Distributed MongoDB: Comprehensive Lecture Notes

Based on Course Material

Fall 2025

Contents

1 Introduction to MongoDB

1.1 History and Definition

MongoDB (derived from "hu**mongo**us") is an open-source, document-oriented database designed for scalability and developer agility. Unlike relational databases (RDBMS) that store data in tables and rows, MongoDB stores data in flexible, JSON-like documents.

Key Characteristics:

- **NoSQL Database:** It falls under the NoSQL umbrella, specifically as a document store.
- **High Performance:** Optimized for high read/write throughput.
- **High Availability:** Achieved through replica sets.
- **Automatic Scaling:** Achieved through horizontal scaling (sharding).

1.2 CAP Theorem Positioning

In the context of the CAP Theorem (Consistency, Availability, Partition Tolerance), MongoDB is generally classified as a **CP** (Consistency and Partition Tolerance) system.

- **Default Behavior:** During a network partition, MongoDB preserves consistency. If the primary node fails, the system may briefly reject writes (sacrificing availability) until a new primary is elected to ensure data remains consistent.
- **Tunable Consistency:** MongoDB allows developers to tune the balance between consistency and availability using *Read Preferences* and *Write Concerns*.

1.3 Motivations: Why MongoDB?

The development of MongoDB was driven by the limitations of traditional RDBMS in modern application development:

1. **Rigid Schema:** RDBMS requires predefined schemas. Changing schema in production (e.g., 'ALTER TABLE') is expensive and causes downtime. MongoDB offers a dynamic schema.
2. **Scalability Issues:** RDBMS are designed to scale vertically (buying bigger servers). MongoDB is designed to scale horizontally (adding more cheap servers).
3. **Object-Relational Impedance Mismatch:** In RDBMS, application objects must be decomposed into tables (normalization). MongoDB documents map naturally to objects in code (Java, JavaScript, Python), eliminating complex joins.

2 Data Model

2.1 Documents and Collections

- **Document:** The basic unit of data in MongoDB. It corresponds to a row in RDBMS but allows nested structures (arrays and sub-documents).
- **Collection:** A grouping of documents. It corresponds to a table in RDBMS. However, unlike tables, collections do not enforce a uniform schema (though schema validation rules can be applied if necessary).

2.2 JSON vs. BSON

MongoDB uses JSON (JavaScript Object Notation) for the user interface but stores data internally as BSON (Binary JSON).

2.2.1 JSON (JavaScript Object Notation)

- **Pros:** Human-readable, text-based, standard data interchange format.
- **Cons:** Text parsing is slow; limited data types (e.g., no distinction between integer and float, no native date type).

2.2.2 BSON (Binary JSON)

BSON is a binary-encoded serialization of JSON-like documents.

- **Efficiency:** Designed to be lightweight, traversable, and efficient to encode/decode.
- **Traversability:** BSON stores length prefixes for documents and arrays, allowing MongoDB to scan and skip fields without parsing the entire document.
- **Rich Type Support:** Supports data types not found in JSON, such as:
  - `Date` (64-bit integer milliseconds)
  - `BinData` (Raw binary data, e.g., images)
  - `ObjectId` (Unique primary keys)
  - `Decimal128` (High-precision math)
  - `Int32` and `Int64`
- **Constraint:** The maximum BSON document size is **16 MB**. This prevents excessive RAM usage per document.

2.3 The `_id` Field

Every MongoDB document requires a primary key stored in the `_id` field.

- **Properties:** Unique within the collection, immutable.
- **Default Type:** `ObjectId`. It is a 12-byte value consisting of:
  - 4-byte timestamp (provides implicit sorting by creation time).
  - 5-byte random value (machine/process identifier).
  - 3-byte incrementing counter (ensures uniqueness within the second).

3 Schema Design: Embedding vs. Linking

One of the most critical decisions in MongoDB schema design is handling relationships.

3.1 One-to-One Relationships

- **Embedding:** Preferred when the data is accessed together.
- **Example:** A user and their address. Instead of a separate Address table, store the address object inside the User document.

3.2 One-to-Many Relationships

1. **Embedding (Denormalization):**
   - Use when the "many" side is small and tightly coupled.
   - *Example:* A Book document embedding an array of Author objects.
   - *Pros:* Single read operation retrieves all data (fast). No joins required.
   - *Cons:* Document size grows; potential data duplication if the embedded entity changes often.
2. **Linking (Normalization/Referencing):**
   - Use when the "many" side is large (unbounded) or the related data is frequently updated independently.
   - *Example:* A Publisher document referenced by ID in thousands of Book documents.
   - *Pros:* Reduces duplication; keeps document sizes small.
   - *Cons:* Requires application-level joins or `$lookup` (slower).

4 Querying and Operators

MongoDB provides a rich query language using JSON-based syntax.

4.1 Logical Operators

- `$and`: Logical AND (implicit in comma-separated query documents).
- `$or`: Logical OR.
- `$not`: Inverts the effect of a query expression.
- `$nor`: Logical NOR (returns documents that fail both conditions).

4.2 Comparison Operators

- `$eq`, `$ne`: Equal, Not Equal.
- `$gt`, `$gte`: Greater than, Greater than or Equal.
- `$lt`, `$lte`: Less than, Less than or Equal.
- `$in`, `$nin`: Match any value in an array.

4.3 Update Operators

MongoDB updates are atomic at the document level.

- **Fields:**
  - `$set`: Sets the value of a field.
  - `$inc`: Increments a numerical field.
  - `$unset`: Removes a field.
  - `$rename`: Renames a field.
- **Arrays:**

- `$push`: Adds an element to an array.
- `$addToSet`: Adds an element only if it doesn't exist (unique).
- `$pull`: Removes items matching a condition.
- `$pop`: Removes the first or last item.

5 Indexing

Indexes are specialized data structures that store a small portion of the data set in an easy-to-traverse form (typically B-Trees). Without indexes, MongoDB must perform a **collection scan** (scan every document), which is inefficient for large datasets.

5.1 Types of Indexes

1. **Single Field Index:** Sorts based on a single field (e.g., `{ score: 1 }`). Supports simple queries and range queries on that field.
2. **Compound Index:** Indexes multiple fields (e.g., `{ userid: 1, score: -1 }`). The order of fields matters (ESR Rule: Equality, Sort, Range).
3. **Multikey Index:** Automatically created when indexing an array field. An index entry is created for *each* element in the array.
4. **Text Index:** Supports keyword search on string content.
5. **Geospatial Index:** Supports location-based queries (e.g., `$near`, `$geoWithin`).

6 Aggregation Framework

The Aggregation Framework allows for advanced data processing and analysis within MongoDB, similar to SQL's `GROUP BY` or Linux pipes.

6.1 The Pipeline Concept

Documents pass through a multi-stage pipeline. Each stage transforms the documents and passes the results to the next stage.

6.2 Common Stages

- **`$match`:** Filters documents (like SQL `WHERE`). Efficiently uses indexes if placed early.
- **`$group`:** Groups documents by a specified identifier key and applies accumulator expressions (e.g., `$sum`, `$avg`) to compute results for each group.
- **`$project`:** Reshapes documents (selects, renames, or creates computed fields).
- **`$sort`:** Sorts the stream of documents.
- **`$limit` / `$skip`:** Controls the number of documents passed.
- **`$lookup`:** Performs a left-outer join with another collection.
- **`$unwind`:** Deconstructs an array field from the input documents to output a document for each element (normalizing the array).

6.3 Map-Reduce (Legacy)

Before the Aggregation Framework, Map-Reduce was used for complex batch processing using JavaScript functions.

- **Phases:** Map (emit key-value pairs) → Reduce (combine values for a key).
- **Drawbacks:** Slower (runs in JS engine, not native C++), harder to debug.
- **Note:** Aggregation Pipeline is now preferred for almost all use cases.

7 Distributed MongoDB: Replication

Replication provides **Redundancy** and **High Availability**.

7.1 Replica Set Architecture

A Replica Set typically consists of three or more nodes:

1. **Primary:** The only node that accepts writes. It records changes in the **Oplog** (Operations Log).
2. **Secondary:** Replicates the Primary's Oplog and applies the operations asynchronously to maintain an identical dataset.
3. **Arbiter (Optional):** A lightweight node that holds no data. It exists solely to participate in elections to ensure a quorum.

7.2 Failover Process

If the Primary fails (stops sending heartbeats):

- Secondary nodes detect the failure.
- An election is triggered.
- A new Primary is elected based on which node has the most up-to-date data.
- Drivers automatically reconnect to the new Primary.

7.3 Read and Write Concerns

- **Write Concern (`w`):** Controls durability.
  - `w: 1`: Acknowledge receipt by Primary (default).
  - `w: "majority"`: Acknowledge only after writing to a majority of nodes. Prevents rollbacks.
- **Read Preference:** Controls where the client reads from.
  - `primary`: Always read from Primary (Strong consistency).
  - `secondary`: Read from Secondary (Eventual consistency, useful for analytics).

8 Distributed MongoDB: Sharding

Sharding provides **Horizontal Scalability** by partitioning data across multiple machines.

8.1 When to Shard?

Shard when the working set exceeds RAM, the disk I/O limits of a single server are reached, or the storage capacity of a single server is exceeded.

8.2 Architecture Components

- **Shards:** Replica sets that hold a subset of the total data.
- **Config Servers:** A replica set that stores metadata (mapping of data chunks to shards).
- **Mongos (Query Router):** Acts as an interface between the client and the sharded cluster. It routes queries to the appropriate shard(s) based on the Shard Key.

8.3 Shard Keys and Balancing

- **Shard Key:** A field chosen to partition the data.
- **Chunks:** Data is split into ranges called chunks (default 64MB).
- **Balancer:** A background process that migrates chunks between shards to ensure even distribution.

8.4 Hotspotting

Choosing a poor shard key leads to hotspots, where one shard receives the majority of writes, negating the benefits of sharding.

- **Monotonically Increasing Key (e.g., Timestamp):** Bad. All new inserts go to the "last" chunk on a single shard.
- **Hashed Sharding:** Good. Hashes the key values to distribute writes randomly, avoiding hotspots.