Indexing

Outline

- Why indexes are important
  - Index all the things
  - Sure it's indexed? Explain…
- Self-imposed limitations
  - Cool story about index keys
  - Miscellaneous notes
- B-tree, LSM + Bloom filter
- A look at the competition
- Index types
  - Single-field, compound, multikey, geospatial, text, hashed, TTL
- Index tiotion

Recap

- Single field index
- Compound index
- Multikey indexes
- Hashed indexes
- Covered queries
- Indexes impact on read and write performance

Issues with multikey indexes

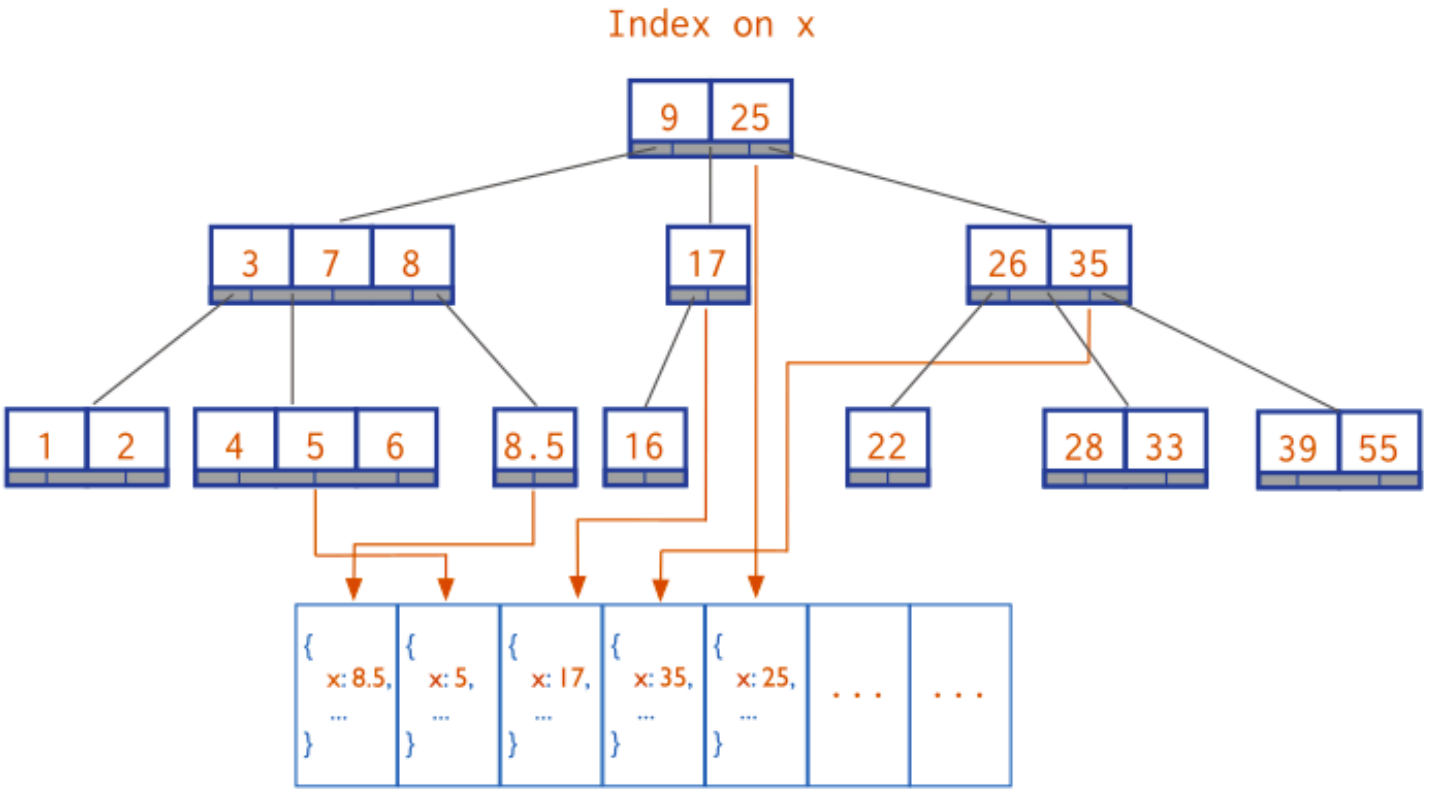
Limitations on Multikey Indexes

- You cannot create a compound index using more than one array-valued field.
- This is because of the combinatorics.
- For a compound index on two array-valued fields you would end up with N * M entries for one document.
- You cannot have a hashed multikey index.
- You cannot have a shard key use a multikey index
- We discuss shard keys in another module.
- The index on the _id field cannot become a multikey index.

What is a Hashed Index?

- Hashed indexes are based on field values like any other index.
- The difference is that the values are hashed and it is the hashed value that is indexed.
- The hashing function collapses sub-documents and computes the hash for the entire value.
- MongoDB can use the hashed index to support equality queries.
- Hashed indexes do not support multi-key indexes, i.e. indexes on array fields.
- Nor do they support range queries.

Why Indexes?


Index on x
9
25
3
7
8
17
26
35
1
2
4
5
6
8.5
16
22
28
33
39
55
{ x: 8.5, ... }
{ x: 5, ... }
{ x: 17, ... }
{ x: 35, ... }
{ x: 25, ... }
. . .
. . .

Why indexes are important

It is a common misconception that MongoDB magically indexes everything, or that it's so fast it doesn't need indexes ;)!

In fact, there is only one default [unique] index per [non-capped] collection called *_id*.

⅔ of all performance problems are due to a missing or incorrect secondary index.

No secondary index ⇒ collection/table scan

Table scans are SLOW and expensive since our storage layer is not optimized for them. They could work fine for a few documents, but performance will degrade over time.

As a general rule of thumb, each index cuts write throughput.

Even seemingly trivial queries move the working set out of memory, so index all the things.

Index all the things

- Limit to 64 indexes per collection (this is an issue with MMAPv1, the limit can conceivably be raised with WiredTiger).
- **Realistically, if you have 64 indexes on a collection, you're doing something wrong.**
- Indexes slow writes as all relevant indexes have to be updated. Can use compound indexes to service multiple queries.
- It is rare to see more than 10 to 12 indexes per application in practice.
- Richard Kreuter claims to have seen a justified case for using 37 indexes in the field.

Are you sure it's indexed?

- Check the logs!
  - Queries > 100ms (slowOpThresholdMs) are recorded by default, even if profiling is disabled
- Use cursor.explain()
  - Runs the query and outputs a document with query plans, execution stats…

Table Scan

```
trexycle(mongod-2.6.7) support> db.issues.find({'jira.fields.project.key':
'CS'}).explain()
{
  "cursor": "BasicCursor",
  "filterSet": false,
  "indexOnly": false,
  "isMultiKey": false,
  "millis": 50018,
  "n": 18617,
  "nChunkSkips": 0,
  "nYields": 58955,
  "nscanned": 143069,
  …
}
```

BasicCursor indicates a table scan, bad!

Number of documents matched (on all criteria specified)

Number of items (documents or index entries) examined

For an effective index, the n / nscanned ratio should be close to 1.

Indexed

```
trexycle(mongod-2.6.7) support> db.issues.find({'jira.fields.project.key': 'CS'}).explain()
{
  "cursor": "BtreeCursor jira.fields.project.key_1",
  "filterSet": false,
  "indexBounds": {
    "jira.fields.project.key": [
      [
        "CS",
        "CS"
      ]
    ]
  },
  "indexOnly": false,
  "isMultiKey": false,
  "millis": 31,
  "n": 18617,
  "nChunkSkips": 0,
  "nYields": 145,
  "nscanned": 18617,
  ...
}
```

BtreeCursor means an index was used!

Number of documents matched (on all criteria specified)

Number of items (documents or index entries) examined

n / nscanned = 1

Using `explain()` for Write Operations

Simulate number of writes that would have occurred, and determine the index(es) used:

```
> db.tweets.explain("executionStats").remove({ "user.followers_count" : 1000
```

```
> db.tweets.explain("executionStats").update({ "user.followers_count" : 1000
  { $set : { "large_following" : true } } )
```

Example with explain()

General Rules of Thumb

- Equality before range.
- Equality before sorting.
- Sorting before range.

Self-imposed limitations

- Index keys, i.e. the values of the indexed fields, must be smaller than 1024 bytes (before 2.0, it was 819 bytes)
  - Compound index keys are a concatenation of field values and can approach this limit
  - Long strings also, e.g. URIs (see next slide for a dark history on the matter)

- Namespace, i.e. the database name + the collection/index name, and hence the index name itself, must be smaller than 123 bytes
  - The default compound index name is a concatenation of field names and can approach this limit

- Compound indexes must have fewer than 32 fields
  - There is consequently a competing dynamic between the compound index fields limit and the index name size limit

More self-imposed limits: http://docs.mongodb.org/manual/reference/limits/

Cool story about index keys

- Since 2.6, inserting a document with an indexed string field value >1024 bytes [of UTF-8 encoded data] throws an exception.
- Before 2.6, we accepted the document but did not create an index entry for it. Oops! Hey, at least we logged the incident.
- When upgrading to 2.6, the offending documents and indexes were left in place. **Only when trying to rebuild the indexes (a common practice among DBAs) and the builds failed did users learn of the offending documents!**
- This resulted in a class of pathological errors from different nodes having slightly different indexes, which upset customers.

Index key drama continued

What if we truncated values that are too long, and referred to the corresponding documents when queries match multiple index keys?

That is a valid idea, but it cannot be done with the existing indexing code. Perhaps in a future release.

Miscellaneous notes

- MongoDB query optimizer is empirical, and sometimes wrong! Use cursor.hint() to force an index

- Enhanced stats available in 2.6 with mongod-level switch: http://docs.mongodb.org/manual/reference/command/indexStats/

B-trees

The B does not stand for binary, these are not binary trees!

They belong to a family of tree structures (B+, B*, …) whose nodes can have more than two children.

Intentionally not self balancing, otherwise all our time would be spent balancing them #disksareslow #diskio!

Instead, make tradeoffs between fanout and depth, i.e. between time to search and amount of disk IO (each disk access takes ~3ms!)

See Wikipedia for details: http://en.wikipedia.org/wiki/B-tree

B-trees – what ours look like

Starts with one block (bucket), 8176 bytes in length.

The block contains index keys.

The index keys point directly to the documents on disk (file, offset).

This is every index in MMAPv1.

B-trees continued

When the block fills, it is split in two. The index keys are split 50-50 between the two blocks.

Leads to fragmentation, i.e. wasted space. The amount depends on the randomness of entries.

An index is a triangle

Random entries fall randomly within the triangle, i.e. in random B-tree buckets.

➔ 50-50 splits are your best bet for random entries

Can we do better if we know something about the data?

Consider the ever-increasing **ObjectId**s inserted into _id, which always go to the bottom right, i.e. the rightmost bucket.

➔ 50-50 splits result in maximal fragmentation, and hence a larger tree in memory and on disk

B-trees continued

To minimize fragmentation, we use a 90-10 split for the rightmost node of the tree, and 50-50 splits otherwise.

If the inserts trend upward, you end up with a minimally fragmented *write balanced* B-tree

➔ More tree fits into memory, fewer disk accesses!

This optimization was added at the behest of Foursquare, and determined to be a generally good thing.

B-trees continued

Can further reduce fragmentation by reindexing or compacting, but these are expensive operations that take a database-level write lock.

Nice data structures because they align with how data is stored in the filesystem, i.e. in blocks.

The index will grow until the write throughput on flush to disk is the bottleneck (we flush once a minute). This is not a MongoDB problem, it affects all DBs.

When document is moved on disk, each index entry for it needs to be repointed to the new location. In practice this is very bad when there are lots of index entries, so please mind those large embedded arrays!

LSM tree + Bloom filter

- New in MongoDB 3.0 with WiredTiger
- Set a maximum size for a B-tree. Once limit reached, create a new tree
  => Good for write heavy workloads.
- Writes to multiple files (trees) that are merged in the background.
- Reads search multiple trees using Bloom filters to narrow down lookups.
- Bloom filter is a probabilistic hash table that doesn't provide false negatives, but can provide false positives. It's an LSM optimization.
- The in-memory and on-disk format data are different and unrelated.

A look at the competition

Our indexes are relatively vanilla implementations of the B-tree algorithm (variable length index keys)

“Other leading brands are databases that wear big boy pants.” – Richard Kreuter

Most other [relational] DBs use B+ trees which have other optimizations Wikipedia can tell you about.

TokuMX uses fractal B-tree…

The Order of Fields Matters

Specifically we want to consider how the index will be used for:

- Equality tests, e.g.,

```
db.example.find( { a : 15, b : 17 } )
```

- Range queries, e.g.,

```
db.example.find( { a : 15, b : { $lt : 85 } } )
```

- Sorting, e.g.,

```
db.example.find( { a : 15, b : 17 } ).sort( { b : -1 } )
```

Designing Compound Indexes

- Let's look at some guiding principles for building compound indexes.
- These will generally produce a good if not optimal index.
- You can optimize after a little experimentation.
- We will explore this in the context of a running example.

Example: A Simple Message Board

Requirements:

- Find all messages in a specified timestamp range.
- Select for whether the messages are anonymous or not.
- Sort by rating from highest to lowest.

Load the Data

```
a = [ { "timestamp" : 1, "username" : "anonymous", "rating" : 3 },
      { "timestamp" : 2, "username" : "anonymous", "rating" : 5 },
      { "timestamp" : 3, "username" : "sam", "rating" : 1 },
      { "timestamp" : 4, "username" : "anonymous", "rating" : 2 },
      { "timestamp" : 5, "username" : "martha", "rating" : 5 } ]
db.messages.insert(a)
```

Start with a Simple Index

Start by building an index on { timestamp : 1 }

```
db.messages.ensureIndex( { timestamp : 1 }, { name : "myindex" } )
```

Now let's query for messages with timestamp in the range 2 through 4 inclusive.

```
db.messages.find( { timestamp : { $gte : 2, $lte : 4 } } ).explain()
```

Analysis:

- Explain plan shows good performance, i.e. `nscanned` = `n`.
- However, this does not satisfy our query.
- Need to query again with { username : "anonymous" } as part of the query.

Query Adding `username`

Let's add the `user` field to our query.

```
db.messages.find( { timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous" } ).explain()
```

`explain` shows:

- `nscanned` > `n`.

Include `username` in Our Index

```
db.messages.dropIndex( "myindex" );
db.messages.ensureIndex( { timestamp : 1, username : 1 },
                         { name : "myindex" } )
db.messages.find( { timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous" } ).explain()
```

`explain` shows:

- `nscanned` is still greater than n.
- Why?

ncanned > n

timestamp	username
1	"anonymous"
2	"anonymous"
3	"sam"
4	"anonymous"
5	"martha"

A Different Compound Index

Drop the index and build a new one with user.

```
db.messages.dropIndex( "myindex" );
db.messages.ensureIndex( { username : 1 , timestamp : 1 },
                         { name : "myindex" } )

db.messages.find( { timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous" } ).explain()
```

`explain` shows:

- nscanned is 2
- n is 2

nscanned == n == 2

username	timestamp
"anonymous"	1
"anonymous"	2
"anonymous"	4
"sam"	2
"martha"	5

Let Selectivity Drive Field Order

- Order fields in a compound index from most selective to least selective.
- Usually, this means equality fields before range fields.
- When dealing with multiple equality values, start with the most selective.
- If a common range query is more selective instead (rare), specify the range component first.

Adding in the Sort

Finally, let's add the sort and run the query.

```
db.messages.find( {
                    timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous"
                  } ).sort( { rating : -1 } ).explain();
```

- Note that the `scanAndOrder` field is set to true.
- This means that MongoDB had to perform a sort in memory.
- In memory sorts for queries that retrieve large numbers of documents can degrade performance significantly.
- Especially, if they are used frequently.


The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

Avoiding an In-Memory Sort

Rebuild the index as follows.

```
db.messages.dropIndex( "myindex" );
db.messages.ensureIndex( { username : 1, rating : 1, timestamp : 1 },
                         { name : "myindex" } );
db.messages.find( {
                    timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous"
                  } ).sort( { rating : -1 } ).explain();
```

- Tradeoff between `nscanned` and `scanAndOrder`
- `nscanned` is 3 and and `n` is 2.
- Best we can do in this case and in this situation is fine.
- However, if `nscanned` is much larger than `n`, this might not be the best index.

Multikey Indexes and Sorting

- If you sort using a multikey index:
  - A document will appear at the first position where a value would place the document.
  - It does not appear multiple times.
- This applies to array values generally.
- It is not a specific property of multikey indexes.

Exercise: Multikey Indexes and Sorting

```
db.testcol.drop()
a = [ { x : [ 1, 11 ] }, { x : [ 2, 10 ] }, { x : [ 3 ] },
      { x : [ 4 ] }, { x : [ 5 ] } ]
db.testcol.insert(a)

db.testcol.ensureIndex( { x : 1 } )

// x : [ 1, 11 ] array comes first. It contains the lowest value.
db.testcol.find().sort( { x : 1 } )

// x : [ 1, 11 ] array still comes first. Contains the highest value.
db.testcol.find().sort( { x : -1 } )
```

Floating Point Numbers

- MongoDB hashed indexes truncate floating point numbers to 64-bit integers before hashing.
- Do not use a hashed index for floating point numbers that cannot be reliably converted to 64-bit integers.
- MongoDB hashed indexes do not support floating point values larger than 2^{53}.

Index intersection

- You are usually better off using a proper compound index.
- Would only be selected if:
  - All predicates return large number of documents, but the intersection is small
  - Most documents are on disk, i.e. not in RAM
- To use one index for the find() clause and one for the sort(), the sort fields must also be in the query.

Building Indexes in Production

Example

Example 2

References

- http://docs.mongodb.org/manual/core/indexes/