Indexing in MongoDB

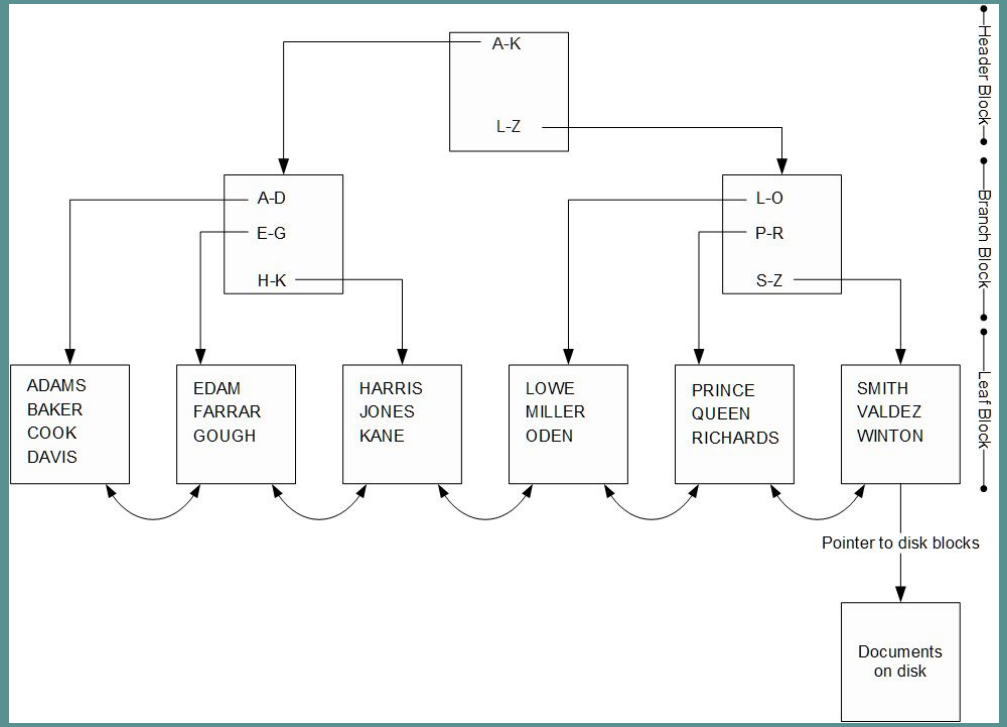
CS585

Joseph Caltabiano
Shine Linn Thant

Indexes

- An object with storage
- Enhance performance
- Efficient if index is selective (e.g. gender vs. dob)
- Default index on _id field
- Can be on fields of any data type

Structure of an Index (B-Tree Indexing)



Advantages & Overheads

- All nodes are at the same level
- Fast lookup
- Expensive for insert
- Allocate new block and shift

References

- https://docs.mongodb.com/manual/core/index-compound/#index-ascending-and-descending
- https://docs.mongodb.com/manual/tutorial/sort-results-with-indexes/#sort-on-multiple-fields
- https://docs.mongodb.com/manual/applications/indexes/
- https://dzone.com/articles/effective-mongodb-indexing-part-1

Types of Indexes

- Single Field Index
```
db.records.createIndex( { score : 1 } )
```
- Unique Index
```
db.products.createIndex( {  "item": 1, "stock": 1 } )
```
- Compound Index
- Multi-key Index
- Hash Index
- 2D Index

Experimental Plan

- Three different-sized datasets
- name, phone, email, gender, salary
- Queries using different indices
- Compare performance
- find(), aggregate($group)

Execution

- no index
- single field index
- one indexed field and one unindexed field
- compound index on both fields
- a compound index on three fields but only use first and last fields

Challenges and caveats

- Getting the execution time of a query is hard
  - Highly inconsistent
  - Different results from shell vs Node.js
  - .aggregate() has no native support for execution time

Statistic collection

```
let res = await collection
  .find({ firstname: "Brady" })
  .explain("executionStats");
```

```
function time(cmd) {
    const before = new Date();
    const res = cmd();
    const after = new Date();
    print(`execution time:\n${cmd}\n${after-before} ms`);
    return res;
}
```

```
time(() => db.large.aggregate({
  '$group': {
    '_id': {
      'gender': '$gender'
      'firstname': '$firstname'
    }
  }
}))
```

Single-field index

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
find({name:x)	none	607	14440	44238
	name	3917	49145	128354
groupby(name)	none	1210	11697	24720
	name	81	10455	24986

Compound index with find

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
find({name:x, phone:y})	name	1	12	123
	name, phone	5619	72935	171310
	name, email, phone	6468	86153	271421
	name, gender	5579	73145	226728

Index cardinality

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
find({name:x, gender:y})	name	1	10	26
	gender	2917	27996	157612
find({gender:y, name:x})	gender, name	8538	71291	172200

Results

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
groupyby({gender:y, name:x})	name	1374	15344	41631
	gender, name	1679	16431	42219
	name, gender	1531	17418	43199
	name: 1, gender: -1	1500	18719	43391
	name: -1, gender: 1	1611	17085	49344

Storage Overheads

Index	**1M**(MB)	**10M**(MB)	**21M**(MB)
name	6	65.6	147.5
name, phone	25.8	202.3	285.2
name, email, phone	53.8	393.6	449.9
gender	4.6	50.3	113
gender, name	6.4	69.6	156.4
name, gender	6.4	69.6	156.4
name:1, gender:-1	6.4	69.6	156.4
name:-1, gender:1	6.4	69.6	156.4

Conclusions

- Choose your indexes carefully
  - Be mindful of cardinality
- Use the most appropriate index for your query
- Write queries to leverage indexes
- Indexes allow for significant reduction in resource use and time at scale