Mongo db Best practices

##### Indexes

Every table must have a index , if there is no index , all rows will be inspected which is a time consuming process

##### Read Ratio/Write Ratio

Identify the **business needs** and design the schema for read-heavy or write-heavy.

Read-heavy

* When an application is read-heavy, we design the schema that **minimizes the number of reads from MongoDB**.

Write-heavy

* When an application is write-heavy, we must ensure that the schema designed **should maximize MongoDB write throughput**.

##### Avoid Application Joins

* **Server-side joins are not supported** in MongoDB.
* Performance can degrade when you are **pulling back and joining a lot of data**.
* When there is a need of so many joins, it advisable to **de-normalize the schema.**

**Pre-Aggregated Data**

Suppose, if there is a lot of aggregation of data in application queries, you must consider pre-aggregation of the data.

Example:

Consider you have a web application and want to know how many users view a page. Rather than summing up the number of views made for a particular page on request:

* Provide an incremental view counter for that page, each time the page is viewed.
* Calculate the number of page views, based on this counter.

Means rather than aggregating / summing up , u better store only pre aggregated data

**Avoid Growing Documents**

When documents created are constantly growing in size, it can impact:

* **Disk IO**
* **Database performance**

In such cases, perform **document buckets** and **document pre-allocation**.

**Over Indexing**

An index also takes up **space and memory**. More Indexes will be an overhead for write operations.

Game rule

* Create schema based on how we can avoid more indexes.
* There should be a balance between query and Indexes.

**One-to-One Relationships Embedded Model**

* The 1:1 relationship defines a relationship between two entities.
* With referencing, the application needs to issue multiple queries to resolve the references. Below example illustrates the advantage of embedding over referencing.

**Normalized Data:**

Consider we have **User** and **addresses** documents

If those 2 datas if u are frequently accessing, then instead of storing separately

Store in a embedded document model.

---User-----

{

\_id: "Jose",

FullName: "Jose Varghese"

}

---Addresses----

{

user\_id: "jose",

House: "#4D Golden Flat",

streetname: "William street",

city: "BANGALORE",

state: "KA",

zip: "560060"

}

**Embedded Data:**

Suppose **address** data is frequently retrieved with the Fullname. In such scenario, it's better to design data model to embed the address data in the user data as mentioned below.

{

\_id: "Jose",

FullName: "Jose Varghese",

Address: {

House: "#4D Golden Flat",

streetname: "William street",

city: "BANGALORE",

state: "KA",

zip: "560060"

}

}

##### One-to-Many Relationships

* 1:N relationship expresses a relationship where one side can hold more than one relationship whereas the reverse relationship can only be single sided.

Consider the following one-to-many relationship between **user** and **address** data, here; the user has multiple address entities.

**Normalized Data:**

---user

{

\_id: "Jose",

FullName: "Jose Varghese"

}

---Addresses 1-----

{

user\_id: "jose",

House: "#4D Golden Flat",

streetname: "William street",

city: "BANGALORE",

state: "KA",

zip: "560060"

}

---Addresses 2-----

{

user\_id: "jose",

House: "1 Some other house name",

streetname: "Alexander street",

city: "London",

state: "NA",

zip: "NA"

}

**Embedded Data: since we always need it store it in a embedded way**

{

\_id: "Jose",

name: "Jose Varghese",

Addresses: [

{

House: "#4D Golden Flat",

streetname: "William street",

city: "BANGALORE",

state: "KA",

zip: "560060"

},

{

House: "1 Some other house name",

streetname: "Alexander street",

city: "London",

state: "NA",

zip: "NA"

}

]

}

While des

Index Introduction

**An index is required for faster retrieval of data. Indexes in MongoDB are sorted and stored as B-tree structure. There should be a balance between Indexes and queries.**

Following are the index types supported in MongoDB.

* **Default\_id**: Each collection contains an index named default\_id
* **Single Field**: Indexes can be either in ascending order or descending order
* **Compound Index**: used for multiple fields
* **Text indexes**: To support text search queries on string content
* **Multikey Index**: These are used to index array data
* **Geospatial Index**: Indexes used are of 2d and 2d spheres

**Index Selection - Factors Consideration**

**When query in MongoDB is not indexed, a full collection scan will be performed. The absence of index can cause significant database performance degradation.**

**sort() method returns documents in ascending or descending order.**

**Performance of Index can be improved by doing the following things:**

* **Documents inspected in memory should be reduced.**
* **The need to perform in-memory sorts must be removed.**
* **Few of the most common Index properties in MongoDB include:**
* 1. Unique Index**helps MongoDB to reject duplicate values for the indexed field.**
* 2. Sparse Index**ensures that only index contains entries for documents that have the indexed field. Documents that do not possess the indexed field will be skipped.**
* 3. TTL Index**is used where there is a need for automatically removing documents from a collection after a certain amount of time.**
* 4. Parse Index**Index the documents in a collection when specified filter expression is met. It is a subset of the *sparse index* and offers lower storage requirements and reduced performance costs for index creation and maintenance.**

Creating and dropping an index

db.file.createIndex({tags: 1});

db.file.dropIndex({tags: 1});

db.Player.createIndex( { score: 1 } )

getting an index

db.Products.getIndexes();

inserting a record

db.Player.insert({

"\_id": "1",

"score": 10340,

"location": { state: "NSW", city: "Sydney" }

})

Compound indexes

db.collectionname.createIndex({<field1>: <type>, <field2>: <type2>, ... } )

## Optimizing Query performance

* Creating indexes
* Limit the number of query results to decrease network demand by utilizing limit() command.
* Use projections to return only necessary data
* Use $hint to select a particular index
* Use increment operator to perform a server-side operation
* Run database profiler and find the query which is running slow and optimize the statement
* Rebuild index if needed
* Set maximum execution times