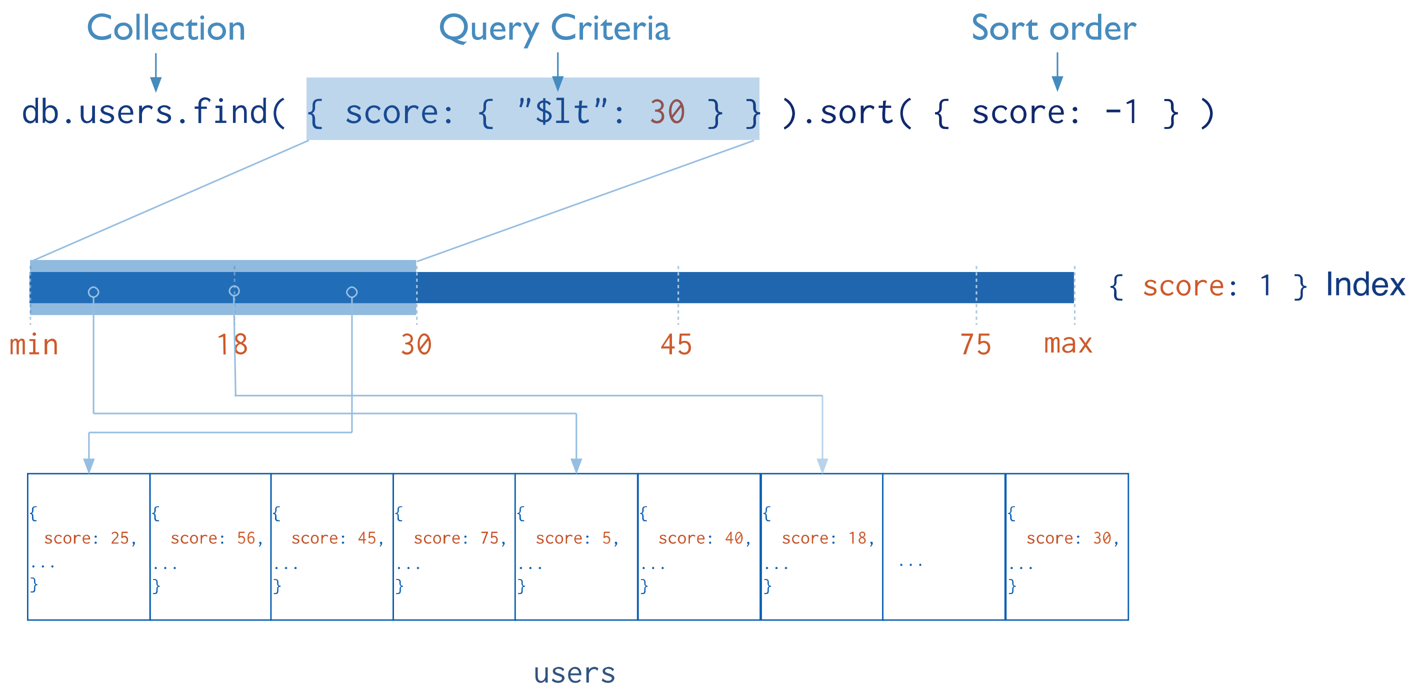
Indexes

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.

Default `_id` Index

MongoDB creates a unique index on the _id field during the creation of a collection. The `_id` index prevents clients from inserting two documents with the same value for the `_id` field. You cannot drop this index on the `_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.

Create an Index

To create an index, use `db.collection.createIndex()` or a similar method from your driver.

```
db.collection.createIndex( <key and index type specification>, <options> )
```

The `db.collection.createIndex()` method only creates an index if an index of the same specification does not already exist.

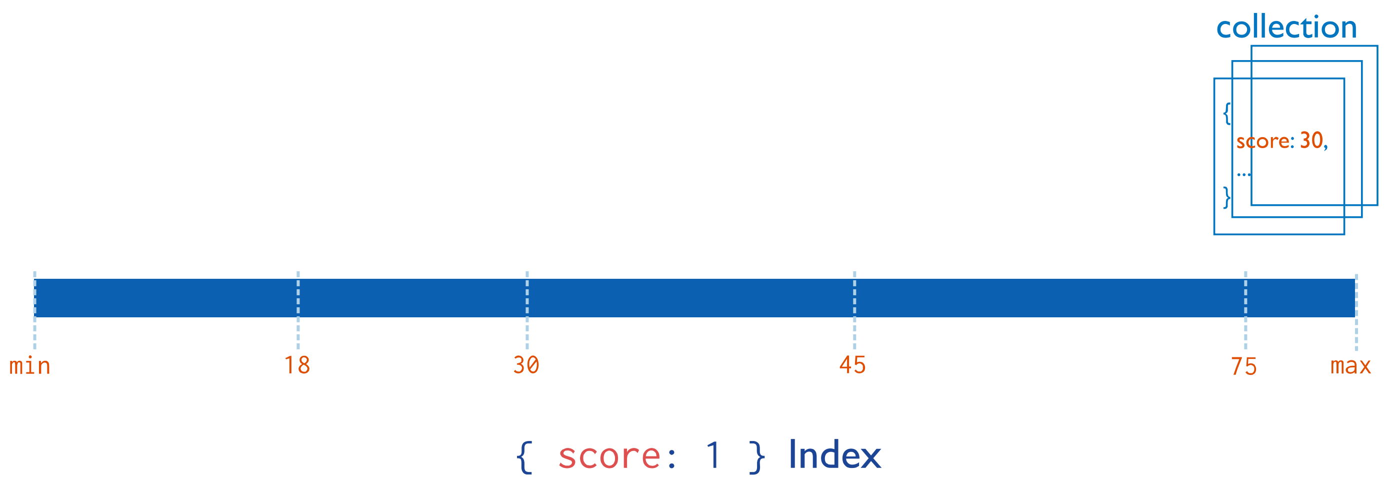
[1] MongoDB indexes use a B-tree data structure.

Index Types

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

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.



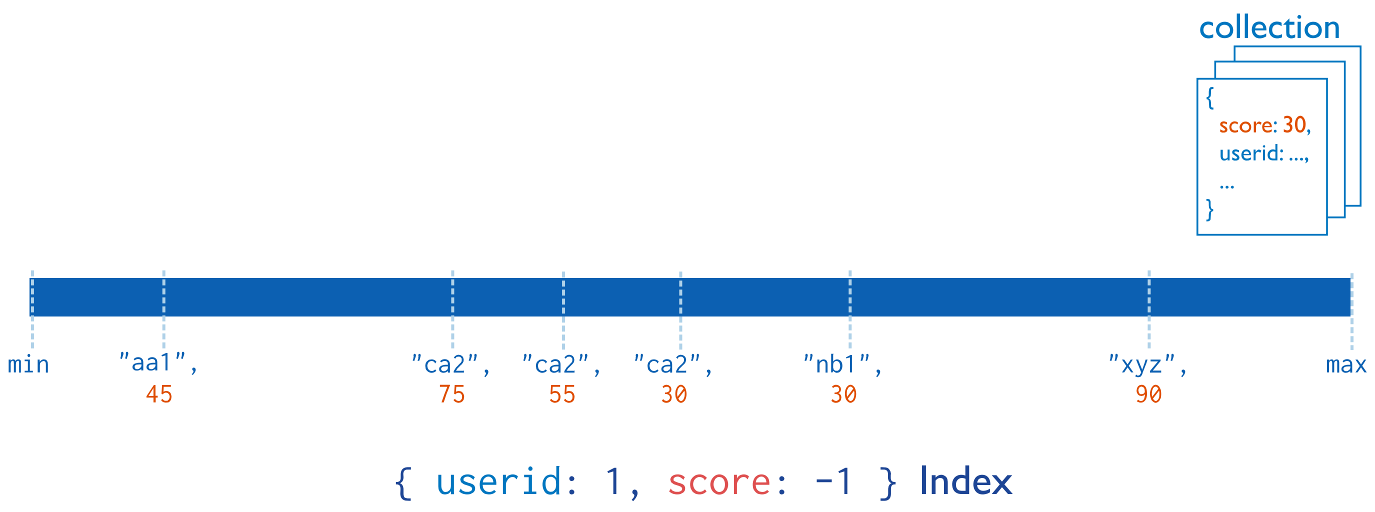
For a single-field index and sort operations, the sort order (i.e. ascending or descending) of the index key does not matter because MongoDB can traverse the index in either direction.

See Single Field Indexes and Sort with a Single Field Index for more information on single-field indexes.

Compound Index

MongoDB also supports user-defined indexes on multiple fields, i.e. compound indexes.

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`.

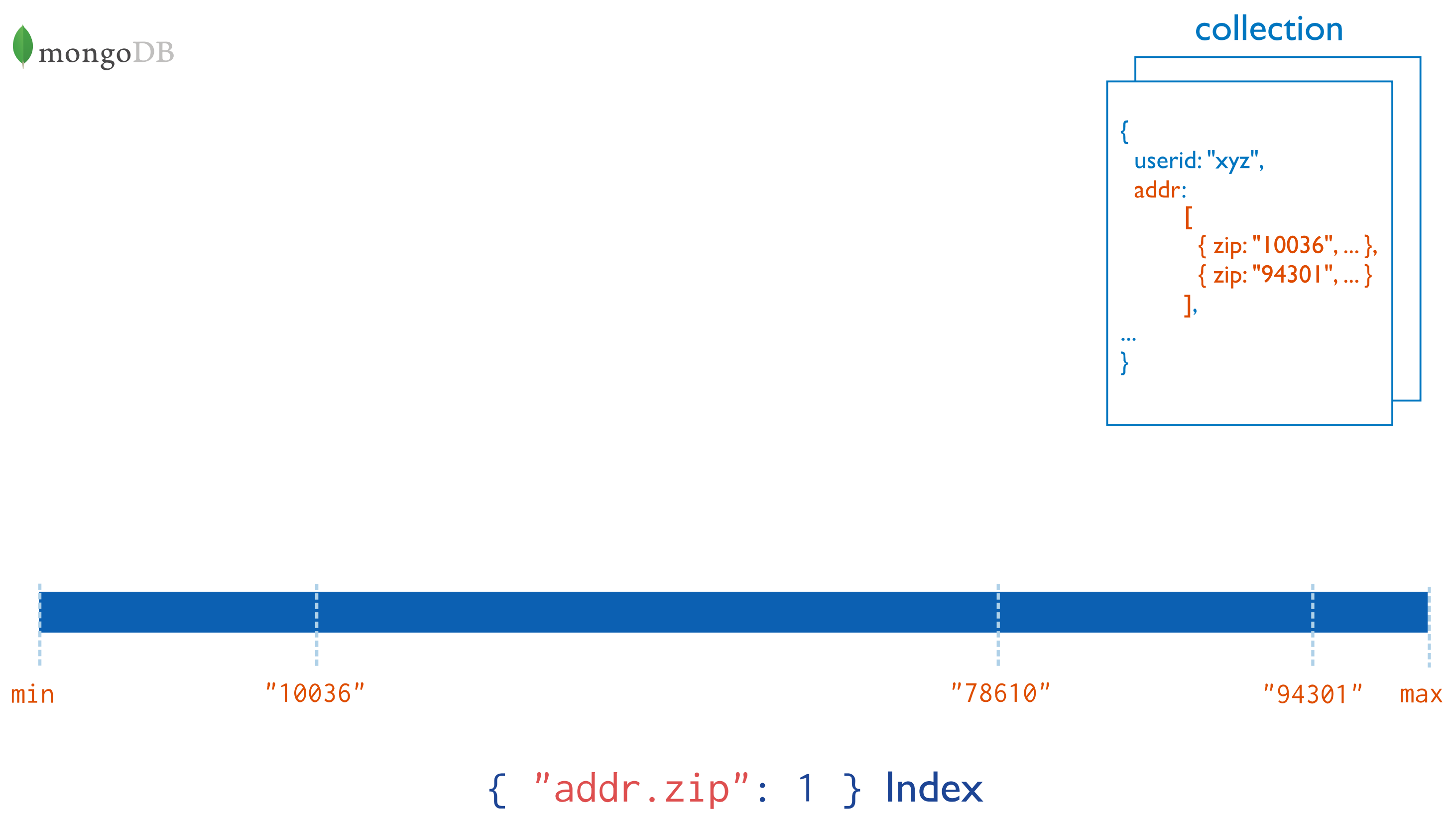


For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support a sort operation. See Sort Order for more information on the impact of index order on results in compound indexes.

See Compound Indexes and Sort on Multiple Fields for more information on compound indexes.

Multikey Index

MongoDB uses multikey indexes 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 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 and Multikey Index Bounds for more information on multikey indexes.

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: 2d indexes that uses planar geometry when returning results and 2dsphere indexes that use spherical geometry to return results.

See 2d Index Internals for a high level introduction to geospatial indexes.

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 for more information on text indexes and search.

Hashed Indexes

To support hash based sharding, MongoDB provides a hashed index 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.

Index Properties

Unique Indexes

The unique 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.

Partial Indexes

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.

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

Sparse Indexes

The sparse 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.

TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for 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: Expire Data from Collections by Setting TTL for implementation instructions.

Index Use

Indexes can improve the efficiency of read operations. The Analyze Query Performance 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.

Indexes and Collation

To use an index for string comparisons, an operation must also specify the same collation. That is, an index with a collation cannot support an operation that performs string comparisons on the indexed fields if the operation specifies a different collation.

For example, the collection `myColl` has an index on a string field `category` with the collation locale `"fr"`.

```
db.myColl.createIndex( { category: 1 }, { collation: { locale: "fr" } } )
```

The following query operation, which specifies the same collation as the index, can use the index:

```
db.myColl.find( { category: "cafe" } ).collation( { locale: "fr" } )
```

However, the following query operation, which by default uses the “simple” binary collator, cannot use the index:

```
db.myColl.find( { category: "cafe" } )
```

For a compound index where the index prefix keys are not strings, arrays, and embedded documents, an operation that specifies a different collation can still use the index to support comparisons on the index prefix keys.

For example, the collection `myColl` has a compound index on the numeric fields `score` and `price` and the string field `category`; the index is created with the collation locale `"fr"` for string comparisons:

```
db.myColl.createIndex(
   { score: 1, price: 1, category: 1 },
   { collation: { locale: "fr" } } )
```

The following operations, which use `"simple"` binary collation for string comparisons, can use the index:

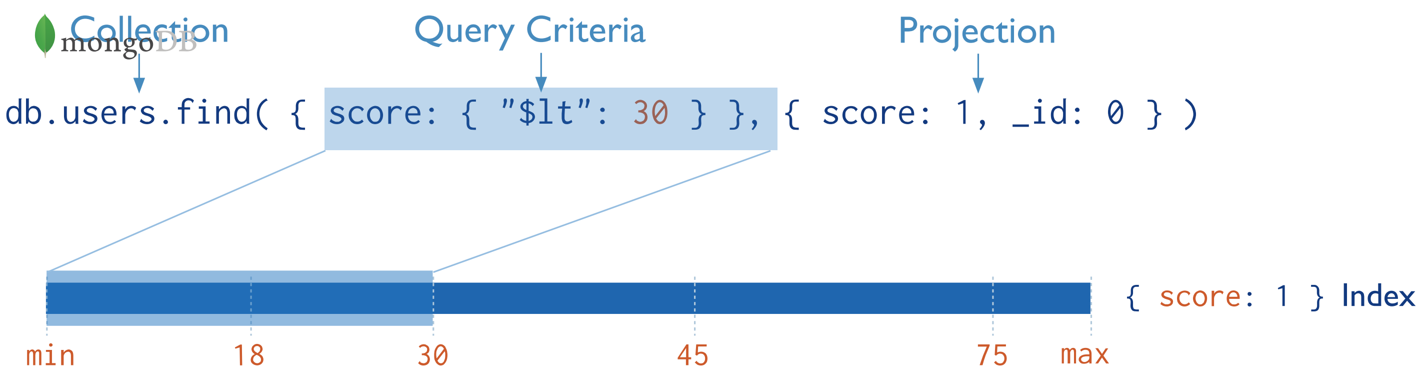
```
db.myColl.find( { score: 5 } ).sort( { price: 1 } )
db.myColl.find( { score: 5, price: { $gt: NumberDecimal( "10" ) } } ).sort( { price: 1 }
```

The following operation, which uses `"simple"` binary collation for string comparisons on the indexed `category` field, can use the index to fulfill only the `score: 5` portion of the query:

```
db.myColl.find( { score: 5, category: "cafe" } )
```

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 Covered Query.

Index Intersection

New in version 2.6.

MongoDB can use the intersection of indexes 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.

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.

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.

To build or rebuild indexes for a replica set, see Build Indexes on Replica Sets.

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

Additional Resources

- Quick Reference Cards