



Project 5: MongoDB Indexes

Team 12: Jackson Powell and Caleb Ralphs



The problem

Queries on large datasets are inefficient as they require full table scans in order to sort through all of the records in the tables or collections. Although query and some other optimization methods reduce this computation overhead, the nature of the full table scan is still computationally expensive.

Solution

Indexes allow queries to optimally organize the data with an additional data structure, allowing the query to return all of the pertinent data corresponding to the query conditions without performing a full table scan.



Importance

- Queries on large databases without indexes require full table/collection scans, not capitalizing on the information contained in some of the fields of each document
- In MongoDB, indexes can limit the number of documents required for inspection for a given query
- For sparse data, indexes allow the query to ignore irrelevant documents not containing values for the indexed field, drastically reducing query computation time



MongoDB



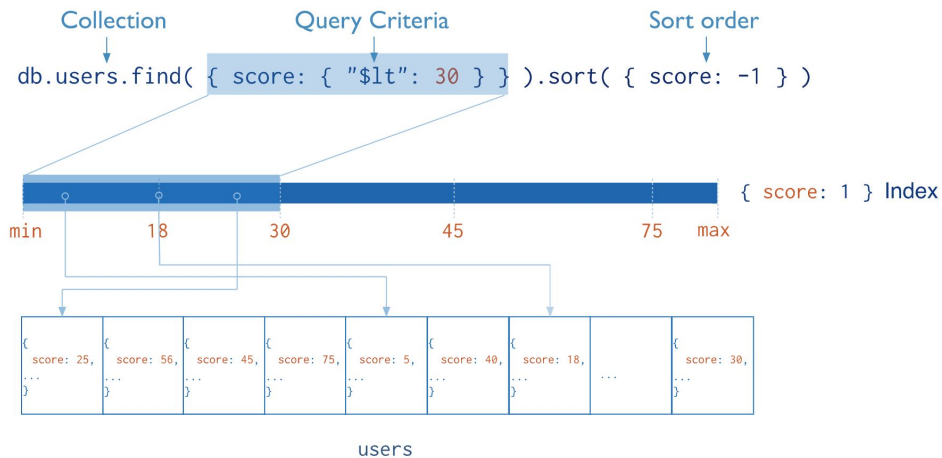
- MongoDB is a NoSQL document oriented database system
- Documents are similar to JSON objects where the values of the fields may include arrays, arrays of other documents, or other documents themselves
- Supports embedded data models to reduce I/O activity on the system

```
{  
  name: "sue",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```

←	field: value
←	field: value
←	field: value
←	field: value

MongoDB Indexes

- Leverages indexes to optimize queries, allowing for inclusion of keys from embedded documents to be used as indexes
- MongoDB indexes store a subset of the collection's data in a B-tree data structure, allowing for easy traversal of the documents
- MongoDB indexes stored values of specified fields or sets of fields
- Indexes are defined at the collection level





Naive Solution

{	"_id"	:	1,	"item"	:	"f1",	type:	"food",	quantity:	500 }
{	"_id"	:	2,	"item"	:	"f2",	type:	"food",	quantity:	100 }
{	"_id"	:	3,	"item"	:	"p1",	type:	"paper",	quantity:	200 }
{	"_id"	:	4,	"item"	:	"p2",	type:	"paper",	quantity:	150 }
{	"_id"	:	5,	"item"	:	"f3",	type:	"food",	quantity:	300 }
{	"_id"	:	6,	"item"	:	"t1",	type:	"toys",	quantity:	500 }
{	"_id"	:	7,	"item"	:	"a1",	type:	"apparel",	quantity:	250 }
{	"_id"	:	8,	"item"	:	"a2",	type:	"apparel",	quantity:	400 }
{	"_id"	:	9,	"item"	:	"t2",	type:	"toys",	quantity:	50 }
{	"_id"	:	10,	"item"	:	"f4",	type:	"food",	quantity:	75 }

Query with no index

```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } )
```

Scans all 10 docs, returning docs 2,3, and 4

Query with an index

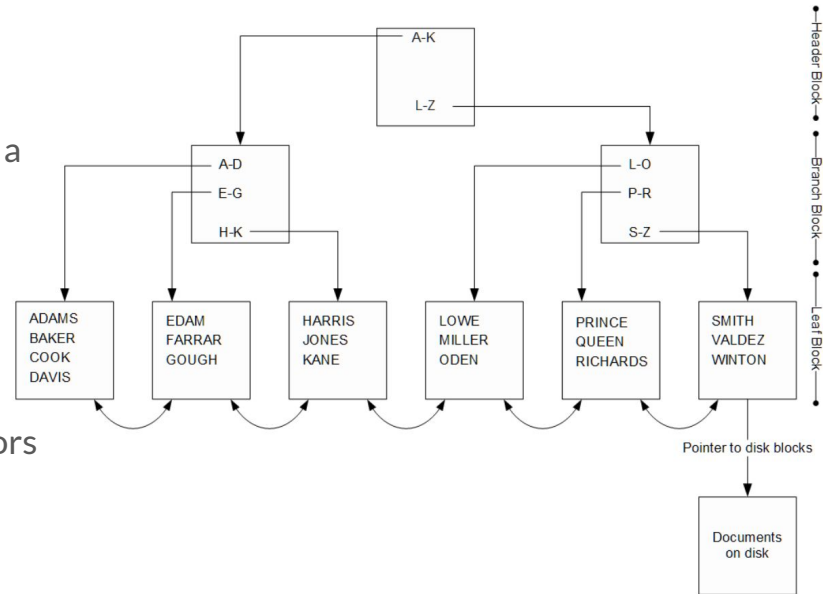
```
db.inventory.createIndex( { quantity: 1 } )
```

```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } )
```

Scans only 3 docs since the index is no longer defaulted to the `_id`, but instead is “quantity”, returning the same 3 docs: 2, 3, and 4

B-tree Indexes

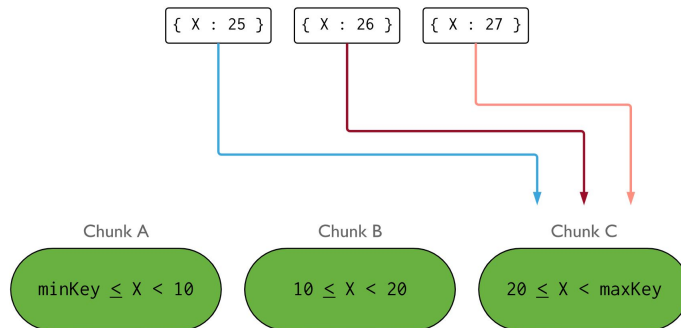
- B-trees are data structures that keep data sorted in a tree format, allowing for access, insertions, and deletions in logarithmic time
- MongoDB, CouchDB, and RDBMS (SQL DB) all use B-Trees as underlying data structure for indexes
- Effective in range queries, such as $\$gt$ and $\$lt$ operators
- Ineffective in inequality queries as each data node requires access, regardless of index value
- Overhead required for inserting, updating, and deleting records



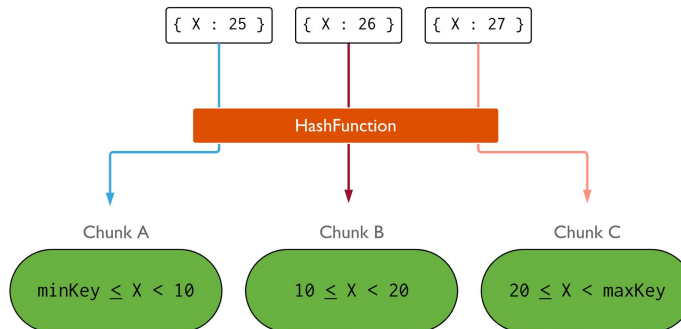
Hash Indexes

- Method for supporting hashed sharding
- Hashed sharding provides more even, random distributions than traditional indexing
- Reduces amount of unbalanced sharding
- Without sharding, data chunks exceeding maximum chunk size may exist
- Without sharding, there may exist empty data chunks, resulting in decreased performance
- Hashed indexes usually require less storage overhead than traditional indexes

With Traditional Indexes



With Hashed Indexes





RDBMS vs MongoDB Indexes

Similarities

- Underlying index data structure is B-tree
- Leverages query optimizer at execution time
- User can specify index column (same as field in MongoDB documents)

Differences

- By default, the RDBMS creates a clustered index on all primary keys in the table, while MongoDB creates a default index on the `_id` field in each document
- RDBMS allows for clustered indexing, which MongoDB does not support because it uses file mapping to manage interactions with data in collections



CouchDB vs MongoDB

CouchDB is another NoSQL document oriented database system

Similarities

- CouchDB also uses B-tree as underlying data structure for indexes
- CouchDB uses views in a way very similar to how indexes are used in RDBMS databases

Differences

- CouchDB uses CRUD API with support for repeated queries on aggregated data using MapReduce framework
- CouchDB only allows for indexing on the `_id` field, whereas MongoDB allows user to specify the index or for the query to select the optimal index



Simplifying Assumptions

To narrow the scope of the project, we do not test or experiment with the following aspects of the MongoDB indexing framework:

- Overhead of MongoDB indexes beyond storage cost, such as time cost of maintaining indexes upon document insertion, update, or deletion
- Unique, sparse, partial, case sensitive, and TTL indexes
- All combinations of index field types for multikey indexes (we only test one multikey index pair)
- Explicit testing of index data caching on query performance
- Tests with multiple processing power and storage conditions (all tests run on same local machine)
- Performance tests on hashed sharding and hashed indexes on shard keys



Tested Indexes


MongoDB index types:

- Single Field - Single field, can be top-level, subfield, or subdocument
- Compound - More than one field
- Multikey - Array field
- Geospatial - GeoJSON or legacy coordinate pairs
- Text - Strings
- Hashed - Hashed values



Experimental Approach

1. Create 3 collections of varying size
2. Create all indexes on the fields of each collection
 - a. `db.collection.createIndex({field : index type or sort direction})`
3. Test the performance of queries using each index
 - a. Hide all indexes to provide a baseline performance
(`db.collection.hideIndex(index name)`)
 - b. Unhide one index to test the query performance with the index
(`db.collection.unhideIndex(index name)`)
 - c. Run a query with and without the index 3 times and report the median performance time.
 - i. Use `cursor.explain("executionStats")` for simple queries and `Data.now()` to time aggregation and recursive queries



Data and Index Generation

Generated documents of the form

```
{
  _id : ObjectId,
  Name : String,
  Home : GeoJSON Point,
  Salary : Number
  Birthday : {Year : number,
              Month : number,
              Day : number},
  Friends: String array,
  Favorite locals : GeoJSON Point array
}
```

Generated 3 collections

Collection	# of documents	Size [Mb]
People1	1 million	417
People10	10 million	4168
People250k	250 thousand	104

Generated indexes of each type

- Single Field - Salary
- Compound - Salary and birthday
- Multikey - Friends, FavoriteLocals
- Geospatial - Home (2dsphere)
- Text - \$** (wildcard - indexes all string values)
- Hashed - Name



Storage Overhead

Collection	_id	Salary	Salary and birthday	Home	Friends	Favorite Locals	Name	Text
People1	9	8	28	18	9	82	18	41
People10	96	75	200	173	86	810	176	406
People250k	2	2	8	4	2	20	4	10

Sizes are shown in megabytes.



Tested Queries

Salary (Simple)

- Range
- Range and sort (1)
- Range and sort (-1)
- Covered range
- Equality
- Inequality

Home (Geospatial)

- \$geoWithin
- \$near
- \$nearSphere

Salary and birthday (Compound)

- Salary equality
- Birthday equality
- Salary and birthday equality
- Salary range
- Salary range and sort (1)

FavoriteLocals (Multikey)

- \$elemMatch
- Value match

Friends (Multikey)

- Value match
- Empty array
- Length 4 array
- Exact array match
- Membership query

Text

- \$text search
- Sort by textScore
- Aggregate after filtering using \$text search

Name (Hashed)

- Equality
- Range
- Inequality
- Aggregation
- Recursive



Performance Results

Simple:

- Inequality and range chose to use index and took longer
- Sorting, equality, and covered queries all faster with index.

Compound:

- Birthday query did not use index
- All other queries were faster with index (4, 4, 1, 3.5 seconds on people10 respectively)

Hashed:

- Equality faster with index (4 s on people10)
- All other queries did not use the index

Text:

- \$text operator requires the use of a text index

Multikey:

- No favoriteLocals query used the index
- The size of the array cannot be accessed using the index
- Friends improved queries 1, 4, and 5 by 5.5, 1.5, and 5.5 seconds on people10

Geospatial:

- \$geoWithin
 - Index: 1 ms
 - No index: 14039 ms
- \$near and \$nearSphere operators requires the use of a geospatial index



References

- <https://docs.mongodb.com/manual/introduction/>
- <https://www.mongodb.com/>
- <https://docs.mongodb.com/manual/indexes/>
- <https://zhangliyong.github.io/posts/2014/02/19/mongodb-index-internals.html>
- <https://dzone.com/articles/effective-mongodb-indexing-part-1>
- <https://docs.mongodb.com/manual/core/hashed-sharding/>