Indexes and MongoDB

Release 2.4.9

MongoDB Documentation Project

January 31, 2014

Contents

Indexes provide high performance read operations for frequently used queries.

This section introduces indexes in MongoDB, describes the types and configuration options for indexes, and describes special types of indexing MongoDB supports. The section also provides tutorials detailing procedures and operational concerns, and providing information on how applications may use indexes.

Index Introduction **(page 3)** An introduction to indexes in MongoDB.

Index Concepts **(page 7)** The core documentation of indexes in MongoDB, including geospatial and text indexes.

> ***Index Types*** **(page 8)** MongoDB provides different types of indexes for different purposes and different types of content.
>
> ***Index Properties*** **(page 23)** The properties you can specify when building indexes.
>
> ***Index Creation*** **(page 25)** The options available when creating indexes.

Indexing Tutorials **(page 28)** Examples of operations involving indexes, including index creation and querying indexes.

Indexing Reference **(page 63)** Reference material for indexes in MongoDB.

1 Index Introduction

Indexes support the efficient resolution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. These *collection scans* are inefficient and require the `mongod` to process a large volume of data for each operation.

Indexes are special data structures [1] that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field.

Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the *collection* level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect. In some cases, MongoDB can use the data from the index to determine which documents match a query. The following diagram illustrates a query that selects documents using an index.

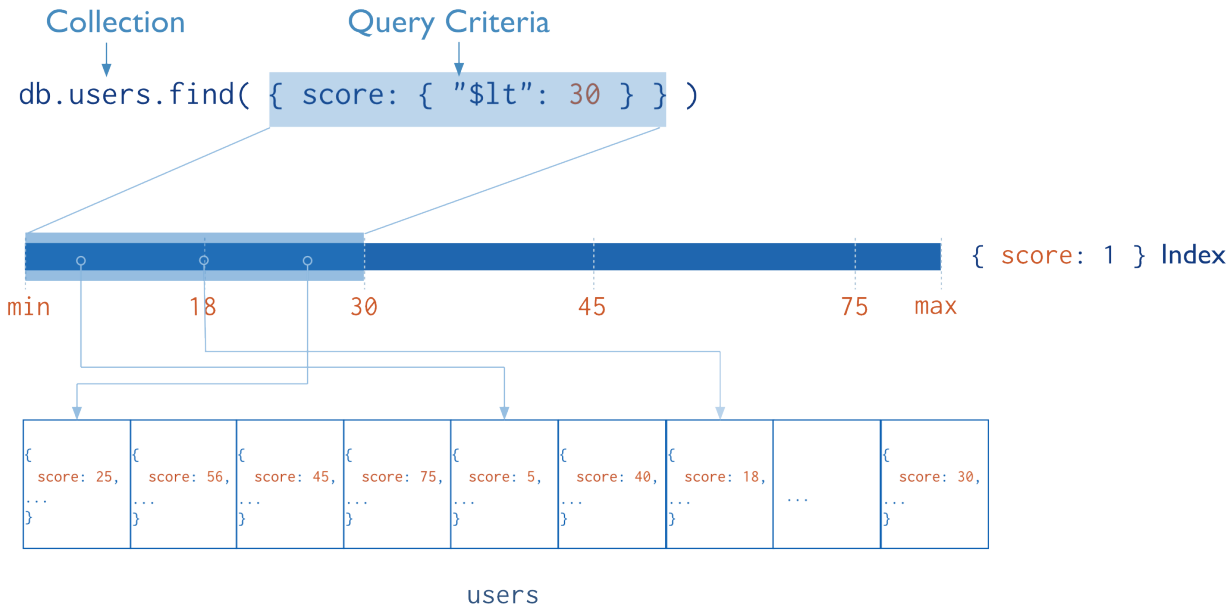

Figure 1: Diagram of a query selecting documents using an index. MongoDB narrows the query by scanning the range of documents with values of `score` less than `30`.

Consider the documentation of the *query optimizer* for more information on the relationship between queries and indexes.

Tip

Create indexes to support common and user-facing queries. Having these indexes will ensure that MongoDB only scans the smallest possible number of documents.

Indexes can also optimize the performance of other operations in specific situations:

Sorted Results

[1] MongoDB indexes use a B-tree data structure.

MongoDB can use indexes to return documents sorted by the index key directly from the index without requiring an additional sort phase.

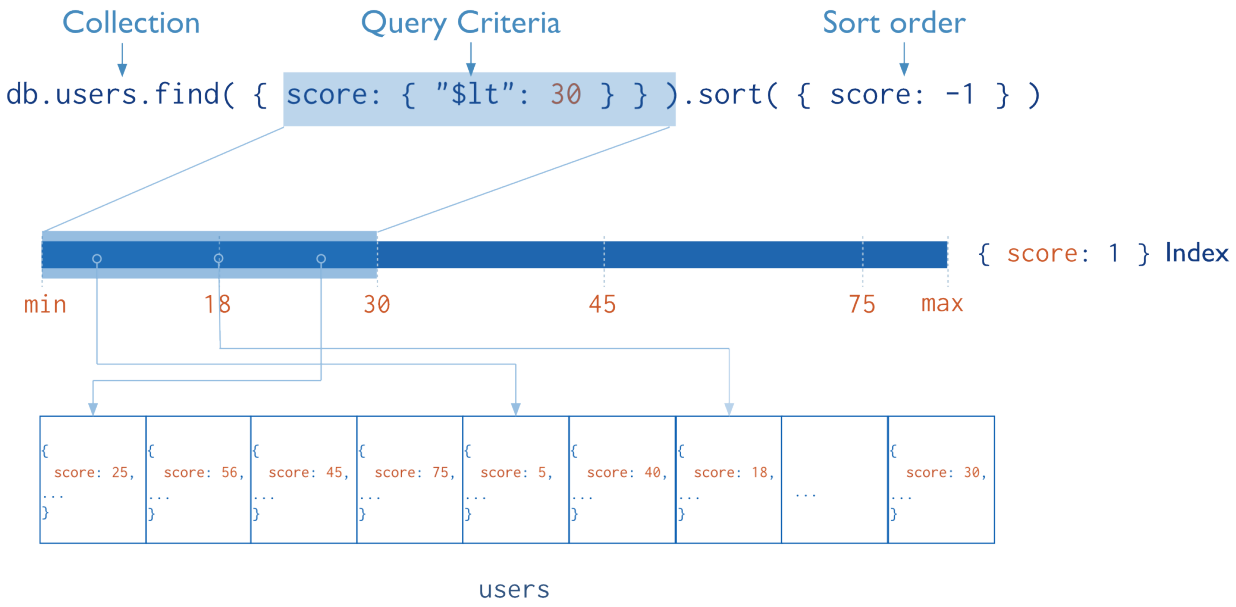

Figure 2: Diagram of a query that uses an index to select and return sorted results. The index stores `score` values in ascending order. MongoDB can traverse the index in either ascending or descending order to return sorted results.

Covered Results

When the query criteria and the *projection* of a query include *only* the indexed fields, MongoDB will return results directly from the index *without* scanning any documents or bringing documents into memory. These covered queries can be *very* efficient. Indexes can also cover `aggregation pipeline operations`.

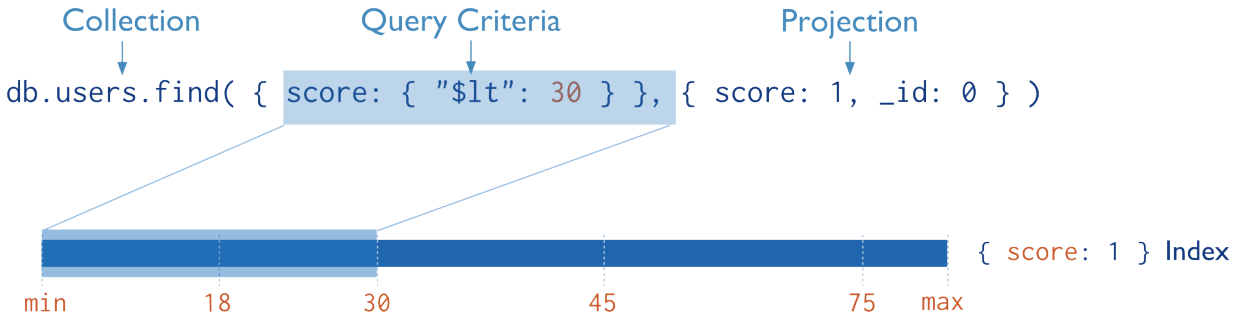

Figure 3: Diagram of a query that uses only the index to match the query criteria and return the results. MongoDB does not need to inspect data outside of the index to fulfill the query.

1.1 Index Types

MongoDB provides a number of different index types to support specific types of data and queries.

Default _id

All MongoDB collections have an index on the _id field that exists by default. If applications do not specify a value for _id the driver or the mongod will create an _id field with an *ObjectID* value.

The _id index is *unique*, and prevents clients from inserting two documents with the same value for the _id field.

Single Field

In addition to the MongoDB-defined _id index, MongoDB supports user-defined indexes on a *single field of a document* (page 9). Consider the following illustration of a single-field index:

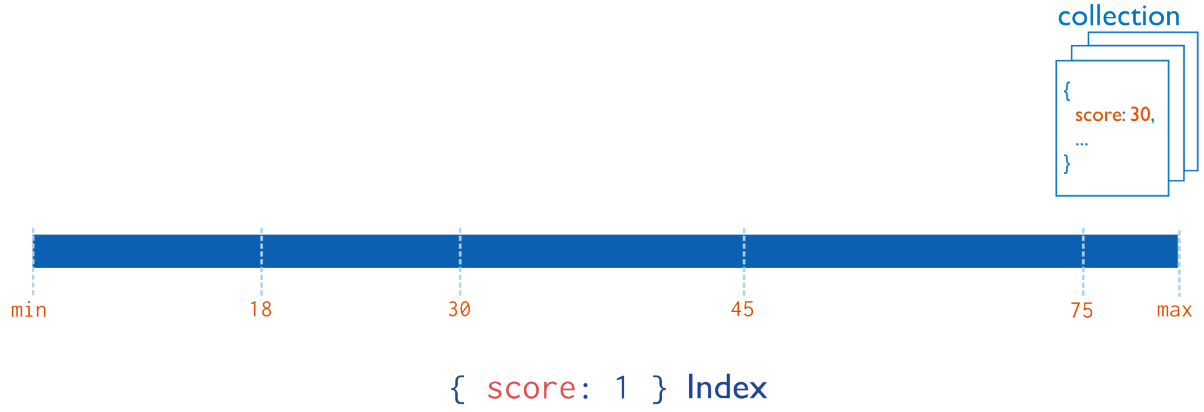

Figure 4: Diagram of an index on the score field (ascending).

Compound Index

MongoDB *also* supports user-defined indexes on multiple fields. These *compound indexes* (page 11) behave like single-field indexes; *however*, the query can select documents based on additional fields. The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each userid value, sort by score. Consider the following illustration of this compound index:

Multikey Index

MongoDB uses *multikey indexes* (page 13) to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array. These *multikey indexes* (page 13) allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Consider the following illustration of a multikey index:

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: *2d indexes* (page 18) that uses planar geometry when returning results and *2sphere indexes* (page 17) that use spherical geometry to return results.

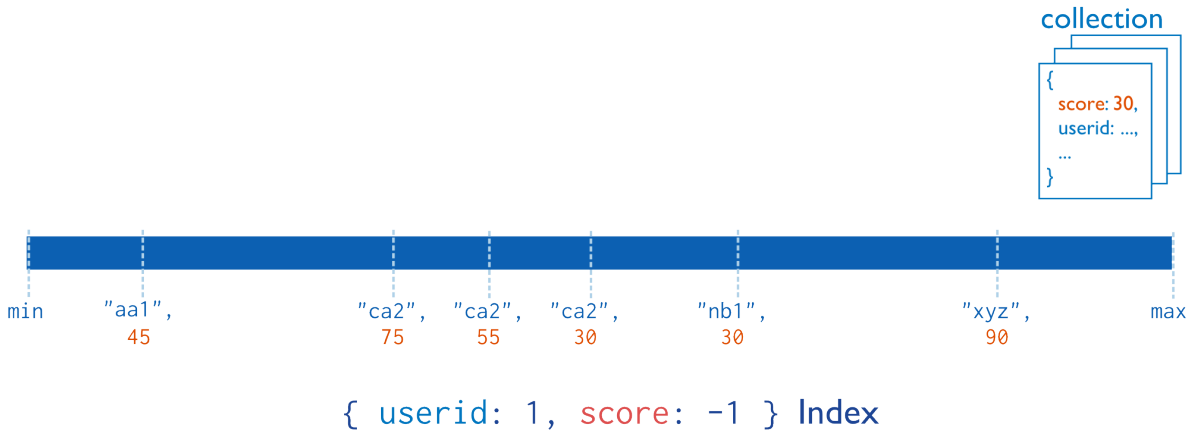

Figure 5: Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

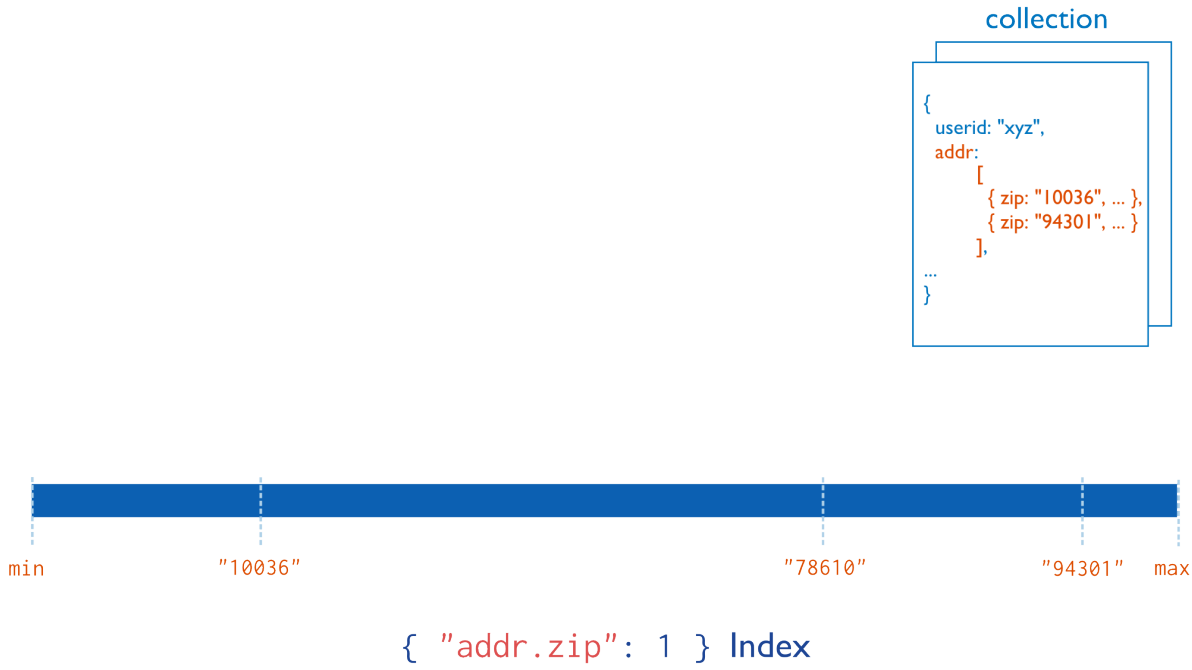

Figure 6: Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

See *2d Index Internals* (page 20) for a high level introduction to geospatial indexes.

Text Indexes

MongoDB provides a *beta* `text` index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. "the", "a", "or") and *stem* the words in a collection to only store root words.

See *Text Indexes* (page 21) for more information on text indexes and search.

Hashed Indexes

To support *hash based sharding*, MongoDB provides a *hashed index* (page 22) type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.

1.2 Index Properties

Unique Indexes

The *unique* (page 23) property for an index causes MongoDB to reject duplicate values for the indexed field. To create a *unique index* (page 23) on a field that already has duplicate values, see *Drop Duplicates* (page 27) for index creation options. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

Sparse Indexes

The *sparse* (page 24) property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field.

You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.

2 Index Concepts

These documents describe and provide examples of the types, configuration options, and behavior of indexes in MongoDB. For an over view of indexing, see *Index Introduction* (page 3). For operational instructions, see *Indexing Tutorials* (page 28). The *Indexing Reference* (page 63) documents the commands and operations specific to index construction, maintenance, and querying in MongoDB, including index types and creation options.

Index Types **(page 8)** MongoDB provides different types of indexes for different purposes and different types of content.

- ***Single Field Indexes*** **(page 9)** A single field index only includes data from a single field of the documents in a collection. MongoDB supports single field indexes on fields at the top level of a document *and* on fields in sub-documents.
- ***Compound Indexes*** **(page 11)** A compound index includes more than one field of the documents in a collection.
- ***Multikey Indexes*** **(page 13)** A multikey index references an array and records a match if a query includes any value in the array.

Geospatial Indexes and Queries **(page 15)** Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.

Text Indexes **(page 21)** Text indexes supports search of string content in documents.

Hashed Index **(page 22)** Hashed indexes maintain entries with hashes of the values of the indexed field.

Index Properties **(page 23)** The properties you can specify when building indexes.

TTL Indexes **(page 23)** The TTL index is used for TTL collections, which expire data after a period of time.

Unique Indexes **(page 23)** A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

Sparse Indexes **(page 24)** A sparse index does not index documents that do not have the indexed field.

Index Creation **(page 25)** The options available when creating indexes.

2.1 Index Types

MongoDB provides a number of different index types. You can create indexes on any field or embedded field within a document or sub-document. You can create *single field indexes* (page 9) or *compound indexes* (page 11). MongoDB also supports indexes of arrays, called *multi-key indexes* (page 13), as well as supports *indexes on geospatial data* (page 15). For a list of the supported index types, see *Index Type Documentation* (page 9).

In general, you should create indexes that support your common and user-facing queries. Having these indexes will ensure that MongoDB scans the smallest possible number of documents.

In the `mongo` shell, you can create an index by calling the `ensureIndex()` method. For more detailed instructions about building indexes, see the *Indexing Tutorials* (page 28) page.

Behavior of Index Types

All indexes in MongoDB are *B-tree* indexes, which can efficiently support equality matches and range queries. The index stores items internally in order sorted by the value of the index field. The ordering of index entries supports efficient range-based operations and allows MongoDB to return sorted results using the order of documents in the index.

Ordering of Indexes

MongoDB indexes may be ascending, (i.e. `1`) or descending (i.e. `-1`) in their ordering. Nevertheless, MongoDB may also traverse the index in either directions. As a result, for single-field indexes, ascending and descending indexes are interchangeable. This is not the case for compound indexes: in compound indexes, the direction of the sort order can have a greater impact on the results.

See *Sort Order* (page 12) for more information on the impact of index order on results in compound indexes.

Redundant Indexes

A single query can only use *one* index, except for queries that use the `$or` operator that can use a different index for each clause.

See also:

Index Limitations.

Index Type Documentation

Single Field Indexes **(page 9)** A single field index only includes data from a single field of the documents in a collection. MongoDB supports single field indexes on fields at the top level of a document *and* on fields in sub-documents.

Compound Indexes **(page 11)** A compound index includes more than one field of the documents in a collection.

Multikey Indexes **(page 13)** A multikey index references an array and records a match if a query includes any value in the array.

Geospatial Indexes and Queries **(page 15)** Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.

Text Indexes **(page 21)** Text indexes supports search of string content in documents.

Hashed Index **(page 22)** Hashed indexes maintain entries with hashes of the values of the indexed field.

Single Field Indexes

MongoDB provides complete support for indexes on any field in a *collection* of *documents*. By default, all collections have an index on the *_id field* (page 10), and applications and users may add additional indexes to support important queries and operations.

MongoDB supports indexes that contain either a single field *or* multiple fields depending on the operations that index supports. This document describes indexes that contain a single field. Consider the following illustration of a single field index.

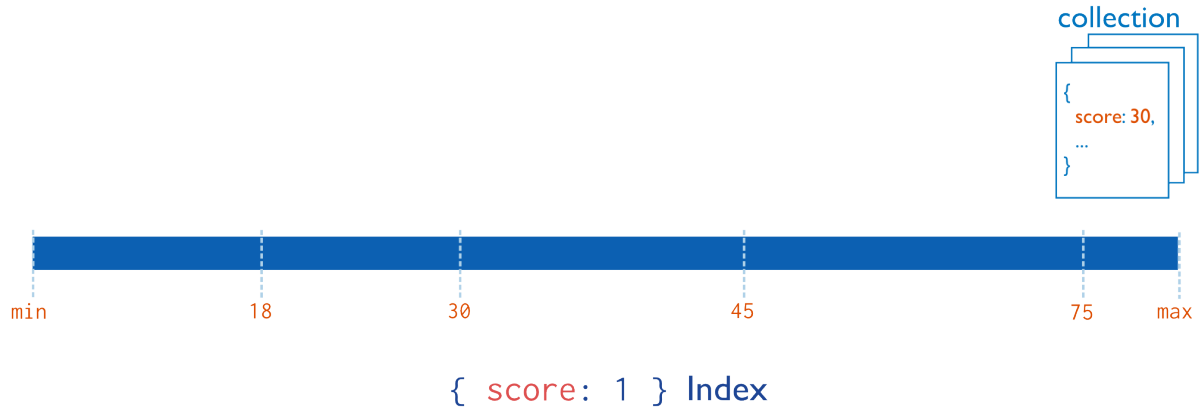

Figure 7: Diagram of an index on the `score` field (ascending).

See also:

Compound Indexes (page 11) for information about indexes that include multiple fields, and *Index Introduction* (page 3) for a higher level introduction to indexing in MongoDB.

Example Given the following document in the `friends` collection:

```
{ "_id" : ObjectID(...),
  "name" : "Alice"
  "age" : 27
}
```

The following command creates an index on the `name` field:

```
db.friends.ensureIndex( { "name" : 1 } )
```

Cases

`_id` Field Index For all collections, MongoDB creates the default `_id` index, which is a *unique index* (page 23) on the `_id` field. MongoDB creates this index by default on all collections. You cannot delete the index on `_id`.

You can think of the `_id` field as the *primary key* for the collection. Every document *must* have a unique `_id` field. You may store any unique value in the `_id` field. The default value of `_id` is an *ObjectID* on every `insert()` operation. An *ObjectId* is a 12-byte unique identifiers suitable for use as the value of an `_id` field.

Note: In *sharded clusters*, if you do *not* use the `_id` field as the *shard key*, then your application **must** ensure the uniqueness of the values in the `_id` field to prevent errors. This is most-often done by using a standard auto-generated *ObjectId*.

Before version 2.2, *capped collections* did not have an `_id` field. In version 2.2 and newer, capped collection do have an `_id` field, except those in the `local` *database*. See *Capped Collections Recommendations and Restrictions* for more information.

Indexes on Embedded Fields You can create indexes on fields embedded in sub-documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from *indexes on sub-documents* (page 10), which include the full content up to the maximum `Index Size` of the sub-document in the index. Instead, indexes on embedded fields allow you to use a "dot notation," to introspect into sub-documents.

Consider a collection named `people` that holds documents that resemble the following example document:

```
{"_id": ObjectId(...)
 "name": "John Doe"
 "address": {
        "street": "Main"
        "zipcode": 53511
        "state": "WI"
        }
}
```

You can create an index on the `address.zipcode` field, using the following specification:

```
db.people.ensureIndex( { "address.zipcode": 1 } )
```

Indexes on Subdocuments You can also create indexes on subdocuments.

For example, the `factories` collection contains documents that contain a `metro` field, such as:

```
{
  _id: ObjectId("523cba3c73a8049bcdbf6007"),
  metro: {
            city: "New York",
            state: "NY"
          },
  name: "Giant Factory"
}
```

The `metro` field is a subdocument, containing the embedded fields `city` and `state`. The following creates an index on the `metro` field as a whole:

```
db.factories.ensureIndex( { metro: 1 } )
```

The following query can use the index on the `metro` field:

```
db.factories.find( { metro: { city: "New York", state: "NY" } } )
```

This query returns the above document. When performing equality matches on subdocuments, field order matters and the subdocuments must match exactly. For example, the following query does not match the above document:

```
db.factories.find( { metro: { state: "NY", city: "New York" } } )
```

See *query-subdocuments* for more information regarding querying on subdocuments.

Compound Indexes

MongoDB supports *compound indexes*, where a single index structure holds references to multiple fields [2] within a collection's documents. The following diagram illustrates an example of a compound index on two fields:

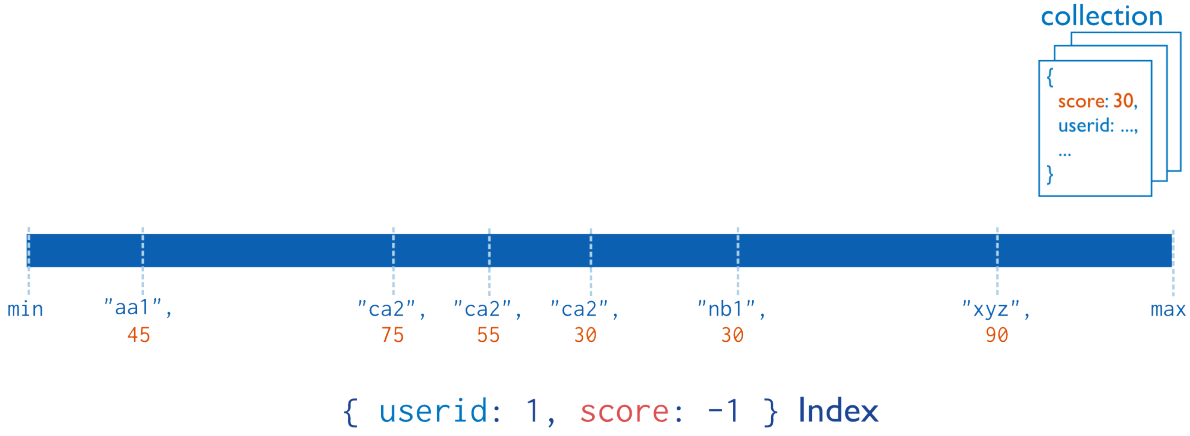

Figure 8: Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Compound indexes can support queries that match on multiple fields.

Example

Consider a collection named `products` that holds documents that resemble the following document:

```
{
 "_id": ObjectId(...)
 "item": "Banana"
 "category": ["food", "produce", "grocery"]
 "location": "4th Street Store"
 "stock": 4
 "type": cases
 "arrival": Date(...)
}
```

[2] MongoDB imposes a `limit of 31 fields for any compound index`.

If applications query on the `item` field as well as query on both the `item` field and the `stock` field, you can specify a single compound index to support both of these queries:

```
db.products.ensureIndex( { "item": 1, "stock": 1 } )
```

Important: You may not create compound indexes that have `hashed` index fields. You will receive an error if you attempt to create a compound index that includes *a hashed index* (page 22).

The order of the fields in a compound index is very important. In the previous example, the index will contain references to documents sorted first by the values of the `item` field and, within each value of the `item` field, sorted by values of the `stock` field. See *Sort Order* (page 12) for more information.

In addition to supporting queries that match on all the index fields, compound indexes can support queries that match on the prefix of the index fields. For details, see *Prefixes* (page 12).

Sort Order Indexes store references to fields in either ascending (`1`) or descending (`-1`) sort order. For single-field indexes, the sort order of keys doesn't matter because MongoDB can traverse the index in either direction. However, for *compound indexes* (page 11), sort order can matter in determining whether the index can support a sort operation.

Consider a collection `events` that contains documents with the fields `username` and `date`. Applications can issue queries that return results sorted first by ascending `username` values and then by descending (i.e. more recent to last) `date` values, such as:

```
db.events.find().sort( { username: 1, date: -1 } )
```

or queries that return results sorted first by descending `username` values and then by ascending `date` values, such as:

```
db.events.find().sort( { username: -1, date: 1 } )
```

The following index can support both these sort operations:

```
db.events.ensureIndex( { "username" : 1, "date" : -1 } )
```

However, the above index cannot support sorting by ascending `username` values and then by ascending `date` values, such as the following:

```
db.events.find().sort( { username: 1, date: 1 } )
```

Prefixes Compound indexes support queries on any prefix of the index fields. Index prefixes are the beginning subset of indexed fields. For example, given the index `{ a: 1, b: 1, c: 1 }`, both `{ a: 1 }` and `{ a: 1, b: 1 }` are prefixes of the index.

If you have a collection that has a compound index on `{ a: 1, b: 1 }`, as well as an index that consists of the prefix of that index, i.e. `{ a: 1 }`, assuming none of the index has a sparse or unique constraints, then you can drop the `{ a: 1 }` index. MongoDB will be able to use the compound index in all of situations that it would have used the `{ a: 1 }` index.

Example

Given the following index:

```
{ "item": 1, "location": 1, "stock": 1 }
```

MongoDB **can** use this index to support queries that include:

- the `item` field,

- the `item` field *and* the `location` field,
- the `item` field *and* the `location` field *and* the `stock` field, or
- only the `item` *and* `stock` fields; however, this index would be less efficient than an index on only `item` and `stock`.

MongoDB **cannot** use this index to support queries that include:

- only the `location` field,
- only the `stock` field, or
- only the `location` *and* `stock` fields.

Multikey Indexes

To index a field that holds an array value, MongoDB adds index items for each item in the array. These *multikey* indexes allow MongoDB to return documents from queries using the value of an array. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Consider the following illustration of a multikey index:

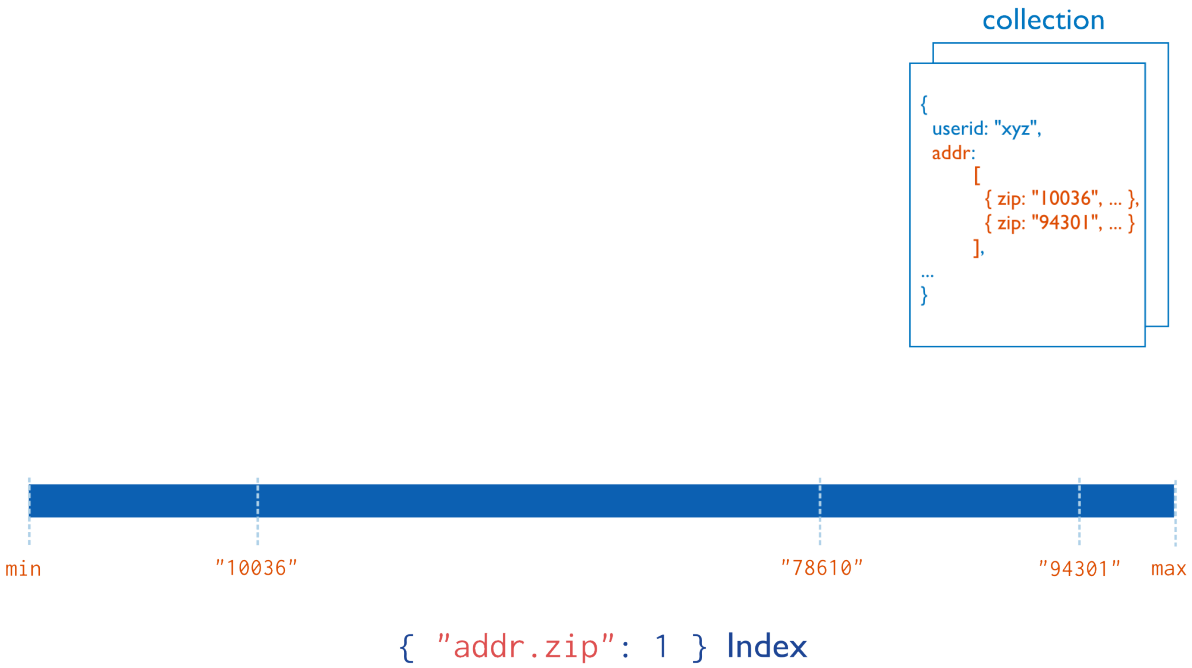

Figure 9: Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

Multikey indexes support all operations supported by other MongoDB indexes; however, applications may use multikey indexes to select documents based on ranges of values for the value of an array. Multikey indexes support arrays that hold both values (e.g. strings, numbers) *and* nested documents.

Limitations

Interactions between Compound and Multikey Indexes While you can create multikey *compound indexes* (page 11), at most one field in a compound index may hold an array. For example, given an index on `{ a: 1, b: 1 }`, the following documents are permissible:

```
{a: [1, 2], b: 1}

{a: 1, b: [1, 2]}
```

However, the following document is impermissible, and MongoDB cannot insert such a document into a collection with the `{a: 1, b: 1 }` index:

```
{a: [1, 2], b: [1, 2]}
```

If you attempt to insert a such a document, MongoDB will reject the insertion, and produce an error that says `cannot index parallel arrays`. MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

Shard Keys

Important: The index of a shard key **cannot** be a multi-key index.

Hashed Indexes `hashed` indexes are not compatible with multi-key indexes.

To compute the hash for a `hashed` index, MongoDB collapses sub-documents and computes the hash for the entire value. For fields that hold arrays or sub-documents, you cannot use the index to support queries that introspect the sub-document.

Examples

Index Basic Arrays Given the following document:

```
{
  "_id" : ObjectId("..."),
  "name" : "Warm Weather",
  "author" : "Steve",
  "tags" : [ "weather", "hot", "record", "april" ]
}
```

Then an index on the `tags` field, `{ tags: 1 }`, would be a multikey index and would include these four separate entries for that document:

- `"weather"`,
- `"hot"`,
- `"record"`, and
- `"april"`.

Queries could use the multikey index to return queries for any of the above values.

Index Arrays with Embedded Documents You can create multikey indexes on fields in objects embedded in arrays, as in the following example:

Consider a `feedback` collection with documents in the following form:

```
{
 "_id": ObjectId(...),
 "title": "Grocery Quality",
 "comments": [
    { author_id: ObjectId(...),
      date: Date(...),
      text: "Please expand the cheddar selection." },
    { author_id: ObjectId(...),
      date: Date(...),
      text: "Please expand the mustard selection." },
    { author_id: ObjectId(...),
      date: Date(...),
      text: "Please expand the olive selection." }
 ]
}
```

An index on the `comments.text` field would be a multikey index and would add items to the index for all embedded documents in the array.

With the index `{ "comments.text": 1 }` on the `feedback` collection, consider the following query:

```
db.feedback.find( { "comments.text": "Please expand the olive selection." } )
```

The query would select the documents in the collection that contain the following embedded document in the `comments` array:

```
{ author_id: ObjectId(...),
  date: Date(...),
  text: "Please expand the olive selection." }
```

Geospatial Indexes and Queries

MongoDB offers a number of indexes and query mechanisms to handle geospatial information. This section introduces MongoDB's geospatial features. For complete examples of geospatial queries in MongoDB, see *Geospatial Index Tutorials* (page 39).

Surfaces Before storing your location data and writing queries, you must decide the type of surface to use to perform calculations. The type you choose affects how you store data, what type of index to build, and the syntax of your queries.

MongoDB offers two surface types:

Spherical To calculate geometry over an Earth-like sphere, store your location data on a spherical surface and use *2dsphere* (page 17) index.

Store your location data as GeoJSON objects with this coordinate-axis order: **longitude, latitude**. The coordinate reference system for GeoJSON uses the *WGS84* datum.

Flat To calculate distances on a Euclidean plane, store your location data as legacy coordinate pairs and use a *2d* (page 18) index.

Location Data If you choose spherical surface calculations, you store location data as either:

GeoJSON Objects Queries on *GeoJSON* objects always calculate on a sphere. The default coordinate reference system for GeoJSON uses the *WGS84* datum.

New in version 2.4: Support for GeoJSON storage and queries is new in version 2.4. Prior to version 2.4, all geospatial data used coordinate pairs.

MongoDB supports the following GeoJSON objects:

- Point
- LineString
- Polygon

Legacy Coordinate Pairs MongoDB supports spherical surface calculations on *legacy coordinate pairs* by converting the data to the GeoJSON Point type.

If you choose flat surface calculations, you can store data only as *legacy coordinate pairs*.

Query Operations MongoDB's geospatial query operators let you query for:

Inclusion MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the `$geoWithin` operator.

Intersection MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the `$geoIntersects` operator.

Proximity MongoDB can query for the points nearest to another point. Proximity queries use the `$near` operator. The `$near` operator requires a `2d` or `2dsphere` index.

Geospatial Indexes MongoDB provides the following geospatial index types to support the geospatial queries.

`2dsphere` *2dsphere* (page 17) indexes support:

- Calculations on a sphere
- Both GeoJSON objects and legacy coordinate pairs
- A compound index with scalar index fields (i.e. ascending or descending) as a prefix or suffix of the `2dsphere` index field

New in version 2.4: `2dsphere` indexes are not available before version 2.4.

See also:

Query a 2dsphere Index (page 39)

`2d` *2d* (page 18) indexes support:

- Calculations using flat geometry
- Legacy coordinate pairs (i.e., geospatial points on a flat coordinate system)
- A compound index with only one additional field, as a suffix of the `2d` index field

See also:

Query a 2d Index (page 42)

Geospatial Indexes and Sharding You *cannot* use a geospatial index as the *shard key* index.

You can create and maintain a geospatial index on a sharded collection if using different fields as the shard key.

Queries using `$near` are not supported for sharded collections. Use `geoNear` instead. You also can query for geospatial data using `$geoWithin`.

Additional Resources The following pages provide complete documentation for geospatial indexes and queries:

***2dsphere Indexes* (page 17)** A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. The index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

***2d Indexes* (page 18)** The `2d` index supports data stored as legacy coordinate pairs and is intended for use in MongoDB 2.2 and earlier.

***Haystack Indexes* (page 19)** A haystack index is a special index optimized to return results over small areas. For queries that use spherical geometry, a `2dsphere` index is a better option than a haystack index.

***2d Index Internals* (page 20)** Provides a more in-depth explanation of the internals of geospatial indexes. This material is not necessary for normal operations but may be useful for troubleshooting and for further understanding.

`2dsphere` Indexes New in version 2.4.

A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. The index supports data stored as both *GeoJSON* objects and as legacy coordinate pairs. The index supports legacy coordinate pairs by converting the data to the GeoJSON `Point` type.

The `2dsphere` index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity.

A *compound* (page 11) `2dsphere` index can reference multiple location and non-location fields within a collection's documents. You can arrange the fields in any order.

The default datum for an earth-like sphere in MongoDB 2.4 is *WGS84*. Coordinate-axis order is **longitude, latitude**.

Important: MongoDB allows *only one* geospatial index per collection. You can create either a `2dsphere` **or** a *2d* (page 18) per collection.

Important: You cannot use a `2dsphere` index as a shard key when sharding a collection. However, you can create and maintain a geospatial index on a sharded collection by using a different field as the shard key.

Store GeoJSON Objects New in version 2.4.

MongoDB supports the following GeoJSON objects:

- *Point*
- *LineString*
- *Polygon*

In order to index GeoJSON data, you must store the data in a location field that you name. The location field contains a subdocument with a `type` field specifying the GeoJSON object type and a `coordinates` field specifying the object's coordinates. Always store coordinates `longitude, latitude` order.

Use the following syntax:

```
{ <location field> : { type : "<GeoJSON type>" ,
                       coordinates : <coordinates>
} }
```

The following example stores a GeoJSON `Point`:

```
{ loc : { type : "Point" ,
          coordinates : [ 40, 5 ]
} }
```

The following example stores a GeoJSON `LineString`:

```
{ loc : { type : "LineString" ,
          coordinates : [ [ 40 , 5 ] , [ 41 , 6 ] ]
} }
```

Polygons consist of an array of GeoJSON `LinearRing` coordinate arrays. These `LinearRings` are closed `LineStrings`. Closed `LineStrings` have at least four coordinate pairs and specify the same position as the first and last coordinates.

The following example stores a GeoJSON `Polygon` with an exterior ring and no interior rings (or holes). Note the first and last coordinate pair with the `[ 0 , 0 ]` coordinate:

```
{ loc :
   { type : "Polygon" ,
     coordinates : [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ] ]
} }
```

For Polygons with multiple rings:

- The first described ring must be the exterior ring.
- The exterior ring cannot self-intersect.
- Any interior ring must be entirely contained by the outer ring.
- Interior rings cannot intersect or overlap each other. Interior rings can share an edge.

The following document represents a polygon with an interior ring as GeoJSON:

```
{ loc :
   { type : "Polygon" ,
     coordinates : [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
                     [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ] ]
} }
```

2d Indexes

Important: MongoDB allows *only one* geospatial index per collection. You can create either a `2d` **or** a *2dsphere* (page 17) per collection.

Use a `2d` index for data stored as points on a two-dimensional plane. The `2d` index is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.

Use a `2d` index if:

- your database has legacy location data from MongoDB 2.2 or earlier, *and*
- you do not intend to store any location data as *GeoJSON* objects.

Do not use a `2d` index if your location data includes GeoJSON objects. To index on both legacy coordinate pairs *and* GeoJSON objects, use a *2dsphere* (page 17) index.

The `2d` index supports calculations on a flat, Euclidean plane. The `2d` index also supports *distance-only* calculations on a sphere, but for *geometric* calculations (e.g. `$geoWithin`) on a sphere, store data as GeoJSON objects and use the `2dsphere` index type.

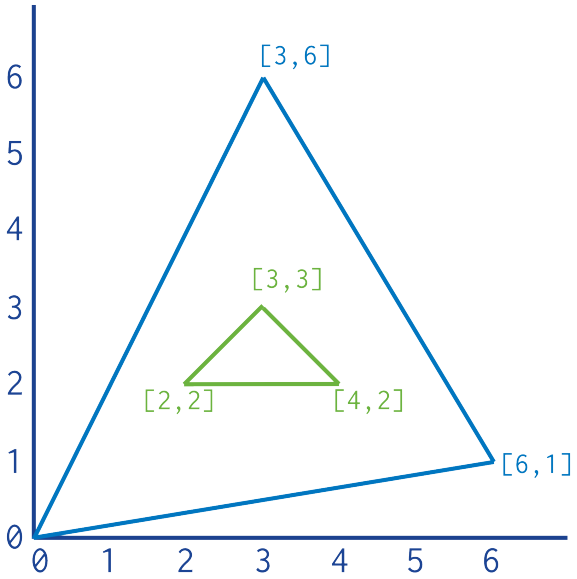

Figure 10: Diagram of a Polygon with internal ring.

A `2d` index can reference two fields. The first must be the location field. A `2d` compound index constructs queries that select first on the location field, and then filters those results by the additional criteria. A compound `2d` index can cover queries.

Note: You cannot use a `2d` index as a shard key when sharding a collection. However, you can create and maintain a geospatial index on a sharded collection by using a different field as the shard key.

Store Points on a 2D Plane To store location data as legacy coordinate pairs, use either an array (preferred):

```
loc : [ <longitude> , <latitude> ]
```

Or an embedded document:

```
loc : { lng : <longitude> , lat : <latitude> }
```

Arrays are preferred as certain languages do not guarantee associative map ordering.

Whether as an array or document, if you use longitude and latitude, store coordinates in this order: **longitude, latitude**.

Haystack Indexes A haystack index is a special index that is optimized to return results over small areas. Haystack indexes improve performance on queries that use flat geometry.

For queries that use spherical geometry, a **2dsphere index is a better option** than a haystack index. *2dsphere indexes* (page 17) allow field reordering; haystack indexes require the first field to be the location field. Also, haystack indexes are only usable via commands and so always return all results at once.

Haystack indexes create "buckets" of documents from the same geographic area in order to improve performance for queries limited to that area. Each bucket in a haystack index contains all the documents within a specified proximity to a given longitude and latitude.

To create a geohaystacks index, see *Create a Haystack Index* (page 44). For information and example on querying a haystack index, see *Query a Haystack Index* (page 45).

2d Index Internals This document provides a more in-depth explanation of the internals of MongoDB's `2d` geospatial indexes. This material is not necessary for normal operations or application development but may be useful for troubleshooting and for further understanding.

Calculation of Geohash Values for 2d Indexes When you create a geospatial index on *legacy coordinate pairs*, MongoDB computes *geohash* values for the coordinate pairs within the specified *location range* (page 42) and then indexes the geohash values.

To calculate a geohash value, recursively divide a two-dimensional map into quadrants. Then assign each quadrant a two-bit value. For example, a two-bit representation of four quadrants would be:

```
01  11

00  10
```

These two-bit values (`00`, `01`, `10`, and `11`) represent each of the quadrants and all points within each quadrant. For a geohash with two bits of resolution, all points in the bottom left quadrant would have a geohash of `00`. The top left quadrant would have the geohash of `01`. The bottom right and top right would have a geohash of `10` and `11`, respectively.

To provide additional precision, continue dividing each quadrant into sub-quadrants. Each sub-quadrant would have the geohash value of the containing quadrant concatenated with the value of the sub-quadrant. The geohash for the upper-right quadrant is `11`, and the geohash for the sub-quadrants would be (clockwise from the top left): `1101`, `1111`, `1110`, and `1100`, respectively.

Multi-location Documents for 2d Indexes New in version 2.0: Support for multiple locations in a document.

While `2d` geospatial indexes do not support more than one set of coordinates in a document, you can use a *multi-key index* (page 13) to index multiple coordinate pairs in a single document. In the simplest example you may have a field (e.g. `locs`) that holds an array of coordinates, as in the following example:

```
{ _id : ObjectId(...),
  locs : [ [ 55.5 , 42.3 ] ,
           [ -74 , 44.74 ] ,
           { lng : 55.5 , lat : 42.3 } ]
}
```

The values of the array may be either arrays, as in `[ 55.5, 42.3 ]`, or embedded documents, as in `{ lng : 55.5 , lat :  42.3 }`.

You could then create a geospatial index on the `locs` field, as in the following:

```
db.places.ensureIndex( { "locs": "2d" } )
```

You may also model the location data as a field inside of a sub-document. In this case, the document would contain a field (e.g. `addresses`) that holds an array of documents where each document has a field (e.g. `loc:`) that holds location coordinates. For example:

```
{ _id : ObjectId(...),
  name : "...",
  addresses : [ {
                  context : "home" ,
                  loc : [ 55.5, 42.3 ]
                } ,
```

```
                {
                 context : "home",
                 loc : [ -74 , 44.74 ]
                }
              ]
}
```

You could then create the geospatial index on the `addresses.loc` field as in the following example:

```
db.records.ensureIndex( { "addresses.loc": "2d" } )
```

For documents with multiple coordinate values, queries may return the same document multiple times if more than one indexed coordinate pair satisfies the query constraints. Use the `uniqueDocs` parameter to `geoNear` or the `$uniqueDocs` operator with `$geoWithin`.

To include the location field with the distance field in multi-location document queries, specify `includeLocs: true` in the `geoNear` command.

See also:

geospatial-query-compatibility-chart

Text Indexes

New in version 2.4.

MongoDB provides `text` indexes to support text search of string content in documents of a collection. `text` indexes are case-insensitive and can include any field whose value is a string or an array of string elements. You can only access the `text` index with the `text` command.

Important:

- Before you can create a text index or *run the text command* (page 22), you need to manually enable the text search. See *Enable Text Search* (page 48) for information on how to enable the text search feature.
- A collection can have at most **one** `text` index.

Create Text Index To create a `text` index, use the `db.collection.ensureIndex()` method. To index a field that contains a string or an array of string elements, include the field and specify the string literal `"text"` in the index document, as in the following example:

```
db.reviews.ensureIndex( { comments: "text" } )
```

For examples of creating `text` indexes on multiple fields, see *Create a text Index* (page 48).

`text` indexes drop language-specific stop words (e.g. in English, "the," "an," "a," "and," etc.) and uses simple language-specific suffix stemming. See *text-search-languages* for the supported languages and *Specify a Language for Text Index* (page 52) for details on specifying languages with `text` indexes.

`text` indexes can cover a text search. If an index covers a text search, MongoDB does not need to inspect data outside of the index to fulfill the search. For details, see *Create text Index to Cover Queries* (page 56).

Storage Requirements and Performance Costs `text` indexes have the following storage requirements and performance costs:

- `text` indexes change the space allocation method for all future record allocations in a collection to `usePowerOf2Sizes`.

- `text` indexes can be large. They contain one index entry for each unique post-stemmed word in each indexed field for each document inserted.
- Building a `text` index is very similar to building a large multi-key index and will take longer than building a simple ordered (scalar) index on the same data.
- When building a large `text` index on an existing collection, ensure that you have a sufficiently high limit on open file descriptors. See the `recommended settings`.
- `text` indexes will impact insertion throughput because MongoDB must add an index entry for each unique post-stemmed word in each indexed field of each new source document.
- Additionally, `text` indexes do not store phrases or information about the proximity of words in the documents. As a result, phrase queries will run much more effectively when the entire collection fits in RAM.

Text Search Text search supports the search of string content in documents of a collection. MongoDB provides the `text` command to perform the text search. The `text` command accesses the `text` index.

The text search process:

- tokenizes and stems the search term(s) during both the index creation and the text command execution.
- assigns a score to each document that contains the search term in the indexed fields. The score determines the relevance of a document to a given search query.

By default, the `text` command returns at most the top 100 matching documents as determined by the scores. The command can search for words and phrases. The command matches on the complete stemmed words. For example, if a document field contains the word `blueberry`, a search on the term `blue` will not match the document. However, a search on either `blueberry` or `blueberries` will match.

For information and examples on various text search patterns, see *Search String Content for Text* (page 49).

Hashed Index

New in version 2.4.

Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses sub-documents and computes the hash for the entire value but does not support multi-key (i.e. arrays) indexes.

Hashed indexes support `sharding` a collection using a *hashed shard key*. Using a hashed shard key to shard a collection ensures a more even distribution of data. See `http://docs.mongodb.org/manual/tutorial/shard-collection-with-a-hashed-shard-key` for more details.

MongoDB can use the `hashed` index to support equality queries, but `hashed` indexes do not support range queries.

You may not create compound indexes that have `hashed` index fields or specify a unique constraint on a `hashed` index; however, you can create both a `hashed` index and an ascending/descending (i.e. non-hashed) index on the same field: MongoDB will use the scalar index for range queries.

Warning: MongoDB `hashed` indexes truncate floating point numbers to 64-bit integers before hashing. For example, a `hashed` index would store the same value for a field that held a value of `2.3`, `2.2`, and `2.9`. To prevent collisions, do not use a `hashed` index for floating point numbers that cannot be reliably converted to 64-bit integers (and then back to floating point). MongoDB `hashed` indexes do not support floating point values larger than 2^{53}.

Create a `hashed` index using an operation that resembles the following:

```
db.active.ensureIndex( { a: "hashed" } )
```

This operation creates a hashed index for the `active` collection on the `a` field.

2.2 Index Properties

In addition to the numerous *index types* (page 8) MongoDB supports, indexes can also have various properties. The following documents detail the index properties that can you can select when building an index.

TTL Indexes **(page 23)** The TTL index is used for TTL collections, which expire data after a period of time.

Unique Indexes **(page 23)** A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

Sparse Indexes **(page 24)** A sparse index does not index documents that do not have the indexed field.

TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for some types of information like machine generated event data, logs, and session information that only need to persist in a database for a limited amount of time.

Considerations

TTL indexes have the following limitations:

- *Compound indexes* (page 11) are *not* supported.
- The indexed field **must** be a date *type*.
- If the field holds an array, and there are multiple date-typed data in the index, the document will expire when the *lowest* (i.e. earliest) matches the expiration threshold.

The TTL index does not guarantee that expired data will be deleted immediately. There may be a delay between the time a document expires and the time that MongoDB removes the document from the database.

The background task that removes expired documents runs *every 60 seconds*. As a result, documents may remain in a collection *after* they expire but *before* the background task runs or completes.

The duration of the removal operation depends on the workload of your `mongod` instance. Therefore, expired data may exist for some time *beyond* the 60 second period between runs of the background task.

In all other respects, TTL indexes are normal indexes, and if appropriate, MongoDB can use these indexes to fulfill arbitrary queries.

Additional Information

`http://docs.mongodb.org/manualtutorial/expire-data`

Unique Indexes

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field. To create a unique index on the `user_id` field of the `members` collection, use the following operation in the `mongo` shell:

```
db.addresses.ensureIndex( { "user_id": 1 }, { unique: true } )
```

By default, `unique` is `false` on MongoDB indexes.

If you use the unique constraint on a *compound index* (page 11), then MongoDB will enforce uniqueness on the *combination* of values rather than the individual value for any or all values of the key.

If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document. Because of the unique constraint, MongoDB will only permit one document that lacks the indexed field. If there is more than one document without a value for the indexed field or is missing the indexed field, the index build will fail with a duplicate key error.

You can combine the unique constraint with the *sparse index* (page 24) to filter these null values from the unique index and avoid the error.

You may not specify a unique constraint on a *hashed index* (page 22).

Sparse Indexes

Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field. The index is "sparse" because it does not include all documents of a collection. By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

The following example in the `mongo` shell creates a sparse index on the `xmpp_id` field of the `members` collection:

```
db.addresses.ensureIndex( { "xmpp_id": 1 }, { sparse: true } )
```

By default, `sparse` is `false` on MongoDB indexes.

Warning: Using these indexes will sometimes result in incomplete results when filtering or sorting results, because sparse indexes are not complete for all documents in a collection.

Note: Do not confuse sparse indexes in MongoDB with block-level[3] indexes in other databases. Think of them as dense indexes with a specific filter.

Tip

You can specify a *sparse* and *unique index* (page 23), that rejects documents that have duplicate values for a field, but allows multiple documents that omit that key.

Examples

Sparse Index On A Collection Can Result In Incomplete Results Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

The collection has a sparse index on the field `score`:

[3] http://en.wikipedia.org/wiki/Database_index#Sparse_index

```
db.scores.ensureIndex( { score: 1 } , { sparse: true } )
```

Then, the following query to return all documents in the `scores` collection sorted by the `score` field gives incomplete results:

```
db.scores.find().sort( { score: -1 } )
```

Because the document for the userid `"newbie"` does not contain the `score` field, the query, which uses the sparse index, will return incomplete results that omit that document:

```
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
```

Sparse Index with Unique Constraint Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

You could create an index with a *unique constraint* (page 23) and sparse filter on the `score` field using the following operation:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true, unique: true } )
```

This index *would permit* inserting documents that had unique values for the `score` field *or* did not include a `score` field. Consider the following `insert operation`:

```
db.scores.insert( { "userid": "PWWfO8lFs1", "score": 43 } )
db.scores.insert( { "userid": "XlSOX66gEy", "score": 34 } )
db.scores.insert( { "userid": "nuZHu2tcRm" } )
db.scores.insert( { "userid": "HIGvEZfdc5" } )
```

However, this index *would not permit* adding the following documents:

```
db.scores.insert( { "userid": "PWWfO8lFs1", "score": 82 } )
db.scores.insert( { "userid": "XlSOX66gEy", "score": 90 } )
```

2.3 Index Creation

MongoDB provides several options that *only* affect the creation of the index. Specify these options in a document as the second argument to the `db.collection.ensureIndex()` method. This section describes the uses of these creation options and their behavior.

Related

Some options that you can specify to `ensureIndex()` options control the *properties of the index* (page 23), which are *not* index creation options. For example, the *unique* (page 23) option affects the behavior of the index after creation.

For a detailed description of MongoDB's index types, see *Index Types* (page 8) and *Index Properties* (page 23) for related documentation.

Background Construction

By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. Any

operation that requires a read or write lock on all databases (e.g. **listDatabases**) will wait for the foreground index build to complete.

For potentially long running index building operations, consider the `background` operation so that the MongoDB database remains available during the index building operation. For example, to create an index in the background of the `zipcode` field of the `people` collection, issue the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```

By default, `background` is `false` for building MongoDB indexes.

You can combine the background option with other options, as in the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true, sparse: true } )
```

Behavior

As of MongoDB version 2.4, a `mongod` instance can build more than one index in the background concurrently.

> Changed in version 2.4: Before 2.4, a `mongod` instance could only build one background index per database at a time.
>
> Changed in version 2.2: Before 2.2, a single `mongod` instance could only build one index at a time.

Background indexing operations run in the background so that other database operations can run while creating the index. However, the `mongo` shell session or connection where you are creating the index *will* block until the index build is complete. To continue issuing commands to the database, open another connection or `mongo` instance.

Queries will not use partially-built indexes: the index will only be usable once the index build is complete.

Note: If MongoDB is building an index in the background, you cannot perform other administrative operations involving that collection, including running `repairDatabase`, dropping the collection (i.e. `db.collection.drop()`), and running `compact`. These operations will return an error during background index builds.

Performance

The background index operation uses an incremental approach that is slower than the normal "foreground" index builds. If the index is larger than the available RAM, then the incremental process can take *much* longer than the foreground build.

If your application includes `ensureIndex()` operations, and an index *doesn't* exist for other operational concerns, building the index can have a severe impact on the performance of the database.

To avoid performance issues, make sure that your application checks for the indexes at start up using the `getIndexes()` method or the equivalent method for your driver[4] and terminates if the proper indexes do not exist. Always build indexes in production instances using separate application code, during designated maintenance windows.

Building Indexes on Secondaries

Background index operations on a *replica set primary* become foreground indexing operations on *secondary members* of the set. All indexing operations on secondaries block replication.

[4] http://api.mongodb.org/

To build large indexes on secondaries the best approach is to restart one secondary at a time in *standalone* mode and build the index. After building the index, restart as a member of the replica set, allow it to catch up with the other members of the set, and then build the index on the next secondary. When all the secondaries have the new index, step down the primary, restart it as a standalone, and build the index on the former primary.

Remember, the amount of time required to build the index on a secondary must be within the window of the *oplog*, so that the secondary can catch up with the primary.

Indexes on secondary members in "recovering" mode are always built in the foreground to allow them to catch up as soon as possible.

See *Build Indexes on Replica Sets* (page 33) for a complete procedure for building indexes on secondaries.

Drop Duplicates

MongoDB cannot create a *unique index* (page 23) on a field that has duplicate values. To force the creation of a unique index, you can specify the `dropDups` option, which will only index the first occurrence of a value for the key, and delete all subsequent values.

Important: As in all unique indexes, if a document does not have the indexed field, MongoDB will include it in the index with a "null" value.

If subsequent fields *do not* have the indexed field, and you have set `{dropDups: true}`, MongoDB will remove these documents from the collection when creating the index. If you combine `dropDups` with the *sparse* (page 24) option, this index will only include documents in the index that have the value, and the documents without the field will remain in the database.

To create a unique index that drops duplicates on the `username` field of the `accounts` collection, use a command in the following form:

```
db.accounts.ensureIndex( { username: 1 }, { unique: true, dropDups: true } )
```

> **Warning:** Specifying `{ dropDups: true }` will delete data from your database. Use with extreme caution.

By default, `dropDups` is `false`.

Index Names

The default name for an index is the concatenation of the indexed keys and each key's direction in the index, 1 or -1.

Example

Issue the following command to create an index on `item` and `quantity`:

```
db.products.ensureIndex( { item: 1, quantity: -1 } )
```

The resulting index is named: `item_1_quantity_-1`.

Optionally, you can specify a name for an index instead of using the default name.

Example

Issue the following command to create an index on `item` and `quantity` and specify `inventory` as the index name:

```
db.products.ensureIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

The resulting index has the name `inventory`.

To view the name of an index, use the `getIndexes()` method.

3 Indexing Tutorials

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection.

The documents in this section outline specific tasks related to building and maintaining indexes for data in MongoDB collections and discusses strategies and practical approaches. For a conceptual overview of MongoDB indexing, see the *Index Concepts* (page 7) document.

Index Creation Tutorials **(page 28)** Create and configure different types of indexes for different purposes.

Index Management Tutorials **(page 35)** Monitor and assess index performance and rebuild indexes as needed.

Geospatial Index Tutorials **(page 39)** Create indexes that support data stored as *GeoJSON* objects and legacy coordinate pairs.

Text Search Tutorials **(page 47)** Build and configure indexes that support full-text searches.

Indexing Strategies **(page 57)** The factors that affect index performance and practical approaches to indexing in MongoDB

3.1 Index Creation Tutorials

Instructions for creating and configuring indexes in MongoDB and building indexes on replica sets and sharded clusters.

Create an Index **(page 29)** Build an index for any field on a collection.

Create a Compound Index **(page 30)** Build an index of multiple fields on a collection.

Create a Unique Index **(page 30)** Build an index that enforces unique values for the indexed field or fields.

Create a Sparse Index **(page 31)** Build an index that omits references to documents that do not include the indexed field. This saves space when indexing fields that are present in only some documents.

Create a Hashed Index **(page 32)** Compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

Build Indexes on Replica Sets **(page 33)** To build indexes on a replica set, you build the indexes separately on the primary and the secondaries, as described here.

Build Indexes in the Background **(page 34)** Background index construction allows read and write operations to continue while building the index, but take longer to complete and result in a larger index.

Build Old Style Indexes **(page 35)** A `{v : 0}` index is necessary if you need to roll back from MongoDB version 2.0 (or later) to MongoDB version 1.8.

Create an Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB creates an index on the `_id` field of every collection by default, but allows users to create indexes for any collection using on any field in a *document*.

This tutorial describes how to create an index on a single field. MongoDB also supports *compound indexes* (page 11), which are indexes on multiple fields. See *Create a Compound Index* (page 30) for instructions on building compound indexes.

Create an Index on a Single Field

To create an index, use `ensureIndex()` or a similar method from your driver[5]. For example the following creates an index on the `phone-number` field of the `people` collection:

```
db.people.ensureIndex( { "phone-number": 1 } )
```

`ensureIndex()` only creates an index if an index of the same specification does not already exist.

All indexes support and optimize the performance for queries that select on this field. For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Tip

The value of the field in the index specification describes the kind of index for that field. For example, a value of `1` specifies an index that orders items in ascending order. A value of `-1` specifies an index that orders items in descending order.

Examples

If you create an index on the `user_id` field in the `records`, this index is, the index will support the following query:

```
db.records.find( { user_id: 2 } )
```

However, the following query, on the `profile_url` field is not supported by this index:

```
db.records.find( { profile_url: 2 } )
```

Additional Considerations

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background, as described in *Background Construction* (page 25). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 33) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 33).

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

See also:

Create a Compound Index (page 30), *Indexing Tutorials* (page 28) and *Index Concepts* (page 7) for more information.

[5] http://api.mongodb.org/

Create a Compound Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB supports indexes that include content on a single field, as well as *compound indexes* (page 11) that include content from multiple fields. Continue reading for instructions and examples of building a compound index.

Build a Compound Index

To create a *compound index* (page 11) use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

Example

The following operation will create an index on the `item`, `category`, and `price` fields of the `products` collection:

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

Additional Considerations

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background, as described in *Background Construction* (page 25). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 33) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 33).

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

Tip

The value of the field in the index specification describes the kind of index for that field. For example, a value of `1` specifies an index that orders items in ascending order. A value of `-1` specifies an index that orders items in descending order.

See also:

Create an Index (page 29), *Indexing Tutorials* (page 28) and *Index Concepts* (page 7) for more information.

Create a Unique Index

MongoDB allows you to specify a *unique constraint* (page 23) on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields. Additionally, if you want to create an index on a collection that has existing data that might have duplicate values for the indexed field, you may choose to combine unique enforcement with *duplicate dropping* (page 27).

Unique Indexes

To create a *unique indexes* (page 23), consider the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { unique: true } )
```

For example, you may want to create a unique index on the `"tax-id":` of the `accounts` collection to prevent storing multiple account records for the same legal entity:

```
db.accounts.ensureIndex( { "tax-id": 1 }, { unique: true } )
```

The *_id index* (page 10) is a unique index. In some situations you may consider using `_id` field itself for this kind of data rather than using a unique index on another field.

In many situations you will want to combine the `unique` constraint with the `sparse` option. When MongoDB indexes a field, if a document does not have a value for a field, the index entry for that item will be `null`. Since unique indexes cannot have duplicate values for a field, without the `sparse` option, MongoDB will reject the second document and all subsequent documents without the indexed field. Consider the following prototype.

```
db.collection.ensureIndex( { a: 1 }, { unique: true, sparse: true } )
```

You can also enforce a unique constraint on *compound indexes* (page 11), as in the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1 }, { unique: true } )
```

These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.

Drop Duplicates

To force the creation of a *unique index* (page 23) index on a collection with duplicate values in the field you are indexing you can use the `dropDups` option. This will force MongoDB to create a *unique* index by deleting documents with duplicate values when building the index. Consider the following prototype invocation of `ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { unique: true, dropDups: true } )
```

See the full documentation of *duplicate dropping* (page 27) for more information.

> **Warning:** Specifying `{ dropDups: true }` may delete data from your database. Use with extreme caution.

Refer to the `ensureIndex()` documentation for additional index creation options.

Create a Sparse Index

Sparse indexes are like non-sparse indexes, except that they omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings. See *Sparse Indexes* (page 24) for more information about sparse indexes and their use.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 28) for more information.

Prototype

To create a *sparse index* (page 24) on a field, use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

Example

The following operation, creates a sparse index on the `users` collection that *only* includes a document in the index if the `twitter_name` field exists in a document.

```
db.users.ensureIndex( { twitter_name: 1 }, { sparse: true } )
```

The index excludes all documents that do not include the `twitter_name` field.

Considerations

Note: Sparse indexes can affect the results returned by the query, particularly with respect to sorts on fields *not* included in the index. See the *sparse index* (page 24) section for more information.

Create a Hashed Index

New in version 2.4.

Hashed indexes (page 22) compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

See

sharding-hashed-sharding for more information about hashed indexes in sharded clusters, as well as *Index Concepts* (page 7) and *Indexing Tutorials* (page 28) for more information about indexes.

Procedure

To create a *hashed index* (page 22), specify `hashed` as the value of the index key, as in the following example:

Example

Specify a hashed index on `_id`

```
db.collection.ensureIndex( { _id: "hashed" } )
```

Considerations

MongoDB supports `hashed` indexes of any single field. The hashing function collapses sub-documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes.

You may not create compound indexes that have `hashed` index fields.

Build Indexes on Replica Sets

Background index creation operations (page 25) become *foreground* indexing operations on *secondary* members of replica sets. The foreground index building process blocks all replication and read operations on the secondaries while they build the index.

Secondaries will begin building indexes *after* the *primary* finishes building the index. In *sharded clusters*, the `mongos` will send `ensureIndex()` to the primary members of the replica set for each shard, which then replicate to the secondaries after the primary finishes building the index.

To minimize the impact of building an index on your replica set, use the following procedure to build indexes on secondaries:

See

Indexing Tutorials (page 28) and *Index Concepts* (page 7) for more information.

Considerations

- Ensure that your *oplog* is large enough to permit the indexing or re-indexing operation to complete without falling too far behind to catch up. See the *oplog sizing* documentation for additional information.
- This procedure *does* take one member out of the replica set at a time. However, this procedure will only affect one member of the set at a time rather than *all* secondaries at the same time.
- Do **not** use this procedure when building a *unique index* (page 23) with the `dropDups` option.

Procedure

Note: If you need to build an index in a *sharded cluster*, repeat the following procedure for each replica set that provides each *shard*.

Stop One Secondary Stop the `mongod` process on one secondary. Restart the `mongod` process *without* the `--replSet` option and running on a different port. [6] This instance is now in "standalone" mode.

For example, if your `mongod` *normally* runs with on the default port of `27017` with the `--replSet` option you would use the following invocation:

```
mongod --port 47017
```

Build the Index Create the new index using the `ensureIndex()` in the `mongo` shell, or comparable method in your driver. This operation will create or rebuild the index on this `mongod` instance

For example, to create an ascending index on the `username` field of the `records` collection, use the following `mongo` shell operation:

```
db.records.ensureIndex( { username: 1 } )
```

See also:

Create an Index (page 29) and *Create a Compound Index* (page 30) for more information.

[6] By running the `mongod` on a different port, you ensure that the other members of the replica set and all clients will not contact the member while you are building the index.

Restart the Program `mongod` When the index build completes, start the `mongod` instance with the `--replSet` option on its usual port:

```
mongod --port 27017 --replSet rs0
```

Modify the port number (e.g. `27017`) or the replica set name (e.g. `rs0`) as needed.

Allow replication to catch up on this member.

Build Indexes on all Secondaries For each secondary in the set, build an index according to the following steps:

1. *Stop One Secondary* (page 33)
2. *Build the Index* (page 33)
3. *Restart the Program mongod* (page 34)

Build the Index on the Primary To build an index on the primary you can either:

1. *Build the index in the background* (page 34) on the primary.
2. Step down the primary using the method:*rs.stepDown()* method in the `mongo` shell to cause the current primary to become a secondary graceful and allow the set to elect another member as primary.

   Then repeat the index building procedure, listed below, to build the index on the primary:

   (a) *Stop One Secondary* (page 33)

   (b) *Build the Index* (page 33)

   (c) *Restart the Program mongod* (page 34)

Building the index on the background, takes longer than the foreground index build and results in a less compact index structure. Additionally, the background index build may impact write performance on the primary. However, building the index in the background allows the set to be continuously up for write operations during while MongoDB builds the index.

Build Indexes in the Background

By default, MongoDB builds indexes in the foreground and prevent all read and write operations to the database while the index builds. Also, no operation that requires a read or write lock on all databases (e.g. **listDatabases**) can occur during a foreground index build.

Background index construction (page 25) allows read and write operations to continue while building the index.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 28) for more information.

Considerations

Background index builds take longer to complete and result in an index that is *initially* larger, or less compact, than an index built in the foreground. Overtime the compactness of indexes built in the background will approach foreground-built indexes.

After MongoDB finishes building the index, background-built indexes are functionally identical to any other index.

Procedure

To create an index in the background, add the `background` argument to the `ensureIndex()` operation, as in the following index:

```
db.collection.ensureIndex( { a: 1 }, { background: true } )
```

Consider the section on *background index construction* (page 25) for more information about these indexes and their implications.

Build Old Style Indexes

Important: Use this procedure *only* if you **must** have indexes that are compatible with a version of MongoDB earlier than 2.0.

MongoDB version 2.0 introduced the `{v:1}` index format. MongoDB versions 2.0 and later support both the `{v:1}` format and the earlier `{v:0}` format.

MongoDB versions prior to 2.0, however, support only the `{v:0}` format. If you need to roll back MongoDB to a version prior to 2.0, you must *drop* and *re-create* your indexes.

To build pre-2.0 indexes, use the `dropIndexes()` and `ensureIndex()` methods. You *cannot* simply reindex the collection. When you reindex on versions that only support `{v:0}` indexes, the `v` fields in the index definition still hold values of `1`, even though the indexes would now use the `{v:0}` format. If you were to upgrade again to version 2.0 or later, these indexes would not work.

Example

Suppose you rolled back from MongoDB 2.0 to MongoDB 1.8, and suppose you had the following index on the `items` collection:

```
{ "v" : 1, "key" : { "name" : 1 }, "ns" : "mydb.items", "name" : "name_1" }
```

The `v` field tells you the index is a `{v:1}` index, which is incompatible with version 1.8.

To drop the index, issue the following command:

```
db.items.dropIndex( { name : 1 } )
```

To recreate the index as a `{v:0}` index, issue the following command:

```
db.foo.ensureIndex( { name : 1 } , { v : 0 } )
```

See also:

2.0-new-index-format.

3.2 Index Management Tutorials

Instructions for managing indexes and assessing index performance and use.

Remove Indexes **(page 36)** Drop an index from a collection.

Rebuild Indexes **(page 36)** In a single operation, drop all indexes on a collection and then rebuild them.

Manage In-Progress Index Creation **(page 37)** Check the status of indexing progress, or terminate an ongoing index build.

Return a List of All Indexes **(page 37)** Obtain a list of all indexes on a collection or of all indexes on all collections in a database.

Measure Index Use **(page 38)** Study query operations and observe index use for your database.

Remove Indexes

To remove an index from a collection use the `dropIndex()` method and the following procedure. If you simply need to rebuild indexes you can use the process described in the *Rebuild Indexes* (page 36) document.

See also:

Indexing Tutorials (page 28) and *Index Concepts* (page 7) for more information about indexes and indexing operations in MongoDB.

Operations

To remove an index, use the `db.collection.dropIndex()` method, as in the following example:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

This will remove the index on the `"tax-id"` field in the `accounts` collection. The shell provides the following document after completing the operation:

```
{ "nIndexesWas" : 3, "ok" : 1 }
```

Where the value of `nIndexesWas` reflects the number of indexes *before* removing this index. You can also use the `db.collection.dropIndexes()` to remove *all* indexes, except for the *_id index* (page 10) from a collection.

These shell helpers provide wrappers around the `dropIndexes` *database command*. Your `client library` may have a different or additional interface for these operations.

Rebuild Indexes

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method to rebuild all indexes on a collection in a single operation. This operation drops all indexes, including the *_id index* (page 10), and then rebuilds all indexes.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 28).

Process

The operation takes the following form:

```
db.accounts.reIndex()
```

MongoDB will return the following document when the operation completes:

```
{
        "nIndexesWas" : 2,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 2,
        "indexes" : [
                {
```

```
                        "key" : {
                                "_id" : 1,
                                "tax-id" : 1
                        },
                        "ns" : "records.accounts",
                        "name" : "_id_"
                }
        ],
        "ok" : 1
}
```

This shell helper provides a wrapper around the `reIndex` *database command*. Your `client library` may have a different or additional interface for this operation.

Additional Considerations

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 33).

Manage In-Progress Index Creation

To see the status of the indexing processes, you can use the `db.currentOp()` method in the `mongo` shell. The value of the `query` field and the `msg` field will indicate if the operation is an index build. The `msg` field also indicates the percent of the build that is complete.

To terminate an ongoing index build, use the `db.killOp()` method in the `mongo` shell.

For more information see `db.currentOp()`.

Changed in version 2.4: Before MongoDB 2.4, you could *only* terminate *background* index builds. After 2.4, you can terminate any index build, including foreground index builds.

Return a List of All Indexes

When performing maintenance you may want to check which indexes exist on a collection. Every index on a collection has a corresponding *document* in the `system.indexes` collection, and you can use standard queries (i.e. `find()`) to list the indexes, or in the `mongo` shell, the `getIndexes()` method to return a list of the indexes on a collection, as in the following examples.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 28) for more information about indexes in MongoDB and common index management operations.

List all Indexes on a Collection

To return a list of all indexes on a collection, use the `db.collection.getIndexes()` method or a similar method for your driver[7].

For example, to view all indexes on the `people` collection:

```
db.people.getIndexes()
```

[7] http://api.mongodb.org/

List all Indexes for a Database

To return a list of all indexes on all collections in a database, use the following operation in the `mongo` shell:

```
db.system.indexes.find()
```

See `system.indexes` for more information about these documents.

Measure Index Use

Synopsis

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides a number of tools that allow you to study query operations and observe index use for your database.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 28) for more information.

Operations

Return Query Plan with `explain()` Append the `explain()` method to any cursor (e.g. query) to return a document with statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

Control Index Use with `hint()` Append the `hint()` to any cursor (e.g. query) with the index as the argument to *force* MongoDB to use a specific index to fulfill the query. Consider the following example:

```
db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } ).hint( { zipcode: 1 } )
```

You can use `hint()` and `explain()` in conjunction with each other to compare the effectiveness of a specific index. Specify the `$natural` operator to the `hint()` method to prevent MongoDB from using *any* index:

```
db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } ).hint( { $natural: 1 } )
```

Instance Index Use Reporting MongoDB provides a number of metrics of index use and operation that you may want to consider when analyzing index use for your database:

- In the output of `serverStatus`:
  - `indexCounters`
  - `scanned`
  - `scanAndOrder`
- In the output of `collStats`:
  - `totalIndexSize`
  - `indexSizes`
- In the output of `dbStats`:
  - `dbStats.indexes`
  - `dbStats.indexSize`

3.3 Geospatial Index Tutorials

Instructions for creating and querying `2d`, `2dsphere`, and haystack indexes.

Create a 2dsphere Index **(page 39)** A `2dsphere` index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

Query a 2dsphere Index **(page 39)** Search for locations within, near, or intersected by a GeoJSON shape, or within a circle as defined by coordinate points on a sphere.

Create a 2d Index **(page 41)** Create a `2d` index to support queries on data stored as legacy coordinate pairs.

Query a 2d Index **(page 42)** Search for locations using legacy coordinate pairs.

Create a Haystack Index **(page 44)** A haystack index is optimized to return results over small areas. For queries that use spherical geometry, a `2dsphere` index is a better option.

Query a Haystack Index **(page 45)** Search based on location and non-location data within a small area.

Calculate Distance Using Spherical Geometry **(page 45)** Convert distances to radians and back again.

Create a `2dsphere` Index

To create a geospatial index for GeoJSON-formatted data, use the `ensureIndex()` method and set the value of the location field for your collection to `2dsphere`. A `2dsphere` index can be a *compound index* (page 11) and does not require the location field to be the first field indexed.

To create the index use the following syntax:

```
db.points.ensureIndex( { <location field> : "2dsphere" } )
```

The following are four example commands for creating a `2dsphere` index:

```
db.points.ensureIndex( { loc : "2dsphere" } )
db.points.ensureIndex( { loc : "2dsphere" , type : 1 } )
db.points.ensureIndex( { rating : 1 , loc : "2dsphere" } )
db.points.ensureIndex( { loc : "2dsphere" , rating : 1 , category : -1 } )
```

The first example creates a simple geospatial index on the location field `loc`. The second example creates a compound index where the second field contains non-location data. The third example creates an index where the location field is not the primary field: the location field does not have to be the first field in a `2dsphere` index. The fourth example creates a compound index with three fields. You can include as many fields as you like in a `2dsphere` index.

Query a `2dsphere` Index

The following sections describe queries supported by the `2dsphere` index. For an overview of recommended geospatial queries, see *geospatial-query-compatibility-chart*.

GeoJSON Objects Bounded by a Polygon

The `$geoWithin` operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

```
db.<collection>.find( { <location field> :
                        { $geoWithin :
                          { $geometry :
                            { type : "Polygon" ,
```

```
                              coordinates : [ <coordinates> ]
                    } } } } )
```

The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates : [ [
                                          [ 0 , 0 ] ,
                                          [ 3 , 6 ] ,
                                          [ 6 , 1 ] ,
                                          [ 0 , 0 ]
                                        ] ]
                } } } } )
```

Intersections of GeoJSON Objects

New in version 2.4.

The `$geoIntersects` operator queries for locations that intersect a specified GeoJSON object. A location intersects the object if the intersection is non-empty. This includes documents that have a shared edge.

The `$geoIntersects` operator uses the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoIntersects :
                           { $geometry :
                             { type : "<GeoJSON object type>" ,
                               coordinates : [ <coordinates> ]
                    } } } } )
```

The following example uses `$geoIntersects` to select all indexed points and shapes that intersect with the polygon defined by the `coordinates` array.

```
db.places.find( { loc :
                  { $geoIntersects :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates: [ [
                                         [ 0 , 0 ] ,
                                         [ 3 , 6 ] ,
                                         [ 6 , 1 ] ,
                                         [ 0 , 0 ]
                                       ] ]
                } } } } )
```

Proximity to a GeoJSON Point

Proximity queries return the points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a `2dsphere` index.

To query for proximity to a GeoJSON point, use either the `$near` operator or `geoNear` command. Distance is in meters.

The `$near` uses the following syntax:

```
db.<collection>.find( { <location field> :
                          { $near :
                            { $geometry :
                               { type : "Point" ,
                                 coordinates : [ <longitude> , <latitude> ] } ,
                              $maxDistance : <distance in meters>
                        } } } )
```

For examples, see `$near`.

The `geoNear` command uses the following syntax:

```
db.runCommand( { geoNear : <collection> ,
                 near : { type : "Point" ,
                          coordinates: [ <longitude>, <latitude> ] } ,
                 spherical : true } )
```

The `geoNear` command offers more options and returns more information than does the `$near` operator. To run the command, see `geoNear`.

Points within a Circle Defined on a Sphere

To select all grid coordinates in a "spherical cap" on a sphere, use `$geoWithin` with the `$centerSphere` operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distance Using Spherical Geometry* (page 45).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                          { $geoWithin :
                            { $centerSphere :
                               [ [ <x>, <y> ] , <radius> ] }
                        } } )
```

The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude `88 W` and latitude `30 N`. The example converts the distance, 10 miles, to radians by dividing by the approximate radius of the earth, 3959 miles:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $centerSphere :
                       [ [ 88 , 30 ] , 10 / 3959 ]
                } } } )
```

Create a `2d` Index

To build a geospatial `2d` index, use the `ensureIndex()` method and specify `2d`. Use the following syntax:

```
db.<collection>.ensureIndex( { <location field> : "2d" ,
                               <additional field> : <value> } ,
                             { <index-specification options> } )
```

The `2d` index uses the following optional index-specification options:

```
{ min : <lower bound> , max : <upper bound> ,
  bits : <bit precision> }
```

Define Location Range for a 2d Index

By default, a 2d index assumes longitude and latitude and has boundaries of -180 inclusive and 180 non-inclusive (i.e. [-180 , 180]). If documents contain coordinate data outside of the specified range, MongoDB returns an error.

Important: The default boundaries allow applications to insert documents with invalid latitudes greater than 90 or less than -90. The behavior of geospatial queries with such invalid points is not defined.

On 2d indexes you can change the location range.

You can build a 2d geospatial index with a location range other than the default. Use the min and max options when creating the index. Use the following syntax:

```
db.collection.ensureIndex( { <location field> : "2d" } ,
                           { min : <lower bound> , max : <upper bound> } )
```

Define Location Precision for a 2d Index

By default, a 2d index on legacy coordinate pairs uses 26 bits of precision, which is roughly equivalent to 2 feet or 60 centimeters of precision using the default range of -180 to 180. Precision is measured by the size in bits of the *geohash* values used to store location data. You can configure geospatial indexes with up to 32 bits of precision.

Index precision does not affect query accuracy. The actual grid coordinates are always used in the final query processing. Advantages to lower precision are a lower processing overhead for insert operations and use of less space. An advantage to higher precision is that queries scan smaller portions of the index to return results.

To configure a location precision other than the default, use the bits option when creating the index. Use following syntax:

```
db.<collection>.ensureIndex( {<location field> : "<index type>"} ,
                             { bits : <bit precision> } )
```

For information on the internals of geohash values, see *Calculation of Geohash Values for 2d Indexes* (page 20).

Query a 2d Index

The following sections describe queries supported by the 2d index. For an overview of recommended geospatial queries, see *geospatial-query-compatibility-chart*.

Points within a Shape Defined on a Flat Surface

To select all legacy coordinate pairs found within a given shape on a flat surface, use the $geoWithin operator along with a shape operator. Use the following syntax:

```
db.<collection>.find( { <location field> :
                        { $geoWithin :
                           { $box|$polygon|$center : <coordinates>
                    } } } )
```

The following queries for documents within a rectangle defined by `[ 0 , 0 ]` at the bottom left corner and by `[ 100 , 100 ]` at the top right corner.

```
db.places.find( { loc :
                  { $geoWithin :
                     { $box : [ [ 0 , 0 ] ,
                                [ 100 , 100 ] ]
                  } } } )
```

The following queries for documents that are within the circle centered on `[ -74 , 40.74 ]` and with a radius of `10`:

```
db.places.find( { loc: { $geoWithin :
                           { $center : [ [-74, 40.74 ] , 10 ]
                } } } )
```

For syntax and examples for each shape, see the following:

- `$box`
- `$polygon`
- `$center` (defines a circle)

Points within a Circle Defined on a Sphere

MongoDB supports rudimentary spherical queries on flat `2d` indexes for legacy reasons. In general, spherical calculations should use a `2dsphere` index, as described in *2dsphere Indexes* (page 17).

To query for legacy coordinate pairs in a "spherical cap" on a sphere, use `$geoWithin` with the `$centerSphere` operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distance Using Spherical Geometry* (page 45).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                        { $geoWithin :
                           { $centerSphere : [ [ <x>, <y> ] , <radius> ] }
                     } } )
```

The following example query returns all documents within a 10-mile radius of longitude `88 W` and latitude `30 N`. The example converts distance to radians by dividing distance by the approximate radius of the earth, 3959 miles:

```
db.<collection>.find( { loc : { $geoWithin :
                                { $centerSphere :
                                   [ [ 88 , 30 ] , 10 / 3959 ]
                     } } } )
```

Proximity to a Point on a Flat Surface

Proximity queries return the 100 legacy coordinate pairs closest to the defined point and sort the results by distance. Use either the `$near` operator or `geoNear` command. Both require a `2d` index.

The `$near` operator uses the following syntax:

```
db.<collection>.find( { <location field> :
                        { $near : [ <x> , <y> ]
                      } } )
```

For examples, see `$near`.

The `geoNear` command uses the following syntax:

```
db.runCommand( { geoNear: <collection>, near: [ <x> , <y> ] } )
```

The `geoNear` command offers more options and returns more information than does the `$near` operator. To run the command, see `geoNear`.

Exact Matches on a Flat Surface

You can use the `db.collection.find()` method to query for an exact match on a location. These queries use the following syntax:

```
db.<collection>.find( { <location field>: [ <x> , <y> ] } )
```

This query will return any documents with the value of `[ <x> , <y> ]`.

Create a Haystack Index

To build a haystack index, use the `bucketSize` option when creating the index. A `bucketSize` of `5` creates an index that groups location values that are within 5 units of the specified longitude and latitude. The `bucketSize` also determines the granularity of the index. You can tune the parameter to the distribution of your data so that in general you search only very small regions. The areas defined by buckets can overlap. A document can exist in multiple buckets.

A haystack index can reference two fields: the location field and a second field. The second field is used for exact matches. Haystack indexes return documents based on location and an exact match on a single additional criterion. These indexes are not necessarily suited to returning the closest documents to a particular location.

To build a haystack index, use the following syntax:

```
db.coll.ensureIndex( { <location field> : "geoHaystack" ,
                       <additional field> : 1 } ,
                     { bucketSize : <bucket value> } )
```

Example

If you have a collection with documents that contain fields similar to the following:

```
{ _id : 100, pos: { lng : 126.9, lat : 35.2 } , type : "restaurant"}
{ _id : 200, pos: { lng : 127.5, lat : 36.1 } , type : "restaurant"}
{ _id : 300, pos: { lng : 128.0, lat : 36.7 } , type : "national park"}
```

The following operations create a haystack index with buckets that store keys within 1 unit of longitude or latitude.

```
db.places.ensureIndex( { pos : "geoHaystack", type : 1 } ,
                       { bucketSize : 1 } )
```

This index stores the document with an `_id` field that has the value `200` in two different buckets:

- In a bucket that includes the document where the `_id` field has a value of `100`
- In a bucket that includes the document where the `_id` field has a value of `300`

To query using a haystack index you use the `geoSearch` command. See *Query a Haystack Index* (page 45).

By default, queries that use a haystack index return 50 documents.

Query a Haystack Index

A haystack index is a special `2d` geospatial index that is optimized to return results over small areas. To create a haystack index see *Create a Haystack Index* (page 44).

To query a haystack index, use the `geoSearch` command. You must specify both the coordinates and the additional field to `geoSearch`. For example, to return all documents with the value `restaurant` in the `type` field near the example point, the command would resemble:

```
db.runCommand( { geoSearch : "places" ,
                 search : { type: "restaurant" } ,
                 near : [-74, 40.74] ,
                 maxDistance : 10 } )
```

Note: Haystack indexes are not suited to queries for the complete list of documents closest to a particular location. The closest documents could be more distant compared to the bucket size.

Note: *Spherical query operations* (page 45) are not currently supported by haystack indexes.

The `find()` method and `geoNear` command cannot access the haystack index.

Calculate Distance Using Spherical Geometry

Note: While basic queries using spherical distance are supported by the `2d` index, consider moving to a `2dsphere` index if your data is primarily longitude and latitude.

The `2d` index supports queries that calculate distances on a Euclidean plane (flat surface). The index also supports the following query operators and command that calculate distances using spherical geometry:

- `$nearSphere`
- `$centerSphere`
- `$near`
- `geoNear` command with the `{ spherical: true }` option.

Important: These three queries use radians for distance. Other query types do not.

For spherical query operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

- *distance to radians*: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
- *radians to distance*: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The radius of the Earth is approximately `3,959` miles or `6,371` kilometers.

The following query would return documents from the `places` collection within the circle described by the center `[ -74, 40.74 ]` with a radius of `100` miles:

```
db.places.find( { loc: { $geoWithin: { $centerSphere: [ [ -74, 40.74 ] ,
                                                         100 / 3959 ] } } } )
```

You may also use the `distanceMultiplier` option to the `geoNear` to convert radians in the `mongod` process, rather than in your application code. See *distance multiplier* (page 46).

The following spherical query, returns all documents in the collection `places` within `100` miles from the point `[ -74, 40.74 ]`.

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true
               }  )
```

The output of the above command would be:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 0.01853688938212826,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

Warning: Spherical queries that wrap around the poles or at the transition from `-180` to `180` longitude raise an error.

Note: While the default Earth-like bounds for geospatial indexes are between `-180` inclusive, and `180`, valid values for latitude are between `-90` and `90`.

Distance Multiplier

The `distanceMultiplier` option of the `geoNear` command returns distances only after multiplying the results by an assigned value. This allows MongoDB to return converted values, and removes the requirement to convert units in application logic.

Using `distanceMultiplier` in spherical queries provides results from the `geoNear` command that do not need radian-to-distance conversion. The following example uses `distanceMultiplier` in the `geoNear` command with a *spherical* (page 45) example:

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true,
                 distanceMultiplier: 3959
               }  )
```

The output of the above operation would resemble the following:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 73.46525170413567,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

3.4 Text Search Tutorials

Instructions for enabling MongoDB's text search feature, and for building and configuring text indexes.

Enable Text Search **(page 48)** You must explicitly enable text search in order to search string content in collections.

Create a text Index **(page 48)** A `text` index allows searches on text strings in the index's specified fields.

Search String Content for Text **(page 49)** Use queries to find strings of text within collections.

Specify a Language for Text Index **(page 52)** The specified language determines the list of stop words and the rules for Text Search's stemmer and tokenizer.

Create text Index with Long Name **(page 53)** Override the `text` index name limit for long index names.

Control Search Results with Weights **(page 54)** Give priority to certain search values by denoting the significance of an indexed field relative to other indexed fields

Limit the Number of Entries Scanned **(page 55)** Search only those documents that match a set of filter conditions.

Create text Index to Cover Queries **(page 56)** Perform text searches that return results without the need to scan documents.

Enable Text Search

New in version 2.4.

The *text search* (page 22) is currently a *beta* feature. As a beta feature:

- You need to explicitly enable the feature before *creating a text index* (page 21) or using the `text` command.
- To enable text search on `replica sets` and `sharded clusters`, you need to enable on **each and every** `mongod` for replica sets and on **each and every** `mongos` for sharded clusters.

Warning:
- Do **not** enable or use text search on production systems.
- Text indexes have significant storage requirements and performance costs. See *Storage Requirements and Performance Costs* (page 21) for more information.

You can enable the text search feature at startup with the `textSearchEnabled` parameter:

```
mongod --setParameter textSearchEnabled=true
```

You may prefer to set the `textSearchEnabled` parameter in the `configuration file`.

Additionally, you can enable the feature in the `mongo` shell with the `setParameter` command. This command does **not** propagate from the primary to the secondaries. You must enable on **each and every** `mongod` for replica sets.

Note: You must set the parameter every time you start the server. You may prefer to add the parameter to the `configuration files`.

Create a `text` Index

You can create a `text` index on the field or fields whose value is a string or an array of string elements. When creating a `text` index on multiple fields, you can specify the individual fields or you can wildcard specifier (`$**`).

Index Specific Fields

The following example creates a `text` index on the fields `subject` and `content`:

```
db.collection.ensureIndex(
                           {
                             subject: "text",
                             content: "text"
                           }
                         )
```

This `text` index catalogs all string data in the `subject` field and the `content` field, where the field value is either a string or an array of string elements.

Index All Fields

To allow for text search on all fields with string content, use the wildcard specifier (`$**`) to index all fields that contain string content.

The following example indexes any string value in the data of every field of every document in `collection` and names the index `TextIndex`:

```
db.collection.ensureIndex(
                            { "$**": "text" },
                            { name: "TextIndex" }
                          )
```

Search String Content for Text

In 2.4, you can enable the text search feature to create `text` indexes and issue text queries using the `text`.

The following tutorial offers various query patterns for using the text search feature.

The examples in this tutorial use a collection `quotes` that has a `text` index on the fields `quote` that contains a string and `related_quotes` that contains an array of string elements.

Note: You cannot combine the `text` command, which requires a special *text index* (page 21), with a query operator that requires a different type of special index. For example you cannot combine `text` with the `$near` operator.

Search for a Term

The following command searches for the word `TOMORROW`:

```
db.quotes.runCommand( "text", { search: "TOMORROW" } )
```

Because `text` command is case-insensitive, the text search will match the following document in the `quotes` collection:

```
{
  "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),
  "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",
  "related_quotes" : [
                       "is this a dagger which I see before me",
                       "the handle toward my hand?"
                     ],
  "src" : {
            "title" : "Macbeth",
            "from" : "Act V, Scene V"
          },
  "speaker" : "macbeth"
}
```

Match Any of the Search Terms

If the search string is a space-delimited text, `text` command performs a logical `OR` search on each term and returns documents that contains any of the terms.

For example, the search string `"tomorrow largo"` searches for the term `tomorrow` **OR** the term `largo`:

```
db.quotes.runCommand( "text", { search: "tomorrow largo" } )
```

The command will match the following documents in the `quotes` collection:

```
{
  "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),
  "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",
  "related_quotes" : [
```

```
                      "is this a dagger which I see before me",
                      "the handle toward my hand?"
                    ],
  "src" : {
              "title" : "Macbeth",
              "from" : "Act V, Scene V"
            },
  "speaker" : "macbeth"
}

{
  "_id" : ObjectId("50ecf0cd8abea0fda30ceab4"),
  "quote" : "Es tan corto el amor y es tan largo el olvido.",
  "related_quotes" : [
                        "Como para acercarla mi mirada la busca.",
                        "Mi corazón la busca, y ella no está conmigo."
                      ],
  "speaker" : "Pablo Neruda",
  "src" : {
              "title" : "Veinte poemas de amor y una canción desesperada",
              "from" : "Poema 20"
            }
}
```

Match Phrases

To match the exact phrase that includes a space(s) as a single term, escape the quotes.

For example, the following command searches for the exact phrase `"and tomorrow"`:

```
db.quotes.runCommand( "text", { search: "\"and tomorrow\"" } )
```

If the search string contains both phrases and individual terms, the `text` command performs a compound logical `AND` of the phrases with the compound logical `OR` of the single terms, including the individual terms from each phrase.

For example, the following search string contains both individual terms `corto` and `largo` as well as the phrase `\"and tomorrow\"`:

```
db.quotes.runCommand( "text", { search: "corto largo \"and tomorrow\"" } )
```

The `text` command performs the equivalent to the following logical operation, where the individual terms `corto`, `largo`, as well as the term `tomorrow` from the phrase `"and tomorrow"`, are part of a logical `OR` expression:

```
(corto OR largo OR tomorrow) AND ("and tomorrow")
```

As such, the results for this search will include documents that only contain the phrase `"and tomorrow"` as well as documents that contain the phrase `"and tomorrow"` and the terms `corto` and/or `largo`. Documents that contain the phrase `"and tomorrow"` as well as the terms `corto` and `largo` will generally receive a higher score for this search.

Match Some Words But Not Others

A *negated* term is a term that is prefixed by a minus sign `-`. If you negate a term, the `text` command will exclude the documents that contain those terms from the results.

Note: If the search text contains *only* negated terms, the `text` command will not return any results.

The following example returns those documents that contain the term `tomorrow` but **not** the term `petty`.

```
db.quotes.runCommand( "text" , { search: "tomorrow -petty" } )
```

Limit the Number of Matching Documents in the Result Set

Note: The result from the `text` command must fit within the maximum `BSON Document Size`.

By default, the `text` command will return up to 100 matching documents, from highest to lowest scores. To override this default limit, use the `limit` option in the `text` command, as in the following example:

```
db.quotes.runCommand( "text", { search: "tomorrow", limit: 2 } )
```

The `text` command will return at most `2` of the *highest scoring* results.

The `limit` can be any number as long as the result set fits within the maximum `BSON Document Size`.

Specify Which Fields to Return in the Result Set

In the `text` command, use the `project` option to specify the fields to include (`1`) or exclude (`0`) in the matching documents.

Note: The `_id` field is always returned unless explicitly excluded in the `project` document.

The following example returns only the `_id` field and the `src` field in the matching documents:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                                project: { "src": 1 } } )
```

Search with Additional Query Conditions

The `text` command can also use the `filter` option to specify additional query conditions.

The following example will return the documents that contain the term `tomorrow` **AND** the `speaker` is `macbeth`:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                                filter: { "speaker" : "macbeth" } } )
```

See also:

Limit the Number of Entries Scanned (page 55)

Search for Text in Specific Languages

You can specify the language that determines the tokenization, stemming, and removal of stop words, as in the following example:

```
db.quotes.runCommand( "text", { search: "amor", language: "spanish" } )
```

See *text-search-languages* for a list of supported languages as well as *Specify a Language for Text Index* (page 52) for specifying languages for the `text` index.

Text Search Output

The `text` command returns a document that contains the result set.

See *text-search-output* for information on the output.

Specify a Language for Text Index

This tutorial describes how to *specify the default language associated with the text index* (page 52) and also how to *create text indexes for collections that contain documents in different languages* (page 52).

Specify the Default Language for a `text` Index

The default language associated with the indexed data determines the list of stop words and the rules for the stemmer and tokenizer. The default language for the indexed data is `english`.

To specify a different language, use the `default_language` option when creating the `text` index. See *text-search-languages* for the languages available for `default_language`.

The following example creates a `text` index on the `content` field and sets the `default_language` to `spanish`:

```
db.collection.ensureIndex(
                            { content : "text" },
                            { default_language: "spanish" }
                          )
```

Create a `text` Index for a Collection in Multiple Languages

Specify the Index Language within the Document If a collection contains documents that are in different languages, include a field in the documents that contain the language to use:

- If you include a field named `language` in the document, by default, the `ensureIndex()` method will use the value of this field to override the default language.
- To use a field with a name other than `language`, you must specify the name of this field to the `ensureIndex()` method with the `language_override` option.

See *text-search-languages* for a list of supported languages.

Include the `language` Field Include a field `language` that specifies the language to use for the individual documents.

For example, the documents of a multi-language collection `quotes` contain the field `language`:

```
{ _id: 1, language: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, language: "spanish", quote: "Nada hay más surreal que la realidad." }
{ _id: 3, language: "english", quote: "is this a dagger which I see before me" }
```

Create a `text` index on the field `quote`:

```
db.quotes.ensureIndex( { quote: "text" } )
```

- For the documents that contain the `language` field, the `text` index uses that language to determine the stop words and the rules for the stemmer and the tokenizer.

- For documents that do not contain the `language` field, the index uses the default language, which is English, to determine the stop words and rules for the stemmer and the tokenizer.

For example, the Spanish word `que` is a stop word. So the following `text` command would not match any document:

```
db.quotes.runCommand( "text", { search: "que", language: "spanish" } )
```

Use any Field to Specify the Language for a Document Include a field that specifies the language to use for the individual documents. To use a field with a name other than `language`, include the `language_override` option when creating the index.

For example, the documents of a multi-language collection `quotes` contain the field `idioma`:

```
{ _id: 1, idioma: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, idioma: "spanish", quote: "Nada hay más surreal que la realidad." }
{ _id: 3, idioma: "english", quote: "is this a dagger which I see before me" }
```

Create a `text` index on the field `quote` with the `language_override` option:

```
db.quotes.ensureIndex( { quote : "text" },
                       { language_override: "idioma" } )
```

- For the documents that contain the `idioma` field, the `text` index uses that language to determine the stop words and the rules for the stemmer and the tokenizer.
- For documents that do not contain the `idioma` field, the index uses the default language, which is English, to determine the stop words and rules for the stemmer and the tokenizer.

For example, the Spanish word `que` is a stop word. So the following `text` command would not match any document:

```
db.quotes.runCommand( "text", { search: "que", language: "spanish" } )
```

Create `text` Index with Long Name

The default name for the index consists of each indexed field name concatenated with `_text`. For example, the following command creates a `text` index on the fields `content`, `users.comments`, and `users.profiles`:

```
db.collection.ensureIndex(
                            {
                              content: "text",
                              "users.comments": "text",
                              "users.profiles": "text"
                            }
                          )
```

The default name for the index is:

```
"content_text_users.comments_text_users.profiles_text"
```

To avoid creating an index with a name that exceeds the `index name length limit`, you can pass the `name` option to the `db.collection.ensureIndex()` method:

```
db.collection.ensureIndex(
                            {
                              content: "text",
                              "users.comments": "text",
                              "users.profiles": "text"
                            },
                            {
```

```
                    name: "MyTextIndex"
                  }
                )
```

Note: To drop the `text` index, use the index name. To get the name of an index, use `db.collection.getIndexes()`.

Control Search Results with Weights

This document describes how to create a `text` index with specified weights for results fields.

By default, the `text` command returns matching documents based on scores, from highest to lowest. For a `text` index, the *weight* of an indexed field denotes the significance of the field relative to the other indexed fields in terms of the score. The score for a given word in a document is derived from the weighted sum of the frequency for each of the indexed fields in that document.

The default weight is 1 for the indexed fields. To adjust the weights for the indexed fields, include the `weights` option in the `db.collection.ensureIndex()` method.

Warning: Choose the weights carefully in order to prevent the need to reindex.

A collection `blog` has the following documents:

```
{ _id: 1,
  content: "This morning I had a cup of coffee.",
  about: "beverage",
  keywords: [ "coffee" ]
}

{ _id: 2,
  content: "Who doesn't like cake?",
  about: "food",
  keywords: [ "cake", "food", "dessert" ]
}
```

To create a `text` index with different field weights for the `content` field and the `keywords` field, include the `weights` option to the `ensureIndex()` method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```
db.blog.ensureIndex(
                      {
                        content: "text",
                        keywords: "text",
                        about: "text"
                      },
                      {
                        weights: {
                                    content: 10,
                                    keywords: 5,
                                  },
                        name: "TextIndex"
                      }
                    )
```

The `text` index has the following fields and weights:

- `content` has a weight of 10,
- `keywords` has a weight of 5, and
- `about` has the default weight of 1.

These weights denote the relative significance of the indexed fields to each other. For instance, a term match in the `content` field has:

- `2` times (i.e. `10:5`) the impact as a term match in the `keywords` field and
- `10` times (i.e. `10:1`) the impact as a term match in the `about` field.

Limit the Number of Entries Scanned

This tutorial describes how to limit the text search to scan only those documents with a field value.

The `text` command includes the `filter` option to further restrict the results of a text search. For a `filter` that specifies equality conditions, this tutorial demonstrates how to perform text searches on only those documents that match the `filter` conditions, as opposed to performing a text search first on all the documents and then matching on the `filter` condition.

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

```
{ _id: 1, dept: "tech", description: "a fun green computer" }
{ _id: 2, dept: "tech", description: "a wireless red mouse" }
{ _id: 3, dept: "kitchen", description: "a green placemat" }
{ _id: 4, dept: "kitchen", description: "a red peeler" }
{ _id: 5, dept: "food", description: "a green apple" }
{ _id: 6, dept: "food", description: "a red potato" }
```

A common use case is to perform text searches by individual departments, such as:

```
db.inventory.runCommand( "text", {
                                    search: "green",
                                    filter: { dept : "kitchen" }
                                  }
                         )
```

To limit the text search to scan only those documents within a specific `dept`, create a compound index that specifies an ascending/descending index key on the field `dept` and a `text` index key on the field `description`:

```
db.inventory.ensureIndex(
                            {
                              dept: 1,
                              description: "text"
                            }
                          )
```

Important:

- The ascending/descending index keys must be listed before, or prefix, the `text` index keys.
- By prefixing the `text` index fields with ascending/descending index fields, MongoDB will **only** index documents that have the prefix fields.
- You cannot include *multi-key* (page 13) index fields or *geospatial* (page 16) index fields.
- The `text` command **must** include the `filter` option that specifies an **equality** condition for the prefix fields.

Then, the text search within a particular department will limit the scan of indexed documents. For example, the following `text` command scans only those documents with `dept` equal to `kitchen`:

```
db.inventory.runCommand( "text", {
                                    search: "green",
                                    filter: { dept : "kitchen" }
                                  }
                       )
```

The returned result includes the statistics that shows that the command scanned 1 document, as indicated by the `nscanned` field:

```
{

  "queryDebugString" : "green||||||",
  "language" : "english",
  "results" : [
                {
                  "score" : 0.75,
                  "obj" : {
                             "_id" : 3,
                             "dept" : "kitchen",
                             "description" : "a green placemat"
                           }
                }
              ],
  "stats" : {
              "nscanned" : 1,
              "nscannedObjects" : 0,
              "n" : 1,
              "nfound" : 1,
              "timeMicros" : 211
            },
  "ok" : 1
}
```

For more information on the result set, see *text-search-output*.

Create `text` Index to Cover Queries

To create a `text` index that can *cover queries* (page 58):

1. Append scalar index fields to a `text` index, as in the following example which specifies an ascending index key on `username`:

   ```
   db.collection.ensureIndex( { comments: "text",
                                username: 1 } )
   ```

   Warning: You cannot include *multi-key* (page 13) index field or *geospatial* (page 16) index field.

2. Use the `project` option in the `text` to return only the fields in the index, as in the following:

   ```
   db.quotes.runCommand( "text", { search: "tomorrow",
                                   project: { username: 1,
                                              _id: 0
                                            }
                                 }
                       )
   ```

Note: By default, the `_id` field is included in the result set. Since the example index did not include the `_id` field, you must explicitly exclude the field in the `project` document.

3.5 Indexing Strategies

The best indexes for your application must take a number of factors into account, including the kinds of queries you expect, the ratio of reads to writes, and the amount of free memory on your system.

When developing your indexing strategy you should have a deep understanding of your application's queries. Before you build indexes, map out the types of queries you will run so that you can build indexes that reference those fields. Indexes come with a performance cost, but are more than worth the cost for frequent queries on large data set. Consider the relative frequency of each query in the application and whether the query justifies an index.

The best overall strategy for designing indexes is to profile a variety of index configurations with data sets similar to the ones you'll be running in production to see which configurations perform best.Inspect the current indexes created for your collections to ensure they are supporting your current and planned queries. If an index is no longer used, drop the index.

MongoDB can only use *one* index to support any given operation. However, each clause of an `$or` query may use a different index.

The following documents introduce indexing strategies:

Create Indexes to Support Your Queries **(page 57)** An index supports a query when the index contains all the fields scanned by the query. Creating indexes that supports queries results in greatly increased query performance.

Use Indexes to Sort Query Results **(page 59)** To support efficient queries, use the strategies here when you specify the sequential order and sort order of index fields.

Ensure Indexes Fit in RAM **(page 61)** When your index fits in RAM, the system can avoid reading the index from disk and you get the fastest processing.

Create Queries that Ensure Selectivity **(page 62)** Selectivity is the ability of a query to narrow results using the index. Selectivity allows MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

Create Indexes to Support Your Queries

An index supports a query when the index contains all the fields scanned by the query. The query scans the index and not the collection. Creating indexes that supports queries results in greatly increased query performance.

This document describes strategies for creating indexes that support queries.

Create a Single-Key Index if All Queries Use the Same, Single Key

If you only ever query on a single key in a given collection, then you need to create just one single-key index for that collection. For example, you might create an index on `category` in the `product` collection:

```
db.products.ensureIndex( { "category": 1 } )
```

Create Compound Indexes to Support Several Different Queries

If you sometimes query on only one key and at other times query on that key combined with a second key, then creating a compound index is more efficient than creating a single-key index. MongoDB will use the compound index for both queries. For example, you might create an index on both `category` and `item`.

```
db.products.ensureIndex( { "category": 1, "item": 1 } )
```

This allows you both options. You can query on just `category`, and you also can query on `category` combined with `item`. A single *compound index* (page 11) on multiple fields can support all the queries that search a "prefix" subset of those fields.

Note: With the exception of queries that use the `$or` operator, a query does not use multiple indexes. A query uses only one index.

Example

The following index on a collection:

```
{ x: 1, y: 1, z: 1 }
```

Can support queries that the following indexes support:

```
{ x: 1 }
{ x: 1, y: 1 }
```

There are some situations where the prefix indexes may offer better query performance: for example if `z` is a large array.

The `{ x: 1, y: 1, z: 1 }` index can also support many of the same queries as the following index:

```
{ x: 1, z: 1 }
```

Also, `{ x: 1, z: 1 }` has an additional use. Given the following query:

```
db.collection.find( { x: 5 } ).sort( { z: 1} )
```

The `{ x: 1, z: 1 }` index supports both the query and the sort operation, while the `{ x: 1, y: 1, z: 1 }` index only supports the query. For more information on sorting, see *Use Indexes to Sort Query Results* (page 59).

Create Indexes that Support Covered Queries

A covered query is a query in which:

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

Because the index "covers" the query, MongoDB can both match the *query conditions* **and** return the results using only the index; MongoDB does not need to look at the documents, only the index, to fulfill the query. An index can also cover an *aggregation pipeline operation* on unsharded collections.

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.

MongoDB automatically uses an index that covers a query when possible. To ensure that an index can *cover* a query, create an index that includes all the fields listed in the *query document* and in the query result. You can specify the fields to return in the query results with a *projection* document. By default, MongoDB includes the `_id` field in the query result. So, if the index does **not** include the `_id` field, then you must exclude the `_id` field (i.e. `_id: 0`) from the query results.

Example

Given collection `users` with an index on the fields `user` and `status`, as created by the following option:

```
db.users.ensureIndex( { status: 1, user: 1 } )
```

Then, this index will cover the following query which selects on the `status` field and returns only the `user` field:

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

In the operation, the projection document explicitly specifies `_id: 0` to exclude the `_id` field from the result since the index is only on the `status` and the `user` fields.

If the projection document does not specify the exclusion of the `_id` field, the query returns the `_id` field. The following query is **not** covered by the index on the `status` and the `user` fields because with the projection document `{ user: 1 }`, the query returns both the `user` field and the `_id` field:

```
db.users.find( { status: "A" }, { user: 1 } )
```

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* (page 13) index and cannot support a covered query.
- any of the indexed fields are fields in subdocuments. To index fields in subdocuments, use *dot notation*. For example, consider a collection `users` with documents of the following form:

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

  The collection has the following indexes:

  ```
  { user: 1 }
  ```

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

  The `{ user: 1 }` index covers the following query:

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

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

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

  The query, however, does use the `{ "user.login": 1 }` index to find matching documents.

To determine whether a query is a covered query, use the `explain()` method. If the `explain()` output displays `true` for the `indexOnly` field, the query is covered by an index, and MongoDB queries only that index to match the query **and** return the results.

For more information see *Measure Index Use* (page 38).

Use Indexes to Sort Query Results

In MongoDB sort operations that sort documents based on an indexed field provide the greatest performance. Indexes in MongoDB, as in other databases, have an order: as a result, using an index to access documents returns in the same order as the index.

To sort on multiple fields, create a *compound index* (page 11). With compound indexes, the results can be in the sorted order of either the full index or an index prefix. An index prefix is a subset of a compound index; the subset consists of one or more fields at the start of the index, in order. For example, given an index `{ a:1, b: 1, c: 1, d: 1 }`, the following subsets are index prefixes:

```
{ a: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 1, c: 1 }
```

For more information on sorting by index prefixes, see *Sort Subset Starts at the Index Beginning* (page 60).

If the query includes **equality** match conditions on an index prefix, you can sort on a subset of the index that starts after or overlaps with the prefix. For example, given an index `{ a: 1, b: 1, c: 1, d: 1 }`, if the query condition includes equality match conditions on `a` and `b`, you can specify a sort on the subsets `{ c: 1 }` or `{ c: 1, d: 1 }`:

```
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1 } )
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1, d: 1 } )
```

In these operations, the equality match and the sort documents together cover the index prefixes `{ a: 1, b: 1, c: 1 }` and `{ a: 1, b: 1, c: 1, d: 1 }` respectively.

You can also specify a sort order that includes the prefix; however, since the query condition specifies equality matches on these fields, they are constant in the resulting documents and do not contribute to the sort order:

```
db.collection.find( { a: 5, b: 3 } ).sort( { a: 1, b: 1, c: 1 } )
db.collection.find( { a: 5, b: 3 } ).sort( { a: 1, b: 1, c: 1, d: 1 } )
```

For more information on sorting by index subsets that are not prefixes, see *Sort Subset Does Not Start at the Index Beginning* (page 61).

Note: For in-memory sorts that do not use an index, the `sort()` operation is significantly slower. The `sort()` operation will abort when it uses 32 megabytes of memory.

Sort With a Subset of Compound Index

If the sort document contains a subset of the compound index fields, the subset can determine whether MongoDB can use the index efficiently to both retrieve and sort the query results. If MongoDB can efficiently use the index to both retrieve and sort the query results, the output from the `explain()` will display `scanAndOrder` as `false` or `0`. If MongoDB can only use the index for retrieving documents that meet the query criteria, MongoDB must manually sort the resulting documents without the use of the index. For in-memory sort operations, `explain()` will display `scanAndOrder` as `true` or `1`.

Sort Subset Starts at the Index Beginning If the sort document is a subset of a compound index and starts from the beginning of the index, MongoDB can use the index to both retrieve and sort the query results.

For example, the collection `collection` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

The following operations include a sort with a subset of the index. Because the sort subset starts at beginning of the index, the operations can use the index for both the query retrieval and sort:

```
db.collection.find().sort( { a:1 } )
db.collection.find().sort( { a:1, b:1 } )
db.collection.find().sort( { a:1, b:1, c:1 } )

db.collection.find( { a: 4 } ).sort( { a: 1, b: 1 } )
db.collection.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )
```

```
db.collection.find( { b: 5 } ).sort( { a: 1, b: 1 } )
db.collection.find( { b: { $gt:5 }, c: { $gt: 1 } } ).sort( { a: 1, b: 1 } )
```

The last two operations include query conditions on the field `b` but does not include a query condition on the field `a`:

```
db.collection.find( { b: 5 } ).sort( { a: 1, b: 1 } )
db.collection.find( { b: { $gt:5 }, c: { $gt: 1 } } ).sort( { a: 1, b: 1 } )
```

Consider the case where the collection has the index `{ b: 1 }` in addition to the `{ a: 1, b: 1, c: 1, d: 1 }` index. Because of the query condition on `b`, it is not immediately obvious which index MongoDB may select as the "best" index. To explicitly specify the index to use, see `hint()`.

Sort Subset Does Not Start at the Index Beginning The sort document can be a subset of a compound index that does **not** start from the beginning of the index. For instance, `{ c: 1 }` is a subset of the index `{ a: 1, b: 1, c: 1, d: 1 }` that omits the preceding index fields `a` and `b`. MongoDB can use the index efficiently **if** the query document includes all the preceding fields of the index, in this case `a` and `b`, in **equality** conditions. In other words, the equality conditions in the query document and the subset in the sort document **contiguously** cover a prefix of the index.

For example, the collection `collection` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

Then following operations can use the index efficiently:

```
db.collection.find( { a: 5 } ).sort( { b: 1, c: 1 } )
db.collection.find( { a: 5, c: 4, b: 3 } ).sort( { d: 1 } )
```

- In the first operation, the query document `{ a: 5 }` with the sort document `{ b: 1, c: 1 }` cover the prefix `{ a:1 , b: 1, c: 1 }` of the index.
- In the second operation, the query document `{ a: 5, c: 4, b: 3 }` with the sort document `{ d: 1 }` covers the full index.

Only the index fields preceding the sort subset must have the equality conditions in the query document. The other index fields may have other conditions. The following operations can efficiently use the index since the equality conditions in the query document and the subset in the sort document **contiguously** cover a prefix of the index:

```
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1 } )
db.collection.find( { a: 5, b: 3, c: { $lt: 4 } } ).sort( { c: 1 } )
```

The following operations specify a sort document of `{ c: 1 }`, but the query documents do not contain equality matches on the **preceding** index fields `a` and `b`:

```
db.collection.find( { a: { $gt: 2 } } ).sort( { c: 1 } )
db.collection.find( { c: 5 } ).sort( { c: 1 } )
```

These operations **will not** efficiently use the index `{ a: 1, b: 1, c: 1, d: 1 }` and may not even use the index to retrieve the documents.

Ensure Indexes Fit in RAM

For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.

To check the size of your indexes, use the `db.collection.totalIndexSize()` helper, which returns data in bytes:

```
> db.collection.totalIndexSize()
4294976499
```

The above example shows an index size of almost 4.3 gigabytes. To ensure this index fits in RAM, you must not only have more than that much RAM available but also must have RAM available for the rest of the *working set*. Also remember:

If you have and use multiple collections, you must consider the size of all indexes on all collections. The indexes and the working set must be able to fit in memory at the same time.

There are some limited cases where indexes do not need to fit in memory. See *Indexes that Hold Only Recent Values in RAM* (page 62).

See also:

`collStats` and `db.collection.stats()`

Indexes that Hold Only Recent Values in RAM

Indexes do not have to fit *entirely* into RAM in all cases. If the value of the indexed field increments with every insert, and most queries select recently added documents; then MongoDB only needs to keep the parts of the index that hold the most recent or "right-most" values in RAM. This allows for efficient index use for read and write operations and minimize the amount of RAM required to support the index.

Create Queries that Ensure Selectivity

Selectivity is the ability of a query to narrow results using the index. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

To ensure selectivity, write queries that limit the number of possible documents with the indexed field. Write queries that are appropriately selective relative to your indexed data.

Example

Suppose you have a field called `status` where the possible values are `new` and `processed`. If you add an index on `status` you've created a low-selectivity index. The index will be of little help in locating records.

A better strategy, depending on your queries, would be to create a *compound index* (page 11) that includes the low-selectivity field and another field. For example, you could create a compound index on `status` and `created_at`.

Another option, again depending on your use case, might be to use separate collections, one for each status.

Example

Consider an index `{ a : 1 }` (i.e. an index on the key `a` sorted in ascending order) on a collection where `a` has three values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 1, b: "cd" }
{ _id: ObjectId(), a: 1, b: "ef" }
{ _id: ObjectId(), a: 2, b: "jk" }
{ _id: ObjectId(), a: 2, b: "lm" }
{ _id: ObjectId(), a: 2, b: "no" }
{ _id: ObjectId(), a: 3, b: "pq" }
{ _id: ObjectId(), a: 3, b: "rs" }
{ _id: ObjectId(), a: 3, b: "tv" }
```

If you query for `{ a: 2, b: "no" }` MongoDB must scan 3 *documents* in the collection to return the one matching result. Similarly, a query for `{ a: { $gt: 1}, b: "tv" }` must scan 6 documents, also to return one result.

Consider the same index on a collection where `a` has *nine* values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 2, b: "cd" }
{ _id: ObjectId(), a: 3, b: "ef" }
{ _id: ObjectId(), a: 4, b: "jk" }
{ _id: ObjectId(), a: 5, b: "lm" }
{ _id: ObjectId(), a: 6, b: "no" }
{ _id: ObjectId(), a: 7, b: "pq" }
{ _id: ObjectId(), a: 8, b: "rs" }
{ _id: ObjectId(), a: 9, b: "tv" }
```

If you query for `{ a: 2, b: "cd" }`, MongoDB must scan only one document to fulfill the query. The index and query are more selective because the values of `a` are evenly distributed *and* the query can select a specific document using the index.

However, although the index on `a` is more selective, a query such as `{ a: { $gt: 5 }, b: "tv" }` would still need to scan 4 documents.

If overall selectivity is low, and if MongoDB must read a number of documents to return results, then some queries may perform faster without indexes. To determine performance, see *Measure Index Use* (page 38).

For a conceptual introduction to indexes in MongoDB see *Index Concepts* (page 7).

4 Indexing Reference

4.1 Indexing Methods in the `mongo` Shell

Name	Description
`db.collection.createIndex()`	Builds an index on a collection. Use `db.collection.ensureIndex()`.
`db.collection.dropIndex()`	Removes a specified index on a collection.
`db.collection.dropIndexes()`	Removes all indexes on a collection.
`db.collection.ensureIndex()`	Creates an index if it does not currently exist. If the index exists `ensureIndex()` does nothing.
`db.collection.getIndexes()`	Returns an array of documents that describe the existing indexes on a collection.
`db.collection.getIndexStats()`	Renders a human-readable view of the data collected by `indexStats` which reflects B-tree utilization.
`db.collection.indexStats()`	Renders a human-readable view of the data collected by `indexStats` which reflects B-tree utilization.
`db.collection.reIndex()`	Rebuilds all existing indexes on a collection.
`db.collection.totalIndexSize()`	Reports the total size used by the indexes on a collection. Provides a wrapper around the `totalIndexSize` field of the `collStats` output.
`cursor.explain()`	Reports on the query execution plan, including index use, for a cursor.
`cursor.hint()`	Forces MongoDB to use a specific index for a query.
`cursor.max()`	Specifies an exclusive upper index bound for a cursor. For use with `cursor.hint()`
`cursor.min()`	Specifies an inclusive lower index bound for a cursor. For use with `cursor.hint()`
`cursor.snapshot()`	Forces the cursor to use the index on the `_id` field. Ensures that the cursor returns each document, with regards to the value of the `_id` field, only once.

4.2 Indexing Database Commands

Name	Description
`dropIndexes`	Removes indexes from a collection.
`compact`	Defragments a collection and rebuilds the indexes.
`reIndex`	Rebuilds all indexes on a collection.
`validate`	Internal command that scans for a collection's data and indexes for correctness.
`indexStats`	Experimental command that collects and aggregates statistics on all indexes.
`geoNear`	Performs a geospatial query that returns the documents closest to a given point.
`geoSearch`	Performs a geospatial query that uses MongoDB's *haystack index* functionality.
`geoWalk`	An internal command to support geospatial queries.
`checkShardingIndex`	Internal command that validates index on shard key.

4.3 Geospatial Query Selectors

Name	Description
`$geoWithin`	Selects geometries within a bounding *GeoJSON* geometry.
`$geoIntersects`	Selects geometries that intersect with a *GeoJSON* geometry.
`$near`	Returns geospatial objects in proximity to a point.
`$nearSphere`	Returns geospatial objects in proximity to a point on a sphere.

4.4 Indexing Query Modifiers

Name	Description
`$explain`	Forces MongoDB to report on query execution plans. See `explain()`.
`$hint`	Forces MongoDB to use a specific index. See `hint()`
`$max`	Specifies an *exclusive* upper limit for the index to use in a query. See `max()`.
`$min`	Specifies an *inclusive* lower limit for the index to use in a query. See `min()`.
`$returnKey`	Forces the cursor to only return fields included in the index.
`$snapshot`	Forces the query to use the index on the `_id` field. See `snapshot()`.

Index

Symbols

C

G

I

R

T