Indexes and MongoDB

Release 3.2.1

MongoDB, Inc.

February 01, 2016

Contents

CHAPTER 1

Index Introduction

On this page

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

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. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

The following diagram illustrates a query that selects and orders the matching documents using an index:

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.

1.1 Index Types

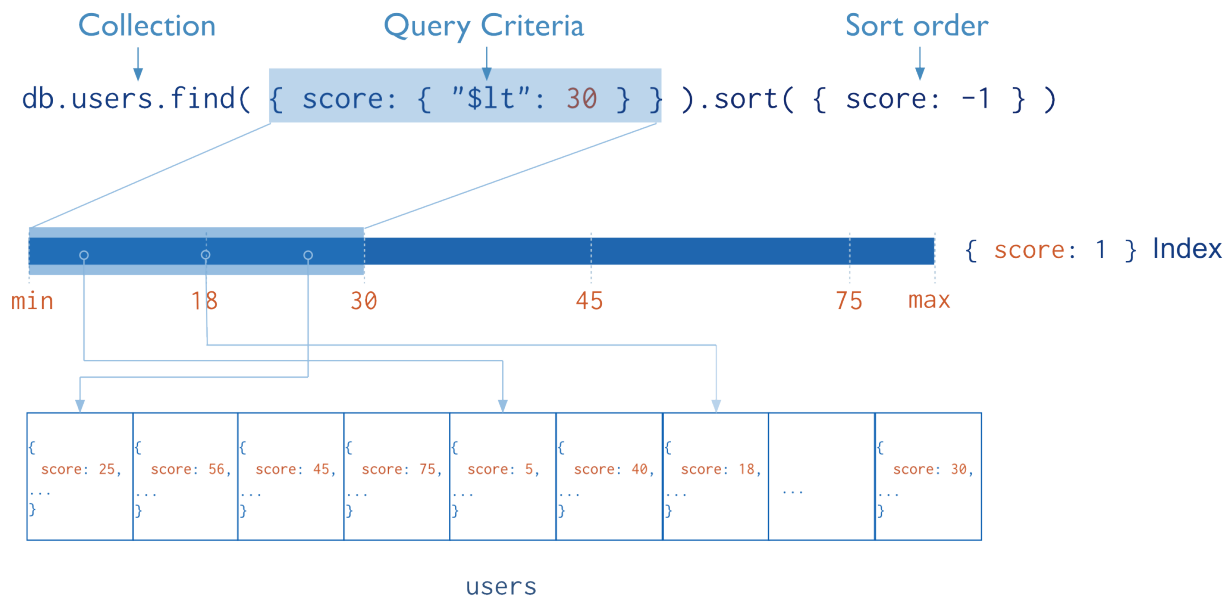
MongoDB provides a number of different index types to support specific types of data and queries.

1.1.1 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.

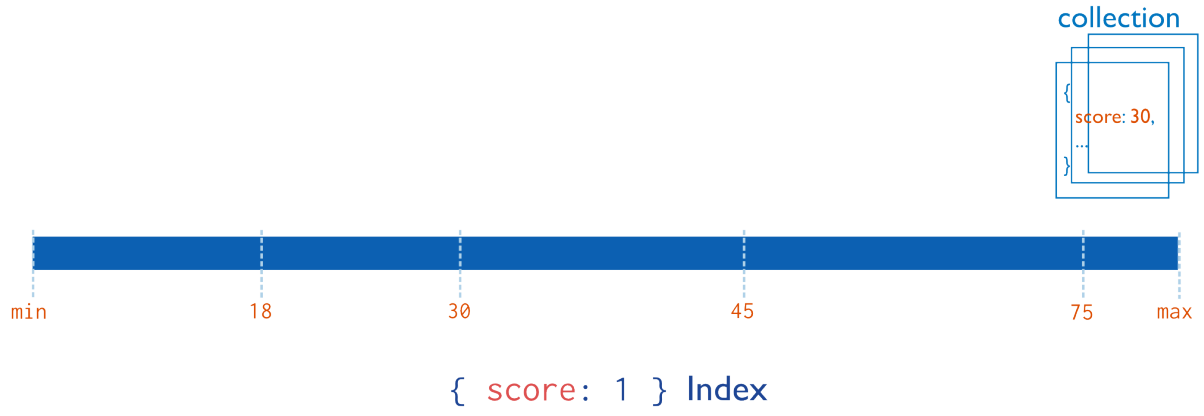
The `_id` index is *unique* and prevents clients from inserting two documents with the same value for the `_id` field.

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



1.1.2 Single Field

In addition to the MongoDB-defined `_id` index, MongoDB supports the creation of user-defined ascending/descending indexes on a *single field of a document* (page 8).



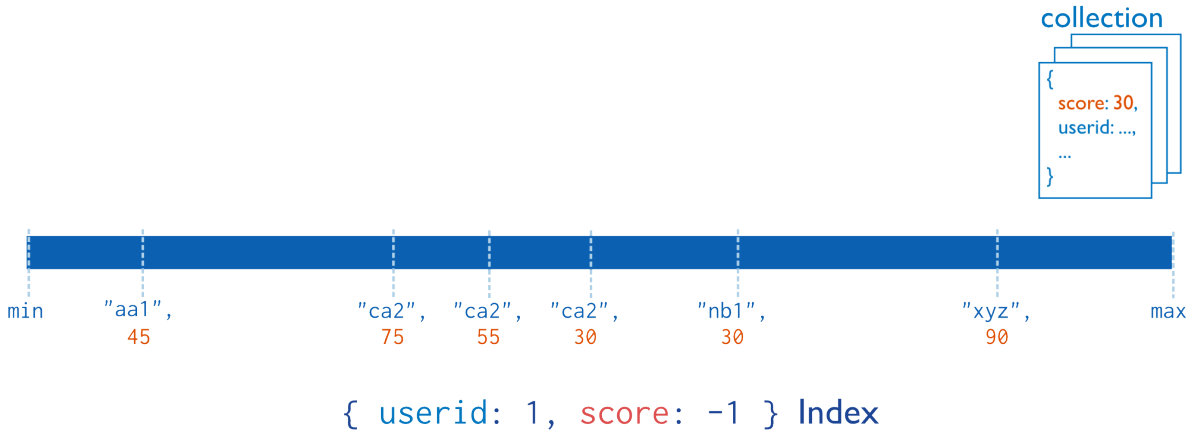
For a single-field index and sort operations, the sort order (i.e. ascending or descending) of the index key does not matter because MongoDB can traverse the index in either direction.

See *Single Field Indexes* (page 8) and *Sort with a Single Field Index* (page 93) for more information on single-field indexes.

1.1.3 Compound Index

MongoDB also supports user-defined indexes on multiple fields, i.e. *compound indexes* (page 10).

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, sorts by `score`.

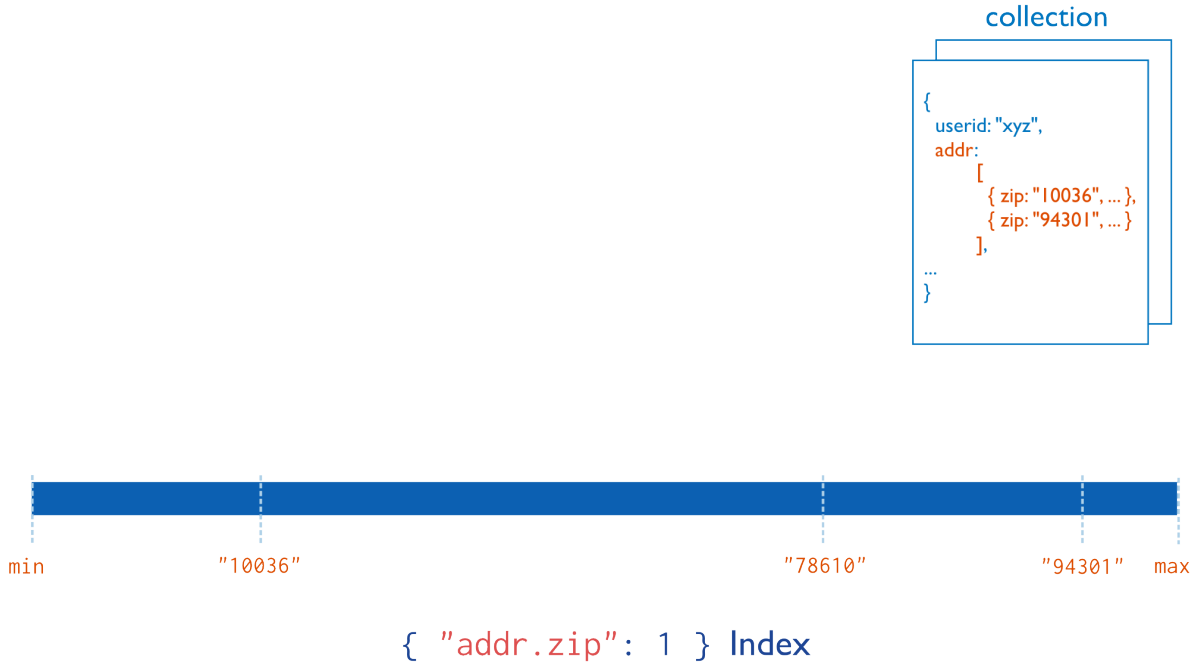


For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support a sort operation. See *Sort Order* (page 11) for more information on the impact of index order on results in compound indexes.

See *Compound Indexes* (page 10) and *Sort on Multiple Fields* (page 94) for more information on compound indexes.

1.1.4 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.



See *Multikey Indexes* (page 13) and *Multikey Index Bounds* (page 43) for more information on multikey indexes.

1.1.5 Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: *2d indexes* (page 21) that uses planar geometry when returning results and *2sphere indexes* (page 19) that use spherical geometry to return results.

See *2d Index Internals* (page 23) for a high level introduction to geospatial indexes.

1.1.6 Text Indexes

MongoDB provides a `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 24) for more information on text indexes and search.

1.1.7 Hashed Indexes

To support *hash based sharding*, MongoDB provides a *hashed index* (page 28) 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

1.2.1 Unique Indexes

The *unique* (page 31) property for an index causes MongoDB to reject duplicate values for the indexed field. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

1.2.2 Partial Indexes

New in version 3.2.

Partial indexes (page 32) only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance.

Partial indexes offer a superset of the functionality of sparse indexes and should be preferred over sparse indexes.

1.2.3 Sparse Indexes

The *sparse* (page 36) 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.

1.2.4 TTL Indexes

TTL indexes (page 29) are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

See: `https://docs.mongodb.org/manual/tutorial/expire-data` for implementation instructions.

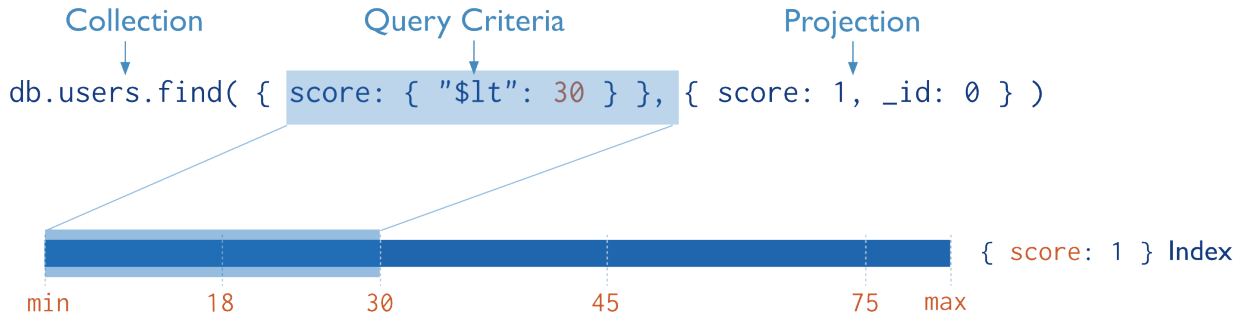
1.3 Index Use

Indexes can improve the efficiency of read operations. The `https://docs.mongodb.org/manual/tutorial/analyze-que` tutorial provides an example of the execution statistics of a query with and without an index.

For information on how MongoDB chooses an index to use, see *query optimizer*.

1.4 Covered Queries

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.



For more information on covered queries, see *read-operations-covered-query*.

1.5 Index Intersection

New in version 2.6.

MongoDB can use the *intersection of indexes* (page 41) to fulfill queries. For queries that specify compound query conditions, if one index can fulfill a part of a query condition, and another index can fulfill another part of the query condition, then MongoDB can use the intersection of the two indexes to fulfill the query. Whether the use of a compound index or the use of an index intersection is more efficient depends on the particular query and the system.

For details on index intersection, see *Index Intersection* (page 41).

1.6 Restrictions

Certain restrictions apply to indexes, such as the length of the index keys or the number of indexes per collection. See *Index Limitations* for details.

CHAPTER 2

Index Concepts

On this page

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 1). For operational instructions, see *Indexing Tutorials* (page 49). The *Indexing Reference* (page 99) 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 8)** 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 10)** A compound index includes more than one field of the documents in a collection.
- ***Multikey Indexes*** **(page 13)** A multikey index is an index on an array field, adding an index key for each value in the array.
- ***Geospatial Indexes and Queries*** **(page 16)** Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.
- ***Text Indexes*** **(page 24)** Text indexes support search of string content in documents.
- ***Hashed Index*** **(page 28)** Hashed indexes maintain entries with hashes of the values of the indexed field and are primarily used with sharded clusters to support hashed shard keys.

Index Properties **(page 29)** The properties you can specify when building indexes.

- ***TTL Indexes*** **(page 29)** The TTL index is used for TTL collections, which expire data after a period of time.
- ***Unique Indexes*** **(page 31)** A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.
- ***Partial Indexes*** **(page 32)** A partial index indexes only documents that meet specified filter criteria.
- ***Sparse Indexes*** **(page 36)** A sparse index does not index documents that do not have the indexed field.

Index Creation **(page 38)** The options available when creating indexes.

Index Intersection **(page 41)** The use of index intersection to fulfill a query.

Multikey Index Bounds **(page 43)** The computation of bounds on a multikey index scan.

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 embedded document.

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 `createIndex()` method. For more detailed instructions about building indexes, see the *Indexing Tutorials* (page 49) page.

Single Field Indexes **(page 8)** 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 10)** A compound index includes more than one field of the documents in a collection.

Multikey Indexes **(page 13)** A multikey index is an index on an array field, adding an index key for each value in the array.

Geospatial Indexes and Queries **(page 16)** Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.

Text Indexes **(page 24)** Text indexes support search of string content in documents.

Hashed Index **(page 28)** Hashed indexes maintain entries with hashes of the values of the indexed field and are primarily used with sharded clusters to support hashed shard keys.

2.1.1 Single Field Indexes

On this page

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 9), 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 this index-type supports. This document describes ascending/descending indexes that contain a single field. Consider the following illustration of a single field index.

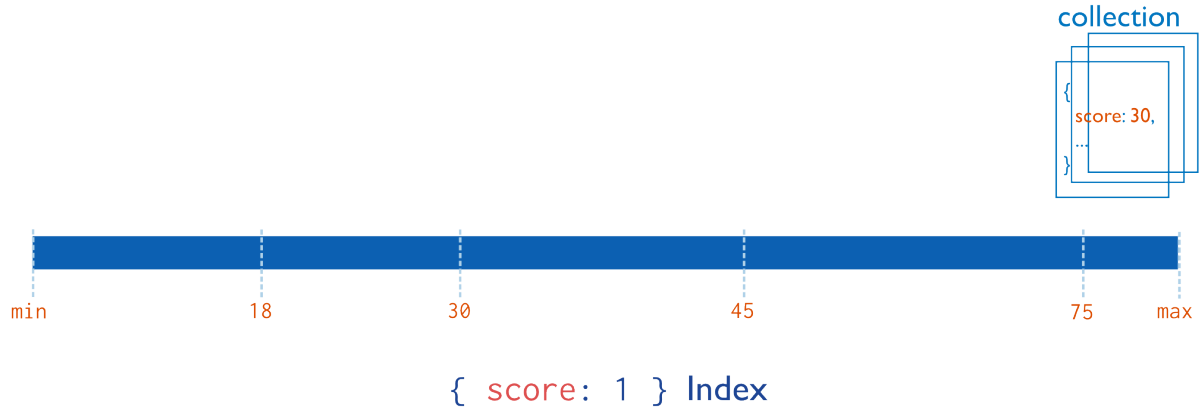
See also:

Compound Indexes (page 10) for information about indexes that include multiple fields, and *Index Introduction* (page 1) 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.createIndex( { "name" : 1 } )
```

Cases

`_id` Field Index

MongoDB creates the `_id` index, which is an ascending *unique index* (page 31) on the `_id` field, for all collections when the collection is created. You cannot remove the index on the `_id` field.

Think of the `_id` field as the *primary key* for a 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* which is generated when the client inserts the document. An *ObjectId* is a 12-byte unique identifier 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 collections 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 within embedded documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from *indexes on embedded documents* (page 10), which include the full content up to the maximum `index size` of the embedded document in the index. Instead, indexes on embedded fields allow you to use a "dot notation," to introspect into embedded 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.createIndex( { "address.zipcode": 1 } )
```

Indexes on Embedded Documents

You can also create indexes on embedded documents.

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

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

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

```
db.factories.createIndex( { 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 embedded documents, field order matters and the embedded documents 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-embedded-documents* for more information regarding querying on embedded documents.

2.1.2 Compound Indexes

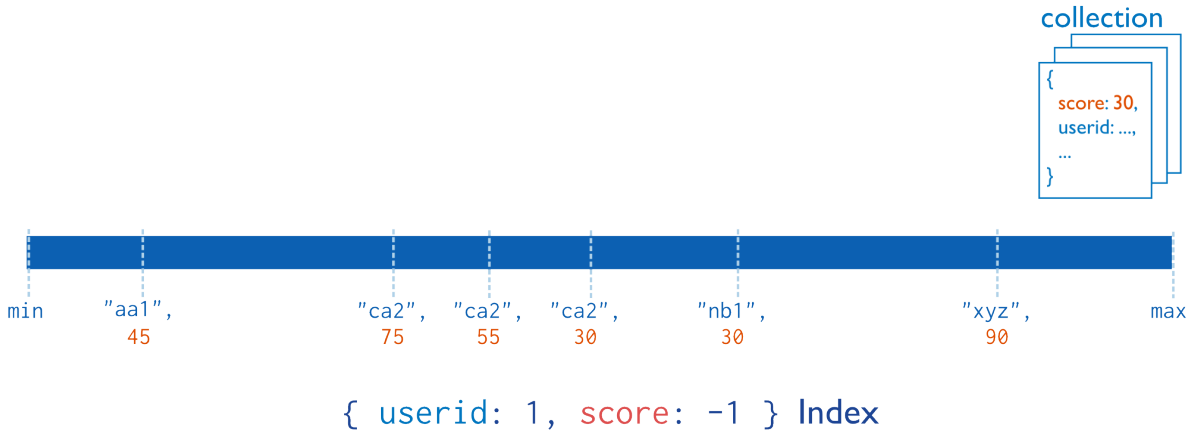
On this page

MongoDB supports *compound indexes*, where a single index structure holds references to multiple fields [1] within a collection's documents. The following diagram illustrates an example of a compound index on two fields:

Compound indexes can support queries that match on multiple fields.

Example

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



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(...)
}
```

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.createIndex( { "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 28).

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 11) 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 10), 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.createIndex( { "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 } )
```

For more information on sort order and compound indexes, see *Use Indexes to Sort Query Results* (page 93).

Prefixes

Index prefixes are the *beginning* subsets of indexed fields. For example, consider the following compound index:

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

The index has the following index prefixes:

- `{ item: 1 }`
- `{ item: 1, location: 1 }`

For a compound index, MongoDB can use the index to support queries on the index prefixes. As such, MongoDB can use the index for queries on the following fields:

- the `item` field,
- the `item` field *and* the `location` field,
- the `item` field *and* the `location` field *and* the `stock` field.

MongoDB can also use the index to support a query on `item` and `stock` fields since `item` field corresponds to a prefix. However, the index would not be as efficient in supporting the query as would be an index on only `item` and `stock`.

However, MongoDB cannot use the index to support queries that include the following fields since without the `item` field, none of the listed fields correspond to a prefix index:

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

If you have a collection that has both a compound index and an index on its prefix (e.g. `{ a: 1, b: 1 }` and `{ a: 1 }`), if neither index has a sparse or unique constraint, then you can remove the index on the prefix (e.g. `{ a: 1 }`). MongoDB will use the compound index in all of the situations that it would have used the prefix index.

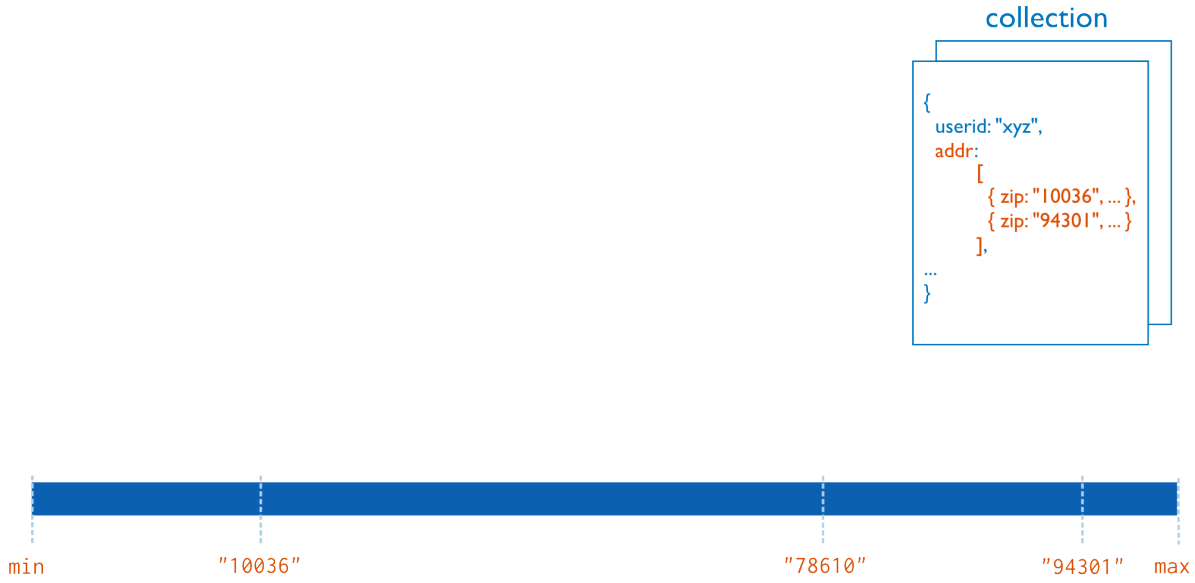
Index Intersection

Starting in version 2.6, MongoDB can use *index intersection* (page 41) to fulfill queries. The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system. See *Index Intersection and Compound Indexes* (page 42) for more details.

2.1.3 Multikey Indexes

On this page

To index a field that holds an array value, MongoDB creates an index key for each element in the array. These *multikey* indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) *and* nested documents.



Create Multikey Index

To create a multikey index, use the `db.collection.createIndex()` method:

```
db.coll.createIndex( { <field>: < 1 or -1 > } )
```

MongoDB automatically creates a multikey index if any indexed field is an array; you do not need to explicitly specify the multikey type.

Index Bounds

If an index is multikey, then computation of the index bounds follows special rules. For details on multikey index bounds, see *Multikey Index Bounds* (page 43).

Limitations

Compound Multikey Indexes

For a *compound* (page 10) multikey index, each indexed document can have *at most* one indexed field whose value is an array. As such, you cannot create a compound multikey index if more than one to-be-indexed field of a document is an array. Or, if a compound multikey index already exists, you cannot insert a document that would violate this restriction.

For example, consider a collection that contains the following document:

```
{ _id: 1, a: [ 1, 2 ], b: [ 1, 2 ], category: "AB - both arrays" }
```

You cannot create a compound multikey index `{ a: 1, b: 1 }` on the collection since both the `a` and `b` fields are arrays.

But consider a collection that contains the following documents:

```
{ _id: 1, a: [1, 2], b: 1, category: "A array" }
{ _id: 2, a: 1, b: [1, 2], category: "B array" }
```

A compound multikey index `{ a: 1, b: 1 }` is permissible since for each document, only one field indexed by the compound multikey index is an array; i.e. no document contains array values for both `a` and `b` fields. After creating the compound multikey index, if you attempt to insert a document where both `a` and `b` fields are arrays, MongoDB will fail the insert.

Shard Keys

You **cannot** specify a multikey index as the shard key index.

Changed in version 2.6: However, if the shard key index is a *prefix* (page 12) of a compound index, the compound index is allowed to become a compound *multikey* index if one of the other keys (i.e. keys that are not part of the shard key) indexes an array. Compound multikey indexes can have an impact on performance.

Hashed Indexes

Hashed (page 28) indexes **cannot** be multikey.

Covered Queries

A *multikey index* (page 13) cannot support a *covered query*.

Query on the Array Field as a Whole

When a query filter specifies an *exact match for an array as a whole*, MongoDB can use the multikey index to look up the first element of the query array but cannot use the multikey index scan to find the whole array. Instead, after using the multikey index to look up the first element of the query array, MongoDB retrieves the associated documents and filters for documents whose array matches the array in the query.

For example, consider an `inventory` collection that contains the following documents:

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

The collection has a multikey index on the `ratings` field:

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

The following query looks for documents where the `ratings` field is the array `[ 5, 9 ]`:

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

MongoDB can use the multikey index to find documents that have `5` at any position in the `ratings` array. Then, MongoDB retrieves these documents and filters for documents whose `ratings` array equals the query array `[ 5, 9 ]`.

Examples

Index Basic Arrays

Consider a `survey` collection with the following document:

```
{ _id: 1, item: "ABC", ratings: [ 2, 5, 9 ] }
```

Create an index on the field `ratings`:

```
db.survey.createIndex( { ratings: 1 } )
```

Since the `ratings` field contains an array, the index on `ratings` is multikey. The multikey index contains the following three index keys, each pointing to the same document:

- `2`,
- `5`, and
- `9`.

Index Arrays with Embedded Documents

You can create multikey indexes on array fields that contain nested objects.

Consider an `inventory` collection with documents of the following form:

```
{
  _id: 1,
  item: "abc",
  stock: [
    { size: "S", color: "red", quantity: 25 },
    { size: "S", color: "blue", quantity: 10 },
    { size: "M", color: "blue", quantity: 50 }
  ]
}
{
  _id: 2,
  item: "def",
  stock: [
```

```
    { size: "S", color: "blue", quantity: 20 },
    { size: "M", color: "blue", quantity: 5 },
    { size: "M", color: "black", quantity: 10 },
    { size: "L", color: "red", quantity: 2 }
  ]
}
{
  _id: 3,
  item: "ijk",
  stock: [
    { size: "M", color: "blue", quantity: 15 },
    { size: "L", color: "blue", quantity: 100 },
    { size: "L", color: "red", quantity: 25 }
  ]
}

...
```

The following operation creates a multikey index on the `stock.size` and `stock.quantity` fields:

```
db.inventory.createIndex( { "stock.size": 1, "stock.quantity": 1 } )
```

The compound multikey index can support queries with predicates that include both indexed fields as well as predicates that include only the index prefix `"stock.size"`, as in the following examples:

```
db.inventory.find( { "stock.size": "M" } )
db.inventory.find( { "stock.size": "S", "stock.quantity": { $gt: 20 } } )
```

For details on how MongoDB can combine multikey index bounds, see *Multikey Index Bounds* (page 43). For more information on behavior of compound indexes and prefixes, see *compound indexes and prefixes* (page 12).

The compound multikey index can also support sort operations, such as the following examples:

```
db.inventory.find( ).sort( { "stock.size": 1, "stock.quantity": 1 } )
db.inventory.find( { "stock.size": "M" } ).sort( { "stock.quantity": 1 } )
```

For more information on behavior of compound indexes and sort operations, see *Use Indexes to Sort Query Results* (page 93).

2.1.4 Geospatial Indexes and Queries

On this page

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 64).

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 19) 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 21) 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.

Changed in version 2.6: Support for additional GeoJSON types: MultiPoint, MultiLineString, MultiPolygon, GeometryCollection.

MongoDB supports the following GeoJSON objects:

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection

Legacy Coordinate Pairs

MongoDB supports spherical surface calculations on *legacy coordinate pairs* using a `2dsphere` index by converting the data to the GeoJSON Point type.

If you choose flat surface calculations via a `2d` index, 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.

Both `2d` and `2dsphere` indexes can support inclusion queries. MongoDB does not require an index for inclusion queries; however, such indexes will improve query performance.

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.

Only `2dsphere` indexes support intersection.

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 19) indexes support:

- Calculations on a sphere
- GeoJSON objects and include backwards compatibility for legacy coordinate pairs
- Compound indexes 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 73)

2d

2d (page 21) indexes support:

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

See also:

Query a 2d Index (page 76)

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 it uses fields other than the shard key fields.

For sharded collections, queries using `$near` and `$nearSphere` are not supported. You can instead use either the `geoNear` command or the `$geoNear` aggregation stage.

You can also query for geospatial data using `$geoWithin`.

Additional Resources

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

2dsphere Indexes **(page 19)** 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 21)** The `2d` index supports data stored as legacy coordinate pairs and is intended for use in MongoDB 2.2 and earlier.

geoHaystack Indexes **(page 22)** 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 23)** 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

On this page

New in version 2.4.

Overview A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. `2dsphere` index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity. See the `https://docs.mongodb.org/manual/reference/operator/query-geospatial` for the query operators that support geospatial queries.

The `2dsphere` index supports data stored as *GeoJSON* (page 100) objects and as legacy coordinate pairs (See also *2dsphere Indexed Field Restrictions* (page 20)). For legacy coordinate pairs, the index converts the data to GeoJSON *Point* (page 101). For details on the supported GeoJSON objects, see *GeoJSON Objects* (page 100).

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

`2dsphere (Version 2)` Changed in version 2.6.

MongoDB 2.6 introduces a version 2 of `2dsphere` indexes. Version 2 is the default version of `2dsphere` indexes created in MongoDB 2.6 and later series. To override the default version 2 and create a version 1 index, include the option `{ "2dsphereIndexVersion": 1 }` when creating the index.

`sparse` Property Changed in version 2.6.

`2dsphere (Version 2)` indexes are *sparse* (page 36) by default and ignores the *sparse: true* (page 36) option. If a document lacks a `2dsphere` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the index. For inserts, MongoDB inserts the document but does not add to the `2dsphere` index.

For a compound index that includes a `2dsphere` index key along with keys of other types, only the `2dsphere` index field determines whether the index references a document.

Earlier versions of MongoDB only support `2dsphere (Version 1)` indexes. `2dsphere (Version 1)` indexes are *not* sparse by default and will reject documents with `null` location fields.

Additional GeoJSON Objects `2dsphere (Version 2)` includes support for additional GeoJSON object: *MultiPoint* (page 102), *MultiLineString* (page 103), *MultiPolygon* (page 103), and *GeometryCollection* (page 104). For details on all supported GeoJSON objects, see *GeoJSON Objects* (page 100).

Considerations

`geoNear` and `$geoNear` Restrictions The `geoNear` command and the `$geoNear` pipeline stage require that a collection have *at most* only one `2dsphere` index and/or only one *2d* (page 21) index whereas *geospatial query operators* (e.g. `$near` and `$geoWithin`) permit collections to have multiple geospatial indexes.

The geospatial index restriction for the `geoNear` command and the `$geoNear` pipeline stage exists because neither the `geoNear` command nor the `$geoNear` pipeline stage syntax includes the location field. As such, index selection among multiple `2d` indexes or `2dsphere` indexes is ambiguous.

No such restriction applies for *geospatial query operators* since these operators take a location field, eliminating the ambiguity.

Shard Key Restrictions 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.

`2dsphere` Indexed Field Restrictions Fields with *2dsphere* (page 19) indexes must hold geometry data in the form of *coordinate pairs* or *GeoJSON* data. If you attempt to insert a document with non-geometry data in a `2dsphere` indexed field, or build a `2dsphere` index on a collection where the indexed field has non-geometry data, the operation will fail.

Create a `2dsphere` Index To create a `2dsphere` index, use the `db.collection.createIndex()` method, specifying the location field as the key and specify the string literal `"2dsphere"` as the index type:

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

Unlike a compound *2d* (page 21) index which can reference one location field and one other field, a *compound* (page 10) `2dsphere` index can reference multiple location and non-location fields.

For more information on creating `2dspshere` indexes, see *Create a 2dsphere Index* (page 70).

`2d` Indexes

On this page

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.

See the `https://docs.mongodb.org/manual/reference/operator/query-geospatial` for the query operators that support geospatial queries.

Considerations The `geoNear` command and the `$geoNear` pipeline stage require that a collection have *at most* only one `2d` index and/or only one *2dsphere index* (page 19) whereas *geospatial query operators* (e.g. `$near` and `$geoWithin`) permit collections to have multiple geospatial indexes.

The geospatial index restriction for the `geoNear` command and the `$geoNear` pipeline stage exists because neither the `geoNear` command nor the `$geoNear` pipeline stage syntax includes the location field. As such, index selection among multiple `2d` indexes or `2dsphere` indexes is ambiguous.

No such restriction applies for *geospatial query operators* since these operators take a location field, eliminating the ambiguity.

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 19) index.

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.

Behavior 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.

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.

Points on a 2D Plane To store location data as legacy coordinate pairs, use an array or an embedded document. When possible, use the array format:

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

Consider the embedded document form:

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

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

For all points, if you use longitude and latitude, store coordinates in **longitude, latitude** order.

`sparse` Property `2d` indexes are *sparse* (page 36) by default and ignores the *sparse: true* (page 36) option. If a document lacks a `2d` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `2d` index. For inserts, MongoDB inserts the document but does not add to the `2d` index.

For a compound index that includes a `2d` index key along with keys of other types, only the `2d` index field determines whether the index references a document.

`geoHaystack` Indexes

On this page

A `geoHaystack` index is a special index that is optimized to return results over small areas. `geoHaystack` 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 19) allow field reordering; `geoHaystack` indexes require the first field to be the location field. Also, `geoHaystack` indexes are only usable via commands and so always return all results at once.

Behavior `geoHaystack` 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 `geoHaystack` index contains all the documents within a specified proximity to a given longitude and latitude.

`sparse` Property `geoHaystack` indexes are *sparse* (page 36) by default and ignore the *sparse: true* (page 36) option. If a document lacks a `geoHaystack` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `geoHaystack` index. For inserts, MongoDB inserts the document but does not add to the `geoHaystack` index.

`geoHaystack` indexes include one `geoHaystack` index key and one non-geospatial index key; however, only the `geoHaystack` index field determines whether the index references a document.

Create `geoHaystack` Index To create a `geoHaystack` index, see *Create a Haystack Index* (page 78). For information and example on querying a haystack index, see *Query a Haystack Index* (page 79).

2d Index Internals

On this page

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 75) 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

Note: *2dsphere* (page 19) indexes can cover multiple geospatial fields in a document, and can express lists of points using *MultiPoint* (page 102) embedded documents.

While 2d geospatial indexes do not support more than one geospatial field 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:

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

You may also model the location data as a field inside of an embedded 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:

```
db.records.save( {
  name : "John Smith",
  addresses : [ {
                  context : "home" ,
                  loc : [ 55.5, 42.3 ]
                } ,
                {
                 context : "work",
                 loc : [ -74 , 44.74 ]
                }
              ]
} )
```

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

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

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

2.1.5 Text Indexes

On this page

Changed in version 3.2.

Starting in MongoDB 3.2, MongoDB introduces a version 3 of the `text` index. Key features of the new version of the index are:

- Improved *case insensitivity* (page 26)
- *Diacritic insensitivity* (page 26)
- Additional *delimiters for tokenization* (page 26)

Starting in MongoDB 3.2, version 3 is the default version for new `text` indexes.

Overview

MongoDB provides `text` indexes to support query operations that perform a text search of string content. `text` indexes can include any field whose value is a string or an array of string elements.

Create Text Index

Important: A collection can have at most **one** `text` index.

To create a `text` index, use the `db.collection.createIndex()` 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.createIndex( { comments: "text" } )
```

You can index multiple fields for the `text` index. The following example creates a `text` index on the fields `subject` and `comments`:

```
db.reviews.createIndex(
   {
     subject: "text",
     comments: "text"
   }
 )
```

A *compound index* (page 10) can include `text` index keys in combination with ascending/descending index keys. For more information, see *Compound Index* (page 27).

Specify Weights

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 text search score.

For each indexed field in the document, MongoDB multiplies the number of matches by the weight and sums the results. Using this sum, MongoDB then calculates the score for the document. See `$meta` operator for details on returning and sorting by text scores.

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.createIndex()` method.

For more information using weights to control the results of a text search, see *Control Search Results with Weights* (page 88).

Wildcard Text Indexes

When creating a `text` index on multiple fields, you can also use the wildcard specifier (`$**`). With a wildcard text index, MongoDB indexes every field that contains string data for each document in the collection. The following example creates a text index using the wildcard specifier:

```
db.collection.createIndex( { "$**": "text" } )
```

This index allows for text search on all fields with string content. Such an index can be useful with highly unstructured data if it is unclear which fields to include in the text index or for ad-hoc querying.

Wildcard text indexes are `text` indexes on multiple fields. As such, you can assign weights to specific fields during index creation to control the ranking of the results. For more information using weights to control the results of a text search, see *Control Search Results with Weights* (page 88).

Wildcard text indexes, as with all text indexes, can be part of a compound indexes. For example, the following creates a compound index on the field `a` as well as the wildcard specifier:

```
db.collection.createIndex( { a: 1, "$**": "text" } )
```

As with all *compound text indexes* (page 27), since the `a` precedes the text index key, in order to perform a `$text` search with this index, the query predicate must include an equality match conditions `a`. For information on compound text indexes, see *Compound Text Indexes* (page 27).

Case Insensitivity

Changed in version 3.2.

The version 3 `text` index supports the common `C`, simple `S`, and for Turkish languages, the special `T` case foldings as specified in Unicode 8.0 Character Database Case Folding[2].

The case foldings expands the case insensitivity of the `text` index to include characters with diacritics, such as `é` and `É`, and characters from non-Latin alphabets, such as characters from Cyrillic alphabet.

Version 3 of the `text` index is also *diacritic insensitive* (page 26). As such, the index also does not distinguish between `é`, `É`, `e`, and `E`.

Previous versions of the `text` index are case insensitive for `[A-z]` only; i.e. case insensitive for non-diacritics Latin characters only . For all other characters, earlier versions of the text index treat them as distinct.

Diacritic Insensitivity

Changed in version 3.2.

With version 3, `text` index is diacritic insensitive. That is, the index does not distinguish between characters that contain diacritical marks and their non-marked counterpart, such as `é`, `ê`, and `e`. More specifically, the `text` index strips the characters categorized as diacritics in Unicode 8.0 Character Database Prop List[3].

Version 3 of the `text` index is also *case insensitive* (page 26) to characters with diacritics. As such, the index also does not distinguish between `é`, `É`, `e`, and `E`.

Previous versions of the `text` index treat characters with diacritics as distinct.

Tokenization Delimiters

Changed in version 3.2.

For tokenization, version 3 `text` index uses the delimiters categorized under `Dash`, `Hyphen`, `Pattern_Syntax`, `Quotation_Mark`, `Terminal_Punctuation`, and `White_Space` in Unicode 8.0 Character Database Prop List[4].

For example, if given a string `"Il a dit qu'il «était le meilleur joueur du monde»"`, the `text` index treats «, », and spaces as delimiters.

Previous versions of the index treat « as part of the term `"«était"` and » as part of the term `"monde»"`.

Index Entries

`text` index tokenizes and stems the terms in the indexed fields for the index entries. `text` index stores one index entry for each unique stemmed term in each indexed field for each document in the collection. The index uses simple *language-specific* (page 27) suffix stemming.

[2] http://www.unicode.org/Public/8.0.0/ucd/CaseFolding.txt
[3] http://www.unicode.org/Public/8.0.0/ucd/PropList.txt
[4] http://www.unicode.org/Public/8.0.0/ucd/PropList.txt

Supported Languages and Stop Words

MongoDB supports text search for various languages. `text` indexes drop language-specific stop words (e.g. in English, `the`, `an`, `a`, `and`, etc.) and use simple language-specific suffix stemming. For a list of the supported languages, see *Text Search Languages* (page 104).

If you specify a language value of `"none"`, then the `text` index uses simple tokenization with no list of stop words and no stemming.

To specify a language for the `text` index, see *Specify a Language for Text Index* (page 82).

`sparse` Property

`text` indexes are *sparse* (page 36) by default and ignore the *sparse: true* (page 36) option. If a document lacks a `text` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `text` index. For inserts, MongoDB inserts the document but does not add to the `text` index.

For a compound index that includes a `text` index key along with keys of other types, only the `text` index field determines whether the index references a document. The other keys do not determine whether the index references the documents or not.

Restrictions

One Text Index Per Collection

A collection can have at most **one** `text` index.

Text Search and Hints

You cannot use `hint()` if the query includes a `$text` query expression.

Text Index and Sort

Sort operations cannot obtain sort order from a `text` index, even from a *compound text index* (page 27); i.e. sort operations cannot use the ordering in the text index.

Compound Index

A *compound index* (page 10) can include a `text` index key in combination with ascending/descending index keys. However, these compound indexes have the following restrictions:

- A compound `text` index cannot include any other special index types, such as *multi-key* (page 13) or *geospatial* (page 18) index fields.
- If the compound `text` index includes keys **preceding** the `text` index key, to perform a `$text` search, the query predicate must include **equality match conditions** on the preceding keys.

See also *Text Index and Sort* (page 27) for additional limitations.

For an example of a compound text index, see *Limit the Number of Entries Scanned* (page 89).

Drop a Text Index

To drop a `text` index, pass the *name* of the index to the `db.collection.dropIndex()` method. To get the name of the index, run the `db.collection.getIndexes()` method.

For information on the default naming scheme for `text` indexes as well as overriding the default name, see *Specify Name for text Index* (page 86).

Storage Requirements and Performance Costs

`text` indexes have the following storage requirements and performance costs:

- `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 Support

The `text` index supports `$text` query operations. For examples of text search, see the `$text reference page`. For examples of `$text` operations in aggregation pipelines, see *Text Search in the Aggregation Pipeline* (page 89).

2.1.6 Hashed Index

New in version 2.4.

Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses embedded 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 `https://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.createIndex( { 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 you can select when building an index.

***TTL Indexes* (page 29)** The TTL index is used for TTL collections, which expire data after a period of time.

***Unique Indexes* (page 31)** A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

***Partial Indexes* (page 32)** A partial index indexes only documents that meet specified filter criteria.

***Sparse Indexes* (page 36)** A sparse index does not index documents that do not have the indexed field.

2.2.1 TTL Indexes

On this page

TTL indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. Data expiration is useful for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

To create a TTL index, use the `db.collection.createIndex()` method with the `expireAfterSeconds` option on a field whose value is either a *date* or an array that contains *date values*.

For example, to create a TTL index on the `lastModifiedDate` field of the `eventlog` collection, use the following operation in the `mongo` shell:

```
db.eventlog.createIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
```

Behavior

Expiration of Data

TTL indexes expire documents after the specified number of seconds has passed since the indexed field value; i.e. the expiration threshold is the indexed field value plus the specified number of seconds.

If the field is an array, and there are multiple date values in the index, MongoDB uses *lowest* (i.e. earliest) date value in the array to calculate the expiration threshold.

If the indexed field in a document is not a *date* or an array that holds a date value(s), the document will not expire.

If a document does not contain the indexed field, the document will not expire.

Delete Operations

A background thread in `mongod` reads the values in the index and removes expired *documents* from the collection.

When the TTL thread is active, you will see `delete` operations in the output of `db.currentOp()` or in the data collected by the *database profiler*.

Timing of the Delete Operation When you build a TTL index in the *background* (page 39), the TTL thread can begin deleting documents while the index is building. If you build a TTL index in the foreground, MongoDB begins removing expired documents as soon as the index finishes building.

The TTL index does not guarantee that expired data will be deleted immediately upon expiration. 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 during the period between the expiration of the document and the running of the background task.

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

Replica Sets On *replica sets*, the TTL background thread *only* deletes documents on the *primary*. However, the TTL background thread does run on secondaries. *Secondary* members replicate deletion operations from the primary.

Support for Queries

A TTL index supports queries in the same way non-TTL indexes do.

Record Allocation

A collection with a TTL index has `usePowerOf2Sizes` enabled, and you cannot modify this setting for the collection. As a result of enabling `usePowerOf2Sizes`, MongoDB must allocate more disk space relative to data size. This approach helps mitigate the possibility of storage fragmentation caused by frequent delete operations and leads to more predictable storage use patterns.

Restrictions

- TTL indexes are a single-field indexes. *Compound indexes* (page 10) do not support TTL and ignores the `expireAfterSeconds` option.
- The `_id` field does not support TTL indexes.
- You cannot create a TTL index on a `capped collection` because MongoDB cannot remove documents from a capped collection.
- You cannot use `createIndex()` to change the value of `expireAfterSeconds` of an existing index. Instead use the `collMod` database command in conjunction with the `index` collection flag. Otherwise, to change the value of the option of an existing index, you must drop the index first and recreate.
- If a non-TTL single-field index already exists for a field, you cannot create a TTL index on the same field since you cannot create indexes that have the same key specification and differ only by the options. To change a non-TTL single-field index to a TTL index, you must drop the index first and recreate with the `expireAfterSeconds` option.

Additional Information

For examples, see `https://docs.mongodb.org/manual/tutorial/expire-data`.

2.2.2 Unique Indexes

On this page

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

To create a unique index, use the `db.collection.createIndex()` method with the `unique` option set to `true`. For example, to create a unique index on the `user_id` field of the `members` collection, use the following operation in the `mongo` shell:

```
db.members.createIndex( { "user_id": 1 }, { unique: true } )
```

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

If you use the unique constraint on a *compound index* (page 10), then MongoDB will enforce uniqueness on the *combination* of values rather than the individual value for any or all values of the key.

Behavior

Unique Constraint Across Separate Documents

The unique constraint applies to separate documents in the collection. That is, the unique index prevents *separate* documents from having the same value for the indexed key, but the index does not prevent a document from having multiple elements or embedded documents in an indexed array from having the same value. In the case of a single document with repeating values, the repeated value is inserted into the index only once.

For example, a collection has a unique index on `a.b`:

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

The unique index permits the insertion of the following document into the collection if no other document in the collection has the `a.b` value of `5`:

```
db.collection.insert( { a: [ { b: 5 }, { b: 5 } ] } )
```

Unique Index and Missing Field

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.

For example, a collection has a unique index on `x`:

```
db.collection.createIndex( { "x": 1 }, { unique: true } )
```

The unique index allows the insertion of a document without the field `x` if the collection does not already contain a document missing the field `x`:

```
db.collection.insert( { y: 1 } )
```

However, the unique index errors on the insertion of a document without the field `x` if the collection already contains a document missing the field `x`:

```
db.collection.insert( { z: 1 } )
```

The operation fails to insert the document because of the violation of the unique constraint on the value of the field `x`:

```
WriteResult({
   "nInserted" : 0,
   "writeError" : {
      "code" : 11000,
      "errmsg" : "E11000 duplicate key error index: test.collection.$a.b_1 dup key: { : null }"
   }
})
```

You can combine the unique constraint with the *sparse index* (page 36) to filter these null values from the unique index and avoid the error.

Unique Partial Indexes

New in version 3.2.

Partial indexes only index the documents in a collection that meet a specified filter expression. If you specify both the `partialFilterExpression` and a *unique constraint* (page 31), the unique constraint only applies to the documents that meet the filter expression. A partial index with a unique constraint does not prevent the insertion of documents that do not meet the unique constraint if the documents do not meet the filter criteria. For an example, see *Partial Index with Unique Constraint* (page 35).

Restrictions

You may not specify a unique constraint on a *hashed index* (page 28).

See also:

Create a Unique Index (page 52)

2.2.3 Partial Indexes

On this page

New in version 3.2.

Partial indexes only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance.

To create a `partial` index, use the `db.collection.createIndex()` method with the new `partialFilterExpression` option. For example, the following example creates a compound index that indexes only the documents with a `rating` field greater than 5.

```
db.restaurants.createIndex(
   { cuisine: 1, name: 1 },
   { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

You can specify a `partialFilterExpression` option for all MongoDB *index types* (page 8).

Behavior

Query Coverage

MongoDB will not use the partial index for a query or sort operation if using the index results in an incomplete result set. To use the partial index, a query must contain the filter expression (or a modified filter expression that specifies a subset of the filter expression) as part of its query condition.

For example, given the following index:

```
db.restaurants.createIndex(
   { cuisine: 1 },
   { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

The following query can use the index since the query predicate includes the condition `rating: { $gte: 8 }` that matches a subset of documents matched by the index filter expression `ratings: { $gt: 5 }`:

```
db.restaurants.find( { cuisine: "Italian", rating: { $gte: 8 } } )
```

However, the following query cannot use the partial index on the `cuisine` field because using the index results in an incomplete result set. Specifically, the query predicate includes the condition `rating: { $lt: 8 }` while the index has the filter `rating: { $gt: 5 }`. That is, the query `{ cuisine: "Italian", rating: { $gte: 8 } }` matches more documents (e.g. an Italian restaurant with a rating equal to 1) than are indexed.

```
db.restaurants.find( { cuisine: "Italian", rating: { $lt: 8 } } )
```

Similarly, the following query cannot use the partial index because the query predicate does not include the filter expression and using the index would return an incomplete result set.

```
db.restaurants.find( { cuisine: "Italian" } )
```

Comparison with the `sparse` Index

Tip

Partial indexes represent a superset of the functionality offered by sparse indexes and should be preferred over sparse indexes.

Partial indexes offer a more expressive mechanism than *Sparse Indexes* (page 36) indexes to specify which documents are indexed.

Sparse indexes selects documents to index *solely* based on the existence of the indexed field, or for compound indexes, the existence of the indexed fields.

Partial indexes determine the index entries based on the specified filter. The filter can include fields other than the index keys and can specify conditions other than just an existence check. For example, a partial index can implement the same behavior as a sparse index:

```
db.contacts.createIndex(
   { name: 1 },
   { partialFilterExpression: { name: { $exists: true } } }
)
```

This partial index supports the same queries as a sparse index on the `name` field.

However, a partial index can also specify filter expressions on fields other than the index key. For example, the following operation creates a partial index, where the index is on the `name` field but the filter expression is on the `email` field:

```
db.contacts.createIndex(
   { name: 1 },
   { partialFilterExpression: { email: { $exists: true } } }
)
```

For the query optimizer to choose this partial index, the query predicate must include a non-null match on the `email` field as well as a condition on the `name` field.

For example, the following query can use the index:

```
db.contacts.find( { name: "xyz", email: { $regex: /\.org$/ } } )
```

However, the following query cannot use the index:

```
db.contacts.find( { name: "xyz", email: { $exists: false } } )
```

Restrictions

In MongoDB, you cannot create multiple versions of an index that differ only in the options. As such, you cannot create multiple partial indexes that differ only by the filter expression.

You cannot specify both the `partialFilterExpression` option and the `sparse` option.

Earlier versions of MongoDB do not support partial indexes. For sharded clusters or replica sets, all nodes must be version 3.2.

`_id` indexes cannot be partial indexes.

Shard key indexes cannot be partial indexes.

Examples

Create a Partial Index On A Collection

Consider a collection `restaurants` containing documents that resemble the following

```
{
   "_id" : ObjectId("5641f6a7522545bc535b5dc9"),
   "address" : {
      "building" : "1007",
      "coord" : [
         -73.856077,
         40.848447
      ],
      "street" : "Morris Park Ave",
      "zipcode" : "10462"
   },
```

```
   "borough" : "Bronx",
   "cuisine" : "Bakery",
   "rating" : { "date" : ISODate("2014-03-03T00:00:00Z"),
                "grade" : "A",
                "score" : 2
              },
   "name" : "Morris Park Bake Shop",
   "restaurant_id" : "30075445"
}
```

You could add a partial index on the `borough` and `cuisine` fields choosing only to index documents where the `rating.grade` field is `A`:

```
db.restaurants.createIndex(
   { borough: 1, cuisine: 1 },
   { partialFilterExpression: { 'rating.grade': { $eq: "A" } } }
)
```

Then, the following query on the `restaurants` collection uses the partial index to return the restaurants in the Bronx with `rating.grade` equal to `A`:

```
db.restaurants.find( { borough: "Bronx", 'rating.grade': "A" } )
```

However, the following query cannot use the partial index because the query expression does not include the `rating.grade` field:

```
db.restaurants.find( { borough: "Bronx", cuisine: "Bakery" } )
```

Partial Index with Unique Constraint

Partial indexes only index the documents in a collection that meet a specified filter expression. If you specify both the `partialFilterExpression` and a *unique constraint* (page 31), the unique constraint only applies to the documents that meet the filter expression. A partial index with a unique constraint does not prevent the insertion of documents that do not meet the unique constraint if the documents do not meet the filter criteria.

For example, a collection `users` contains the following documents:

```
{ "_id" : ObjectId("56424f1efa0358a27fa1f99a"), "username" : "david", "age" : 29 }
{ "_id" : ObjectId("56424f37fa0358a27fa1f99b"), "username" : "amanda", "age" : 35 }
{ "_id" : ObjectId("56424fe2fa0358a27fa1f99c"), "username" : "rajiv", "age" : 57 }
```

The following operation creates an index that specifies a *unique constraint* (page 31) on the `username` field and a partial filter expression `age: { $gte: 21 }`.

```
db.users.createIndex(
   { username: 1 },
   { unique: true, partialFilterExpression: { age: { $gte: 21 } } }
)
```

The index prevents the insertion of the following documents since documents already exist with the specified usernames and the `age` fields are greater than `21`:

```
db.users.insert( { username: "david", age: 27 } )
db.users.insert( { username: "amanda", age: 25 } )
db.users.insert( { username: "rajiv", age: 32 } )
```

However, the following documents with duplicate usernames are allowed since the unique constraint only applies to documents with `age` greater than or equal to 21.

```
db.users.insert( { username: "david", age: 20 } )
db.users.insert( { username: "amanda" } )
db.users.insert( { username: "rajiv", age: null } )
```

2.2.4 Sparse Indexes

On this page

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.

Important: Changed in version 3.2: Starting in MongoDB 3.2, MongoDB provides the option to create *partial indexes* (page 32). Partial indexes offer a superset of the functionality of sparse indexes. If you are using MongoDB 3.2 or later, *partial indexes* (page 32) should be preferred over sparse indexes.

To create a `sparse` index, use the `db.collection.createIndex()` method with the `sparse` option set to `true`. For example, the following operation in the `mongo` shell creates a sparse index on the `xmpp_id` field of the `addresses` collection:

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

Note: Do not confuse sparse indexes in MongoDB with block-level[5] indexes in other databases. Think of them as dense indexes with a specific filter.

Behavior

`sparse` Index and Incomplete Results

Changed in version 2.6.

If a sparse index would result in an incomplete result set for queries and sort operations, MongoDB will not use that index unless a `hint()` explicitly specifies the index.

For example, the query `{ x: { $exists: false } }` will not use a sparse index on the `x` field unless explicitly hinted. See *Sparse Index On A Collection Cannot Return Complete Results* (page 37) for an example that details the behavior.

Indexes that are `sparse` by Default

2dsphere (version 2) (page 20), *2d* (page 21), *geoHaystack* (page 22), and *text* (page 24) indexes are always `sparse`.

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

`sparse` Compound Indexes

Sparse *compound indexes* (page 10) that only contain ascending/descending index keys will index a document as long as the document contains at least one of the keys.

For sparse compound indexes that contain a geospatial key (i.e. *2dsphere* (page 19), *2d* (page 21), or *geoHaystack* (page 22) index keys) along with ascending/descending index key(s), only the existence of the geospatial field(s) in a document determine whether the index references the document.

For sparse compound indexes that contain *text* (page 24) index keys along with ascending/descending index keys, only the existence of the `text` index field(s) determine whether the index references a document.

`sparse` and `unique` Properties

An index that is both `sparse` and *unique* (page 31) prevents collection from having documents with duplicate values for a field but allows multiple documents that omit the key.

Examples

Create a Sparse Index On A Collection

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`:

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

Then, the following query on the `scores` collection uses the sparse index to return the documents that have the `score` field less than (`$lt`) `90`:

```
db.scores.find( { score: { $lt: 90 } } )
```

Because the document for the userid `"newbie"` does not contain the `score` field and thus does not meet the query criteria, the query can use the sparse index to return the results:

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

Sparse Index On A Collection Cannot Return Complete 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`:

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

Because the document for the userid "newbie" does not contain the score field, the sparse index does not contain an entry for that document.

Consider the following query to return **all** documents in the scores collection, sorted by the score field:

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

Even though the sort is by the indexed field, MongoDB will **not** select the sparse index to fulfill the query in order to return complete results:

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

To use the sparse index, explicitly specify the index with hint():

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

The use of the index results in the return of only those documents with the score field:

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

See also:

explain() and https://docs.mongodb.org/manual/tutorial/analyze-query-plan

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 31) and sparse filter on the score field using the following operation:

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

This index *would permit* the insertion of documents that had unique values for the score field *or* did not include a score field. As such, given the existing documents in the scores collection, the index permits the following insert operations:

```
db.scores.insert( { "userid": "AAAAAAA", "score": 43 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 34 } )
db.scores.insert( { "userid": "CCCCCCC" } )
db.scores.insert( { "userid": "DDDDDDD" } )
```

However, the index *would not permit* the addition of the following documents since documents already exists with score value of 82 and 90:

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

2.3 Index Creation

On this page

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.createIndex()` method. This section describes the uses of these creation options and their behavior.

Related

Some options that you can specify to `createIndex()` options control the *properties of the index* (page 29), which are *not* index creation options. For example, the *unique* (page 31) 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 29) for related documentation.

2.3.1 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.createIndex( { 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.createIndex( { 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 `createIndex()` 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[6] and terminates if the proper indexes do not exist. Always build indexes in production instances using separate application code, during designated maintenance windows.

Interrupted Index Builds

If a background index build is in progress when the `mongod` process terminates, when the instance restarts the index build will restart as foreground index build. If the index build encounters any errors, such as a duplicate key error, the `mongod` will exit with an error.

To start the `mongod` after a failed index build, use the `storage.indexBuildRetry` or `--noIndexBuildRetry` to skip the index build on start up. .. _index-creation-building-indexes-on-secondaries:

Building Indexes on Secondaries

Changed in version 2.6: Secondary members can now build indexes in the background. Previously all index builds on secondaries were in the foreground.

Background index operations on a *replica set secondaries* begin after the *primary* completes building the index. If MongoDB builds an index in the background on the primary, the secondaries will then build that index in the background.

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.

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 55) for a complete procedure for building indexes on secondaries.

2.3.2 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.createIndex( { item: 1, quantity: -1 } )
```

[6] https://api.mongodb.org/

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.createIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

The resulting index has the name `inventory`.

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

2.4 Index Intersection

On this page

New in version 2.6.

MongoDB can use the intersection of multiple indexes to fulfill queries. [7] In general, each index intersection involves two indexes; however, MongoDB can employ multiple/nested index intersections to resolve a query.

To illustrate index intersection, consider a collection `orders` that has the following indexes:

```
{ qty: 1 }
{ item: 1 }
```

MongoDB can use the intersection of the two indexes to support the following query:

```
db.orders.find( { item: "abc123", qty: { $gt: 15 } } )
```

To determine if MongoDB used index intersection, run `explain()`; the results of *explain()* will include either an `AND_SORTED` stage or an `AND_HASH` stage.

2.4.1 Index Prefix Intersection

With index intersection, MongoDB can use an intersection of either the entire index or the index prefix. An index prefix is a subset of a compound index, consisting of one or more keys starting from the beginning of the index.

Consider a collection `orders` with the following indexes:

```
{ qty: 1 }
{ status: 1, ord_date: -1 }
```

To fulfill the following query which specifies a condition on both the `qty` field and the `status` field, MongoDB can use the intersection of the two indexes:

[7] In previous versions, MongoDB could use only a single index to fulfill most queries. The exception to this is queries with `$or` clauses, which could use a single index for each `$or` clause.

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } )
```

2.4.2 Index Intersection and Compound Indexes

Index intersection does not eliminate the need for creating *compound indexes* (page 10). However, because both the list order (i.e. the order in which the keys are listed in the index) and the sort order (i.e. ascending or descending), matter in *compound indexes* (page 10), a compound index may not support a query condition that does not include the *index prefix keys* (page 12) or that specifies a different sort order.

For example, if a collection `orders` has the following compound index, with the `status` field listed before the `ord_date` field:

```
{ status: 1, ord_date: -1 }
```

The compound index can support the following queries:

```
db.orders.find( { status: { $in: ["A", "P" ] } } )
db.orders.find(
   {
     ord_date: { $gt: new Date("2014-02-01") },
     status: {$in:[ "P", "A" ] }
   }
)
```

But not the following two queries:

```
db.orders.find( { ord_date: { $gt: new Date("2014-02-01") } } )
db.orders.find( { } ).sort( { ord_date: 1 } )
```

However, if the collection has two separate indexes:

```
{ status: 1 }
{ ord_date: -1 }
```

The two indexes can, either individually or through index intersection, support all four aforementioned queries.

The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system.

See also:

compound indexes (page 10), *Create Compound Indexes to Support Several Different Queries* (page 92)

2.4.3 Index Intersection and Sort

Index intersection does not apply when the `sort()` operation requires an index completely separate from the query predicate.

For example, the `orders` collection has the following indexes:

```
{ qty: 1 }
{ status: 1, ord_date: -1 }
{ status: 1 }
{ ord_date: -1 }
```

MongoDB cannot use index intersection for the following query with sort:

```
db.orders.find( { qty: { $gt: 10 } } ).sort( { status: 1 } )
```

That is, MongoDB does not use the `{ qty: 1 }` index for the query, and the separate `{ status: 1 }` or the `{ status: 1, ord_date: -1 }` index for the sort.

However, MongoDB can use index intersection for the following query with sort since the index `{ status: 1, ord_date: -1 }` can fulfill part of the query predicate.

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } ).sort( { ord_date: -1 } )
```

2.5 Multikey Index Bounds

On this page

The bounds of an index scan define the portions of an index to search during a query. When multiple predicates over an index exist, MongoDB will attempt to combine the bounds for these predicates by either *intersection* or *compounding* in order to produce a scan with smaller bounds.

2.5.1 Intersect Bounds for Multikey Index

Bounds intersection refers to a logical conjunction (i.e. `AND`) of multiple bounds. For instance, given two bounds `[ [ 3, Infinity ] ]` and `[ [ -Infinity, 6 ] ]`, the intersection of the bounds results in `[ [ 3, 6 ] ]`.

Given an *indexed* (page 13) array field, consider a query that specifies multiple predicates on the array and can use a *multikey index* (page 13). MongoDB can intersect *multikey index* (page 13) bounds if an `$elemMatch` joins the predicates.

For example, a collection `survey` contains documents with a field `item` and an array field `ratings`:

```
{ _id: 1, item: "ABC", ratings: [ 2, 9 ] }
{ _id: 2, item: "XYZ", ratings: [ 4, 3 ] }
```

Create a *multikey index* (page 13) on the `ratings` array:

```
db.survey.createIndex( { ratings: 1 } )
```

The following query uses `$elemMatch` to require that the array contains at least one *single* element that matches both conditions:

```
db.survey.find( { ratings : { $elemMatch: { $gte: 3, $lte: 6 } } } )
```

Taking the predicates separately:

- the bounds for the greater than or equal to 3 predicate (i.e. `$gte: 3`) are `[ [ 3, Infinity ] ]`;
- the bounds for the less than or equal to 6 predicate (i.e. `$lte: 6`) are `[ [ -Infinity, 6 ] ]`.

Because the query uses `$elemMatch` to join these predicates, MongoDB can intersect the bounds to:

```
ratings: [ [ 3, 6 ] ]
```

If the query does *not* join the conditions on the array field with `$elemMatch`, MongoDB cannot intersect the multikey index bounds. Consider the following query:

```
db.survey.find( { ratings : { $gte: 3, $lte: 6 } } )
```

The query searches the `ratings` array for at least one element greater than or equal to 3 and at least one element less than or equal to 6. Because a single element does not need to meet both criteria, MongoDB does *not* intersect the bounds and uses either `[ [ 3, Infinity ] ]` or `[ [ -Infinity, 6 ] ]`. MongoDB makes no guarantee as to which of these two bounds it chooses.

2.5.2 Compound Bounds for Multikey Index

Compounding bounds refers to using bounds for multiple keys of *compound index* (page 10). For instance, given a compound index `{ a: 1, b: 1 }` with bounds on field `a` of `[ [ 3, Infinity ] ]` and bounds on field `b` of `[ [ -Infinity, 6 ] ]`, compounding the bounds results in the use of both bounds:

```
{ a: [ [ 3, Infinity ] ], b: [ [ -Infinity, 6 ] ] }
```

If MongoDB cannot compound the two bounds, MongoDB always constrains the index scan by the bound on its leading field, in this case, `a: [ [ 3, Infinity ] ]`.

Compound Index on an Array Field

Consider a compound multikey index; i.e. a *compound index* (page 10) where one of the indexed fields is an array. For example, a collection `survey` contains documents with a field `item` and an array field `ratings`:

```
{ _id: 1, item: "ABC", ratings: [ 2, 9 ] }
{ _id: 2, item: "XYZ", ratings: [ 4, 3 ] }
```

Create a *compound index* (page 10) on the `item` field and the `ratings` field:

```
db.survey.createIndex( { item: 1, ratings: 1 } )
```

The following query specifies a condition on both keys of the index:

```
db.survey.find( { item: "XYZ", ratings: { $gte: 3 } } )
```

Taking the predicates separately:

- the bounds for the `item: "XYZ"` predicate are `[ [ "XYZ", "XYZ" ] ]`;
- the bounds for the `ratings: { $gte: 3 }` predicate are `[ [ 3, Infinity ] ]`.

MongoDB can compound the two bounds to use the combined bounds of:

```
{ item: [ [ "XYZ", "XYZ" ] ], ratings: [ [ 3, Infinity ] ] }
```

Compound Index on Fields from an Array of Embedded Documents

If an array contains embedded documents, to index on fields contained in the embedded documents, use the *dotted field name* in the index specification. For instance, given the following array of embedded documents:

```
ratings: [ { score: 2, by: "mn" }, { score: 9, by: "anon" } ]
```

The dotted field name for the `score` field is `"ratings.score"`.

Compound Bounds of Non-array Field and Field from an Array

Consider a collection `survey2` contains documents with a field `item` and an array field `ratings`:

```
{
  _id: 1,
  item: "ABC",
  ratings: [ { score: 2, by: "mn" }, { score: 9, by: "anon" } ]
}
{
  _id: 2,
  item: "XYZ",
  ratings: [ { score: 5, by: "anon" }, { score: 7, by: "wv" } ]
}
```

Create a *compound index* (page 10) on the non-array field `item` as well as two fields from an array `ratings.score` and `ratings.by`:

```
db.survey2.createIndex( { "item": 1, "ratings.score": 1, "ratings.by": 1 } )
```

The following query specifies a condition on all three fields:

```
db.survey2.find( { item: "XYZ",  "ratings.score": { $lte: 5 }, "ratings.by": "anon" } )
```

Taking the predicates separately:

- the bounds for the `item: "XYZ"` predicate are `[ [ "XYZ", "XYZ" ] ]`;
- the bounds for the `score: { $lte: 5 }` predicate are `[ [ -Infinity, 5 ] ]`;
- the bounds for the `by: "anon"` predicate are `[ "anon", "anon" ]`.

MongoDB can compound the bounds for the `item` key with *either* the bounds for `"ratings.score"` or the bounds for `"ratings.by"`, depending upon the query predicates and the index key values. MongoDB makes no guarantee as to which bounds it compounds with the `item` field. For instance, MongoDB will either choose to compound the `item` bounds with the `"ratings.score"` bounds:

```
{
  "item" : [ [ "XYZ", "XYZ" ] ],
  "ratings.score" : [ [ -Infinity, 5 ] ],
  "ratings.by" : [ [ MinKey, MaxKey ] ]
}
```

Or, MongoDB may choose to compound the `item` bounds with `"ratings.by"` bounds:

```
{
  "item" : [ [ "XYZ", "XYZ" ] ],
  "ratings.score" : [ [ MinKey, MaxKey ] ],
  "ratings.by" : [ [ "anon", "anon" ] ]
}
```

However, to compound the bounds for `"ratings.score"` with the bounds for `"ratings.by"`, the query must use `$elemMatch`. See *Compound Bounds of Index Fields from an Array* (page 45) for more information.

Compound Bounds of Index Fields from an Array

To compound together the bounds for index keys from the same array:

- the index keys must share the same field path up to but excluding the field names, and

- the query must specify predicates on the fields using `$elemMatch` on that path.

For a field in an embedded document, the *dotted field name*, such as `"a.b.c.d"`, is the field path for `d`. To compound the bounds for index keys from the same array, the `$elemMatch` must be on the path up to *but excluding* the field name itself; i.e. `"a.b.c"`.

For instance, create a *compound index* (page 10) on the `ratings.score` and the `ratings.by` fields:

```
db.survey2.createIndex( { "ratings.score": 1, "ratings.by": 1 } )
```

The fields `"ratings.score"` and `"ratings.by"` share the field path `ratings`. The following query uses `$elemMatch` on the field `ratings` to require that the array contains at least one *single* element that matches both conditions:

```
db.survey2.find( { ratings: { $elemMatch: { score: { $lte: 5 }, by: "anon" } } } )
```

Taking the predicates separately:

- the bounds for the `score: { $lte: 5 }` predicate is `[ -Infinity, 5 ]`;
- the bounds for the `by: "anon"` predicate is `[ "anon", "anon" ]`.

MongoDB can compound the two bounds to use the combined bounds of:

```
{ "ratings.score" : [ [ -Infinity, 5 ] ], "ratings.by" : [ [ "anon", "anon" ] ] }
```

Query Without `$elemMatch`

If the query does *not* join the conditions on the indexed array fields with `$elemMatch`, MongoDB *cannot* compound their bounds. Consider the following query:

```
db.survey2.find( { "ratings.score": { $lte: 5 }, "ratings.by": "anon" } )
```

Because a single embedded document in the array does not need to meet both criteria, MongoDB does *not* compound the bounds. When using a compound index, if MongoDB cannot constrain all the fields of the index, MongoDB always constrains the leading field of the index, in this case `"ratings.score"`:

```
{
  "ratings.score": [ [ -Infinity, 5 ] ],
  "ratings.by": [ [ MinKey, MaxKey ] ]
}
```

`$elemMatch` on Incomplete Path

If the query does not specify `$elemMatch` on the path of the embedded fields, up to but excluding the field names, MongoDB **cannot** compound the bounds of index keys from the same array.

For example, a collection `survey3` contains documents with a field `item` and an array field `ratings`:

```
{
  _id: 1,
  item: "ABC",
  ratings: [ { score: { q1: 2, q2: 5 } }, { score: { q1: 8, q2: 4 } } ]
}
{
  _id: 2,
  item: "XYZ",
  ratings: [ { score: { q1: 7, q2: 8 } }, { score: { q1: 9, q2: 5 } } ]
}
```

Create a *compound index* (page 10) on the `ratings.score.q1` and the `ratings.score.q2` fields:

```
db.survey3.createIndex( { "ratings.score.q1": 1, "ratings.score.q2": 1 } )
```

The fields `"ratings.score.q1"` and `"ratings.score.q2"` share the field path `"ratings.score"` and the `$elemMatch` must be on that path.

The following query, however, uses an `$elemMatch` but not on the required path:

```
db.survey3.find( { ratings: { $elemMatch: { 'score.q1': 2, 'score.q2': 8 } } } )
```

As such, MongoDB **cannot** compound the bounds, and the `"ratings.score.q2"` field will be unconstrained during the index scan. To compound the bounds, the query must use `$elemMatch` on the path `"ratings.score"`:

```
db.survey3.find( { 'ratings.score': { $elemMatch: { 'q1': 2, 'q2': 8 } } } )
```

Compound `$elemMatch` Clauses

Consider a query that contains multiple `$elemMatch` clauses on different field paths, for instance, `"a.b": { $elemMatch: ... }, "a.c": { $elemMatch: ... }`. MongoDB cannot combine the bounds of the `"a.b"` with the bounds of `"a.c"` since `"a.b"` and `"a.c"` also require `$elemMatch` on the path `a`.

For example, a collection `survey4` contains documents with a field `item` and an array field `ratings`:

```
{
  _id: 1,
  item: "ABC",
  ratings: [
    { score: { q1: 2, q2: 5 }, certainty: { q1: 2, q2: 3 } },
    { score: { q1: 8, q2: 4 }, certainty: { q1: 10, q2: 10 } }
  ]
}
{
  _id: 2,
  item: "XYZ",
  ratings: [
    { score: { q1: 7, q2: 8 }, certainty: { q1: 5, q2: 5 } },
    { score: { q1: 9, q2: 5 }, certainty: { q1: 7, q2: 7 } }
  ]
}
```

Create a *compound index* (page 10) on the `ratings.score.q1` and the `ratings.score.q2` fields:

```
db.survey4.createIndex( {
   "ratings.score.q1": 1,
   "ratings.score.q2": 1,
   "ratings.certainty.q1": 1,
   "ratings.certainty.q2": 1
} )
```

Consider the following query with two `$elemMatch` clauses:

```
db.survey4.find(
  {
    "ratings.score": { $elemMatch: { q1: 5, q2: 5 } },
    "ratings.certainty": { $elemMatch: { q1: 7, q2: 7 } },
  }
)
```

Taking the predicates separately:

- the bounds for the "ratings.score" predicate are the compound bounds:

```
{ "ratings.score.q1" : [ [ 5, 5 ] ], "ratings.score.q2" : [ [ 5, 5 ] ] }
```

- the bounds for the "ratings.certainty" predicate are the compound bounds:

```
{ "ratings.certainty.q1" : [ [ 7, 7 ] ], "ratings.certainty.q2" : [ [ 7, 7 ] ] }
```

However, MongoDB cannot compound the bounds for "ratings.score" and "ratings.certainty" since $elemMatch does not join the two. Instead, MongoDB constrains the leading field of the index "ratings.score.q1" which can be compounded with the bounds for "ratings.score.q2":

```
{
  "ratings.score.q1" : [ [ 5, 5 ] ],
  "ratings.score.q2" : [ [ 5, 5 ] ],
  "ratings.certainty.q1" : [ [ MinKey, MaxKey ] ],
  "ratings.certainty.q2" : [ [ MinKey, MaxKey ] ]
}
```

2.6 Additional Resources

- Quick Reference Cards[8]

[8] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs

CHAPTER 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 49)** Create and configure different types of indexes for different purposes.

Index Management Tutorials **(page 58)** Monitor and assess index performance and rebuild indexes as needed.

Geospatial Index Tutorials **(page 64)** Create indexes that support data stored as *GeoJSON* objects and legacy coordinate pairs.

Text Search Tutorials **(page 81)** Build and configure indexes that support full-text searches.

Indexing Strategies **(page 91)** 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 50)** Build an index for any field on a collection.

Create a Compound Index **(page 51)** Build an index of multiple fields on a collection.

Create a Unique Index **(page 52)** Build an index that enforces unique values for the indexed field or fields.

Create a Partial Index **(page 52)** Build an index that only indexes documents that meet specified filter criteria. This can reduce index size and improve performance.

Create a Sparse Index **(page 53)** 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 54)** 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 55)** 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 57)** 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 58)** A `{v : 0}` index is necessary if you need to roll back from MongoDB version 2.0 (or later) to MongoDB version 1.8.

3.1.1 Create an Index

On this page

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

This tutorial describes how to create an index on a single field. MongoDB also supports *compound indexes* (page 10), which are indexes on multiple fields. See *Create a Compound Index* (page 51) for instructions on building compound indexes.

Create an Index on a Single Field

To create an index, use `createIndex()` or a similar method from your driver[1]. The `createIndex()` method only creates an index if an index of the same specification does not already exist.

For example, the following operation creates an index on the `userid` field of the `records` collection:

```
db.records.createIndex( { userid: 1 } )
```

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. For additional index types, see *Index Types* (page 8).

The created index will support queries that select on the field `userid`, such as the following:

```
db.records.find( { userid: 2 } )
db.records.find( { userid: { $gt: 10 } } )
```

But the created index does not support the following query on the `profile_url` field:

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

For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Additional Considerations

Although indexes can improve query performances, indexes also present some operational considerations. See *Operational Considerations for Indexes* for more information.

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 39). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 55) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 55).

[1] https://api.mongodb.org/

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 51), *Indexing Tutorials* (page 49) and *Index Concepts* (page 7) for more information.

3.1.2 Create a Compound Index

On this page

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 10) 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 10) use an operation that resembles the following prototype:

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

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. For additional index types, see *Index Types* (page 8).

Example

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

```
db.products.createIndex( { 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 39). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 55) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 55).

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 an Index (page 50), *Indexing Tutorials* (page 49) and *Index Concepts* (page 7) for more information.

3.1.3 Create a Unique Index

On this page

MongoDB allows you to specify a *unique constraint* (page 31) on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields.

MongoDB cannot create a *unique index* (page 31) on the specified index field(s) if the collection already contains data that would violate the unique constraint for the index.

Unique Index on a Single Field

To create a *unique index* (page 31), consider the following prototype:

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

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

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

The *_id index* (page 9) is a unique index. In some situations you may consider using the `_id` field itself for this kind of data rather than using a unique index on another field.

Unique Compound Index

You can also enforce a unique constraint on *compound indexes* (page 10), as in the following prototype:

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

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

Unique Index and Missing Field

If a document does not have a value for a field, the index entry for that item will be `null` in any index that includes it. Thus, in many situations you will want to combine the `unique` constraint with the `sparse` option. *Sparse indexes* (page 36) skip over any document that is missing the indexed field, rather than storing `null` for the index entry. 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.createIndex( { a: 1 }, { unique: true, sparse: true } )
```

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

3.1.4 Create a Partial Index

On this page

New in version 3.2.

Partial indexes only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance. See *Partial Indexes* (page 32) for more information about partial indexes and their use.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 49) for more information.

Prototype

To create a *partial index* (page 32) on a field, use the `partialFilterExpression` option when creating the index, as in the following:

```
db.collection.createIndex(
   { a: 1 },
   { partialFilterExpression: { b: { $gt: 5 } } }
)
```

Example

The following operation creates a sparse index on the `users` collection that *only* includes a document in the index if the `archived` field is `false`.

```
db.users.createIndex( { username: 1 }, { archived: false } )
```

The index only includes documents where the `archived` field is `false`.

Considerations

Note: To use the partial index, a query **must** contain the filter expression (or a modified filter expression that specifies a subset of the filter expression) as part of its query condition. As such, MongoDB will not use the partial index if the index results in an incomplete result set for the query or sort operation.

3.1.5 Create a Sparse Index

On this page

Important: Changed in version 3.2: *Partial indexes* (page 32) offer a superset of the functionality of sparse indexes. If you are using MongoDB 3.2 or later, you should use *partial indexes* (page 32) rather than sparse.

Sparse indexes 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 36) for more information about sparse indexes and their use.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 49) for more information.

Prototype

To create a *sparse index* (page 36) on a field, use an operation that resembles the following prototype:

```
db.collection.createIndex( { 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.createIndex( { 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 36) section for more information.

3.1.6 Create a Hashed Index

On this page

New in version 2.4.

Hashed indexes (page 28) 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 49) for more information about indexes.

Procedure

To create a *hashed index* (page 28), specify `hashed` as the value of the index key, as in the following example:

Example

Specify a hashed index on `_id`

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

Considerations

MongoDB supports `hashed` indexes of any single field. The hashing function collapses embedded 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.

3.1.7 Build Indexes on Replica Sets

On this page

For replica sets, secondaries will begin building indexes *after* the *primary* finishes building the index. In *sharded clusters*, the `mongos` will send `createIndex()` 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:

See

Indexing Tutorials (page 49) 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.
- Before version 2.6 *Background index creation operations* (page 39) become *foreground* indexing operations on *secondary* members of replica sets. After 2.6, background index builds replicate as background index builds on the secondaries.

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. [2] 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 `createIndex()` 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.createIndex( { username: 1 } )
```

See also:

Create an Index (page 50) and *Create a Compound Index* (page 51) for more information.

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

Changed in version 2.6: Secondary members can now *build indexes in the background* (page 57). Previously all index builds on secondaries were in the foreground.

For each secondary in the set, build an index according to the following steps:

1. *Stop One Secondary* (page 56)
2. *Build the Index* (page 56)
3. *Restart the Program mongod* (page 56)

[2] 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.

Build the Index on the Primary

To build an index on the primary you can either:

1. *Build the index in the background* (page 57) on the primary.
2. Step down the primary using the `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 56)

   (b) *Build the Index* (page 56)

   (c) *Restart the Program mongod* (page 56)

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 while MongoDB builds the index.

3.1.8 Build Indexes in the Background

On this page

By default, MongoDB builds indexes in the foreground, which prevents 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 39) allows read and write operations to continue while building the index.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 49) 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. Over time, 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 `createIndex()` operation, as in the following index:

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

Consider the section on *background index construction* (page 39) for more information about these indexes and their implications.

3.1.9 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 `createIndex()` 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.createIndex( { 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 59)** Drop an index from a collection.

Modify an Index **(page 59)** Modify an existing index.

Rebuild Indexes **(page 60)** In a single operation, drop all indexes on a collection and then rebuild them.

Manage In-Progress Index Creation **(page 61)** Check the status of indexing progress, or terminate an ongoing index build.

Return a List of All Indexes **(page 62)** Obtain a list of all indexes on a collection or of all indexes on all collections in a database.

Measure Index Use **(page 63)** Study query operations and observe index use for your database.

3.2.1 Remove Indexes

On this page

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 60) document.

See also:

Indexing Tutorials (page 49) and *Index Concepts* (page 7) for more information about indexes and indexing operations in MongoDB.

Remove a Specific Index

To remove an index, use the `db.collection.dropIndex()` method.

For example, the following operation removes an ascending index on the `tax-id` field in the `accounts` collection:

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

The operation returns a document with the status of the operation:

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

Where the value of `nIndexesWas` reflects the number of indexes *before* removing this index.

For *text* (page 24) indexes, pass the index name to the `db.collection.dropIndex()` method. See *Use the Index Name to Drop a text Index* (page 87) for details.

Remove All Indexes

You can also use the `db.collection.dropIndexes()` to remove *all* indexes, except for the *_id index* (page 9) 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.

3.2.2 Modify an Index

To modify an existing index, you need to drop and recreate the index.

Step 1: Create a unique index.

Use the `createIndex()` method create a unique index.

```
db.orders.createIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 },
   { unique: true }
)
```

The method returns a document with the status of the results. The method only creates an index if the index does not already exist. See *Create an Index* (page 50) and *Index Creation Tutorials* (page 49) for more information on creating indexes.

Step 2: Attempt to modify the index.

To modify an existing index, you **cannot** just re-issue the `createIndex()` method with the updated specification of the index.

For example, the following operation attempts to remove the `unique` constraint from the previously created index by using the `createIndex()` method.

```
db.orders.createIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 }
)
```

The status document returned by the operation shows an error.

Step 3: Drop the index.

To modify the index, you must drop the index first.

```
db.orders.dropIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 }
)
```

The method returns a document with the status of the operation. Upon successful operation, the `ok` field in the returned document should specify a `1`. See *Remove Indexes* (page 59) for more information about dropping indexes.

Step 4: Recreate the index without the `unique` constraint.

Recreate the index without the `unique` constraint.

```
db.orders.createIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 }
)
```

The method returns a document with the status of the results. Upon successful operation, the returned document should show the `numIndexesAfter` to be greater than `numIndexesBefore` by one.

See also:

Index Introduction (page 1), *Index Concepts* (page 7).

3.2.3 Rebuild Indexes

On this page

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 9), and then rebuilds all indexes.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 49).

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 55).

3.2.4 Manage In-Progress Index Creation

On this page

View Index Creation Operations

To see the status of an indexing process, you can use the `db.currentOp()` method in the `mongo` shell. To filter the current operations for index creation operations, see *currentOp-index-creation* for an example.

The `msg` field will include the percent of the build that is complete.

Terminate Index Creation

To terminate an ongoing index build, use the `db.killOp()` method in the `mongo` shell. For index builds, the effects of `db.killOp()` may not be immediate and may occur well after much of the index build operation has completed.

You cannot terminate a *replicated* index build on secondary members of a replica set. To minimize the impact of building an index on replica sets, see *Build Indexes on Replica Sets* (page 55).

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

See also:

`db.currentOp()`, `db.killOp()`

3.2.5 Return a List of All Indexes

On this page

When performing maintenance you may want to check which indexes exist on a collection. In the `mongo` shell, you can use the `getIndexes()` method to return a list of the indexes on a collection.

See also:

Index Concepts (page 7) and *Indexing Tutorials* (page 49) 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[3].

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

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

List all Indexes for a Database

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

```
db.getCollectionNames().forEach(function(collection) {
   indexes = db[collection].getIndexes();
   print("Indexes for " + collection + ":");
   printjson(indexes);
});
```

MongoDB 3.0 deprecates direct access to the `system.indexes` collection.

For MongoDB 3.0 deployments using the *WiredTiger* storage engine, if you run `db.getCollectionNames()` and `db.collection.getIndexes()` from a version of the `mongo` shell before 3.0 or a version of the driver prior to *3.0 compatible version*, `db.getCollectionNames()` and `db.collection.getIndexes()` will return

[3] https://api.mongodb.org/

no data, even if there are existing collections and indexes. For more information, see *3.0-compatibility-drivers-wired-tiger*.

3.2.6 Measure Index Use

On this page

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 49) for more information.

Operations

Return Query Plan with `explain()`

Use the `db.collection.explain()` or the `cursor.explain()` method in *executionStats* mode to return statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

Run `db.collection.explain()` or the `cursor.explain()` method in *allPlansExecution* mode to view partial execution statistics collected during plan selection.

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

Control Index Use with `hint()`

To *force* MongoDB to use a particular index for a `db.collection.find()` operation, specify the index with the `hint()` method. Append the `hint()` method to the `find()` method. Consider the following example:

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

To view the execution statistics for a specific index, append to the `db.collection.find()` the `hint()` method followed by `cursor.explain()`, e.g.:

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

Or, append `hint()` method to `db.collection.explain().find()`:

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

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`:
  - `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.

Find Restaurants with Geospatial Queries **(page 65)** Use *Geospatial* queries to find a user's current neighborhood and list nearby restaurants.

Create a 2dsphere Index **(page 70)** A `2dsphere` index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

Query a 2dsphere Index **(page 73)** 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 75)** Create a `2d` index to support queries on data stored as legacy coordinate pairs.

Query a 2d Index **(page 76)** Search for locations using legacy coordinate pairs.

Create a Haystack Index **(page 78)** 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 79)** Search based on location and non-location data within a small area.

Calculate Distance Using Spherical Geometry **(page 79)** Convert distances to radians and back again.

3.3.1 Find Restaurants with Geospatial Queries

On this page

Overview

MongoDB's *geospatial* indexing allows you to efficiently execute spatial queries on a collection that contains geospatial shapes and points. This tutorial will briefly introduce the concepts of geospatial indexes, and then demonstrate their use with `$geoWithin`, `$geoIntersects`, and `geoNear`.

To showcase the capabilities of geospatial features and compare different approaches, this tutorial will guide you through the process of writing queries for a simple geospatial application.

Suppose you are designing a mobile application to help users find restaurants in New York City. The application must:

- Determine the user's current neighborhood using `$geoIntersects`,
- Show the number of restaurants in that neighborhood using `$geoWithin`, and
- Find restaurants within a specified distance of the user using `$nearSphere`.

This tutorial will use a `2dsphere` index to query for this data on spherical geometry.

Differences Between Flat and Spherical Geometry

Geospatial queries can use either flat or spherical geometries, depending on both the query and the type of index in use. `2dsphere` indexes support only spherical geometries, while `2d` indexes support both flat and spherical geometries.

However, queries using spherical geometries will be more performant and accurate with a `2dsphere` index, so you should always use `2dsphere` indexes on geographical geospatial fields.

The following table shows what kind of geometry each geospatial operator will use:

Query Type	Geometry Type	Notes
`$near` (*GeoJSON* point, `2dsphere` index)	Spherical	
`$near` (*legacy coordinates*, `2d` index)	Flat	
`$nearSphere` (*GeoJSON* point, `2dsphere` index)	Spherical	
`$nearSphere` (*legacy coordinates*, `2d` index)	Spherical	Use *GeoJSON* points instead.
`$geoWithin` : { `$geometry`: ... }	Spherical	
`$geoWithin` : { `$box`: ... }	Flat	
`$geoWithin` : { `$polygon`: ... }	Flat	
`$geoWithin` : { `$center`: ... }	Flat	
`$geoWithin` : { `$centerSphere`: ... }	Spherical	
`$geoIntersects`	Spherical	

The `geoNear` command and the `$geoNear` aggregation operator both operate in radians when using *legacy coordinates*, and meters when using *GeoJSON* points.

Distortion

Spherical geometry will appear distorted when visualized on a map due to the nature of projecting a three dimensional sphere, such as the earth, onto a flat plane.

For example, take the specification of the spherical square defined by the longitude latitude points `(0,0)`, `(80,0)`, `(80,80)`, and `(0,80)`. The following figure depicts the area covered by this region:

Searching for Restaurants

Prerequisites

Download the example datasets from https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/neighborhoods.json and https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/restaurants.json. These contain the collections `restaurants` and `neighborhoods` respectively.

After downloading the datasets, import them into the database:

```
mongoimport <path to restaurants.json> -c restaurants
mongoimport <path to neighborhoods.json> -c neighborhoods
```

The `geoNear` command requires a geospatial index, and almost always improves performance of `$geoWithin` and `$geoIntersects` queries.

Because this data is geographical, create a `2dsphere` index on each collection using the `mongo` shell:

```
db.restaurants.createIndex({ location: "2dsphere" })
db.neighborhoods.createIndex({ coordinates: "2dsphere" })
```

Exploring the Data

Inspect an entry in the newly-created `restaurants` collection from within the `mongo` shell:

```
db.restaurants.findOne()
```

This query returns a document like the following:

```
{
   location: {
      type: "Point",
      coordinates: [-73.856077, 40.848447]
   },
   name: "Morris Park Bake Shop"
}
```

This restaurant document corresponds to the location shown in the following figure:

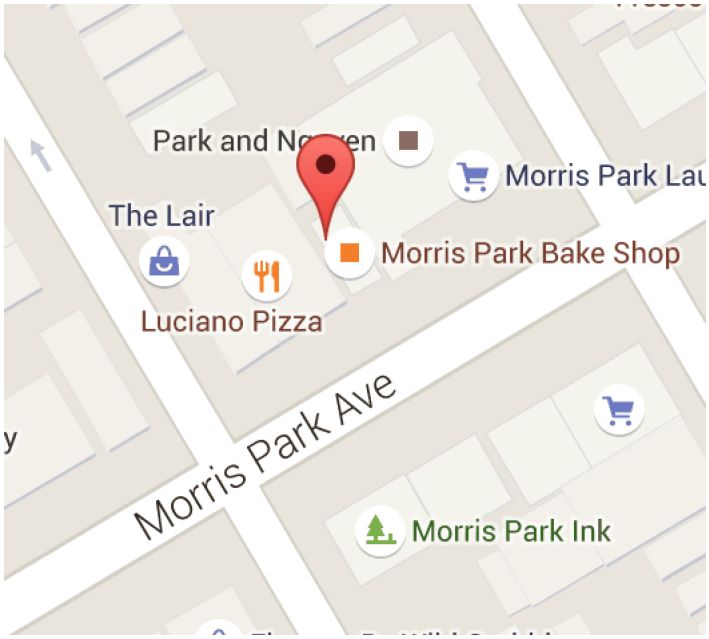
Because the tutorial uses a `2dsphere` index, the geometry data in the `location` field must follow the doc:*GeoJSON format </reference/geojson>*.

Now inspect an entry in the `neighborhoods` collection:

```
db.neighborhoods.findOne()
```

This query will return a document like the following:

```
{
   geometry: {
      type: "Polygon",
      coordinates: [[
         [ -73.99, 40.75 ],
         ...
         [ -73.98, 40.76 ],
         [ -73.99, 40.75 ]
      ]]
    },
```




```
    name: "Hell's Kitchen"
}
```

This geometry corresponds to the region depicted in the following figure:

Find the Current Neighborhood

Assuming the user's mobile device can give a reasonably accurate location for the user, it is simple to find the user's current neighborhood with `$geoIntersects`.

Suppose the user is located at -73.93414657 longitude and 40.82302903 latitude. To find the current neighborhood, you will specify a point using the special `$geometry` field in *GeoJSON* format:

```
db.neighborhoods.findOne({ geometry: { $geoIntersects: { $geometry: { type: "Point", coordinates: [
```

This query will return the following result:

```
{
    "_id" : ObjectId("55cb9c666c522cafdb053a68"),
    "geometry" : {
        "type" : "Polygon",
        "coordinates" : [
            [
                [
                    -73.93383000695911,
                    40.81949109558767
                ],
                ...
            ]
        ]
    },
    "name" : "Central Harlem North-Polo Grounds"
}
```


Hudson River
Lincoln Center for
the Performing Arts
West Dr
W 57th St
Intrepid Sea, Air
& Space Museum
HELL'S KITCHEN
Lincoln Tunnel (Toll road)
9A
495
11th Ave
W 42nd St
Times Square
The
of M
Madame Tussauds
New York
10th Ave
GARMENT
DISTRICT
MIDTOWN
5th Ave
Bryant Park
Frying Pan
Starrett-Lehigh
Ninth Ave
W 34th St

Find all Restaurants in the Neighborhood

You can also query to find all restaurants contained in a given neighborhood. Run the following in the `mongo` shell to find the neighborhood containing the user, and then count the restaurants within that neighborhood:

```
var neighborhood = db.neighborhoods.findOne( { geometry: { $geoIntersects: { $geometry: { type: "Poi
db.restaurants.find( { location: { $geoWithin: { $geometry: neighborhood.geometry } } } ).count()
```

This query will tell you that there are 127 restaurants in the requested neighborhood, visualized in the following figure:

Find Restaurants within a Distance

To find restaurants within a specified distance of a point, you can use either `$geoWithin` with `$centerSphere` to return results in unsorted order, or `nearSphere` with `$maxDistance` if you need results sorted by distance.

Unsorted with `$geoWithin`

To find restaurants within a circular region, use `$geoWithin` with `$centerSphere`. `$centerSphere` is a MongoDB-specific syntax to denote a circular region by specifying the center and the radius in radians.

`$geoWithin` does not return the documents in any specific order, so it may show the user the furthest documents first.

The following will find all restaurants within five miles of the user:

```
db.restaurants.find({ location:
   { $geoWithin:
      { $centerSphere: [ [ -73.93414657, 40.82302903 ], 5 / 3963.2 ] } } })
```

`$centerSphere`'s second argument accepts the radius in radians, so you must divide it by the radius of the earth in miles. See *Calculate Distance Using Spherical Geometry* (page 79) for more information on converting between distance units.

Sorted with `$nearSphere`

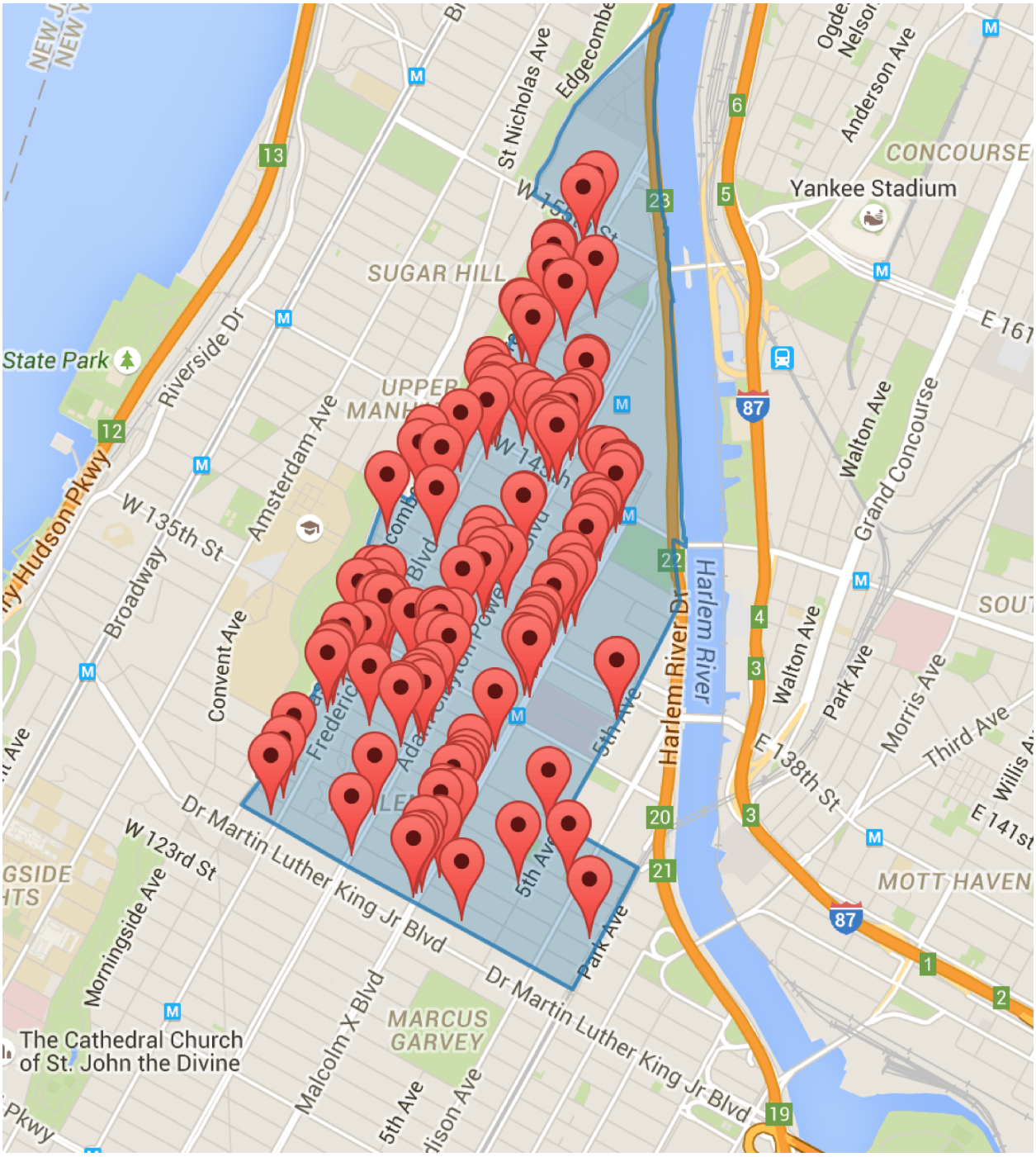
You may also use `$nearSphere` and specify a `$maxDistance` term in meters. This will return all restaurants within five miles of the user in sorted order from nearest to farthest:

```
var METERS_PER_MILE = 1609.34
db.restaurants.find({ location: { $nearSphere: { $geometry: { type: "Point", coordinates: [ -73.9341
```

3.3.2 Create a `2dsphere` Index

On this page

To create a geospatial index for GeoJSON-formatted data, use the `db.collection.createIndex()` method to create a *2dsphere index* (page 19). In the index specification document for the `db.collection.createIndex()` method, specify the location field as the index key and specify the string literal `"2dsphere"` as the value:


SUGAR HILL
UPPER MANH
Yankee Stadium
CONCOURSE
State Park
Riverside Dr
Amsterdam Ave
Broadway
W 135th St
W 145th
St Nicholas Ave
Edgecombe
Anderson Ave
Walton Ave
Grand Concourse
Harlem River
Harlem River Dr
Park Ave
Morris Ave
Third Ave
E 138th St
E 141st
MOTT HAVEN
Convent Ave
5th Ave
Dr Martin Luther King Jr Blvd
W 123rd St
Morningside Ave
Malcolm X Blvd
MARCUS GARVEY
The Cathedral Church of St. John the Divine
87
13
12
23
22
20
21
19

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

The following procedure presents steps to populate a collection with documents that contain a GeoJSON data field and create *2dsphere indexes* (page 19). Although the procedure populates the collection first, you can also create the indexes before populating the collection.

Procedure

First, populate a collection `places` with documents that store location data as *GeoJSON Point* (page 101) in a field named `loc`. The coordinate order is longitude, then latitude.

```
db.places.insert(
   {
      loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
      name: "Central Park",
      category : "Parks"
   }
)

db.places.insert(
   {
      loc : { type: "Point", coordinates: [ -73.88, 40.78 ] },
      name: "La Guardia Airport",
      category : "Airport"
   }
)
```

Then, create the *2dsphere* (page 19) index.

Create a `2dsphere` Index

For example, the following creates a *2dsphere* (page 19) index on the location field `loc`:

```
db.places.createIndex( { loc : "2dsphere" } )
```

Create a Compound Index with `2dsphere` Index Key

A *compound index* (page 10) can include a `2dsphere` index key in combination with non-geospatial index keys. For example, the following operation creates a compound index where the first key `loc` is a `2dsphere` index key, and the remaining keys `category` and `names` are non-geospatial index keys, specifically descending (`-1`) and ascending (`1`) keys respectively.

```
db.places.createIndex( { loc : "2dsphere" , category : -1, name: 1 } )
```

Unlike the *2d* (page 21) index, a compound `2dsphere` index does not require the location field to be the first field indexed. For example:

```
db.places.createIndex( { category : 1 , loc : "2dsphere" } )
```

Considerations

Fields with *2dsphere* (page 19) indexes must hold geometry data in the form of *coordinate pairs* or *GeoJSON* data. If you attempt to insert a document with non-geometry data in a `2dsphere` indexed field, or build a `2dsphere` index on a collection where the indexed field has non-geometry data, the operation will fail.

The `geoNear` command and the `$geoNear` pipeline stage require that a collection have *at most* only one `2dsphere` index and/or only one *2d* (page 21) index whereas *geospatial query operators* (e.g. `$near` and `$geoWithin`) permit collections to have multiple geospatial indexes.

The geospatial index restriction for the `geoNear` command and the `$geoNear` pipeline stage exists because neither the `geoNear` command nor the `$geoNear` pipeline stage syntax includes the location field. As such, index selection among multiple `2d` indexes or `2dsphere` indexes is ambiguous.

No such restriction applies for *geospatial query operators* since these operators take a location field, eliminating the ambiguity.

As such, although this tutorial creates multiple `2dsphere` indexes, to use the `geoNear` command or the `$geoNear` pipeline stage against the example collection, you will need to `drop` all but one of the `2dsphere` indexes.

To query using the `2dsphere` index, see *Query a 2dsphere Index* (page 73).

3.3.3 Query a `2dsphere` Index

On this page

The following sections describe queries supported by the `2dsphere` index.

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 79).

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 equatorial radius of the earth, 3963.2 miles:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $centerSphere :
                       [ [ -88 , 30 ] , 10 / 3963.2 ]
                } } } )
```

3.3.4 Create a `2d` Index

On this page

To build a geospatial `2d` index, use the :method:.createIndex() <db.collection.createIndex()>' method and specify `2d`. Use the following syntax:

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

For information on the internals of geohash values, see *Calculation of Geohash Values for 2d Indexes* (page 23).

3.3.5 Query a 2d Index

On this page

The following sections describe queries supported by the 2d index.

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 19).

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 79).

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 equatorial radius of the earth, 3963.2 miles:

```
db.<collection>.find( { loc : { $geoWithin :
                                { $centerSphere :
                                   [ [ 88 , 30 ] , 10 / 3963.2 ]
                      } } } )
```

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

Changed in version 2.6: Previously, `2d` indexes would support exact-match queries for coordinate pairs.

You cannot use a `2d` index to return an exact match for a coordinate pair. Use a scalar, ascending or descending, index on a field that stores coordinates to return exact matches.

In the following example, the `find()` operation will return an exact match on a location if you have a `{ 'loc': 1}` index:

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

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

3.3.6 Create a Haystack Index

A haystack index must 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.createIndex( { <location field> : "geoHaystack" ,
                       <additional field> : 1 } ,
                     { bucketSize : <bucket value> } )
```

To build a haystack index, you must specify 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.

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.createIndex( { 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 79).

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

3.3.7 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 78).

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 79) are not currently supported by haystack indexes.

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

3.3.8 Calculate Distance Using Spherical Geometry

On this page

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 equatorial radius of the Earth is approximately `3,963.2` miles or `6,378.1` 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 / 3963.2 ] } } } )
```

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 80).

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 79) example:

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true,
                 distanceMultiplier: 3963.2
               } )
```

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.

Create a text Index **(page 81)** A `text` index allows searches on text strings in the index's specified fields.

Specify a Language for Text Index **(page 82)** The specified language determines the list of stop words and the rules for Text Search's stemmer and tokenizer.

Text Search with Basis Technology Rosette Linguistics Platform **(page 84)** Enable text search support for Arabic, Farsi (specifically Dari and Iranian Persian dialects), Urdu, Simplified Chinese, and Traditional Chinese. MongoDB Enterprise feature only.

Specify Name for text Index **(page 86)** Override the `text` index name limit for long index names.

Control Search Results with Weights **(page 88)** 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 89)** Create an index to support queries that includes `$text` expressions and equality conditions.

Text Search in the Aggregation Pipeline **(page 89)** Perform various text search in the aggregation pipeline.

3.4.1 Create a `text` Index

On this page

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 use wildcard specifier (`$**`).

Index Specific Fields

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

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

Note: In order to drop a `text` index, use the index name. See *Use the Index Name to Drop a text Index* (page 87) for more information.

3.4.2 Specify a Language for Text Index

On this page

This tutorial describes how to *specify the default language associated with the text index* (page 83) and also how to *create text indexes for collections that contain documents in different languages* (page 83).

Specify the Default Language for a `text` Index

The default language associated with the indexed data determines the rules to parse word roots (i.e. stemming) and ignore stop words. 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* (page 104) for the languages available for `default_language`.

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

```
db.quotes.createIndex(
   { content : "text" },
   { default_language: "spanish" }
)
```

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

Changed in version 2.6: Added support for language overrides within embedded documents.

Specify the Index Language within the Document

If a collection contains documents or embedded documents that are in different languages, include a field named `language` in the documents or embedded documents and specify as its value the language for that document or embedded document.

MongoDB will use the specified language for that document or embedded document when building the `text` index:

- The specified language in the document overrides the default language for the `text` index.
- The specified language in an embedded document override the language specified in an enclosing document or the default language for the index.

See *Text Search Languages* (page 104) for a list of supported languages.

For example, a collection `quotes` contains multi-language documents that include the `language` field in the document and/or the embedded document as needed:

```
{
   _id: 1,
   language: "portuguese",
   original: "A sorte protege os audazes.",
   translation:
     [
        {
           language: "english",
           quote: "Fortune favors the bold."
        },
        {
           language: "spanish",
           quote: "La suerte protege a los audaces."
        }
    ]
}
{
   _id: 2,
   language: "spanish",
   original: "Nada hay más surrealista que la realidad.",
```

```
   translation:
      [
        {
          language: "english",
          quote: "There is nothing more surreal than reality."
        },
        {
          language: "french",
          quote: "Il n'y a rien de plus surréaliste que la réalité."
        }
      ]
}
{
   _id: 3,
   original: "is this a dagger which I see before me.",
   translation:
   {
      language: "spanish",
      quote: "Es este un puñal que veo delante de mí."
   }
}
```

If you create a `text` index on the `quote` field with the default language of English.

```
db.quotes.createIndex( { original: "text", "translation.quote": "text" } )
```

Then, for the documents and embedded documents that contain the `language` field, the `text` index uses that language to parse word stems and other linguistic characteristics.

For embedded documents that do not contain the `language` field,

- If the enclosing document contains the `language` field, then the index uses the document's language for the embedded document.
- Otherwise, the index uses the default language for the embedded documents.

For documents that do not contain the `language` field, the index uses the default language, which is English.

Use any Field to Specify the Language for a Document

To use a field with a name other than `language`, include the `language_override` option when creating the index.

For example, give the following command to use `idioma` as the field name instead of `language`:

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

The documents of the `quotes` collection may specify a language with the `idioma` field:

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

3.4.3 Text Search with Basis Technology Rosette Linguistics Platform

On this page

Enterprise Feature

Available in MongoDB Enterprise only.

Overview

New in version 3.2.

MongoDB Enterprise provides support for the following languages: Arabic, Farsi (specifically Dari and Iranian Persian dialects), Urdu, Simplified Chinese, and Traditional Chinese.

To provide support for these languages, MongoDB Enterprise integrates Basis Technology Rosette Linguistics Platform (RLP) to perform normalization, word breaking, sentence breaking, and stemming or tokenization depending on the language.

MongoDB Enterprise supports RLP SDK 7.11.1 on Red Hat Enterprise Linux 6.x. For information on providing support on other platforms, contact your sales representative.

See also:

Text Search Languages (page 104), *Specify a Language for Text Index* (page 82)

Prerequisites

To use MongoDB with RLP, MongoDB requires a license for the Base Linguistics component of RLP and one or more languages specified above. MongoDB does not require a license for all 6 languages listed above.

Support for any of the specified languages is conditional on having a valid RLP license for the language. For instance, if there is only an RLP license provided for Arabic, then MongoDB will only enable support for Arabic and will not enable support for any other RLP based languages. For any language which lacks a valid license, the MongoDB log will contain a warning message. Additionally, you can set the MongoDB log verbosity level to `2` to log debug messages that identify each supported language.

You do not need the Language Extension Pack as MongoDB does not support these RLP languages at this time.

Contact Basis Technology at info@basistech.com[4] to get a copy of RLP and a license for one or more languages. For more information on how to contact Basis Technology, see http://www.basistech.com/contact/.

Procedure

Step 1: Download Rosette Linguistics Platform from Basis Technology.

From Basis Technology, obtain the links to download the RLP C++ SDK package file, the documentation package file, and the license file (`rlp-license.xml`) for Linux x64. Basis Technology provides the download links in an email.

[4] info@basistech.com

Using the links, download the RLP C++ SDK package file, the documentation package file, and the license file (`rlp-license.xml`) for Linux x64.

Note: These links automatically expire after 30 days.

Step 2: Install the RLP binaries.

Untar the RLP binaries and place them in a directory; this directory is referred to as the installation directory or `BT_ROOT`. For this example, we will use `/opt/basis` as the `BT_ROOT`.

```
tar zxvC /opt/basis rlp-7.11.1-sdk-amd64-glibc25-gcc41.tar.gz
```

Step 3: Move the RLP license into the RLP licenses directory.

Move the RLP license file `rlp-license.xml` to the `<BT_ROOT>/rlp/rlp/licenses` directory; in our example, move the file to the `/opt/basis/rlp/rlp/licenses/` directory.

```
mv rlp-license.xml /opt/basis/rlp/rlp/licenses/
```

Step 4: Run `mongod` with RLP support.

To enable support for RLP, use the `--basisTechRootDirectory` option to specify the `BT_ROOT` directory.

Include any additional settings as appropriate for your deployment.

```
mongod --basisTechRootDirectory=/opt/basis
```

Additional Information

For installation help, see the RLP Quick Start manual or Chapter 2 of the Rosette Linguistics Platform Application Developer's Guide.

For debugging any RLP specific issues, you can set the `rlpVerbose` parameter to `true` (i.e. `--setParameter rlpVerbose=true`) to view `INFO` messages from RLP.

Warning: Enabling `rlpVerbose` has a performance overhead and should only be enabled for troubleshooting installation issues.

3.4.4 Specify Name for `text` Index

On this page

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.createIndex(
   {
     content: "text",
     "users.comments": "text",
     "users.profiles": "text"
   }
)
```

The default name for the index is:

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

The `text` index, like other indexes, must fall within the `index name length limit`.

Specify a Name for `text` Index

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.createIndex()` method:

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

Use the Index Name to Drop a `text` Index

Whether the *text* (page 24) index has the default name or you specified a name for the *text* (page 24) index, to drop the *text* (page 24) index, pass the index name to the `db.collection.dropIndex()` method.

For example, consider the index created by the following operation:

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

Then, to remove this text index, pass the name `"MyTextIndex"` to the `db.collection.dropIndex()` method, as in the following:

```
db.collection.dropIndex("MyTextIndex")
```

To get the names of the indexes, use the `db.collection.getIndexes()` method.

3.4.5 Control Search Results with Weights

Text search 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.

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 text search score.

For each indexed field in the document, MongoDB multiplies the number of matches by the weight and sums the results. Using this sum, MongoDB then calculates the score for the document. See `$meta` operator for details on returning and sorting by text scores.

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.createIndex()` 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 `createIndex()` method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```
db.blog.createIndex(
   {
     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.

3.4.6 Limit the Number of Entries Scanned

This tutorial describes how to create indexes to limit the number of index entries scanned for queries that includes a `$text` expression and equality conditions.

A collection `inventory` contains the following documents:

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

Consider the common use case that performs text searches by *individual* departments, such as:

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

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

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

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

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

Note:

- A compound `text` index cannot include any other special index types, such as *multi-key* (page 13) or *geospatial* (page 18) index fields.
- If the compound `text` index includes keys **preceding** the `text` index key, to perform a `$text` search, the query predicate must include **equality match conditions** on the preceding keys.

See also:

Text Indexes (page 24)

3.4.7 Text Search in the Aggregation Pipeline

On this page

New in version 2.6. In the aggregation pipeline, text search is available via the use of the `$text` query operator in the `$match` stage.

Restrictions

Text search in the aggregation pipeline has the following restrictions:

- The `$match` stage that includes a `$text` must be the **first** stage in the pipeline.
- A `text` operator can only occur once in the stage.
- The `text` operator expression cannot appear in `$or` or `$not` expressions.
- The text search, by default, does not return the matching documents in order of matching scores. Use the `$meta` aggregation expression in the `$sort` stage.

Text Score

The `$text` operator assigns a score to each document that contains the search term in the indexed fields. The score represents the relevance of a document to a given text search query. The score can be part of a `$sort` pipeline specification as well as part of the projection expression. The `{ $meta: "textScore" }` expression provides information on the processing of the `$text` operation. See `$meta` aggregation for details on accessing the score for projection or sort.

The metadata is only available after the `$match` stage that includes the `$text` operation.

Examples

The following examples assume a collection `articles` that has a text index on the field `subject`:

```
db.articles.createIndex( { subject: "text" } )
```

Calculate the Total Views for Articles that Contains a Word

The following aggregation searches for the term `cake` in the `$match` stage and calculates the total `views` for the matching documents in the `$group` stage.

```
db.articles.aggregate(
    [
      { $match: { $text: { $search: "cake" } } },
      { $group: { _id: null, views: { $sum: "$views" } } }
    ]
)
```

Return Results Sorted by Text Search Score

To sort by the text search score, include a `$meta` expression in the `$sort` stage. The following example matches on *either* the term `cake` or `tea`, sorts by the `textScore` in descending order, and returns only the `title` field in the results set.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "cake tea" } } },
     { $sort: { score: { $meta: "textScore" } } },
     { $project: { title: 1, _id: 0 } }
   ]
)
```

The specified metadata determines the sort order. For example, the `"textScore"` metadata sorts in descending order. See `$meta` for more information on metadata as well as an example of overriding the default sort order of the metadata.

Match on Text Score

The `"textScore"` metadata is available for projections, sorts, and conditions subsequent the `$match` stage that includes the `$text` operation.

The following example matches on *either* the term `cake` or `tea`, projects the `title` and the `score` fields, and then returns only those documents with a `score` greater than `1.0`.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "cake tea" } } },
     { $project: { title: 1, _id: 0, score: { $meta: "textScore" } } },
     { $match: { score: { $gt: 1.0 } } }
   ]
)
```

Specify a Language for Text Search

The following aggregation searches in spanish for documents that contain the term `saber` but not the term `claro` in the `$match` stage and calculates the total `views` for the matching documents in the `$group` stage.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "saber -claro", $language: "es" } } },
     { $group: { _id: null, views: { $sum: "$views" } } }
   ]
)
```

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.

Generally, MongoDB only uses *one* index to fulfill most queries. However, each clause of an `$or` query may use a different index, and starting in 2.6, MongoDB can use an *intersection* (page 41) of multiple indexes.

The following documents introduce indexing strategies:

***Create Indexes to Support Your Queries* (page 92)** 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 93)** 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 95)** 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 96)** 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.

3.5.1 Create Indexes to Support Your Queries

On this page

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 support 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.createIndex( { "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.createIndex( { "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 10) on multiple fields can support all the queries that search a "prefix" subset of those fields.

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 93).

Starting in version 2.6, MongoDB can use *index intersection* (page 41) to fulfill queries. The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system. See *Index Intersection and Compound Indexes* (page 42) for more details.

3.5.2 Use Indexes to Sort Query Results

On this page

In MongoDB, sort operations can obtain the sort order by retrieving documents based on the ordering in an index. If the query planner cannot obtain the sort order from an index, it will sort the results in memory. Sort operations that use an index often have better performance than those that do not use an index. In addition, sort operations that do *not* use an index will abort when they use 32 megabytes of memory.

Sort with a Single Field Index

If an ascending or a descending index is on a single field, the sort operation on the field can be in either direction.

For example, create an ascending index on the field `a` for a collection `records`:

```
db.records.createIndex( { a: 1 } )
```

This index can support an ascending sort on `a`:

```
db.records.find().sort( { a: 1 } )
```

The index can also support the following descending sort on `a` by traversing the index in reverse order:

```
db.records.find().sort( { a: -1 } )
```

Sort on Multiple Fields

Create a *compound index* (page 10) to support sorting on multiple fields.

You can specify a sort on all the keys of the index or on a subset; however, the sort keys must be listed in the *same order* as they appear in the index. For example, an index key pattern `{ a: 1, b: 1 }` can support a sort on `{ a: 1, b: 1 }` but *not* on `{ b: 1, a: 1 }`.

The sort must specify the *same sort direction* (i.e.ascending/descending) for all its keys as the index key pattern or specify the *reverse sort direction* for all its keys as the index key pattern. For example, an index key pattern `{ a: 1, b: 1 }` can support a sort on `{ a: 1, b: 1 }` and `{ a: -1, b: -1 }` but *not* on `{ a: -1, b: 1 }`.

Sort and Index Prefix

If the sort keys correspond to the index keys or an index *prefix*, MongoDB can use the index to sort the query results. A *prefix* of a compound index is a subset that consists of one or more keys at the start of the index key pattern.

For example, create a compound index on the `data` collection:

```
db.data.createIndex( { a:1, b: 1, c: 1, d: 1 } )
```

Then, the following are prefixes for that index:

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

The following query and sort operations use the index prefixes to sort the results. These operations do not need to sort the result set in memory.

Example	Index Prefix
`db.data.find().sort( { a: 1 } )`	`{ a: 1 }`
`db.data.find().sort( { a: -1 } )`	`{ a: 1 }`
`db.data.find().sort( { a: 1, b: 1 } )`	`{ a: 1, b: 1 }`
`db.data.find().sort( { a: -1, b: -1 } )`	`{ a: 1, b: 1 }`
`db.data.find().sort( { a: 1, b: 1, c: 1 } )`	`{ a: 1, b: 1, c: 1 }`
`db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )`	`{ a: 1, b: 1 }`

Consider the following example in which the prefix keys of the index appear in both the query predicate and the sort:

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

In such cases, MongoDB can use the index to retrieve the documents in order specified by the sort. As the example shows, the index prefix in the query predicate can be different from the prefix in the sort.

Sort and Non-prefix Subset of an Index

An index can support sort operations on a non-prefix subset of the index key pattern. To do so, the query must include **equality** conditions on all the prefix keys that precede the sort keys.

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

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

The following operations can use the index to get the sort order:

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

As the last operation shows, only the index fields *preceding* the sort subset must have the equality conditions in the query document; the other index fields may specify other conditions.

If the query does **not** specify an equality condition on an index prefix that precedes or overlaps with the sort specification, the operation will **not** efficiently use the index. For example, 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.data.find( { a: { $gt: 2 } } ).sort( { c: 1 } )
db.data.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.

3.5.3 Ensure Indexes Fit in RAM

On this page

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 96).

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.

3.5.4 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 10) 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 63).

For a conceptual introduction to indexes in MongoDB see *Index Concepts* (page 7).

CHAPTER 4

Indexing Reference

On this page

4.1 Indexing Methods in the mongo Shell

Name	Description
db.collection.createIndex()	Builds an index on a collection.
db.collection.dropIndex()	Removes a specified index on a collection.
db.collection.dropIndexes()	Removes all indexes on a collection.
db.collection.getIndexes()	Returns an array of documents that describe the existing indexes on a collection.
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 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
`createIndexes`	Builds one or more indexes for a collection.
`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.
`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.
`checkShardingIndex`	Internal command that validates index on shard key.

4.3 Geospatial Query Selectors

Name	Description
`$geoWithin`	Selects geometries within a bounding *GeoJSON geometry* (page 100). The *2dsphere* (page 19) and *2d* (page 21) indexes support `$geoWithin`.
`$geoIntersects`	Selects geometries that intersect with a *GeoJSON* geometry. The *2dsphere* (page 19) index supports `$geoIntersects`.
`$near`	Returns geospatial objects in proximity to a point. Requires a geospatial index. The *2dsphere* (page 19) and *2d* (page 21) indexes support `$near`.
`$nearSphere`	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The *2dsphere* (page 19) and *2d* (page 21) indexes support `$nearSphere`.

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`	Guarantees that a query returns each document no more than once. See `snapshot()`.

4.5 Other Index References

GeoJSON Objects **(page 100)** Supported GeoJSON objects.

Text Search Languages **(page 104)** Supported languages for *text indexes* (page 24) and `$text` query operations.

4.5.1 GeoJSON Objects

On this page

Overview

MongoDB supports the GeoJSON object types listed on this page.

To specify GeoJSON data, use a document with a `type` field specifying the GeoJSON object type and a `coordinates` field specifying the object's coordinates:

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

Important: Always list coordinates in `longitude, latitude` order.

The default coordinate reference system for GeoJSON uses the *WGS84* datum.

Point

New in version 2.4.

The following example specifies a GeoJSON Point[1]:

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

LineString

New in version 2.4.

The following example specifies a GeoJSON LineString[2]:

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

Polygon

New in version 2.4.

Polygons[3] 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.

[1] http://geojson.org/geojson-spec.html#point
[2] http://geojson.org/geojson-spec.html#linestring
[3] http://geojson.org/geojson-spec.html#polygon

The line that joins two points on a curved surface may or may not contain the same set of co-ordinates that joins those two points on a flat surface. The line that joins two points on a curved surface will be a geodesic. Carefully check points to avoid errors with shared edges, as well as overlaps and other types of intersections.

Polygons with a Single Ring

The following example specifies a GeoJSON `Polygon` with an exterior ring and no interior rings (or holes). The first and last coordinates must match in order to close the polygon:

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

For Polygons with a single ring, the ring cannot self-intersect.

Polygons with Multiple Rings

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 cannot share an edge.

The following example represents a GeoJSON polygon with an interior ring:

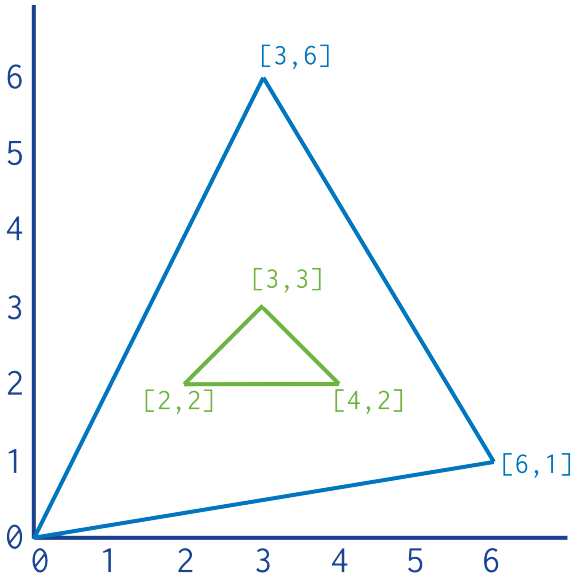
```
{
  type : "Polygon",
  coordinates : [
     [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
     [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ]
  ]
}
```

MultiPoint

New in version 2.6: Requires *2dsphere (Version 2)* (page 20)

GeoJSON *MultiPoint <http://geojson.org/geojson-spec.html#multipoint>* embedded documents encode a list of points.

```
{
  type: "MultiPoint",
  coordinates: [
     [ -73.9580, 40.8003 ],
     [ -73.9498, 40.7968 ],
     [ -73.9737, 40.7648 ],
     [ -73.9814, 40.7681 ]
  ]
}
```



MultiLineString

New in version 2.6: Requires *2dsphere (Version 2)* (page 20)

The following example specifies a GeoJSON MultiLineString[4]:

```
{
  type: "MultiLineString",
  coordinates: [
     [ [ -73.96943, 40.78519 ], [ -73.96082, 40.78095 ] ],
     [ [ -73.96415, 40.79229 ], [ -73.95544, 40.78854 ] ],
     [ [ -73.97162, 40.78205 ], [ -73.96374, 40.77715 ] ],
     [ [ -73.97880, 40.77247 ], [ -73.97036, 40.76811 ] ]
  ]
}
```

MultiPolygon

New in version 2.6: Requires *2dsphere (Version 2)* (page 20)

The following example specifies a GeoJSON MultiPolygon[5]:

```
{
  type: "MultiPolygon",
  coordinates: [
     [ [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -73.9814, 40.7681 ],
     [ [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -73.958, 40.8003 ] ] ]
  ]
}
```

[4] http://geojson.org/geojson-spec.html#multilinestring
[5] http://geojson.org/geojson-spec.html#multipolygon

GeometryCollection

New in version 2.6: Requires *2dsphere (Version 2)* (page 20)

The following example stores coordinates of GeoJSON type GeometryCollection[6]:

```
{
  type: "GeometryCollection",
  geometries: [
     {
       type: "MultiPoint",
       coordinates: [
          [ -73.9580, 40.8003 ],
          [ -73.9498, 40.7968 ],
          [ -73.9737, 40.7648 ],
          [ -73.9814, 40.7681 ]
       ]
     },
     {
       type: "MultiLineString",
       coordinates: [
          [ [ -73.96943, 40.78519 ], [ -73.96082, 40.78095 ] ],
          [ [ -73.96415, 40.79229 ], [ -73.95544, 40.78854 ] ],
          [ [ -73.97162, 40.78205 ], [ -73.96374, 40.77715 ] ],
          [ [ -73.97880, 40.77247 ], [ -73.97036, 40.76811 ] ]
       ]
     }
  ]
}
```

4.5.2 Text Search Languages

The *text index* (page 24) and the `$text` operator supports the following languages:

Changed in version 2.6: MongoDB introduces version 2 of the text search feature. With version 2, text search feature supports using the two-letter language codes defined in ISO 639-1. Version 1 of text search only supported the long form of each language name.

Changed in version 3.2: MongoDB Enterprise includes support for Arabic, Farsi (specifically Dari and Iranian Persian dialects), Urdu, Simplified Chinese, and Traditional Chinese. To support the new languages, the text search feature uses the three-letter language codes defined in ISO 636-3. To enable support for these languages, see *Text Search with Basis Technology Rosette Linguistics Platform* (page 84).

[6] http://geojson.org/geojson-spec.html#geometrycollection

Language Name	ISO 639-1 (Two letter codes)	ISO 636-3 (Three letter codes)	RLP names (Three letter codes)
`danish`	`da`		
`dutch`	`nl`		
`english`	`en`		
`finnish`	`fi`		
`french`	`fr`		
`german`	`de`		
`hungarian`	`hu`		
`italian`	`it`		
`norwegian`	`nb`		
`portuguese`	`pt`		
`romanian`	`ro`		
`russian`	`ru`		
`spanish`	`es`		
`swedish`	`sv`		
`turkish`	`tr`		
`arabic`		`ara`	
`dari`		`prs`	
`iranian persian`		`pes`	
`urdu`		`urd`	
`simplified chinese` or `hans`			`zhs`
`traditional chinese` or `hant`			`zht`

Note: If you specify a language value of `"none"`, then the text search uses simple tokenization with no list of stop words and no stemming.

See also:

Specify a Language for Text Index (page 82)