MongoDB Indexing: Concepts, Types, and Best Practices

1 Introduction to Indexing

Indexing in MongoDB is a technique used to improve the speed of data retrieval operations. It works similarly to an index in a book, allowing the database to locate data without scanning the entire collection.

- Without index → Collection Scan (COLLSCAN)
- With index → Index Scan (IXSCAN)

2 When NOT to Use Indexing

2.1 Small Collections

- Full scan is already fast
- Index adds unnecessary overhead

Example: A student collection with only 100 records.

2.2 High Write/Update Frequency

- Every insert/update must update indexes
- Too many indexes slow down writes

Example: Real-time IoT sensor data streaming.

2.3 Low Selectivity Fields

- Fields with few unique values

```
gender: "Male", "Female"
status: "Active", "Inactive"
```

2.4 Frequently Changing Fields

- Index maintenance cost is high

Example: `last_seen_time` updated every second.

2.5 Too Many Indexes

- Increased memory usage
- Slower insert/update operations

2.6 Queries Returning Large Data

- If query returns more than 30–40% of data, index is inefficient

```
db.users.find({country: "India"})
```

3 Types of Indexing in MongoDB

3.1 Single Field Index

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

Use Case: Search by one field (e.g., roll number).

3.2 Compound Index

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

- Order matters (prefix rule)

Example: E-commerce filtering (category + price).

3.3 Multikey Index

```
db.students.createIndex({subjects: 1})
```

Use Case: Array fields like subjects or tags.

3.4 Text Index

```
db.posts.createIndex({content: "text"})
db.posts.find({$text: {$search: "AI clustering"}})
```

Use Case: Search engines, blogs.

3.5 Geospatial Index

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

Use Case: Location-based services (Uber, Maps).

3.6 Hashed Index

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

- Good for equality queries
- Not suitable for range queries

3.7 Unique Index

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

Use Case: Prevent duplicate values.

3.8 Sparse Index

```
db.users.createIndex({phone: 1}, {sparse: true})
```

Use Case: Optional fields.

3.9 TTL Index

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

Use Case: Automatic data deletion (logs, cache).

3.10 Partial Index

```
db.users.createIndex(
  {age: 1},
  {partialFilterExpression: {status: "Active"}}
)
```

Use Case: Index subset of documents.

3.11 Wildcard Index

```
db.collection.createIndex({"$**": 1})
```

Use Case: Dynamic schema.

4 Summary Table

Index Type	Use Case	Example
Single	Simple queries	Search by name
Compound	Multi-field queries	name + age
Multikey	Arrays	subjects
Text	Search	blog content
Geospatial	Location	maps
Hashed	Equality	user_id
Unique	No duplicates	email
Sparse	Optional fields	phone
TTL	Auto delete	logs
Partial	Conditional	active users
Wildcard	Dynamic schema	JSON

5 Practical Insights

- Index improves read performance but slows writes
- Avoid excessive indexing
- Choose indexes based on query patterns

For IoT Systems:

- Use index on `device_id`
- Use index on `timestamp`
- Use compound index: {`device_id, timestamp`}
- Use TTL index for temporary data

6 Conclusion

Indexing is essential for performance optimization in MongoDB, but it must be used carefully. Proper selection of indexes based on application requirements ensures efficient querying without degrading write performance.