ASSIGNMENT NO: 6

AIM : Design Use MongoDB queries for: 1.Indexing 2.Aggregate.

INDEX TERMS: MongoDB, Indexing, Aggregate.

THEORY

Indexes provide high performance read operations for frequently used queries. This section introduces indexes in MongoDB, describes the types and configuration options for indexes, and describes special types of indexing MongoDB supports. The section also provides tutorials detailing procedures and operational concerns, and providing information on how applications may use indexes. Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. These collection scans are inefficient because they require mongod to process a larger volume of data than an index for each operation. Indexes are special data structures 1 that store a small portion of the collection's data set in an easy to traverse form.

The index stores the value of a specific field or set of fields, ordered by the value of the field. Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect. In some cases, MongoDB can use the data from the index to determine which documents match a query. The following diagram illustrates a query that selects documents using an index.

Index Types

MongoDB provides a number of different index types to support specific types of data and queries.

Default _id

All MongoDB collections have an index on the _id field that exists by default. If applications do not specify a value for _id the driver or the mongod will create an _id field with an ObjectId value.

The _id index is unique, and prevents clients from inserting two documents with the same value for the _id field.

Single Field

In addition to the MongoDB-defined _id index, MongoDB supports userdefined indexes on a single field of a document

Consider the following illustration of a single-field index:

Diagram of an index on the score field (ascending).

Compound Index
MongoDB also supports user-defined indexes on multiple fields. These compound indexes behave like single-field indexes; however, the query can select documents based on additional fields. The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score:-1 }, the index sorts first by userid and then, within each userid value, sort by score. Consider the following illustration of this compound index:

Multikey Index
MongoDB uses multikey indexes to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array. These multikey indexes allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Geospatial Index
To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: 2d indexes that uses planar geometry when returning results and 2sphere indexes that use spherical geometry to return results.

Text Indexes
MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific stop words (e.g. “the”, “a”, “or”) and stem the words in a collection to only store root words.

Hashed Indexes
To support hash based sharding, MongoDB provides a hashed index type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries.
Example Given the following document in the friends collection:

```
{ "_id" : ObjectId(...),
"name" : "Alice"
"age" : 27
}
```

The following command creates an index on the name field:

```
db.friends.ensureIndex( { "name" : 1 } )
```

Indexes on Embedded Fields
You can create indexes on fields embedded in sub-documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from indexes on sub-documents, which include the full content up to the maximum index size of the sub-document in the index. Instead, indexes on embedded fields allow you to use a “dot notation,” to introspect into subdocuments.
Consider a collection named people that holds documents that resemble the following example document:

```
{"_id": ObjectId(...)
"name": "John Doe"
"address": {
"street": "Main",
"zipcode": "53511",
"state": "WI"
}
```

}
You can create an index on the address.zipcode field, using the following specification:
db.people.ensureIndex({ "address.zipcode": 1 })

Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

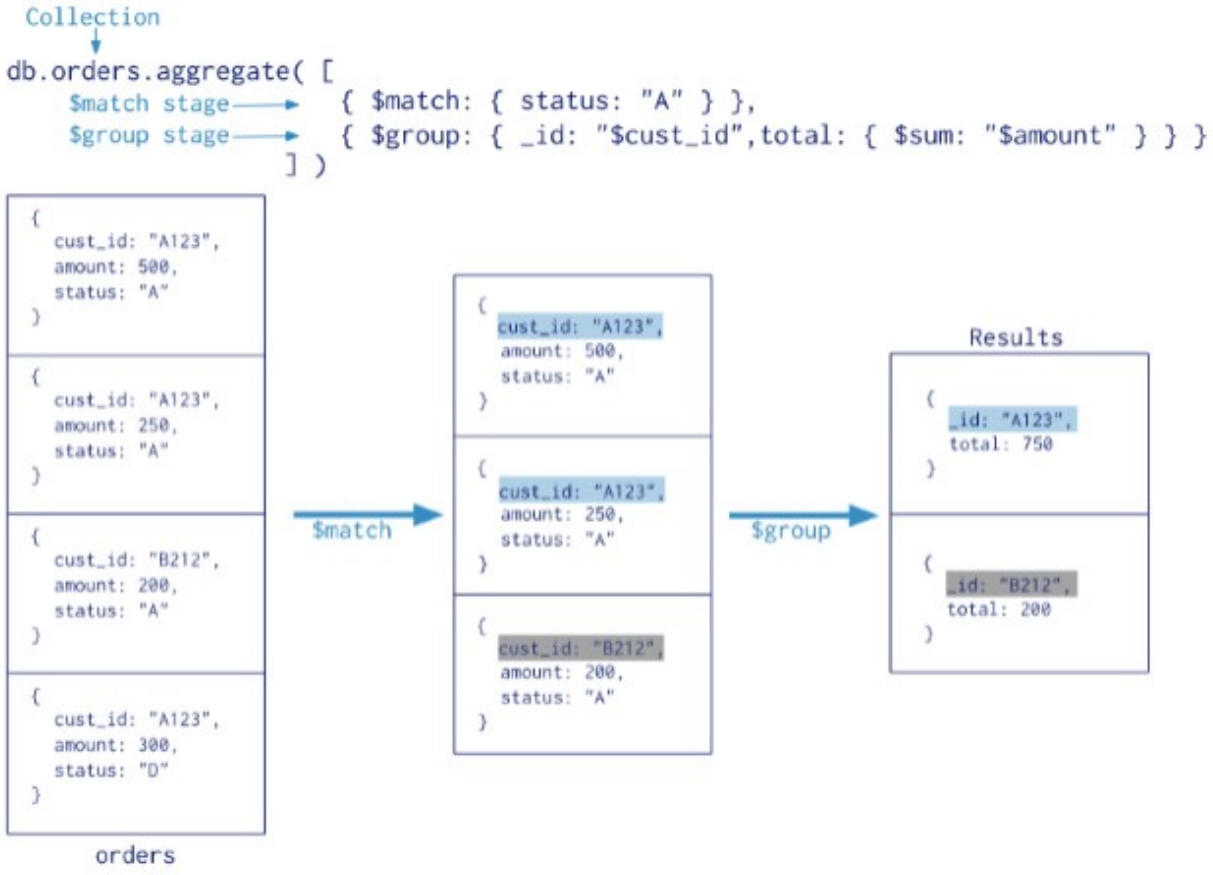
MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function , and single purpose aggregation methods and commands.

Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the mongod instance simplifies application code and limits resource requirements. Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents.

Aggregation Pipelines

MongoDB 2.2 introduced a new aggregation framework, modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document. Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.
Figure:



1. $project:
It reshapes a document stream. $project can rename, add, or remove fields as well as create computed values and sub-documents.
>db.articles.aggregate({"$project": {"author" :1, "_id": 0}})
By default, "_id" is always returned if it exists in the incoming document

2. $match:
Filters the document stream, and only allows matching documents to pass into the next pipeline stage. $match uses standard MongoDB queries. $match filters documents so that you can run an aggregation on a subset of documents. For example, if you only want to find out stats about users in Oregon, you might add a "$match" expression such as {$match : {"state" : "OR"}}. "$match" can use all of the usual query operators ("$gt", "$lt", "$in", etc.). Generally, good practice is to put "$match" expressions as early as possible in the pipeline. This has two benefits: it allows you to filter out unneeded documents quickly and the query can use indexes if it is run before any projections or groupings.

3. $limit: Restricts the number of documents in an aggregation pipeline.

4. **$Skip:** Skips over a specified number of documents from the pipeline and returns the rest.

5. **$unwind:** Takes an array of documents and returns them as a stream of documents. Unwinding turns each field of an array into a separate document.

Pipeline operators & Indexes:
The $match and $sort pipeline operators can take advantage of an index when they occur at the beginning of the pipeline. If your aggregation operation requires only a subset of the data in a collection, use the $match, $limit, and $skip stages to restrict the documents that enter at the beginning of the pipeline. When placed at the beginning of a pipeline, $match operations use suitable indexes to scan only the matching documents in a collection.

FAQS:
Q.1. What isdropDups parameterusedwhile creatingindex?
Q2. What isaggregation?
Q3. What isthe purposeof $unwin

Conclusion:
Outcome of the experiment is students are able to understand and implement various Aggregation function and Indexing.