Database Query Optimization

What is Database Query Optimization?

- Improving the efficiency and performance of database queries.

- The goal is to minimize the amount of time and system resources required to execute a query

- Importance:
  - Faster Query Execution = Short response time & Improved User Experience
  - Decrease strain on system resource by minimizing disk I/O, Memory Consumption etc
  - Handle large number of workloads and scale

Query Optimization in MongoDB

Query Execution Process

1. Query Parsing
2. Query Optimization
3. Query Execution
4. Result Transmission

Query Execution Process in MongoDB

1. Query Parsing
   - When a query is received, it is parsed to understand the structure and semantics.
   - Checked against the collection’s schema to ensure its validity (valid references, etc)

2. Query Optimization
   - Query optimizer analyzes the query and determines the most efficient query plan to minimize the number of documents examined
   - Takes available indexes, query predicates, sort orders etc
   - Cost of using indexes is estimated and most suitable index(s) are selected

Query Execution Process in MongoDB

3. Query Execution:

   - Based on the optimal plan, MongoDB executes the query.
   - Locate matching documents based on query filtering, projection etc

4. Result Transmission:

   - Data is read from disk/memory and returned

Diagnosing Slow Queries

Analyze queries using the EXPLAIN helper

- `db.collection.<method>.explain(verbosity)`

Verbosity Modes:

- `"queryPlanner"` (Default) - Run the query optimizer to choose the winning plan and return the details without executing it
- `"executionStats"` - Choose and execute the winning plan and returns the statistics
- `"allPlansExecution"` - Choose and execute the winning plan and return the stats along with all other candidate plans evaluated during the plan selection process

```
Example - db.users.find({name: "Jane"}).explain()
```

MongoDB Query Optimization Techniques

- Proper Data Modelling
- Indexing
- Covered Queries
- Optimizing Aggregation Pipelines

1. Proper Data Modelling

- When to embed a document or create a reference between separate documents
- Embedding
  - 1:1 relationship can be imbedded in the same document,
  - 1:many where the data are always accessed together (“many” are viewed in the context of their parent)
  - Might complicate data updates (update for each), stores redundant data (Trade write performance for read performance)

- Referencing
  - Preferable for many to many relations, Reduced document size
  - Easier updates, independent querying
  - Resolving references, round trips to the server can come at a cost

1. Proper Data Modelling

```
// Blog Document
{
  _id: ObjectId("post_id"),
  title: "My Blog Post",
  content: "This is my blog content.",
  comments: [
    ObjectId("comment_id1"),
    ObjectId("comment_id2")
  ]
}

// Comment Documents
{
  _id: ObjectId("comment_id1"),
  author: "User1",
  text: "Great post!",
},
{
  _id: ObjectId("comment_id2"),
  author: "User2",
  text: "I enjoyed reading this.",
}
```

```
{
  _id: ObjectId("post_id"),
  title: "My Blog Post",
  content: "This is my blog content.",
  comments: [
    {
      _id: ObjectId("comment_id1"),
      author: "User1",
      text: "Great post!",
    },
    {
      _id: ObjectId("comment_id2"),
      author: "User2",
      text: "I enjoyed reading this.",
    }
  ]
}
```

2. Indexing

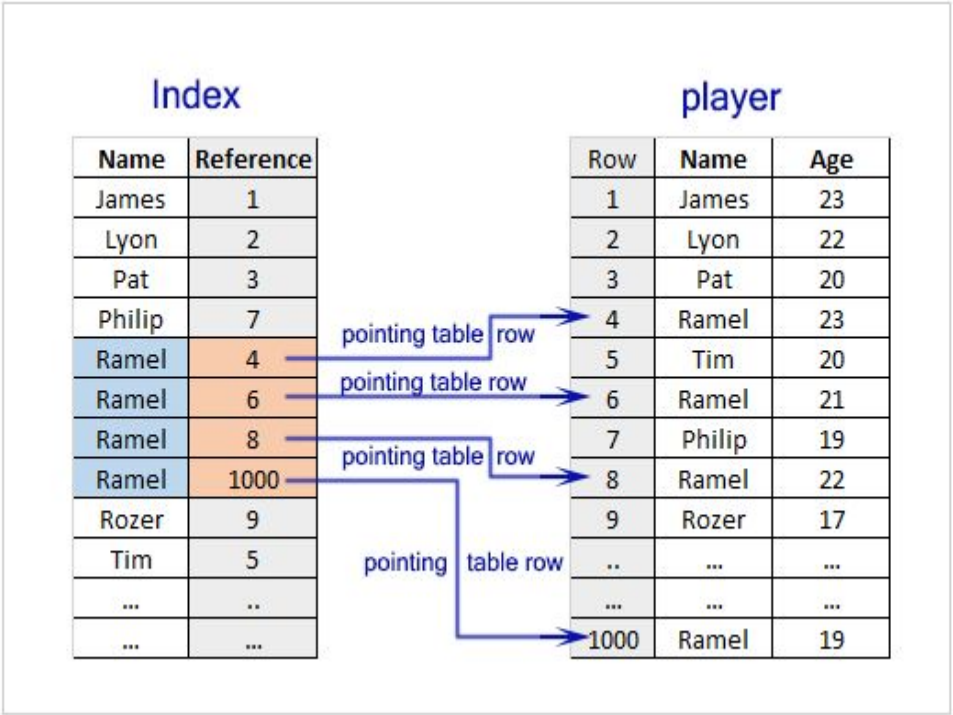
- Queries are slowed down by unnecessary Collection Scans
- Data structures that contain a set of keys from documents and their values. Stored in memory
- Limit the number of documents MongoDB has to scan (by-pass full document scan) to improve speed of retrieval / read operations
- Create an index for whatever field the query should run on. Defaults to _id

2. Indexing

```
db.users.createIndex({name: 1})

db.users.find({name: "Ramel"})

db.users.getIndexes()
db.users.dropIndex({name: 1})
```



Type of Indexes

- Single Index
  - Collect and sort data from a single field in each document in a collection
  - `db.collection.createIndex({name: 1})`

- Compound Index
  - Collect and sort data from a >=2 fields in each document in a collection
  - `db.collection.createIndex({<field1>: <sortOrder>,<field2>: sortOrder,.....,<fieldN>: <sortOrder>})`

Type of Indexes - Compound Index

```
db.collection.createIndex({<field1>: <sortOrder>,<field2>: sortOrder,......., <fieldN>: <sortOrder>})
```

- Data is grouped by the first field in the index, and then by each subsequent field
- Queries can be supported on individual or combination of the indexes where the prefix is always the first field

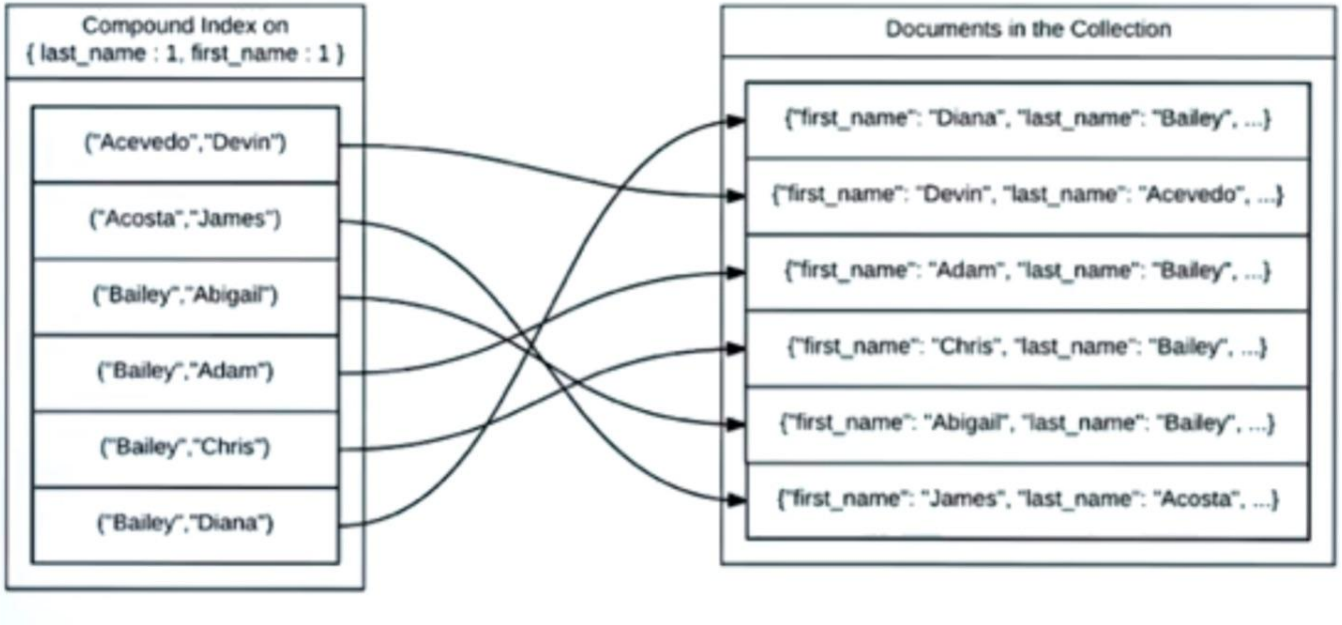
Example

```
db.collection.createIndex({“item”: 1, “location”: 1, “stock”: 1})
```

- **item**
- **item and location**
- **Item and stock**
- **item, location and stock**

Type of Indexes - Compound Index

```
db.users.createIndex({last_name:1, first_name: 1})
```



ESR (Equality, Sort, Range) Rule

- Order of the fields affects the effectiveness in compound indexes

- Equality (High Selectivity)
  - Exact match on a single value to limit the number of documents
  - Best placed as the first index field
- Sort
  - Follows equality match to reduce amount of documents that need to be sorted
- Range (Loosely selective)

ESR (Equality, Sort, Range) Rule

```
db.cars.find({
    manufacturer: “Ford”, // Equality match
    cost: {$gt: 15000}  // Range
    }).sort({model: 1}) // Sorting
```

- Following ESR rule, the optimal index would be
    - `db.cars.createIndex({manufacturer: 1, model: 1, cost: 1})`

Type of Indexes

- Geospatial Index
  - For applications who continuously query a field with geospatial data
  - Beneficial when querying with **`$near`** or **`$geoNear`**
  - **`db.collection.createIndex({ <location field> : "2dsphere" })`**

- Text Index
  - Improve performance when searching words/phrases (string content)
  - **`db.collection.createIndex({ <field> : "text" })`**

Indexing Pros / When to use

- Repeatedly running queries on the same field & return a subset of documents
- Faster sorting of query results
- Large Collection sets
- Text Search & Geospatial Queries

Indexing Cons

- Query Selectivity - Indexed field has low selectivity (duplicate values)
- Write / Update performance overhead- Negative impact on write operations as indexes should be updated to reflect the changes in the collection
- Index Selection & Planning - Improper index selection or over indexing can lead to increased index selection process, unnecessary index updates etc
- Increased Memory Usage - Too much index = working set will not fit in the memory

3. Covered Queries

Indexing commonly queries fields

- Can the query can be satisfied **without scanning any documents** and use index only scans
- Only satisfied if every field the query needs to access is part of the index

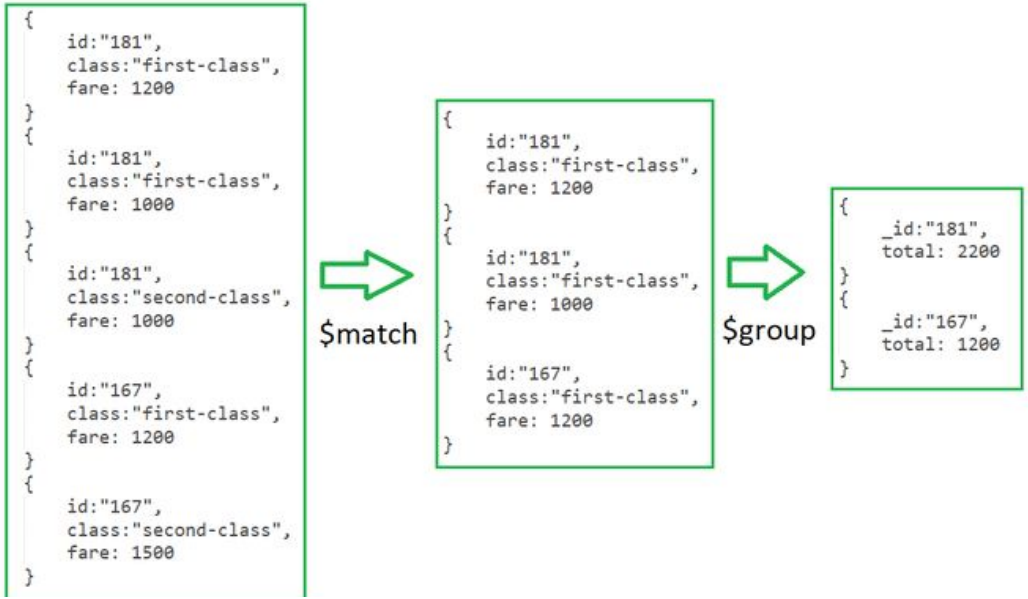
    - `db.users.createIndex({ gender:1, name : 1})`
    - `db.users.find({gender: "F"}, {name: 1, _id:0})`

4. Optimizing Aggregation Pipelines

- Process data through a sequence of stages

- Pipeline stage order matters
  - Filtering early in the pipeline stage
  - Using projection to limit output,
  - Leveraging indexes

```
db.train.aggregate([
{$match: {class: "first-class"}}, // Match stage
{$group: {_id: "id", total: {$sum: "$fare"}}} // Group Stage
])
```



4. Optimizing Aggregation Pipelines

- Pipeline stage ordering
  - Early stages should reduce result set ($match, $limit, etc)
  - $sort as early as possible (Before any kind of transformations)
- Index utilization
  - Index fields used in $match, $sort
- Use Projection
  - Avoid unnecessary fields in the output
- Avoid large results
  - $limit, $skip
- Set **`maxTimeMs`**

Best Practices in Optimizing our queries

- Use .lean() method to return plain JSON instead of Mongoose documents
- Limit the results to reduce network demand
- Drop unnecessary indexes
- Use projection to return only necessary data
- Caching frequent query results (Redis, etc)
- Query profiling and Monitoring to identify bottlenecks (mongostat, etc)

END