Indexing in MongoDB

Tova Just

Overview

1. Creating indexes
2. Compound indexes
3. Find() and Sort()
4. Best Practices
5. Inefficiencies and Drawbacks

Creating an Index in MongoDB

```
> db.users.createindex({"username":1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1, "numIndexesAfter" : 2,
    "ok":1
}
```

- should take no longer than a few seconds

Which fields to index?

- Previous example: indexed on “username”
- Consider what queries are made often and need to be fast
- Find a common set of keys
- Index on that set of keys

Compound Indexes

- Built on two or more keys
- Why?
  - useful if your query has multiple sort directions or multiple keys in the criteria

- For example: sorting
- `db.users.find().sort({"age":1,"username":1})`
  - Sorts by “age” and then by “username,” so an index on just “username” is not very helpful
- Instead, make an index on age AND username:
- `db.users.createindex({"age":1,"username":1})`

More Indexing Syntax

- Indexing in different directions:
- `{"age" : 1, "username" : -1}` creates an index with age ascending and username descending
  - Equivalent to `{"age" : -1, "username" : 1}`
  - For single-key indexes, this doesn’t matter; Mongo just reads the index backwards

- Indexing Embedded Docs:
  - `db.users.createindex({"loc.city":1})`
- If index the entire subdocument, only helpful if querying entire subdoc in order, ie `db.users.find({"loc" : {"ip" : "123.456. 789.000", "city" : "Shelbyville", "state" : "NY"}}})`

```
{
  "username" : "sid",
  "loc":{
    "ip" : "1.2.3.4",
    "city" : "Springfield",
    "state" : "NY"
  }
}
```

Find() and Sort() using Indexes

- In general, if an index is used for a query, the resulting documents are returned in index order
- However, can specify a different sorting order:
- `db.users.find({"age" : {"$gte" : 21, "$lte" : 30}})`
  `.sort({"username" : 1})`
  - MongoDB will use the index to match the criteria
  - BUT: the index doesn't return the usernames in sorted order
  - MongoDB needs to sort the results in memory before returning them
  - Usually less efficient
- If you have more than 32 MB of results, MongoDB will show an error
  - Fix this by creating an index to support the sort
- .explain(“executionStats”) lets you see the internal workings of Mongo executing a query

General Best Practices

- If you can **avoid in-memory sort** with a better index design, you should
- Order of keys in index: **[equality_filter, sort_filter, multivalue_filter]**
    - For the query: `db.students.find({student_id:{$gt:500000}, class_id:54}).sort({final_grade:1})`
    - Make this index: `db.students.createindex({class_id:1,final_grade:1,student_id:1})`
    - Examines keys for more documents than end up being in the result set, but we save execution time because the index ensures we have sorted documents (no in-memory sort)
- *Index Cardinality*: how many distinct values there are for a field in a collection
- **Create indexes on high-cardinality keys**, or **put high-cardinality keys first** in compound indexes

How MongoDB selects an index

- Query comes in
- MongoDB looks at query’s *shape* (what fields are being searched on, whether or not there is a sort, etc.)
- A set of candidate indexes are identified
- In parallel threads, one query plan is run for each candidate index
- The plans are raced against each other for a trial period
- The first query plan to reach a goal state is the winner
- Going forward, it will be selected as the index to use for queries that have that same query shape

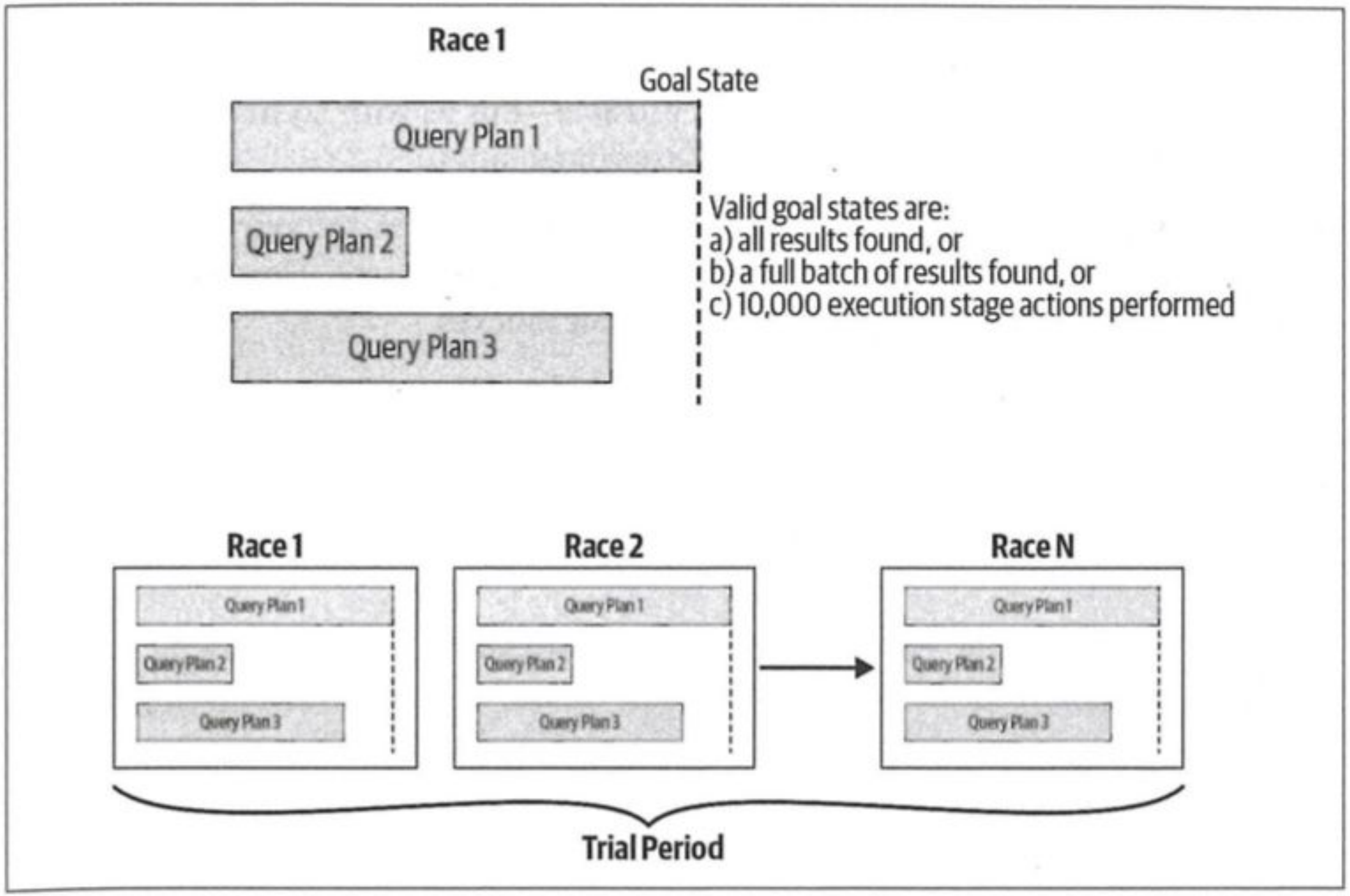

Figure 5-1. How the MongoDB Query Planner selects an index, visualized as a race

The Price of Indexing

- An index can make queries much faster
- However:
  - When your data changes, MongoDB has to update indexes (in addition to updating the document)
  - Write operations (inserts, updates, and deletes) that modify an indexed field will take longer
  - Typically, the tradeoff is worth it

Inefficient Queries

- Some queries can use indexes more efficiently than others
- Inefficient
  - Negation: $ne, $not, $nin
  - Try to find another clause using an index to add to the query, so that nonindexed matching is performed on a smaller set
- $or: two separate queries are performed and then merged
  - Merging: needs to look through results of both queries and remove duplicates
  - Less efficient than a single query, so use $in instead

When (and when not) to use indexing

Indexes work well for:

- Large collections
- Large documents
- Selective queries

Indexing is inefficient (Collection scans work better):

- Small collections
- Small documents
- Nonselective queries

Why?

Using an index requires two lookups: (1) to look at the index entry, and (2) following the index's pointer to the document. Collection scan only requires one: looking at the document.
In the *worst case* (**returning all of the documents in a collection**) using an index would take **twice as many lookups** and would be significantly slower than a collection scan.

Resources

1. Bradshaw, Brazil and Chodorow. “Chapter 5: Indexes.” *MongoDB: The definitive guide*. 3rd ed., O'Reilly Media, 2019.