Query Optimizing Using Indexing in MongoDB

**Introduction:**

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 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.

MongoDB provides complete support for indexes on any field in a [collection](https://docs.mongodb.com/manual/reference/glossary/#term-collection) of [documents](https://docs.mongodb.com/manual/reference/glossary/#term-document). By default, all collections have an index on the [\_id field](https://docs.mongodb.com/manual/indexes/#index-type-id), and applications and users may add additional indexes to support important queries and operations.

**Objectives:**

Objective of the experiment is to reduce the cost of MongoDB queries on a dataset using the following indexing strategies.

* Single field indexing
* Compound indexing

**Single Field Indexing:**

**db.collection.createIndex( { field:1 } )**

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

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

The order of items in single field index does not show any effect on performance.  For single-field indexes, the sort order of keys doesn’t matter because MongoDB can traverse the index in either direction.

**Compound Index:**

MongoDB supports compound indexes where a single index structure holds references to multiple fields within a collection’s documents. Compound indexes can support queries that match on multiple fields. MongoDB imposes a limit of 32 fields for any compound index.

To create a [compound index](https://docs.mongodb.com/manual/core/index-compound/#index-type-compound) use an operation that resembles the following prototype:

**db.collection.createIndex({<field1>:<type>,<field2>:<type2>,...})**

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

The order of the fields listed in a compound index is important. 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. 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 [compound indexes](https://docs.mongodb.com/manual/core/index-compound/#index-type-compound), sort order can matter in determining whether the index can support a sort operation.

Consider a collection that contains documents with the fields

**db.collection.find().sort( { field1: 1, field2: -1 } )**

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.

**Dataset Used:**

Dataset name – Black Friday

About this file:

The dataset is about study of sales through consumer behaviors. Dataset of observations about the black Friday in a retail store, it contains different kinds of variables either numerical or categorical. It contains missing values.

Columns:

User\_ID - User ID

Product\_ID - Product ID

Gender - Sex of User

Age - Age in bins

Occupation – Occupation

City\_Category - Category of the City (A,B,C)

Stay\_In\_Current\_City\_Years - Number of years stay in current city

Marital\_Status - Marital Status

Product\_Category\_1 - Product Category

Product\_Category\_2 - Product may belongs to other category also

Product\_Category\_3 - Product may belongs to other category also

Purchase - Purchase amount in dollars

**Experiment On Single Field Indexing:**

Collection Name: BlackFriday

*Query Without Indexing*:

**db.getCollection('BlackFriday').find({"User\_ID" : 1000319})**

When the above query is executed, the cost in repeated executions is as follows:

0.217, 0.118, 0.12, 0.138, 0.125, ……………..

The average of the above executed costs = 0.143

*Query With Indexing:*

Create index using the following query

**db.getCollection('BlackFriday').createIndex({"User\_ID" : 1})**

After creating the index in User\_ID field, execute the same query.

**db.getCollection('BlackFriday').find({"User\_ID" : 1000319})**

When the above query is executed, the cost in repeated executions is as follows:

0.006, 0.002, 0.004, 0.003, 0.002, …………

The average of the above executed costs = 0.004

We can clearly notice a huge difference between the average cost of the same query with indexing and without indexing. Hence the single field indexing works.

**Experiment On Compound Index:**

Collection Name: BlackFriday

*Query Without Indexing:*

**db.BlackFriday.find({"Product\_ID" : "P00361942","Purchase" : 4229})**

When the above query is executed, the cost in repeated executions is as follows:

3.14, 2.84, 3.37, 4.59, 3.75, …………………

The average of the above executed costs = 3.538

*Query With Indexing:*

Create index using the following query

**db.BlackFriday.createIndex({"Product\_ID" : 1,"Purchase" : 1})**

After creating the index in User\_ID field, execute the same query.

**db.BlackFriday.find({"Product\_ID" : "P00361942","Purchase" : 4229})**

When the above query is executed, the cost in repeated executions is as follows:

0.003, 0.001, 0.001, 0.001, 0.001, …………

The average of the above executed costs = 0.0014

The order of the fields in compound index is very important. The order should match the order of the fields in the query that is being used.

**Prefix of Compound Index:**

**db.BlackFriday.createIndex({"Product\_ID" :1,"Purchase" : 1})**

In the above query “Product\_ID” is called as the prefix. This prefix does not need to have a separate index. The prefix of compound index acts as an index which automatically reduces the cost of an operation.

*Query Execution With The Prefix Of Compound Index:*

**db.getCollection('BlackFriday').find({"Product\_ID" : "P00361942"})**

Though there is no index created on Product\_ID, the above query returns results in very less time as follows:

0.003, 0.003, 0.004, 0.003, 0.003, ..........................

The average of the above executed costs = 0.00032

The average cost of the same query would be more then the above value if it is not the prefix of the compound index.

**References:**

Text Book: Database systems concepts (by Abraham Silberschartz)

Website**:**<https://docs.mongodb.com/>

**Technologies:** MongoDB

**Tools:**

Robo 3T – 1.2 (for experiments)