Lecture Notes:
NoSQL Databases and MongoDB Querying

Why NoSQL?

NoSQL stands for "Not Only SQL" and refers to a variety of databases that depart from the rigid structure of relational databases.

Limitation in SQL	NoSQL Alternative
Rigid schemas	Flexible, dynamic schemas
Vertical scaling (single server)	Horizontal scaling (distributed systems)
Poor performance on unstructured data	Native support for JSON, XML, nested documents
Complex joins for hierarchical data	Direct nesting or embedding

NoSQL arose to meet the demands of web-scale, cloud-native, and real-time data applications.

CAP Theorem

It states that in a distributed system, you can only guarantee two of the following three at any time:

- **Consistency**: Every node sees the same data at the same time
- **Availability**: Every request receives a response (success/failure)
- **Partition Tolerance**: The system works despite network partitioning

Type	Guarantees	Examples
CP (Consistency + Partition)	Data always consistent, may sacrifice availability	MongoDB, HBase
AP (Availability + Partition)	Always responds, data may be inconsistent temporarily	CouchDB, Cassandra
CA (Consistency + Availability)	Only possible if system is not distributed	Traditional RDBMS

In distributed systems, **partitioning is inevitable**, so trade-offs must be made between consistency and availability.

Types of NoSQL Databases and Use Cases

Type	/Description	Use Case Examples	Examples
Document	Store data as documents (JSON/BSON)	User profiles, product catalogues	MongoDB, CouchDB
Key-Value	Pairs of key and data blob	Caching, session storage	Redis, DynamoDB
Column-Family	Column-based storage structure	Time-series, log analytics	Apache Cassandra, HBase
Graph	Nodes and relationships	Social networks, recommendation systems	Neo4j, ArangoDB

NoSQL databases are **optimised for access patterns** rather than strict schema design.

MongoDB

MongoDB is a **document-oriented NoSQL database** that stores data in flexible, JSON-like documents called **BSON (Binary JSON)**.

MongoDB vs SQL

Concept	SQL	MongoDB
Table	Table	Collection
Row	Record	Document
Column	Field	Field
Schema	Fixed	Dynamic
Joins	Foreign keys	Embedding or $lookup
Query Language	SQL	Mongo Query Language (MQL)

Structure of MongoDB

Basic Components:

- **Database**: Top-level working database
- **Collection**: A set of documents (e.g., `students`)
- **Document**: JSON-like data unit with nested fields

```json
{
  "_id": "s123",
  "name": "Asha",
  "age": 22,
  "courses": ["SQL", "Python"],
```

```
  "address": {
    "city": "Mumbai",
    "pincode": 400001
  }
}
```

MongoDB documents can vary in structure, they don't need to follow a rigid schema.

MongoDB Data Types:

- String, Number, Boolean, Date, Array, Object, Null
- ObjectId (unique identifier generated automatically)

NoSQL databases like MongoDB emerged to meet the challenges of **scale, flexibility, and semi-structured data**. The **CAP theorem** explains trade-offs in distributed systems. MongoDB uses a **document model**, allowing for **schema-less, nested structures** that suit modern applications well. Knowing when and how to use NoSQL is critical for designing efficient data pipelines.

Here is a structured and progressive **lecture note** titled **"MongoDB Basics & CRUD Operations"**, which builds on your earlier lecture and introduces key concepts through practical demonstrations and examples.

CRUD Operations in MongoDB

CRUD = **Create**, **Read**, **Update**, **Delete**. These are the fundamental operations for working with data in any database system.

Create

Insert one or multiple documents

```javascript
db.students.insertOne({
  name: "Asha",
  age: 21,
  department: "Physics"
})
db.students.insertMany([
  { name: "Ravi", age: 24, department: "Maths" },
  { name: "Meena", age: 22, department: "Chemistry" }
])
```

Read

Querying Documents

```javascript
db.students.find({ department: "Physics" })
```

Update Documents

Update a document using `$set`:

```javascript
db.students.updateOne(
  { name: "Asha" },
  { $set: { age: 22 } }
)
```

Update many documents using `$inc`:

```javascript
db.students.updateMany(
  { department: "Maths" },
  { $inc: { age: 1 } }
)
```

We can use other update operators such as `$unset` or `$rename`.

Delete Documents

Delete One:

```javascript
db.students.deleteOne({ name: "Asha" })
```

Delete Many:

```javascript
db.students.deleteMany({ department: "Physics" })
```

Mongoimport Command

```shell
mongoimport --db <name> --collection <name> --type csv
--headerline --file <file.csv>
```

Comparison Operators

Operator	Description	Example
`$gt`	Greater than	`{ age: { $gt: 20 } }`
`$lt`	Less than	`{ age: { $lt: 30 } }`
`$gte`	Greater or equal	`{ age: { $gte: 25 } }`
`$lte`	Less or equal	`{ age: { $lte: 22 } }`
`$eq`	Equal	`{ age: { $eq: 21 } }`
`$ne`	Not equal	`{ age: { $ne: 21 } }`
`$in`	Matches any in list	`{ department: { $in: ["Maths", "Physics"] } }`

Counting & Sorting

Count:

```javascript
db.students.countDocuments({ age: { $lt: 23 } })
```

Sort:

```JavaScript
db.students.find().sort({ age: -1 })  // descending

db.students.find().sort({ name: 1 })  // ascending
```

Logical Operators

Operator	Description	Example
`$and`	All conditions must match	`{ $and: [ { age: { $gt: 20 } }, { age: { $lt: 25 } } ] }`
`$or`	At least one matches	`{ $or: [ { department: "Maths" }, { age: 21 } ] }`
`$not`	Negates condition	`{ age: { $not: { $gt: 23 } } }`

Logical operators are **composable** and can be nested inside queries.

Pattern Matching with Regular Expressions

Basic Regex Query:

```JavaScript
db.students.find({ name: { $regex: "^A" } })
// names starting with A
```

Common Patterns:

- `[]` : a set of characters
- `\` : special sequence
- `^` : starts with
- `$` : ends with
- `.` : any character except a newline

Examples:

Pattern	Matches
`^A`	Begins with "A"
`n$`	Ends with "n"
`^.{5}$`	Exactly 5 characters

Here is your structured **lecture note on Aggregations in MongoDB**, aligned with the rest of your MongoDB sequence, and crafted for clarity, demonstration, and application in DS/ML pipelines.

Aggregation

Aggregation is the process of transforming and computing over documents to produce summarised results.

MongoDB uses a **pipeline-based** approach with multiple stages (`$match`, `$group`, `$sort`, etc.) applied **in sequence**.

```JavaScript
db.collection.aggregate([
  { <stage1> },
  { <stage2> },
  ...
])
```

The `$group` Stage

The `$group` stage groups documents by a specified expression and can perform aggregations like `sum`, `avg`, `max`, `min`, `count`.

Example: Count students per department

```JavaScript
db.students.aggregate([
  { $group: { _id: "$department", total: { $sum: 1 } } } ])
```

Field Paths and the $ Prefix

To refer to a **field inside a document** during aggregation, prefix it with `$`.

`"$department"` → refers to the the value of the field "department"

This is different from just `"department"`, which is treated as a **string literal**.

Aggregation With `_id: null`

Use `_id: null` when you want to **aggregate across all documents** (i.e., no grouping).

Example: Average age of all students

```javascript
db.students.aggregate([
  { $group: { _id: null, averageAge: { $avg: "$age" } } }
])
```

Aggregation With `_id: <field>`

Use `_id: "$<field>"` to group by a field.

Example: Total students in each department

```javascript
db.students.aggregate([
  { $group: { _id: "$department", count: { $sum: 1 } } }
])
```

Filtering With `$match` and `$group`

Filtering Before Grouping

Efficient when you want to reduce data first.

```javascript
db.students.aggregate([
  { $match: { age: { $gt: 22 } } },
  { $group: { _id: "$department", total: { $sum: 1 } } }
])
```

Filtering After Grouping

Use another `$match` after `$group` to filter grouped results.

```javascript
db.students.aggregate([
  { $group: { _id: "$department", avgAge: { $avg: "$age" } } },
  { $match: { avgAge: { $gt: 23 } } }
])
```

Always remember: `$match` works on the shape of the current pipeline stage's output.

Note:

- MongoDB's **aggregation pipeline** lets you process data through sequential stages.
- Use `$_id: null` to compute overall stats; use `$_id: "$field"` to group by.
- Always prefix field names with `$` inside expressions.
- `$match` can be used **before** to filter input or **after** to filter grouped output.

Ask an LLM

- Explain MongoDB's document model to someone used to relational databases.
- What is the role of `_id`? Can it be changed? Why is it useful for distributed systems?
- What are the pros and cons of MongoDB's eventual consistency model?
- When would you *not* use MongoDB? What are its weaknesses for analytical workloads?
- How is MongoDB's handling of joins different from SQL? How does `$lookup` work internally?
- How does MongoDB ensure performance at scale without fixed schemas?
- If your dataset grows rapidly (GBs per day), what strategies would you apply in MongoDB to maintain performance?