NoSQL databases - MongoDB

Secondary Indexes

- Secondary indexes in MongoDB are implemented as B-trees.
- B-tree indexes are optimized for a variety of queries, including range scans and queries with sort clauses
- By permitting multiple secondary indexes, MongoDB allows users to optimize for a wide variety of queries
- With MongoDB, you can create up to 64 indexes per collection
- ascending, descending, unique, compound-key, and even geospatial indexes are supported

Different Indexes Types

MongoDB provide support for different kinds of indexes to support specific types of data and queries.

- Default _id
- Single Field
- Compound Index
- Multikey Index
- Geospatial Index
- Text Indexes
- Hashed Indexes

Single field Index

- MongoDB provides complete support for indexes on any field in a collection of documents

- By default, all collections have an index on the _id field

- Applications and users may add additional indexes to support important queries and operations.

- The following command creates an index on the name field for the users collection

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

Compound Indexes

- MongoDB supports creating compound indexes by combing multiple fields

- Compound indexes can support queries that match on multiple fields.

  db.productsInfo.createIndex({ "itemName": 1, “noOfItems": 1 })

- The order of the fields in a compound index is very important.

- Documents sorted first by the values of the first field and, within each value of the first field, sorted by values of the second field

- MongoDB imposes a limit of 31 fields for any compound index

Multikey Indexes

- Multikey index is supported on a field that holds an array value

- MongoDB creates an index key for each element in the array

- Multikey indexes helps running efficient queries against array fields

- Multikey indexes can be generated over arrays that hold both scalar values (e.g. strings, numbers) and nested documents.

  ```
  db.collectionName.createIndex( { <field>: < 1 or -1 > } )
  ```

- if any indexed field is an array then MongoDB automatically creates a multikey index

Hashed Indexes

- Hashed indexes stores hash values of the indexed field.
- Hash Index can’t be created on multi-key (i.e. arrays) field
- The hashing function flattens embedded documents and generate the hash value
- MongoDB can use the hashed index to support equality queries
- Hashed indexes do not support range queries
- Hashed indexes support sharding a collection using a hashed shard key
- Using a hashed shard key to shard a collection ensures a more even distribution of data.

```
db.items.createIndex( { item: "hashed" } )
```

TTL(Time to Live) Indexes

- TTL indexes are special single-field indexes to automatically remove documents from a collection after a certain amount of time.

- It is useful for certain types of data such as application and server logs, and session information that only need to persist in a database for a finite amount of time.

- Use the db.collection.createIndex() method with the expireAfterSeconds option on a field whose value is either a date or an array that contains date value

  ```
  db.eventlog.createIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
  ```

- TTL indexes are a single-field indexes. Compound indexes do not support TTL

Unique Indexes

- Using unique index, MongoDB rejects all documents that contain a duplicate value for the indexed field

```
db.items.createIndex( { “item": 1 }, { unique: true } )
```

- If you use the unique constraint on a compound index, then MongoDB will enforce uniqueness on the combination of values rather than the individual value for any or all values of the key.
- It stores a null value for the document, If a document does not have a value for the indexed field in a unique index
- Because of the unique constraint, MongoDB will only permit one document that lacks the indexed field
- we can also combine the unique constraint with the sparse index to filter these null values from the unique index and avoid the error

Text Indexes

- A MongoDB provides text indexes to support text search of string content in documents of a collection
- Text indexes can be created on any field whose value is a string or an array of string elements
- To perform queries that access the text index, use the $text query operator.
- To create a text index on a field “customer_name” that contains a string or an array of string elements of the customer_info collection:

```
db.customer_info.createIndex({”customer_name”: “text”})
```

- To perform the text search

```
db.customer_info.find({$text:{$search:”John”}})
```

- A collection can have at most one text index.

.

Multi Languages Text Search

Supported Languages and Stop Words

- Mongo DB supports text search for various languages and drop language-specific stop words
- For English language it uses simple language-specific suffix stemming and drop stop words like “the”, “an”, “a”, “and”, etc.)
- db.quotes.createIndex({ content : "text" }, default_language: "spanish" })
- Text Search Languages: Danish,dutch,English,finish, French, German,Italian, Norwegian, Portuguese, Romanian, Russian, Spanish. Swedish,Turkish, Hungarian, Portuguese

Geospatial Indexes

- MongoDB provides a special type of index for coordinate plane queries, called a geospatial index
- **db.collection.createIndex({ <location field> : "2dsphere" })**

Geospatial Query Operators

- **Inclusion**: MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the $geoWithin operator.
- **Intersection** : MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the $geoIntersects operator.
- **Proximity** MongoDB can query for the points nearest to another point. Proximity queries use the $near operator.

Geospatial $geoWithin Query

- The $geoWithin operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

db.<collection>.find({ <location field> : { $geoWithin :{ $geometry :{ type : "Polygon" ,coordinates : [<coordinates>]} } } })

- The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

db.places.find({ loc :{ $geoWithin :{ $geometry :{ type : "Polygon" ,coordinates : [[[0 , 0] ,[3 , 6] ,[6 , 1] ,[0 , 0]]]} } } })

Geospatial Proximity Query

- Proximity queries return the points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a 2dsphere index
- To query for proximity to a GeoJSON point, use either the $near operator or geoNear command. Distance is in meters.

```
db.<collection>.find( { <location field> :{ $near :{ $geometry :{ type : "Point" ,coordinates : [
<longitude> , <latitude> ] } ,$maxDistance : <distance in meters>} } } )
```

- The geoNear command uses the following syntax:

```
db.runCommand( { geoNear : <collection> ,near : { type : "Point" ,coordinates: [ <longitude>,
<latitude> ] } ,spherical : true } )
```

- To select all grid coordinates in a “spherical cap” on a sphere, use $geoWithin with the $centerSphere operator.

```
db.<collection>.find( { <location field> :{ $geoWithin :{ $centerSphere :[ [ <x>, <y> ] , <radius> ]
}} }
```

Sparse Indexes

- Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value.
- It ignores and skips over those document which don’t have the indexed field.
- The index is called as “sparse” because it does not include all documents of a collection.

  db.sales.createIndex({ “sales_id": 1 }, { sparse: true })
- 2d (geospatial) and text indexes are sparse by Default
- Sparse and unique Properties: An index that has both sparse and unique properties defined,prevents collection from having documents with duplicate values for a field but allows multiple documents that omit the key.

Index Creation

- By default, creating an index blocks all other operations by holding X exclusive lock on that collection
- The Collection that holds the collection is unavailable for read or write operations until the index build completes.
- For potentially long running index building operations, consider the creating the indexes in the background

db.people.createIndex({ zipcode: 1}, {background: true})

- By default, background is false for building MongoDB indexes.
- You can combine the background option with other options, as in the following:

db.people.createIndex({ zipcode: 1}, {background: true, sparse: true })

Removing an Index

- To remove an index from a collection use the dropIndex() method
  db.accounts.dropIndex({ "tax-id": 1 })

- You can also use the db.collection.dropIndexes() to remove all indexes, except for the _id index
  db.items.dropIndexes()

References:

https://www.mongodb.com/docs/manual/core/document/

Thank You