Indexing In Mongo db

Single Field Index

Indexes a single field for faster queries.

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

2️⃣ Unique Index

Ensures all values in the indexed field are unique.

```
db.collection.createIndex({ email: 1 }, { unique: true })
```

3️⃣ Compound Index

Indexes multiple fields together for queries that filter/sort on multiple fields.

```
db.collection.createIndex({ a: 1, b: 1 })
```

4️⃣ Multikey Index

Used for array fields. MongoDB indexes each element in the array.

```
db.products.createIndex({ tags: 1 })
```

5️⃣ Hashed Index

Useful for sharding or equality lookups.

```
db.users.createIndex({ userId: "hashed" })
```

6️⃣ TTL (Time-To-Live) Index

Automatically deletes documents after a specified time (in seconds).

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

7️⃣ Show Indexes

List all indexes in a collection.

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

8️⃣ Drop Index

Delete an index by name.

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

9️⃣ Partial Index

Indexes only documents that match a filter condition. Saves space and improves performance for selective queries.

Syntax:

```
db.collection.createIndex(
  { fieldName: 1 },  // index key
  { partialFilterExpression: { conditionField: { $operator: value } } }
```

```
)
```

Example:

```
db.orders.createIndex(
  { status: 1 },
  { partialFilterExpression: { status: { $eq: "pending" } } }
)
```

> Only documents with status: "pending" will be indexed.

Sparse Index

- Only includes documents where the indexed field **exists and is not** `null`.
- Example:

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

- Documents **without** `nickname` are not included in the index.

In MongoDB you **can give an index any name you want** using the `name` option when creating the index.

Syntax:

```
db.collection.createIndex(
  { fieldName: 1 },   // the fields to index
  { name: "myCustomIndexName" }  // your custom index name
)
```

Example:

```
db.users.createIndex(
  { email: 1 },
  { unique: true, name: "unique_email_index" }
)
```

- Here, the index on `email` is named `"unique_email_index"` instead of the default auto-generated name (`email_1`).
- Using a custom name can make **index management and dropping indexes easier**, especially in large collections.

🔟 Covered Query

A query is **covered** if all fields used in the **filter** and **projection** exist in the index.

- MongoDB does **not** need to read the actual documents.
- Faster query execution.

Rules for Covered Query:

1. Filter fields are in the index.
2. Projection fields are in the index.
3. `_id` is excluded unless part of the index.

Example (Covered Query):

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

// Query
db.users.find({ name: "Ali" }, { name: 1, age: 1, _id: 0 });
```

✅ Covered: Filter and projection fields exist in the index.

Example (Not Covered Query):

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

// Query
db.users.find({ name: "Ali" }, { name: 1, age: 1, contact: 1, _id: 0 });
```

❌ Not covered: `contact` is not in the index, so MongoDB reads documents.

⚡ Summary

- **Indexes:** speed up queries.
- **Unique index:** enforces uniqueness.
- **Compound index:** supports multi-field queries.
- **Multikey index:** for arrays.
- **Hashed index:** equality lookups.
- **TTL index:** auto-deletes documents.
- **Partial index:** index only relevant documents.
- **Covered query:** query answered entirely from the index → fastest.

WINNING PLAN CONCEPT

If i have multiple indexes for same document , which index will be selected ? Mongo db do is that before the query runs , it selects some things from the docs by using each index and evalaute the performance through each index and choose index for search as the index with highest performance .

It does not evaluate before each query , after 1 time each caches the index

This index remains upto or caches resets after

- 2000 Writes operations
- mongo server is restart
- Other indexes are manupulated

- Index is reset

> db.collection.find({name:jibran}).(”allPlansExecution”)

MULTI -KEY INDEX

An index in MongoDB that is automatically created on a **field that contains an array**, which indexes **each element of the array separately** so queries on individual array values are f

{ name: ["Jibran", "Ali", "Sara"] }
Index on name → MongoDB indexes "Jibran", "Ali", and "Sara" individually.

TEXT-INDEX

{ {name : “jibran” , job = “i am a youtuber”},

{name : “samar” , job=” i am a youtuber and IT manager”},

{name: “qasim” , job =” i am a news reader”} }

To search jibran i have to write complete job = i am a youtube to cover both cases youtuber or manager i have to or: job = i am a youtuber or i am a youtuber or it manager

but if i want like i just search youtuber only and both jirban and samar comes then use text indexes

NOTE : THERE WILL BE ONLY 1 TEXT INDEX PER DOC

Create a Text Index

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

Example:

```
db.articles.createIndex({ title: "text", content: "text" })
```

- You can index **one or more string fields** using `"text"`.
- For multiple fields, MongoDB combines them into a **single text index**.

Drop all indexes except the default `_id` index

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

- This will remove **all indexes** on the collection **except the `_id` index**, which is mandatory.
- You don’t need to know the names of the indexes.

2. Query Using Text Index

```
db.collection.find({ $text: { $search: "search string" } })
```

Example:

```
db.articles.find({ $text: { $search: "MongoDB tutorial" } })
```

- Returns documents where `title` or `content` contains `"MongoDB"` or `"tutorial"`.
- sort by **relevance** using `score`:

```
db.articles.find(
  { $text: { $search: "MongoDB" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

Text Score

- When you perform a **text search** using a **text index**, MongoDB calculates a **relevance score** for each document.
- This score indicates **how well the document matches the search terms**.
- It's called the `textScore`.

What is field weight?

- In a text index with multiple fields, **weights** control **how much a field contributes to the text score**.
- Higher weight → matches in that field are considered **more relevant**.

---

2. Syntax to assign weights

```
db.collection.createIndex(
  { title: "text", content: "text" },   // fields to index
  { weights: { title: 10, content: 5 } } // weights assigned
)
```

- Here:
  - `title` is **more important** than `content`
  - A match in `title` increases `textScore` more than a match in `content`.

What is **field weight** in MongoDB text index?

- It means: **some fields are more important than others** when searching text.
- A higher weight = MongoDB gives **more points** if that field matches.

⚡ Example

Suppose you have documents:

```
{ "title": "Fast Car", "content": "This car is very old but strong" }
{ "title": "Old Bicycle", "content": "This bicycle is fast and cheap" }
```

Create a text index with weights:

```
db.items.createIndex(
  { title: "text", content: "text" },
  { weights: { title: 5, content: 1 } }
)
```

🔍 Search:

```
db.items.find(
  { $text: { $search: "fast" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

✅ What happens?

- If "fast" is in **title**, it gets **more points** (weight = 5).
- If "fast" is only in **content**, it gets **fewer points** (weight = 1).

So MongoDB will rank documents where the word is in **title** higher than if it's only in **content**.

3. Using textScore with weighted fields

```
db.collection.find(
  { $text: { $search: "MongoDB" } },
  { score: { $meta: "textScore" } }
```

```
).sort({ score: { $meta: "textScore" } })
```

- MongoDB calculates the **textScore** using the weights.
- Documents where `"MongoDB"` appears in `title` will rank higher than those where it only appears in `content`.

1️⃣ Foreground Indexing (default behavior)

- When you create an index **without the** `background` **option**, MongoDB builds it in the **foreground**.
- This **blocks all other operations** on the collection until the index is built.
- Only use this for **small collections**, because it can freeze writes/reads temporarily.

```
// Foreground index (default)
db.users.createIndex({ name: 1 });
```

Characteristics:

Feature	Foreground
Collection access during index creation	Blocked
Index build speed	Fast (because no concurrent writes)
Suitable for	Small collections, offline maintenance

2️⃣ Background Indexing

- When you create an index **with** `{ background: true }`, MongoDB builds it in the **background**.
- The collection remains **readable and writable** during index creation.
- Slower than foreground but safer for **production systems**.

```
// Background index
db.users.createIndex({ email: 1 }, { background: true });
```

Characteristics:

Feature	Background
Collection access during index creation	Allowed
Index build speed	Slower (due to concurrency)
Suitable for	Large collections, production systems

NOTE: queires depended on that index will be blocked only