

DBS_MidSem2025

Exported on: 2/22/2026, 7:49:35 PM

User

Explain Q5) (a)

Gemini

In MongoDB, **Scale-Out** (also known as horizontal scaling) is primarily achieved through a process called **Sharding**.

While scaling-up (vertical scaling) involves adding more power (CPU, RAM) to a single server, scaling-out involves distributing data across multiple machines to handle increased traffic and massive datasets.

How Sharding Works

Sharding breaks down large collections into smaller chunks and distributes them across multiple servers called **shards**. Here are the key components that make this happen:

1. Shards

Each shard is a separate server (or more commonly, a **replica set**) that holds a subset of the total data. Because each shard is its own entity, the system's total storage and throughput capacity grow as you add more shards.

2. The Shard Key

To decide which document goes to which shard, MongoDB uses a **Shard Key**. This is a specific field (or fields) in your documents. MongoDB uses two main strategies for distribution:

- **Ranged Sharding:** Data is divided based on ranges of the shard key values. This is great for range-based queries but can lead to "hotspots" if data is written sequentially.
- **Hashed Sharding:** A hash of the shard key is used to distribute data. This ensures a very even distribution of data across shards, which is better for heavy write loads.

3. Mongos (The Query Router)

Since the data is scattered, the application needs to know where to find it. **Mongos** acts as a traffic cop. It receives requests from the application, consults the config servers to see which shard has the data, and routes the query accordingly.

4. Config Servers

These servers store the **metadata** for the cluster. They keep track of which chunks of data live on which shards. The `mongos` routers use this metadata to manage the cluster's state.

The "Balancing" Act

MongoDB automatically manages data distribution through a background process called the **Balancer**. If one shard becomes significantly fuller than others, the balancer migrates "chunks" of data from the overloaded shard to a less-loaded one without any manual intervention or downtime.

Would you like me to explain part (b) regarding Aggregation pipelines, or perhaps dive deeper into how to choose a good Shard Key?

User

Explain Q5) (b)

Gemini

In MongoDB, **Aggregation** is a powerful framework used to process data records and return computed results. Think of it as the MongoDB equivalent of the `GROUP BY` and `JOIN` clauses in SQL, but much more flexible.

It essentially allows you to "pipe" your data through various stages to transform it, filter it, and eventually output a summary.

The Aggregation Pipeline

The most common way to perform aggregation is through the **Aggregation Pipeline**. In this model, documents from a collection enter a multi-stage pipeline that transforms the documents into aggregated results.

Each stage performs a specific operation:

- `$match` : Filters the documents (similar to a `find` query).
- `$group` : Groups documents by a specified identifier and applies "accumulator" functions (like sum, average, or count).
- `$sort` : Orders the resulting documents.
- `$project` : Reshapes the documents (adding, removing, or renaming fields).
- `$limit / $skip` : Controls the number of results.

Example: If you have a collection of "Sales," you could `$match` only the sales from 2024, `$group` them by "Product ID," and then calculate the total revenue using `$sum`.

Why is Aggregation Useful?

The usefulness of the aggregation framework stems from its ability to handle complex data processing directly on the database server:

1. **Efficiency:** It processes data where it lives. Instead of downloading millions of raw documents to your application to calculate a total, MongoDB does the math and sends you only the final result.
2. **Versatility:** It can perform complex transformations, such as joining data from different collections (`$lookup`), unwinding arrays into individual documents (`$unwind`), and performing mathematical or string operations.
3. **Real-time Analytics:** Because it is highly optimized, it can be used to generate real-time reports, dashboards, and data summaries without needing a separate ETL (Extract, Transform, Load) process.
4. **Pipeline Modularization:** You can easily add or remove stages to debug your data processing flow step-by-step.

Since you're working through these questions, would you like me to explain part (c) regarding "Covered Queries" next?

User

Explain Q5) (c)

Gemini

In MongoDB, a **Covered Query** is a query that can be satisfied **entirely by using an index**, without ever needing to look at the actual documents stored on the disk.

To understand why this is a "gold standard" for performance, you have to look at how MongoDB usually fetches data.

How it Works

Normally, when you run a query, MongoDB:

1. Uses an **Index** to find the location of the documents.
2. Follows a pointer to the **Collection** (the actual data on disk) to fetch the requested fields.

In a **Covered Query**, the index itself contains all the fields requested in the query and the projection. Because the index is usually stored in RAM, MongoDB can return the result immediately.

Criteria for a Covered Query

For a query to be "covered," it must meet two strict conditions:

1. All fields used in the **query criteria** (the filter) are part of the index.
2. All fields returned in the **projection** (the output) are also part of that same index.
3. The `_id` field must be explicitly excluded from the projection (unless it is part of the index), because it is included by default.

Example: If you have an index on `{ name: 1, age: 1 }`, this query is covered: `db.users.find({ name: "Samyak" }, { name: 1, age: 1, _id: 0 })`

Why is it Important?

The importance of covered queries lies in **extreme optimization**:

- **Zero Disk I/O:** Reading from an index (which stays in memory) is significantly faster than fetching documents from the disk or the filesystem cache.
- **Reduced Memory Pressure:** By only touching the index, the database doesn't have to load large documents into the "WiredTiger" cache, leaving more room for other operations.
- **Lower Latency:** It provides the fastest possible response time for a read operation.
- **Scalability:** As your data grows to terabytes, covered queries ensure that common lookups remain near-instant, regardless of how large the underlying documents are.

That covers all of Q5! Since you're also looking at Q6, would you like me to start with the differences between Signature-based and Anomaly-based IDS?

User

Explain Q6)

Gemini

Since you're diving into Intrusion Detection Systems (IDS) for Q6, it's helpful to look at how these systems decide what constitutes a "threat." Here is the breakdown of the entire question:

a) Signature-based vs. Anomaly-based IDS

The primary difference lies in **how** they identify a threat:

Feature	Signature-based (Misuse)	Anomaly-based
Detection Logic	Compares traffic against a database of known attack patterns (signatures).	Compares current behavior against a "baseline" of normal activity.
New Threats	Ineffective against Zero-day attacks (unseen threats).	Effective at catching new or evolved threats.
False Positives	Very low; if it matches a signature, it's almost certainly an attack.	High; any unusual but legitimate activity triggers an alarm.
Maintenance	Requires constant updates to the signature database.	Requires a "training period" to learn the network's normal behavior.

 Export to Sheets

**b) Ways to improve Anomaly-based IDS efficiency**

Anomaly detection is powerful but "noisy." You can improve it by:

- Adaptive Learning:** Use Machine Learning algorithms that continuously update the "normal" baseline. This prevents the system from flagging legitimate network growth or architectural changes as attacks.
- Multi-Dimensional Profiling:** Instead of just looking at one metric (like traffic volume), correlate multiple factors—such as protocol types, source/destination ports, and time of day—to reduce "false alarms."

c) Is there anything interesting about False Positives?

Yes, there is a concept called the **Base Rate Fallacy**. In a large network, actual attacks are extremely rare compared to legitimate traffic (the "base rate"). Even an IDS with a 99% accuracy rate can produce so many false positives that the security team suffers from "**Alert Fatigue**." If 99% of your 10,000 daily alerts are false, the one

real attack is easily missed. This makes the *Reduction of False Positives* arguably more important than the *Detection Rate* in high-traffic environments.

d) Issues in testing Intrusion Detection Systems

Testing an IDS is notoriously difficult for several reasons:

- **Lack of Realistic Datasets:** Most public datasets (like KDD Cup '99) are outdated and don't reflect modern encrypted traffic or sophisticated APTs (Advanced Persistent Threats).
- **Environment Scaling:** An IDS that works perfectly in a small lab may fail or drop packets when tested against a high-speed backbone network.
- **Traffic Encryption:** As more traffic becomes encrypted (HTTPS/TLS), the IDS cannot "see" the payload, making it much harder to test its deep-packet inspection capabilities.
- **The "Moving Target" Problem:** Attackers constantly change their methods to evade detection, meaning a test result from today might be irrelevant by tomorrow.

Would you like me to go into more detail on how Machine Learning is specifically used to solve the efficiency issues in part (b)?