14. Indexing and Performance Tuning

- Understanding indexing in MongoDB
- Creating and managing indexes
- Optimizing query performance

1. Understanding Indexing in MongoDB

Indexes in MongoDB are special data structures that store a portion of the data in a collection in an easy-to-traverse format. They improve the efficiency of query operations, allowing the database to quickly locate and retrieve data without scanning every document in a collection.

Key Concepts of Indexing:

- **Index Structure:**
  - MongoDB uses **B-trees** for indexes, providing efficient read and write operations.
- **Default Index:**
  - MongoDB automatically creates an index on the _id field of every collection, ensuring uniqueness.
- **Why Indexing is Important:**
  - Without an index, MongoDB performs a **collection scan**, which examines every document in a collection to find the relevant data, leading to slower queries.
- **Trade-offs of Indexing:**
  - **Pros:**
    - Faster query execution.
    - Efficient sorting and filtering.
  - **Cons:**
    - Increased storage space for the index data.
    - Overhead during write operations, as indexes must be updated.

---

2. Creating and Managing Indexes

Creating Indexes:

Basic Index:

- To create a simple index on a single field

```
db.collection.createIndex({ fieldName: 1 })  // 1 for ascending, -1 for descending
```

Compound Index:

- Indexes on multiple fields

```
db.collection.createIndex({ field1: 1, field2: -1 })
```

Useful for queries that filter on multiple fields.

Multikey Index:

Automatically created for array fields, allowing indexing of each element

```
db.collection.createIndex({ arrayField: 1 })
```

Text Index:

For full-text search on string fields

```
db.collection.createIndex({ fieldName: "text" })
```

Geospatial Index:

For queries involving location data

```
db.collection.createIndex({ location: "2dsphere" })
```

Managing Indexes:

View Indexes

```
db.collection.getIndexes()
```

Drop an Index

```
db.collection.dropIndex("indexName")
```

Drop All Indexes

```
db.collection.dropIndexes()
```

Index Naming:

MongoDB assigns a default name to each index based on its fields, but you can provide a custom name for clarity:

```
db.collection.createIndex({ fieldName: 1 }, { name: "customIndexName" }
```

Adding Expiry to Indexes in MongoDB

MongoDB supports **TTL (Time-To-Live) Indexes**, which are special indexes that automatically delete documents after a specified period. This is particularly useful for managing datasets with time-sensitive information, such as logs, sessions, or temporary data.

Key Concepts of TTL Indexes

1. **Purpose:** Automatically delete documents after a specific amount of time to free up storage and keep the dataset current.
2. **Behavior:** TTL indexes work by periodically checking the indexed field and deleting documents with expired timestamps.
3. **Supported Data Type:** The indexed field must store **date values** (ISODate).

Creating a TTL Index

- A TTL index is created on a field that contains a date value with an additional expireAfterSeconds option.

```
db.collection.createIndex(
   { fieldName: 1 },
   { expireAfterSeconds: seconds }
)
```

Example:

1. Add a document with a createdAt field:

```
db.sessionData.insertOne({
   sessionId: "abc123",
   createdAt: new Date(),  // Current timestamp
   userId: 101
 })
```

Create a TTL index to expire documents 3600 seconds (1 hour) after the createdAt field

```
db.sessionData.createIndex(
   { createdAt: 1 },
   { expireAfterSeconds: 3600 }
)
```

Result:

- The document will be automatically removed from the sessionData collection 1 hour after the createdAt timestamp.

Updating TTL Behavior

- **Change TTL Value:**
  - To modify the expiration time, you need to drop the existing TTL index and recreate it with a new expireAfterSeconds value:

```
db.collection.dropIndex("indexName")
db.collection.createIndex(
   { fieldName: 1 },
   { expireAfterSeconds: newSeconds }
)
```

Best Practices for TTL Indexes

1. **Monitor Deletions:**
    - MongoDB's TTL deletions occur in the background and may not happen precisely at the expiration time, depending on system load and the TTLMonitor frequency (default is 60 seconds).
2. **Use Dedicated Fields:**
    - Avoid reusing fields with other indexes for TTL to prevent conflicts.
3. **Ensure Correct Data Type:**
    - The indexed field must store **Date** objects (ISODate format) to work correctly with TTL.

Limitations of TTL Indexes

1. **Single TTL Index per Collection:** A collection can have only one TTL index.
2. **Fixed Expiration:** TTL cannot have dynamic expiration per document. For custom expirations, you must calculate and set the expiration timestamp in the indexed field manually.
3. **TTL Monitor Frequency:** Expired documents are not removed immediately but are cleaned up during periodic background checks.

Use Cases of TTL Indexes

1. **Session Management:**
    - Automatically remove expired user sessions or tokens.
2. **Log Cleanup:**
    - Keep only recent logs by deleting older ones after a certain period.
3. **Caching:**
    - Remove outdated cache entries without manual intervention.

3. Optimizing Query Performance

Analyzing Query Performance:

- **Explain Plans:**
    - Use the explain() method to analyze query execution and determine if indexes are being used effectively:

```
db.collection.find({ fieldName: value }).explain("executionStats")
```

Stages:

- COLLSCAN (Collection Scan): Indicates no index is being used.
- IXSCAN (Index Scan): Indicates an index is being used.

Query Profiling:

- Enable profiling to log slow queries

```
db.setProfilingLevel(1)  // 1 = log slow queries
db.system.profile.find().pretty()
```

Optimizing Queries:

- **Use Indexes Properly:**
  - Ensure fields in the query filter, sort, or join have appropriate indexes.
- **Avoid Full Collection Scans:**
  - Use filters that match indexed fields.
- **Optimize Query Shape:**
  - Keep query predicates simple and avoid operations that prevent index usage, like $not or complex regular expressions.
- **Projection Optimization:**
  - Retrieve only the required fields using projections

```
db.collection.find({ fieldName: value }, { field1: 1, field2: 1 })
```

Index Design Strategies:

- **Field Order in Compound Indexes:**
  - Place the most selective fields (fields with high cardinality) first.
- **Sparse Indexes:**
  - Indexes only documents with the indexed field present:

```
db.collection.createIndex({ fieldName: 1 }, { sparse: true })
```

Partial Indexes:

- Index only a subset of documents

```
db.collection.createIndex({ fieldName: 1 }, { partialFilterExpression: { fieldName: { $exists: true } } })
```

Covering Indexes:

- Include fields in the index that are used in the query's filter, sort, and projection, enabling the query to be served directly from the index

Interview Questions and Answers

Q1: What is an index in MongoDB, and why is it important?

- **Answer:** An index is a special data structure in MongoDB that allows the database to efficiently retrieve documents based on a query. Indexes reduce the need for collection scans, significantly improving query performance. However, they come at the cost of additional storage space and overhead during write operations, as the index needs to be updated whenever the data changes.

Q2: What types of indexes are available in MongoDB?

- **Answer:**
  - **Single Field Index:** Index on a single field (default _id).
  - **Compound Index:** Index on multiple fields.
  - **Multikey Index:** Index on array fields.
  - **Text Index:** For full-text search on string fields.
  - **Geospatial Index:** For location-based queries (2d or 2dsphere).
  - **Hashed Index:** For hashed sharding or equality queries.
  - **TTL Index:** Automatically removes documents after a specific time.

Q3: What is the default index created in MongoDB collections?

- **Answer:** By default, MongoDB creates an index on the _id field of every collection. This index ensures the uniqueness of _id values and improves the efficiency of queries that target _id.

Q4: How do you create an index on a single field in MongoDB?

- **Answer:** Use the createIndex() method:

```
db.collection.createIndex({ fieldName: 1 }); // 1 for ascending order, -1 for descending
```

Q5: What is a compound index, and how is it created?

- **Answer:** A compound index is an index on multiple fields. It is used when queries filter or sort on multiple fields. Create a compound index like this:

```
db.collection.createIndex({ field1: 1, field2: -1 });
```

Q6: How do you view the indexes on a collection?

- **Answer:** Use the getIndexes() method:

```
db.collection.getIndexes();
```

Q7: How do you drop an index?

- **Answer:** Use the dropIndex() method, specifying the index name:

```
db.collection.dropIndex("indexName");
```

Q8: What is a TTL index, and why is it used?

- **Answer:** A TTL (Time-To-Live) index is a special index that automatically deletes documents after a specified time period. It is used for managing time-sensitive data like logs, session data, or temporary cache.

Q9: How do you create a TTL index?

- **Answer:** Use the expireAfterSeconds option while creating an index on a date field:

```
db.collection.createIndex(
    { createdAt: 1 },
    { expireAfterSeconds: 3600 } // Documents expire after 1 hour
  );
```

Q10: What are the limitations of TTL indexes?

- **Answer:**
  - A collection can have only one TTL index.
  - The indexed field must store a Date object.
  - Deletions are not immediate and depend on the TTLMonitor frequency (default is 60 seconds).
  - TTL expiration time is static and cannot vary per document.

Q11: How can you analyze the performance of a query in MongoDB?

- **Answer:** Use the explain() method with queries to analyze execution:

```
db.collection.find({ fieldName: value }).explain("executionStats");
```

  The output shows whether an index is used (IXSCAN) or if a collection scan (COLLSCAN) occurs.

Q12: What is a covering index, and why is it beneficial?

- **Answer:** A covering index is an index that contains all the fields used in a query's filter, sort, and projection. It allows MongoDB to answer the query directly from the index without accessing the actual documents, improving performance.

Q13: What are the best practices for designing indexes in MongoDB?

- **Answer:**
  - Use indexes for fields frequently used in queries.
  - Prefer compound indexes for queries filtering on multiple fields.
  - Place the most selective fields first in compound indexes.
  - Avoid creating too many indexes, as they can degrade write performance.
  - Regularly monitor and drop unused indexes.
  - Use sparse or partial indexes to reduce index size for specific cases.

Q14: How does MongoDB handle query performance for sorting operations?

- **Answer:** MongoDB can use an index to perform efficient sorting. To enable indexed sorting:
  - Ensure the sort field is part of an existing index.

- Combine sorting and filtering in queries to leverage compound indexes.

Q15: What is the difference between a sparse index and a partial index?

- **Answer:**
  - **Sparse Index:** Indexes only documents where the indexed field exists.

```
db.collection.createIndex({ fieldName: 1 }, { sparse: true });
```

  - **Partial Index:** Indexes documents matching a specific filter expression.

```
db.collection.createIndex(
    { fieldName: 1 },
    { partialFilterExpression: { status: "active" } }
  );
```

Q16: A query on a collection is slow, and explain() shows a COLLSCAN. What would you do?

- **Answer:**
  - Identify the fields used in the query filter and create an appropriate index.
  - Verify if the query shape matches the index (e.g., field order in compound indexes).
  - Check for typos or issues preventing index usage (e.g., $not or unsupported operators).

Q17: How would you optimize a query that filters by one field and sorts by another?

- **Answer:**
  - Use a compound index with the filter field followed by the sort field:

```
db.collection.createIndex({ filterField: 1, sortField: -1 });
```