype.cursor()

Parameters

- [options] «Object»

Returns:

- «QueryCursor»

Returns a wrapper around a mongodb driver cursor. A QueryCursor exposes a Streams3 interface, as well as a `.next()` function.

The `.cursor()` function triggers pre find hooks, but **not** post find hooks.

Example

```
// There are 2 ways to use a cursor. First, as a stream:
Thing.
  find({ name: /^hello/ }).
  cursor().
  on('data', function(doc) { console.log(doc); }).
  on('end', function() { console.log('Done!'); });

// Or you can use `.next()` to manually get the next doc in the stream.
// `.next()` returns a promise, so you can use promises or callbacks.
const cursor = Thing.find({ name: /^hello/ }).cursor();
cursor.next(function(error, doc) {
  console.log(doc);
});

// Because `.next()` returns a promise, you can use co
// to easily iterate through all documents without loading them
// all into memory.
const cursor = Thing.find({ name: /^hello/ }).cursor();
for (let doc = await cursor.next(); doc != null; doc = await cursor.next()) {
  console.log(doc);
}
```

Valid options

- `transform`: optional function which accepts a mongoose document. The return value of the function will be emitted on `data` and returned by `.next()`.

Query.prototype.deleteMany()

Parameters

- [filter] «Object|Query» mongodb selector
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function» optional params are (error, mongooseDeleteResult)

Returns:

- «Query» this

Declare and/or execute this query as a `deleteMany()` operation. Works like remove, except it deletes *every* document that matches `filter` in the collection, regardless of the value of `single`.

This function triggers `deleteMany` middleware.

Example

```
await Character.deleteMany({ name: /Stark/, age: { $gte: 18 } });

// Using callbacks:
Character.deleteMany({ name: /Stark/, age: { $gte: 18 } }, callback);
```

This function calls the MongoDB driver's `Collection#deleteMany()` function. The returned promise resolves to an object that contains 3 properties:

- `ok`: `1` if no errors occurred
- `deletedCount`: the number of documents deleted
- `n`: the number of documents deleted. Equal to `deletedCount`.

Example

```
const res = await Character.deleteMany({ name: /Stark/, age: { $gte: 18 } });
// `0` if no docs matched the filter, number of docs deleted otherwise
res.deletedCount;
```

Query.prototype.deleteOne()

Parameters

- [filter] «Object|Query» mongodb selector
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function» optional params are (error, mongooseDeleteResult)

Returns:

- «Query» this

Declare and/or execute this query as a `deleteOne()` operation. Works like remove, except it deletes at most one document regardless of the `single` option.

This function triggers `deleteOne` middleware.

Example

```
await Character.deleteOne({ name: 'Eddard Stark' });

// Using callbacks:
Character.deleteOne({ name: 'Eddard Stark' }, callback);
```

This function calls the MongoDB driver's `Collection#deleteOne()` function. The returned promise resolves to an object that contains 3 properties:

- `ok`: `1` if no errors occurred
- `deletedCount`: the number of documents deleted
- `n`: the number of documents deleted. Equal to `deletedCount`.

Example

```
const res = await Character.deleteOne({ name: 'Eddard Stark' });
// `1` if MongoDB deleted a doc, `0` if no docs matched the filter `{ name: ... }`
res.deletedCount;
```

Query.prototype.distinct()

Parameters

- [field] «String»
- [filter] «Object|Query»
- [callback] «Function» optional params are (error, arr)

Returns:

- «Query» this

Declares or executes a distinct() operation.

Passing a `callback` executes the query.

This function does not trigger any middleware.

Example

```
distinct(field, conditions, callback)
distinct(field, conditions)
distinct(field, callback)
distinct(field)
distinct(callback)
distinct()
```

Query.prototype.elemMatch()

Parameters

- path «String|Object|Function»
- filter «Object|Function»

Returns:

- «Query» this

Specifies an `$elemMatch` condition

Example

```
query.elemMatch('comment', { author: 'autobot', votes: {$gte: 5}})

query.where('comment').elemMatch({ author: 'autobot', votes: {$gte: 5}})

query.elemMatch('comment', function (elem) {
  elem.where('author').equals('autobot');
  elem.where('votes').gte(5);
})

query.where('comment').elemMatch(function (elem) {
  elem.where({ author: 'autobot' });
  elem.where('votes').gte(5);
})
```

Query.prototype.equals()

Parameters

- val «Object»

Returns:

- «Query» this

Specifies the complementary comparison value for paths specified with `where()`

Example

```
User.where('age').equals(49);

// is the same as

User.where('age', 49);
```

Query.prototype.error()

Parameters

- err «Error|null» if set, `exec()` will fail fast before sending the query to MongoDB

Returns:

- «Query» this

Gets/sets the error flag on this query. If this flag is not null or undefined, the `exec()` promise will reject without executing.

Example:

```
Query().error(); // Get current error value
Query().error(null); // Unset the current error
Query().error(new Error('test')); // `exec()` will resolve with test
Schema.pre('find', function() {
  if (!this.getQuery().userId) {
    this.error(new Error('Not allowed to query without setting userId'));
  }
});
```

Note that query casting runs **after** hooks, so cast errors will override custom errors.

Example:

```
const TestSchema = new Schema({ num: Number });
const TestModel = db.model('Test', TestSchema);
TestModel.find({ num: 'not a number' }).error(new Error('woops')).exec(function(error) {
  // `error` will be a cast error because `num` failed to cast
});
```

Query.prototype.estimatedDocumentCount()

Parameters

- [options] «Object» passed transparently to the MongoDB driver
- [callback] «Function» optional params are (error, count)

Returns:

- «Query» this

Specifies this query as a `estimatedDocumentCount()` query. Faster than using `countDocuments()` for large collections because `estimatedDocumentCount()` uses collection metadata rather than scanning the entire collection.

`estimatedDocumentCount()` does **not** accept a filter. `Model.find({ foo: bar }).estimatedDocumentCount()` is equivalent to `Model.find().estimatedDocumentCount()`

This function triggers the following middleware.

- `estimatedDocumentCount()`

Example:

```
await Model.find().estimatedDocumentCount();
```

Query.prototype.exec()

Parameters

- [operation] «String|Function»
- [callback] «Function» optional params depend on the function being called

Returns:

- «Promise»

Executes the query

Examples:

```
const promise = query.exec();
const promise = query.exec('update');

query.exec(callback);
query.exec('find', callback);
```

Query.prototype.exists()

Parameters

- [path] «String»
- val «Boolean»

Returns:

- «Query» this

Specifies an `$exists` condition

Example

```
// { name: { $exists: true }}
Thing.where('name').exists()
Thing.where('name').exists(true)
Thing.find().exists('name')

// { name: { $exists: false }}
Thing.where('name').exists(false);
Thing.find().exists('name', false);
```

Query.prototype.explain()

Parameters

- [verbose] «String» The verbosity mode. Either 'queryPlanner', 'executionStats', or 'allPlansExecution'. The default is 'queryPlanner'

Returns:

- «Query» this

Sets the `explain` option, which makes this query return detailed execution stats instead of the actual query result. This method is useful for determining what index your queries use.

Calling `query.explain(v)` is equivalent to `query.setOptions({ explain: v })`

Example:

```
const query = new Query();
const res = await query.find({ a: 1 }).explain('queryPlanner');
console.log(res);
```

Query.prototype.find()

Parameters

- [filter] «Object|ObjectId» mongodb selector. If not specified, returns all documents.
- [callback] «Function»

Returns:

- «Query» this

Find all documents that match `selector`. The result will be an array of documents.

If there are too many documents in the result to fit in memory, use `Query.prototype.cursor()`

Example

```
// Using async/await
const arr = await Movie.find({ year: { $gte: 1980, $lte: 1989 } });

// Using callbacks
Movie.find({ year: { $gte: 1980, $lte: 1989 } }, function(err, arr) {});
```

Query.prototype.findOne()

Parameters

- [filter] «Object» mongodb selector
- [projection] «Object» optional fields to return
- [options] «Object» see `setOptions()`
- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Declares the query a findOne operation. When executed, the first found document is passed to the callback.

Passing a `callback` executes the query. The result of the query is a single document.

- *Note:* `conditions` is optional, and if `conditions` is null or undefined, mongoose will send an empty `findOne` command to MongoDB, which will return an arbitrary document. If you're querying by `_id`, use `Model.findById()` instead.

This function triggers the following middleware.

- `findOne()`

Example

```
const query  = Kitten.where({ color: 'white' });
query.findOne(function (err, kitten) {
  if (err) return handleError(err);
  if (kitten) {
    // doc may be null if no document matched
  }
});
```

Query.prototype.findOneAndDelete()

Parameters

- [conditions] «Object»
- [options] «Object»
  - [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
  - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option

- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Issues a MongoDB findOneAndDelete command.

Finds a matching document, removes it, and passes the found document (if any) to the callback. Executes if `callback` is passed.

This function triggers the following middleware.

- `findOneAndDelete()`

This function differs slightly from `Model.findOneAndRemove()` in that `findOneAndRemove()` becomes a MongoDB `findAndModify()` command, as opposed to a `findOneAndDelete()` command. For most mongoose use cases, this distinction is purely pedantic. You should use `findOneAndDelete()` unless you have a good reason not to.

Available options

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `rawResult`: if true, resolves to the raw result from the MongoDB driver

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if `new = true
}
```

Examples

```
A.where().findOneAndDelete(conditions, options, callback) // executes
A.where().findOneAndDelete(conditions, options)  // return Query
A.where().findOneAndDelete(conditions, callback) // executes
A.where().findOneAndDelete(conditions) // returns Query
A.where().findOneAndDelete(callback)   // executes
A.where().findOneAndDelete()           // returns Query
```

Query.prototype.findOneAndRemove()

Parameters

- [conditions] «Object»
- [options] «Object»

- [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
- [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
- [options.strict] «Boolean|String» overwrites the schema's strict mode option

- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Issues a mongodb findAndModify remove command.

Finds a matching document, removes it, passing the found document (if any) to the callback. Executes if `callback` is passed.

This function triggers the following middleware.

- `findOneAndRemove()`

Available options

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `rawResult`: if true, resolves to the raw result from the MongoDB driver

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if `new = true
}
```

Examples

```
A.where().findOneAndRemove(conditions, options, callback) // executes
A.where().findOneAndRemove(conditions, options)  // return Query
A.where().findOneAndRemove(conditions, callback) // executes
A.where().findOneAndRemove(conditions) // returns Query
A.where().findOneAndRemove(callback)   // executes
A.where().findOneAndRemove()           // returns Query
```

Query.prototype.findOneAndReplace()

Parameters

- [filter] «Object»

- [replacement] «Object»
- [options] «Object»
  - [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
  - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.new=false] «Boolean» By default, `findOneAndUpdate()` returns the document as it was **before** `update` was applied. If you set `new: true`, `findOneAndUpdate()` will instead give you the object after `update` was applied.
  - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()` and the Mongoose lean tutorial.
  - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.
  - [options.returnOriginal=null] «Boolean» An alias for the `new` option. `returnOriginal: false` is equivalent to `new: true`.
- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Issues a MongoDB findOneAndReplace command.

Finds a matching document, removes it, and passes the found document (if any) to the callback. Executes if `callback` is passed.

This function triggers the following middleware.

- `findOneAndReplace()`

Available options

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `rawResult`: if true, resolves to the raw result from the MongoDB driver

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if `new = true
}
```

Examples

```
A.where().findOneAndReplace(filter, replacement, options, callback); // executes
A.where().findOneAndReplace(filter, replacement, options); // return Query
A.where().findOneAndReplace(filter, replacement, callback); // executes
A.where().findOneAndReplace(filter); // returns Query
A.where().findOneAndReplace(callback); // executes
A.where().findOneAndReplace(); // returns Query
```

Query.prototype.findOneAndUpdate()

Parameters

- [filter] «Object|Query»
- [doc] «Object»
- [options] «Object»
  - [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
  - [options.new=false] «Boolean» By default, `findOneAndUpdate()` returns the document as it was **before** `update` was applied. If you set `new: true`, `findOneAndUpdate()` will instead give you the object after `update` was applied.
  - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()` and the Mongoose lean tutorial.
  - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.

- [options.returnOriginal=null] «Boolean» An alias for the `new` option. `returnOriginal: false` is equivalent to `new: true`.

- [callback] «Function» optional params are (error, doc), *unless* `rawResult` is used, in which case params are (error, writeOpResult)

Returns:

- «Query» this

Issues a mongodb findAndModify update command.

Finds a matching document, updates it according to the `update` arg, passing any `options`, and returns the found document (if any) to the callback. The query executes if `callback` is passed.

This function triggers the following middleware.

- `findOneAndUpdate()`

Available options

- `new`: bool - if true, return the modified document rather than the original. defaults to false (changed in 4.0)
- `upsert`: bool - creates the object if it doesn't exist. defaults to false.
- `fields`: {Object|String} - Field selection. Equivalent to `.select(fields).findOneAndUpdate()`
- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `runValidators`: if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert`: `true` by default. If `setDefaultsOnInsert` and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created.
- `rawResult`: if true, returns the raw result from the MongoDB driver

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if `new = true
}
```

Examples

```
query.findOneAndUpdate(conditions, update, options, callback) // executes
query.findOneAndUpdate(conditions, update, options)  // returns Query
query.findOneAndUpdate(conditions, update, callback) // executes
query.findOneAndUpdate(conditions, update)           // returns Query
query.findOneAndUpdate(update, callback)             // returns Query
query.findOneAndUpdate(update)                       // returns Query
query.findOneAndUpdate(callback)                     // executes
query.findOneAndUpdate()                             // returns Query
```

Query.prototype.geometry()

Parameters

- object «Object» Must contain a `type` property which is a String and a `coordinates` property which is an Array. See the examples.

Returns:

- «Query» this

Specifies a `$geometry` condition

Example

```
const polyA = [[[ 10, 20 ], [ 10, 40 ], [ 30, 40 ], [ 30, 20 ]]]
query.where('loc').within().geometry({ type: 'Polygon', coordinates: polyA })

// or
const polyB = [[ 0, 0 ], [ 1, 1 ]]
query.where('loc').within().geometry({ type: 'LineString', coordinates: polyB })

// or
const polyC = [ 0, 0 ]
query.where('loc').within().geometry({ type: 'Point', coordinates: polyC })

// or
query.where('loc').intersects().geometry({ type: 'Point', coordinates: polyC })
```

The argument is assigned to the most recent path passed to `where()`.

NOTE:

`geometry()` **must** come after either `intersects()` or `within()`.

The `object` argument must contain `type` and `coordinates` properties. - type {String} - coordinates {Array}

Query.prototype.get()

Parameters

- path «String|Object» path or object of key/value pairs to get

Returns:

- «Query» this

For update operations, returns the value of a path in the update's `$set`. Useful for writing getters/setters that can work with both update operations and `save()`.

Example:

```
const query = Model.updateOne({}, { $set: { name: 'Jean-Luc Picard' } });
query.get('name'); // 'Jean-Luc Picard'
```

Query.prototype.getFilter()

Returns:

- «Object» current query filter

Returns the current query filter (also known as conditions) as a POJO.

Example:

```
const query = new Query();
query.find({ a: 1 }).where('b').gt(2);
query.getFilter(); // { a: 1, b: { $gt: 2 } }
```

Query.prototype.getOptions()

Returns:

- «Object» the options

Gets query options.

Example:

```
const query = new Query();
query.limit(10);
query.setOptions({ maxTimeMS: 1000 })
query.getOptions(); // { limit: 10, maxTimeMS: 1000 }
```

Query.prototype.getPopulatedPaths()

Returns:

- «Array» an array of strings representing populated paths

Gets a list of paths to be populated by this query

Example:

```
bookSchema.pre('findOne', function() {
   let keys = this.getPopulatedPaths(); // ['author']
 });
 ...
 Book.findOne({}).populate('author');
```

Example:

```
// Deep populate
 const q = L1.find().populate({
   path: 'level2',
   populate: { path: 'level3' }
 });
 q.getPopulatedPaths(); // ['level2', 'level2.level3']
```

Query.prototype.getQuery()

Returns:

- «Object» current query filter

Returns the current query filter. Equivalent to `getFilter()`.

You should use `getFilter()` instead of `getQuery()` where possible. `getQuery()` will likely be deprecated in a future release.

Example:

```
const query = new Query();
query.find({ a: 1 }).where('b').gt(2);
query.getQuery(); // { a: 1, b: { $gt: 2 } }
```

Query.prototype.getUpdate()

Returns:

- «Object» current update operations

Returns the current update operations as a JSON object.

Example:

```
const query = new Query();
query.update({}, { $set: { a: 5 } });
query.getUpdate(); // { $set: { a: 5 } }
```

Query.prototype.gt()

Parameters

- [path] «String»
- val «Number»

Specifies a `$gt` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Example

```
Thing.find().where('age').gt(21)

// or
Thing.find().gt('age', 21)
```

Query.prototype.gte()

Parameters

- [path] «String»
- val «Number»

Specifies a `$gte` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.hint()

Parameters

- val «Object» a hint object

Returns:

- «Query» this

Sets query hints.

Example

```
query.hint({ indexA: 1, indexB: -1})
```

Note

Cannot be used with `distinct()`

Query.prototype.in()

Parameters

- [path] «String»
- val «Array»

Specifies an `$in` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.intersects()

Parameters

- [arg] «Object»

Returns:

- «Query» this

Declares an intersects query for `geometry()`.

Example

```
query.where('path').intersects().geometry({
    type: 'LineString'
  , coordinates: [[180.0, 11.0], [180, 9.0]]
})

query.where('path').intersects({
    type: 'LineString'
  , coordinates: [[180.0, 11.0], [180, 9.0]]
})
```

NOTE:

MUST be used after `where()`.

NOTE:

In Mongoose 3.7, `intersects` changed from a getter to a function. If you need the old syntax, use this.

Query.prototype.j()

Parameters

- val «boolean»

Returns:

- «Query» this

Requests acknowledgement that this operation has been persisted to MongoDB's on-disk journal.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndReplace()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern.j` option

Example:

```
await mongoose.model('Person').deleteOne({ name: 'Ned Stark' }).j(true);
```

Query.prototype.lean()

Parameters

- bool «Boolean|Object» defaults to true

Returns:

- «Query» this

Sets the lean option.

Documents returned from queries with the `lean` option enabled are plain javascript objects, not Mongoose Documents. They have no `save` method, getters/setters, virtuals, or other Mongoose features.

Example:

```
new Query().lean() // true
new Query().lean(true)
new Query().lean(false)

const docs = await Model.find().lean();
docs[0] instanceof mongoose.Document; // false
```

Lean is great for high-performance, read-only cases, especially when combined with cursors.

If you need virtuals, getters/setters, or defaults with `lean()`, you need to use a plugin. See:

- mongoose-lean-virtuals
- mongoose-lean-getters
- mongoose-lean-defaults

Query.prototype.limit()

Parameters

- val «Number»

Specifies the maximum number of documents the query will return.

Example

```
query.limit(20)
```

Note

Cannot be used with `distinct()`

Query.prototype.lt()

Parameters

- [path] «String»
- val «Number»

Specifies a `$lt` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.lte()

Parameters

- [path] «String»
- val «Number»

Specifies a `$lte` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.map()

Parameters

- fn «Function» function to run to transform the query result

Returns:

- «Query» this

Runs a function `fn` and treats the return value of `fn` as the new value for the query to resolve to.

Any functions you pass to `map()` will run **after** any post hooks.

Example:

```
const res = await MyModel.findOne().transform(res => {
  // Sets a `loadedAt` property on the doc that tells you the time the
  // document was loaded.
  return res == null ?
    res :
    Object.assign(res, { loadedAt: new Date() });
});
```

Query.prototype.maxDistance()

Parameters

- [path] «String»
- val «Number»

Specifies a `maxDistance` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.maxScan()

Parameters

- val «Number»

Specifies the maxScan option.

Example

```
query.maxScan(100)
```

Note

Cannot be used with `distinct()`

Query.prototype.maxTimeMS()

Parameters

- [ms] «Number» The number of milliseconds

Returns:

- «Query» this

Sets the maxTimeMS option. This will tell the MongoDB server to abort if the query or write op has been running for more than `ms` milliseconds.

Calling `query.maxTimeMS(v)` is equivalent to `query.setOptions({ maxTimeMS: v })`

Example:

```
const query = new Query();
// Throws an error 'operation exceeded time limit' as long as there's
// >= 1 doc in the queried collection
const res = await query.find({ $where: 'sleep(1000) || true' }).maxTimeMS(100);
```

Query.prototype.maxscan()

DEPRECATED Alias of `maxScan`

Query.prototype.merge()

Parameters

- source «Query|Object»

Returns:

- «Query» this

Merges another Query or conditions object into this one.

When a Query is passed, conditions, field selection and options are merged.

Query.prototype.mod()

Parameters

- [path] «String»
- val «Array» must be of length 2, first element is `divisor`, 2nd element is `remainder`.

Returns:

- «Query» this

Specifies a `$mod` condition, filters documents for documents whose `path` property is a number that is equal to `remainder` modulo `divisor`.

Example

```
// All find products whose inventory is odd
Product.find().mod('inventory', [2, 1]);
Product.find().where('inventory').mod([2, 1]);
// This syntax is a little strange, but supported.
Product.find().where('inventory').mod(2, 1);
```

Query.prototype.model

Type:

- «property»

The model this query is associated with.

Example:

```
const q = MyModel.find();
q.model === MyModel; // true
```

Query.prototype.mongooseOptions()

Parameters

- options «Object» if specified, overwrites the current options

Returns:

- «Object» the options

Getter/setter around the current mongoose-specific options for this query Below are the current Mongoose-specific options.

- `populate`: an array representing what paths will be populated. Should have one entry for each call to `Query.prototype.populate()`
- `lean`: if truthy, Mongoose will not hydrate any documents that are returned from this query. See `Query.prototype.lean()` for more information.
- `strict`: controls how Mongoose handles keys that aren't in the schema for updates. This option is `true` by default, which means Mongoose will silently strip any paths in the update that aren't in the schema. See the `strict` mode docs for more information.
- `strictQuery`: controls how Mongoose handles keys that aren't in the schema for the query `filter`. This option is `false` by default for backwards compatibility, which means Mongoose will allow `Model.find({ foo: 'bar' })` even if `foo` is not in the schema. See the `strictQuery` docs for more information.
- `nearSphere`: use `$nearSphere` instead of `near()`. See the `Query.prototype.nearSphere()` docs

Mongoose maintains a separate object for internal options because Mongoose sends `Query.prototype.options` to the MongoDB server, and the above options are not relevant for the MongoDB server.

Query.prototype.ne()

Parameters

- [path] «String»
- val «any»

Specifies a `$ne` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.near()

Parameters

- [path] «String»
- val «Object»

Returns:

- «Query» this

Specifies a `$near` or `$nearSphere` condition

These operators return documents sorted by distance.

Example

```
query.where('loc').near({ center: [10, 10] });
query.where('loc').near({ center: [10, 10], maxDistance: 5 });
query.where('loc').near({ center: [10, 10], maxDistance: 5, spherical: true });
query.near('loc', { center: [10, 10], maxDistance: 5 });
```

Query.prototype.nearSphere()

DEPRECATED Specifies a `$nearSphere` condition

Example

```
query.where('loc').nearSphere({ center: [10, 10], maxDistance: 5 });
```

Deprecated. Use `query.near()` instead with the `spherical` option set to `true`.

Example

```
query.where('loc').near({ center: [10, 10], spherical: true });
```

Query.prototype.nin()

Parameters

- [path] «String»
- val «Array»

Specifies an `$nin` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.nor()

Parameters

- array «Array» array of conditions

Returns:

- «Query» this

Specifies arguments for a `$nor` condition.

Example

```
query.nor([{ color: 'green' }, { status: 'ok' }])
```

Query.prototype.or()

Parameters

- array «Array» array of conditions

Returns:

- «Query» this

Specifies arguments for an `$or` condition.

Example

```
query.or([{ color: 'red' }, { status: 'emergency' }])
```

Query.prototype.orFail()

Parameters

- [err] «Function|Error» optional error to throw if no docs match `filter`. If not specified, `orFail()` will throw a `DocumentNotFoundError`

Returns:

- «Query» this

Make this query throw an error if no documents match the given `filter`. This is handy for integrating with async/await, because `orFail()` saves you an extra `if` statement to check if no document was found.

Example:

```
// Throws if no doc returned
await Model.findOne({ foo: 'bar' }).orFail();

// Throws if no document was updated
await Model.updateOne({ foo: 'bar' }, { name: 'test' }).orFail();

// Throws "No docs found!" error if no docs match `{ foo: 'bar' }`
await Model.find({ foo: 'bar' }).orFail(new Error('No docs found!'));

// Throws "Not found" error if no document was found
await Model.findOneAndUpdate({ foo: 'bar' }, { name: 'test' }).
  orFail(() => Error('Not found'));
```

Query.prototype.polygon()

Parameters

- [path] «String|Array»
    - [coordinatePairs...] «Array|Object»

Returns:

- «Query» this

Specifies a `$polygon` condition

Example

```
query.where('loc').within().polygon([10,20], [13, 25], [7,15])
query.polygon('loc', [10,20], [13, 25], [7,15])
```

Query.prototype.populate()

Parameters

- path «Object|String» either the path to populate or an object specifying all parameters
- [select] «Object|String» Field selection for the population query
- [model] «Model» The model you wish to use for population. If not specified, populate will look up the model by the name in the Schema's `ref` field.

- [match] «Object» Conditions for the population query
- [options] «Object» Options for the population query (sort, etc)
  - [options.path=null] «String» The path to populate.
  - [options.retainNullValues=false] «boolean» by default, Mongoose removes null and undefined values from populated arrays. Use this option to make `populate()` retain `null` and `undefined` array entries.
  - [options.getters=false] «boolean» if true, Mongoose will call any getters defined on the `localField`. By default, Mongoose gets the raw value of `localField`. For example, you would need to set this option to `true` if you wanted to add a `lowercase` getter to your `localField`.
  - [options.clone=false] «boolean» When you do `BlogPost.find().populate('author')`, blog posts with the same author will share 1 copy of an `author` doc. Enable this option to make Mongoose clone populated docs before assigning them.
  - [options.match=null] «Object|Function» Add an additional filter to the populate query. Can be a filter object containing MongoDB query syntax, or a function that returns a filter object.
  - [options.transform=null] «Function» Function that Mongoose will call on every populated document that allows you to transform the populated document.
  - [options.options=null] «Object» Additional options like `limit` and `lean`.

Returns:

- «Query» this

Specifies paths which should be populated with other documents.

Example:

```
let book = await Book.findOne().populate('authors');
book.title; // 'Node.js in Action'
book.authors[0].name; // 'TJ Holowaychuk'
book.authors[1].name; // 'Nathan Rajlich'

let books = await Book.find().populate({
  path: 'authors',
  // `match` and `sort` apply to the Author model,
  // not the Book model. These options do not affect
  // which documents are in `books`, just the order and
  // contents of each book document's `authors`.
  match: { name: new RegExp('.*h.*', 'i') },
  sort: { name: -1 }
});
books[0].title; // 'Node.js in Action'
// Each book's `authors` are sorted by name, descending.
books[0].authors[0].name; // 'TJ Holowaychuk'
books[0].authors[1].name; // 'Marc Harter'

books[1].title; // 'Professional AngularJS'
```

```
// Empty array, no authors' name has the letter 'h'
books[1].authors; // []
```

Paths are populated after the query executes and a response is received. A separate query is then executed for each path specified for population. After a response for each query has also been returned, the results are passed to the callback.

Query.prototype.post()

Parameters

- fn «Function»

Returns:

- «Promise»

Add post middleware to this query instance. Doesn't affect other queries.

Example:

```
const q1 = Question.find({ answer: 42 });
q1.post(function middleware() {
  console.log(this.getFilter());
});
await q1.exec(); // Prints "{ answer: 42 }"

// Doesn't print anything, because `middleware()` is only
// registered on `q1`.
await Question.find({ answer: 42 });
```

Query.prototype.pre()

Parameters

- fn «Function»

Returns:

- «Promise»

Add pre middleware to this query instance. Doesn't affect other queries.

Example:

```
const q1 = Question.find({ answer: 42 });
q1.pre(function middleware() {
  console.log(this.getFilter());
```

```
});
await q1.exec(); // Prints "{ answer: 42 }"

// Doesn't print anything, because `middleware()` is only
// registered on `q1`.
await Question.find({ answer: 42 });
```

Query.prototype.projection()

Parameters

- arg «Object|null»

Returns:

- «Object» the current projection

Get/set the current projection (AKA fields). Pass `null` to remove the current projection.

Unlike `projection()`, the `select()` function modifies the current projection in place. This function overwrites the existing projection.

Example:

```
const q = Model.find();
q.projection(); // null

q.select('a b');
q.projection(); // { a: 1, b: 1 }

q.projection({ c: 1 });
q.projection(); // { c: 1 }

q.projection(null);
q.projection(); // null
```

Query.prototype.read()

Parameters

- pref «String» one of the listed preference options or aliases
- [tags] «Array» optional tags for this query

Returns:

- «Query» this

Determines the MongoDB nodes from which to read.

Preferences:

```
primary - (default) Read from primary only. Operations will produce an error if primary is unavail
secondary            Read from secondary if available, otherwise error.
primaryPreferred     Read from primary if available, otherwise a secondary.
secondaryPreferred   Read from a secondary if available, otherwise read from the primary.
nearest              All operations read from among the nearest candidates, but unlike other modes
```

Aliases

```
p   primary
pp  primaryPreferred
s   secondary
sp  secondaryPreferred
n   nearest
```

Example:

```
new Query().read('primary')
new Query().read('p')  // same as primary

new Query().read('primaryPreferred')
new Query().read('pp') // same as primaryPreferred

new Query().read('secondary')
new Query().read('s')  // same as secondary

new Query().read('secondaryPreferred')
new Query().read('sp') // same as secondaryPreferred

new Query().read('nearest')
new Query().read('n')  // same as nearest

// read from secondaries with matching tags
new Query().read('s', [{ dc:'sf', s: 1 },{ dc:'ma', s: 2 }])
```

Read more about how to use read preferences here and here.

Query.prototype.readConcern()

Parameters

- level «String» one of the listed read concern level or their aliases

Returns:

- «Query» this

Sets the readConcern option for the query.

Example:

```
new Query().readConcern('local')
new Query().readConcern('l')  // same as local

new Query().readConcern('available')
new Query().readConcern('a')  // same as available

new Query().readConcern('majority')
new Query().readConcern('m')  // same as majority

new Query().readConcern('linearizable')
new Query().readConcern('lz') // same as linearizable

new Query().readConcern('snapshot')
new Query().readConcern('s')  // same as snapshot
```

Read Concern Level:

```
local         MongoDB 3.2+ The query returns from the instance with no guarantee guarantee that th
available     MongoDB 3.6+ The query returns from the instance with no guarantee guarantee that th
majority      MongoDB 3.2+ The query returns the data that has been acknowledged by a majority of
linearizable  MongoDB 3.4+ The query returns data that reflects all successful majority-acknowledg
snapshot      MongoDB 4.0+ Only available for operations within multi-document transactions. Upon
```

Aliases

```
l   local
a   available
m   majority
lz  linearizable
s   snapshot
```

Read more about how to use read concern here.

Query.prototype.regex()

Parameters

- [path] «String»
- val «String|RegExp»

Specifies a `$regex` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.remove()

Parameters

- [filter] «Object|Query» mongodb selector
- [callback] «Function» optional params are (error, mongooseDeleteResult)

Returns:

- «Query» this

Declare and/or execute this query as a remove() operation. `remove()` is deprecated, you should use `deleteOne()` or `deleteMany()` instead.

This function does not trigger any middleware

Example

```
Character.remove({ name: /Stark/ }, callback);
```

This function calls the MongoDB driver's `Collection#remove()` function. The returned promise resolves to an object that contains 3 properties:

- `ok`: `1` if no errors occurred
- `deletedCount`: the number of documents deleted
- `n`: the number of documents deleted. Equal to `deletedCount`.

Example

```
const res = await Character.remove({ name: /Stark/ });
// Number of docs deleted
res.deletedCount;
```

Note

Calling `remove()` creates a Mongoose query, and a query does not execute until you either pass a callback, call `Query#then()`, or call `Query#exec()`.

```
// not executed
const query = Character.remove({ name: /Stark/ });

// executed
Character.remove({ name: /Stark/ }, callback);
Character.remove({ name: /Stark/ }).remove(callback);

// executed without a callback
Character.exec();
```

Query.prototype.replaceOne()

Parameters

- [filter] «Object»
- [doc] «Object» the update command
- [options] «Object»
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.
- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as a replaceOne() operation. Same as `update()`, except MongoDB will replace the existing document and will not accept any atomic operators (`$set`, etc.)

Note replaceOne will *not* fire update middleware. Use `pre('replaceOne')` and `post('replaceOne')` instead.

Example:

```
const res = await Person.replaceOne({ _id: 24601 }, { name: 'Jean Valjean' });
res.n; // Number of documents matched
res.nModified; // Number of documents modified
```

This function triggers the following middleware.

- `replaceOne()`

Query.prototype.select()

Parameters

- arg «Object|String|Array<String>»

Returns:

- «Query» this

Specifies which document fields to include or exclude (also known as the query "projection")

When using string syntax, prefixing a path with `-` will flag that path as excluded. When a path does not have the `-` prefix, it is included. Lastly, if a path is prefixed with `+`, it forces inclusion of the path, which is useful for paths excluded at the schema level.

A projection *must* be either inclusive or exclusive. In other words, you must either list the fields to include (which excludes all others), or list the fields to exclude (which implies all other fields are included). The `_id` field is the only exception because MongoDB includes it by default.

Example

```
// include a and b, exclude other fields
query.select('a b');
// Equivalent syntaxes:
query.select(['a', 'b']);
query.select({ a: 1, b: 1 });

// exclude c and d, include other fields
query.select('-c -d');

// Use `+` to override schema-level `select: false` without making the
// projection inclusive.
const schema = new Schema({
  foo: { type: String, select: false },
  bar: String
});
// ...
query.select('+foo'); // Override foo's `select: false` without excluding `bar`

// or you may use object notation, useful when
// you have keys already prefixed with a "-"
query.select({ a: 1, b: 1 });
query.select({ c: 0, d: 0 });
```

Query.prototype.selected()

Returns:

- «Boolean»

Determines if field selection has been made.

Query.prototype.selectedExclusively()

Returns:

- «Boolean»

Determines if exclusive field selection has been made.

```
query.selectedExclusively() // false
query.select('-name')
query.selectedExclusively() // true
query.selectedInclusively() // false
```

Query.prototype.selectedInclusively()

Returns:

- «Boolean»

Determines if inclusive field selection has been made.

```
query.selectedInclusively() // false
query.select('name')
query.selectedInclusively() // true
```

Query.prototype.session()

Parameters

- [session] «ClientSession» from `await conn.startSession()`

Returns:

- «Query» this

Sets the MongoDB session associated with this query. Sessions are how you mark a query as part of a transaction.

Calling `session(null)` removes the session from this query.

Example:

```
const s = await mongoose.startSession();
await mongoose.model('Person').findOne({ name: 'Axl Rose' }).session(s);
```

Query.prototype.set()

Parameters

- path «String|Object» path or object of key/value pairs to set
- [val] «Any» the value to set

Returns:

- «Query» this

Adds a `$set` to this query's update without changing the operation. This is useful for query middleware so you can add an update regardless of whether you use `updateOne()`, `updateMany()`, `findOneAndUpdate()`, etc.

Example:

```
// Updates `{ $set: { updatedAt: new Date() } }`
new Query().updateOne({}, {}).set('updatedAt', new Date());
new Query().updateMany({}, {}).set({ updatedAt: new Date() });
```

Query.prototype.setOptions()

Parameters

- options «Object»

Returns:

- «Query» this

Sets query options. Some options only make sense for certain operations.

Options:

The following options are only for `find()`:

- tailable
- sort
- limit
- skip
- allowDiskUse
- batchSize
- readPreference
- hint
- comment
- snapshot
- maxscan

The following options are only for write operations: `update()`, `updateOne()`, `updateMany()`, `replaceOne()`, `findOneAndUpdate()`, and `findByIdAndUpdate()`:

- upsert
- writeConcern
- timestamps: If `timestamps` is set in the schema, set this option to `false` to skip timestamps for that particular update. Has no effect if `timestamps` is not enabled in the schema options.
- omitUndefined: delete any properties whose value is `undefined` when casting an update. In other words, if this is set, Mongoose will delete `baz` from the update in `Model.updateOne({}, { foo: 'bar', baz: undefined })` before sending the update to the server.
- overwriteDiscriminatorKey: allow setting the discriminator key in the update. Will use the correct discriminator schema if the update changes the discriminator key.

The following options are only for `find()`, `findOne()`, `findById()`, `findOneAndUpdate()`, and `findByIdAndUpdate()`:

- lean
- populate
- projection
- sanitizeProjection

The following options are only for all operations **except** `update()`, `updateOne()`, `updateMany()`, `remove()`, `deleteOne()`, and `deleteMany()`:

- maxTimeMS

The following options are for `findOneAndUpdate()` and `findOneAndRemove()`

- rawResult

The following options are for all operations

- strict
- collation
- session
- explain

Query.prototype.setQuery()

Parameters

- new «Object» query conditions

Returns:

- «undefined»

Sets the query conditions to the provided JSON object.

Example:

```
const query = new Query();
query.find({ a: 1 })
query.setQuery({ a: 2 });
query.getQuery(); // { a: 2 }
```

Query.prototype.setUpdate()

Parameters

- new «Object» update operation

Returns:

- «undefined»

Sets the current update operation to new value.

Example:

```
const query = new Query();
query.update({}, { $set: { a: 5 } });
query.setUpdate({ $set: { b: 6 } });
query.getUpdate(); // { $set: { b: 6 } }
```

Query.prototype.size()

Parameters

- [path] «String»
- val «Number»

Specifies a `$size` query condition.

When called with one argument, the most recent path passed to `where()` is used.

Example

```
const docs = await MyModel.where('tags').size(0).exec();
assert(Array.isArray(docs));
console.log('documents with 0 tags', docs);
```

Query.prototype.skip()

Parameters

- val «Number»

Specifies the number of documents to skip.

Example

```
query.skip(100).limit(20)
```

Note

Cannot be used with `distinct()`

Query.prototype.slaveOk()

Parameters

- v «Boolean» defaults to true

Returns:

- «Query» this

DEPRECATED Sets the slaveOk option.

Deprecated in MongoDB 2.2 in favor of read preferences.

Example:

```
query.slaveOk() // true
query.slaveOk(true)
query.slaveOk(false)
```

Query.prototype.slice()

Parameters

- [path] «String»
- val «Number» number/range of elements to slice

Returns:

- «Query» this

Specifies a `$slice` projection for an array.

Example

```
query.slice('comments', 5)
query.slice('comments', -5)
query.slice('comments', [10, 5])
query.where('comments').slice(5)
query.where('comments').slice([-10, 5])
```

Query.prototype.snapshot()

Returns:

- «Query» this

Specifies this query as a `snapshot` query.

Example

```
query.snapshot() // true
query.snapshot(true)
query.snapshot(false)
```

Note

Cannot be used with `distinct()`

Query.prototype.sort()

Parameters

- arg «Object|String»

Returns:

- «Query» this

Sets the sort order

If an object is passed, values allowed are `asc`, `desc`, `ascending`, `descending`, `1`, and `-1`.

If a string is passed, it must be a space delimited list of path names. The sort order of each path is ascending unless the path name is prefixed with `-` which will be treated as descending.

Example

```
// sort by "field" ascending and "test" descending
query.sort({ field: 'asc', test: -1 });

// equivalent
query.sort('field -test');
```

Note

Cannot be used with `distinct()`

Query.prototype.tailable()

Parameters

- bool «Boolean» defaults to true
- [opts] «Object» options to set
  - [opts.numberOfRetries] «Number» if cursor is exhausted, retry this many times before giving up
  - [opts.tailableRetryInterval] «Number» if cursor is exhausted, wait this many milliseconds before retrying

Sets the tailable option (for use with capped collections).

Example

```
query.tailable() // true
query.tailable(true)
query.tailable(false)
```

Note

Cannot be used with `distinct()`

Query.prototype.then()

Parameters

- [resolve] «Function»
- [reject] «Function»

Returns:

- «Promise»

Executes the query returning a `Promise` which will be resolved with either the doc(s) or rejected with the error.

More about `then()` in JavaScript.

Query.prototype.toConstructor()

Returns:

- «Query» subclass-of-Query

Converts this query to a customized, reusable query constructor with all arguments and options retained.

Example

```
// Create a query for adventure movies and read from the primary
// node in the replica-set unless it is down, in which case we'll
// read from a secondary node.
const query = Movie.find({ tags: 'adventure' }).read('primaryPreferred');

// create a custom Query constructor based off these settings
const Adventure = query.toConstructor();

// Adventure is now a subclass of mongoose.Query and works the same way but with the
// default query parameters and options set.
Adventure().exec(callback)

// further narrow down our query results while still using the previous settings
Adventure().where({ name: /^Life/ }).exec(callback);

// since Adventure is a stand-alone constructor we can also add our own
// helper methods and getters without impacting global queries
Adventure.prototype.startsWith = function (prefix) {
  this.where({ name: new RegExp('^' + prefix) })
  return this;
}
Object.defineProperty(Adventure.prototype, 'highlyRated', {
  get: function () {
    this.where({ rating: { $gt: 4.5 }});
    return this;
  }
})
Adventure().highlyRated.startsWith('Life').exec(callback)
```

Query.prototype.update()

Parameters

- [filter] «Object»
- [doc] «Object» the update command
- [options] «Object»
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.
- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as an update() operation.

All paths passed that are not atomic operations will become `$set` ops.

This function triggers the following middleware.

- `update()`

Example

```
Model.where({ _id: id }).update({ title: 'words' })

// becomes

Model.where({ _id: id }).update({ $set: { title: 'words' }})
```

Valid options:

- `upsert` (boolean) whether to create the doc if it doesn't match (false)
- `multi` (boolean) whether multiple documents should be updated (false)
- `runValidators` (boolean) if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` (boolean) `true` by default. If `setDefaultsOnInsert` and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created.
- `strict` (boolean) overrides the `strict` option for this update
- `read`
- `writeConcern`

Note

Passing an empty object `{}` as the doc will result in a no-op. The update operation will be ignored and the callback executed without sending the command to MongoDB.

Note

The operation is only executed when a callback is passed. To force execution without a callback, we must first call update() and then execute it by using the `exec()` method.

```
const q = Model.where({ _id: id });
q.update({ $set: { name: 'bob' }}).update(); // not executed

q.update({ $set: { name: 'bob' }}).exec(); // executed

// keys that are not [atomic](https://docs.mongodb.com/manual/tutorial/model-data-for-atomic-opera
// this executes the same command as the previous example.
q.update({ name: 'bob' }).exec();

// multi updates
Model.where()
     .update({ name: /^match/ }, { $set: { arr: [] }}, { multi: true }, callback)

// more multi updates
Model.where()
     .setOptions({ multi: true })
     .update({ $set: { arr: [] }}, callback)

// single update by default
Model.where({ email: 'address@example.com' })
     .update({ $inc: { counter: 1 }}, callback)
```

API summary

```
update(filter, doc, options, cb) // executes
update(filter, doc, options)
update(filter, doc, cb) // executes
update(filter, doc)
update(doc, cb) // executes
update(doc)
update(cb) // executes
update(true) // executes
update()
```

Query.prototype.updateMany()

Parameters

- [filter] «Object»
- [update] «Object|Array» the update command

- [options] «Object»
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.
- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as an updateMany() operation. Same as `update()`, except MongoDB will update *all* documents that match `filter` (as opposed to just the first one) regardless of the value of the `multi` option.

Note updateMany will *not* fire update middleware. Use `pre('updateMany')` and `post('updateMany')` instead.

Example:

```
const res = await Person.updateMany({ name: /Stark$/ }, { isDeleted: true });
res.n; // Number of documents matched
res.nModified; // Number of documents modified
```

This function triggers the following middleware.

- `updateMany()`

Query.prototype.updateOne()

Parameters

- [filter] «Object»
- [update] «Object|Array» the update command
- [options] «Object»
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option

- [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
- [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
- [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.

- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as an updateOne() operation. Same as `update()`, except it does not support the `multi` option.

- MongoDB will update *only* the first document that matches `filter` regardless of the value of the `multi` option.
- Use `replaceOne()` if you want to overwrite an entire document rather than using atomic operators like `$set`.

Note updateOne will *not* fire update middleware. Use `pre('updateOne')` and `post('updateOne')` instead.

Example:

```
const res = await Person.updateOne({ name: 'Jean-Luc Picard' }, { ship: 'USS Enterprise' });
res.n; // Number of documents matched
res.nModified; // Number of documents modified
```

This function triggers the following middleware.

- `updateOne()`

Query.prototype.use$geoWithin

Type:

- «property»

Flag to opt out of using `$geoWithin`.

```
mongoose.Query.use$geoWithin = false;
```

MongoDB 2.4 deprecated the use of `$within`, replacing it with `$geoWithin`. Mongoose uses `$geoWithin` by default (which is 100% backward compatible with `$within`). If you are running an older version of MongoDB, set this flag to `false` so your `within()` queries continue to work.

Query.prototype.w()

Parameters

- val «String|number» 0 for fire-and-forget, 1 for acknowledged by one server, 'majority' for majority of the replica set, or any of the more advanced options.

Returns:

- «Query» this

Sets the specified number of `mongod` servers, or tag set of `mongod` servers, that must acknowledge this write before this write is considered successful.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndReplace()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern.w` option

Example:

```
// The 'majority' option means the `deleteOne()` promise won't resolve
// until the `deleteOne()` has propagated to the majority of the replica set
await mongoose.model('Person').
  deleteOne({ name: 'Ned Stark' }).
  w('majority');
```

Query.prototype.where()

Parameters

- [path] «String|Object»
- [val] «any»

Returns:

- «Query» this

Specifies a `path` for use with chaining.

Example

```
// instead of writing:
User.find({age: {$gte: 21, $lte: 65}}, callback);

// we can instead write:
User.where('age').gte(21).lte(65);

// passing query conditions is permitted
User.find().where({ name: 'vonderful' })

// chaining
User
.where('age').gte(21).lte(65)
.where('name', /^vonderful/i)
.where('friends').slice(10)
.exec(callback)
```

Query.prototype.within()

Returns:

- «Query» this

Defines a `$within` or `$geoWithin` argument for geo-spatial queries.

Example

```
query.where(path).within().box()
query.where(path).within().circle()
query.where(path).within().geometry()

query.where('loc').within({ center: [50,50], radius: 10, unique: true, spherical: true });
query.where('loc').within({ box: [[40.73, -73.9], [40.7, -73.988]] });
query.where('loc').within({ polygon: [[],[],[],[]] });

query.where('loc').within([], [], []) // polygon
query.where('loc').within([], []) // box
query.where('loc').within({ type: 'LineString', coordinates: [...] }); // geometry
```

MUST be used after `where()`.

NOTE:

As of Mongoose 3.7, `$geoWithin` is always used for queries. To change this behavior, see Query.use$geoWithin.

NOTE:

In Mongoose 3.7, `within` changed from a getter to a function. If you need the old syntax, use this.

Query.prototype.writeConcern()

Parameters

- writeConcern «Object» the write concern value to set

Returns:

- «Query» this

Sets the 3 write concern parameters for this query

- `w`: Sets the specified number of `mongod` servers, or tag set of `mongod` servers, that must acknowledge this write before this write is considered successful.
- `j`: Boolean, set to `true` to request acknowledgement that this operation has been persisted to MongoDB's on-disk journal.
- `wtimeout`: If `w > 1`, the maximum amount of time to wait for this write to propagate through the replica set before this operation fails. The default is `0`, which means no timeout.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndReplace()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern` option

Example:

```
// The 'majority' option means the `deleteOne()` promise won't resolve
// until the `deleteOne()` has propagated to the majority of the replica set
await mongoose.model('Person').
  deleteOne({ name: 'Ned Stark' }).
  writeConcern({ w: 'majority' });
```

Query.prototype.wtimeout()

Parameters

- ms «number» number of milliseconds to wait

Returns:

- «Query» this

If `w > 1`, the maximum amount of time to wait for this write to propagate through the replica set before this operation fails. The default is `0`, which means no timeout.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndReplace()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern.wtimeout` option

Example:

```
// The `deleteOne()` promise won't resolve until this `deleteOne()` has
// propagated to at least `w = 2` members of the replica set. If it takes
// longer than 1 second, this `deleteOne()` will fail.
await mongoose.model('Person').
  deleteOne({ name: 'Ned Stark' }).
  w(2).
  wtimeout(1000);
```