MongoDB, Redis를 연계한 트래픽 처리

hwpark@wiken.co.kr

NoSQL

What is NoSQL?

- NoSQL databases are designed to address specific requirements such as scalability, high availability, and flexible data models.
- Basically a large serialized object store
- In general, doesn't support complicated queries
  - Joins generally not available
- Schema is no longer rigid but flexible
  - Recommends denormalization
- Designed to be distributed (cloud-scale) out of the box
- Drops ACID requirements
  - Any database can answer any query
  - Any write query can operate against any database and will "eventually" propagate to other distributed servers

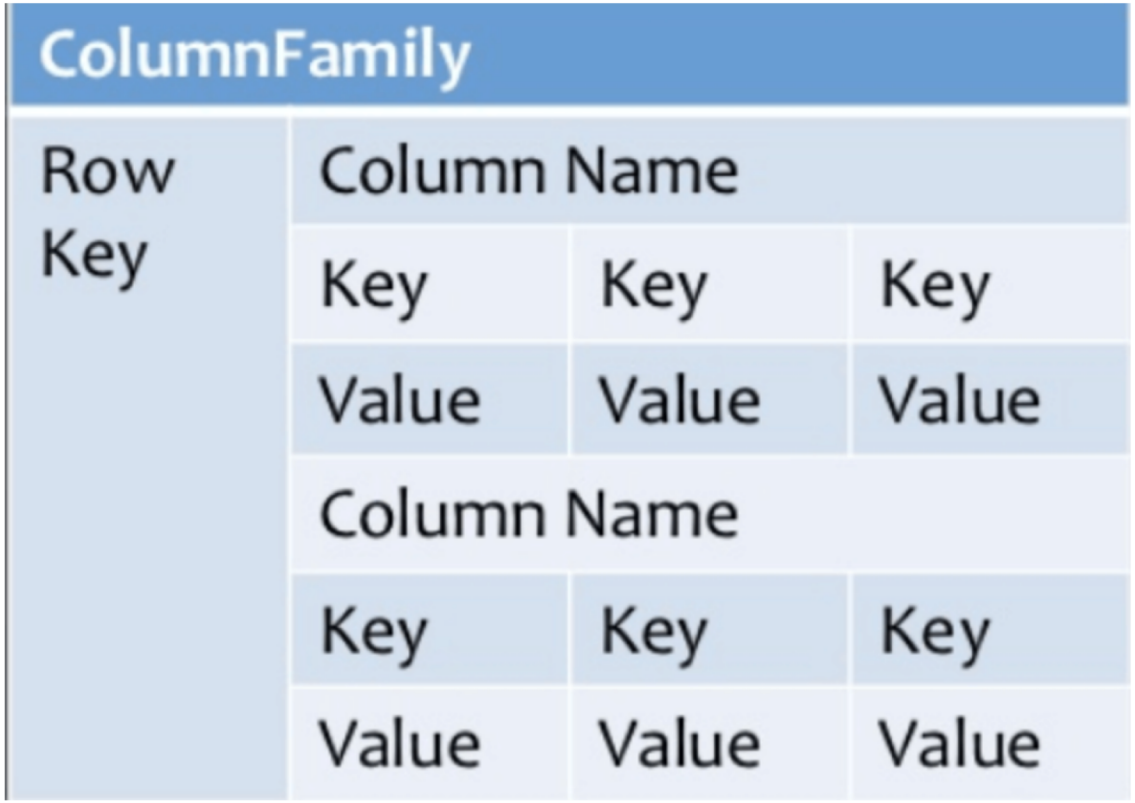
Types of NoSQL Databases

- Key-Value Store
  - A key that refers to a payload
  - MemcacheDB, Redis, Voldemorte

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

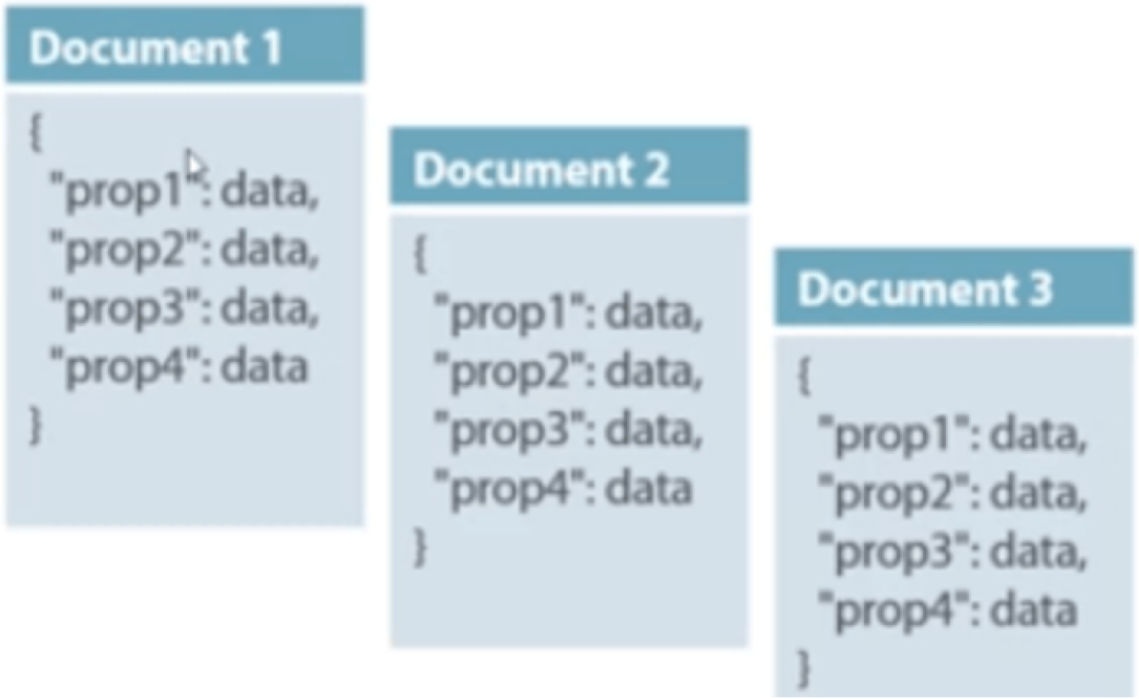
Types of NoSQL Databases

- ColumnStore
  - Column data is saved together, as opposed to row data
  - Super useful for data analytics
  - Hbase, Bigtable, Cassandra



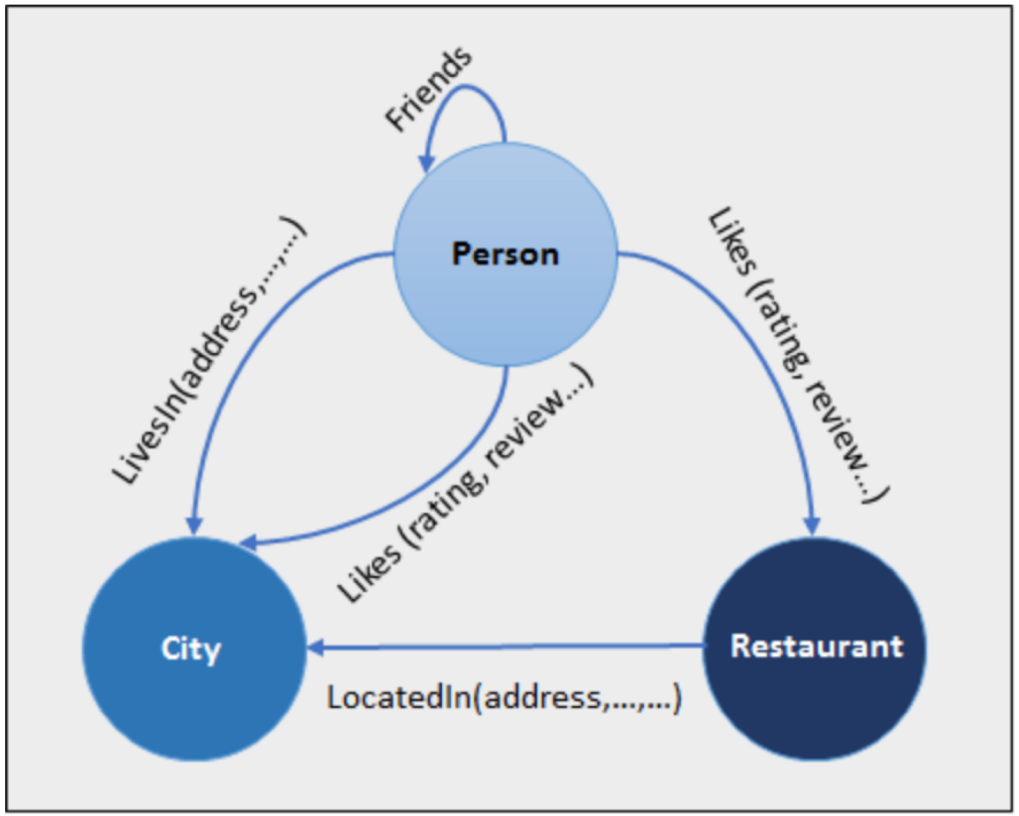
Types of NoSQL Databases

- Document/XML/ObjectStore
  - Key (and possibly other indexes) point at a serialized object
  - DB can operate against values in document
  - MongoDB, CouchDB, Elasticsearch

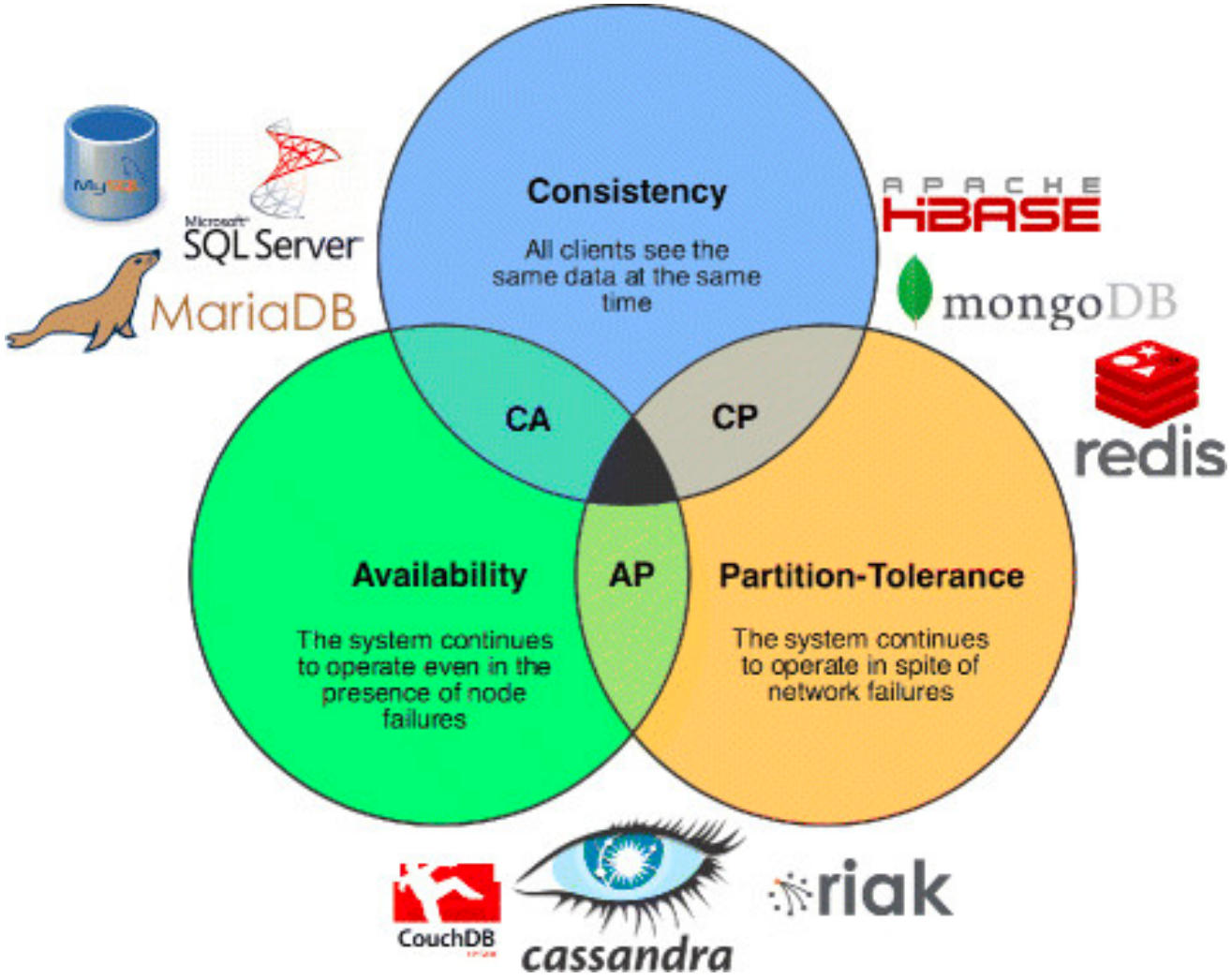


Types of NoSQL Databases

- GraphStore
  - Nodes are stored independently, and the relationship between nodes (edges) are stored with data
  - Neo4j



Types of NoSQL - CAP Theorem Tradeoff



Introduction to MongoDB

- Understanding and Implementing MongoDB

- Welcome to our course "Understanding and Implementing MongoDB & Redis". This module is designed to introduce you to MongoDB, its primary features, use-cases, and how it fits into the modern data management landscape.

The Big Picture

- High-Level Data Management:
  - Data management encompasses storage, processing, querying, scalability, and analysis.
  - Consider these aspects when choosing a database system.
- Traditional Databases vs NoSQL:
  - Traditional relational databases use a table structure and SQL for data access.
  - They offer strong ACID guarantees but may lack scalability and flexibility.
  - NoSQL databases prioritize scalability, speed, and flexibility over some ACID properties.
  - NoSQL is well-suited for large data volumes and complex data structures.
- Need for MongoDB:
  - MongoDB is a NoSQL database with a flexible document-oriented model.
  - It provides scalability through sharding and durability with built-in replication.
  - MongoDB is ideal for content management, user data management, and real-time analytics.

Overview

- **Overview of MongoDB**: We will delve into what MongoDB is, what makes it unique, and why it has gained such popularity in the tech world.
- **MongoDB Use Cases**: We'll then explore a variety of use cases for MongoDB, spanning different industries and applications, to understand where and why it excels.
- **MongoDB Data Model**: We will study MongoDB's unique data model and see how it contributes to MongoDB's flexibility and scalability.
- **Creating and Querying MongoDB Documents**: Finally, we will get hands-on and learn about creating and querying MongoDB documents, the basic operations that form the crux of MongoDB usage

MongoDB Common Terminology

- **_id** – Document's primary key
  - This is a field required in every MongoDB document
  - Unique value in the MongoDB document
  - Automatically created if omitted
- **Collection** – This is a grouping of MongoDB documents
  - Equivalent of a table in other RDBMS such as Oracle or MS SQL
  - A collection exists within a single database
  - Collections don't enforce any sort of structure
- **Cursor** – Pointer to the result set of a query
  - Clients can iterate through a cursor to retrieve results

MongoDB Common Terminology

- **Database** – Container for collections
  - Each database gets its own set of files on the file system
  - A MongoDB server can store multiple databases
- **Document** - Equivalent to a Row in RDBMS
  - A record in a MongoDB collection is basically called a document
  - The document, in turn, will consist of field name and values.
- **Field** - A name-value pair in a document
  - A document has zero or more fields
  - Fields are analogous to columns in relational databases.
- **JSON** – This is known as JavaScript Object Notation.
  - Human-readable, plain text format for expressing structured data

What is MongoDB?

- **Definition**:
  - MongoDB is an open-source, NoSQL database that uses a document-oriented data model.
  - Unlike relational databases, MongoDB uses BSON (Binary JSON)-like documents and allows for greater flexibility and scalability.

What is MongoDB?

- **Key Features**: Some of the primary features of MongoDB include:
  - **Document-oriented Storage**: Data in MongoDB is stored in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.
  - **Ad hoc Queries**: MongoDB supports a rich set of query operations, including field, range queries, and regular expression searches.
  - **Indexing**: You can index any field in a MongoDB document, significantly improving search performance.
  - **Replication and High Availability**: MongoDB uses native replication for high availability, including automatic failover.
  - **Horizontal Scaling**: MongoDB can be scaled across platforms, allowing for the management of large data sets.
- **Unique Selling Proposition**

Why MongoDB?

- Key Advantages:
  - Schema-less: MongoDB allows collections to hold different documents with varying fields, content, and sizes.
  - Ease of scale-out: Sharding data across multiple servers makes scaling out in MongoDB simple.
  - Rich queries and analytics: MongoDB supports advanced queries, including joins and secondary indexes.
  - Speed: MongoDB uses internal memory for storing the working set, enabling faster data access.

MongoDB Use Cases

- **Case Studies**: Various organizations across different sectors have successfully integrated MongoDB into their operations:
  - **Healthcare**: MyHealthDirect, a digital care coordination company, leveraged MongoDB to unify patient data and build an event-driven healthcare system.
  - **Finance**: Square, a mobile payment company, used MongoDB to manage its enormous database with high demand for write loads.
  - **Retail**: Urban Outfitters uses MongoDB to keep track of its diverse set of products, customers, and recommendations, all in real time.
- **Success Stories:**
  - **Adobe**: Adobe Experience Manager uses MongoDB to store, manage, and access billions of digital assets, providing a seamless customer experience.
  - **InVision**: InVision built its design prototyping tool using MongoDB due to its ability to handle large and complex data sets and schemas.

MongoDB Data Model

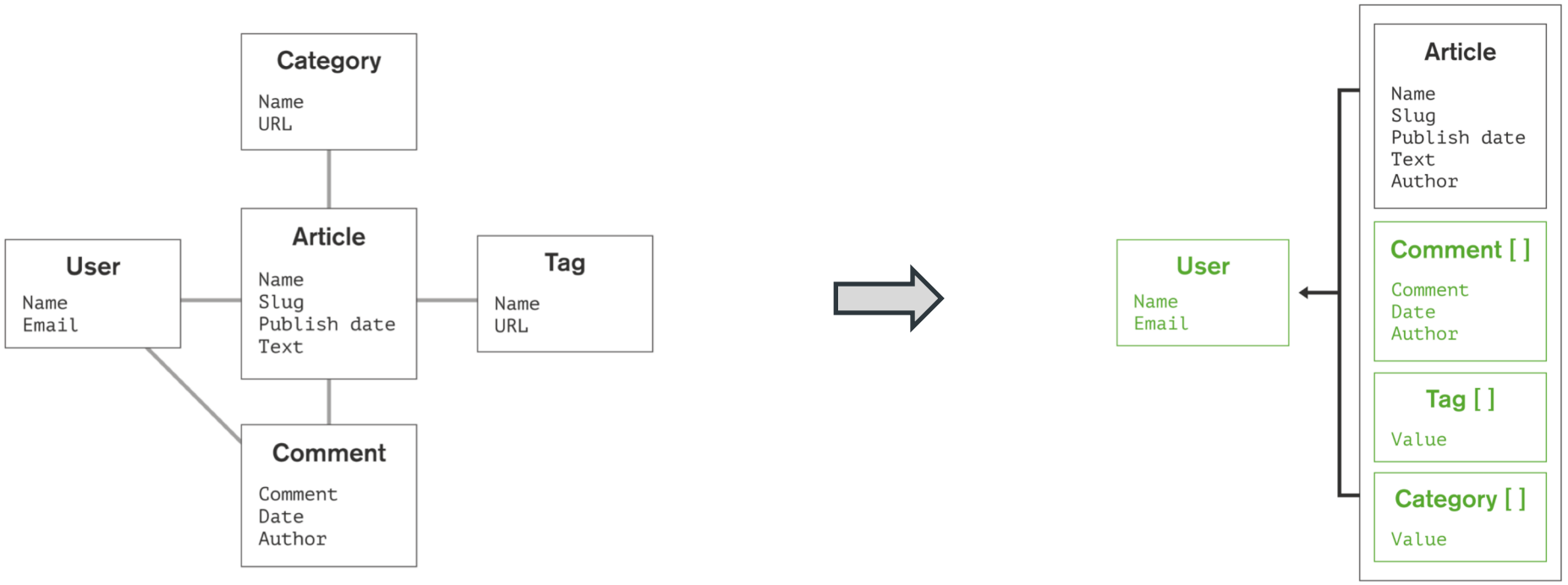
- **Document Model**: MongoDB uses a flexible document model that can accommodate complex hierarchical relationships and an array of data types.

- **BSON Format**: MongoDB stores data in a format known as BSON (Binary JSON). BSON extends the JSON model to provide additional data types, ordered fields, and to be efficient for encoding and decoding within different languages.

- **Collections and Databases**: in MongoDB, a collection is a group of MongoDB Documents and is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema and documents within a collection can have different fields.

MongoDB Data Model – Data as Documents

- MongoDB stores data as documents in a binary representation called BSON (Binary JSON).
- The BSON encoding extends JSON to include additional types
  - int, long, date, and floating point
  - BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays, binary data and sub-documents.



Creating MongoDB Documents

- **Creating Documents**: MongoDB stores structured data as BSON documents, i.e., data structures composed of field and value pairs
- **InsertOne() and InsertMany() Methods**: MongoDB provides two primary methods for inserting data into a collection: insertOne() and insertMany()

```
db.collection('users').insertOne({
    name: 'John Doe',
    age: 28,
    email: 'johndoe@example.com'
})
```

```
db.collection('users').insertMany([{
    name: 'John Doe',
    age: 28,
    email: 'johndoe@example.com'
}, {
    name: 'Jane Doe',
    age: 27,
    email: 'janedoe@example.com'
}])
```

Querying MongoDB Documents

- **Find() Method**: MongoDB uses the find() method to query documents within a collection.

```
db.collection('users').find()
db.collection('users').find({ age: 28 })
```

- **Query Types**: MongoDB offers a variety of query types, including exact match, range, and logical AND/OR operations.

```
// Exact Match
db.collection('users').find({ name: 'John Doe' })
// Range
db.collection('users').find({ age: { $gt: 20, $lt: 30 } })
// Logical AND
db.collection('users').find({ $and: [{ age: { $gt: 20 } }, { age: { $lt: 30 } }] })
```

Querying MongoDB Documents

- **Projections**: Projections in MongoDB are a way to specify which fields you want returned in the output of your find() queries.

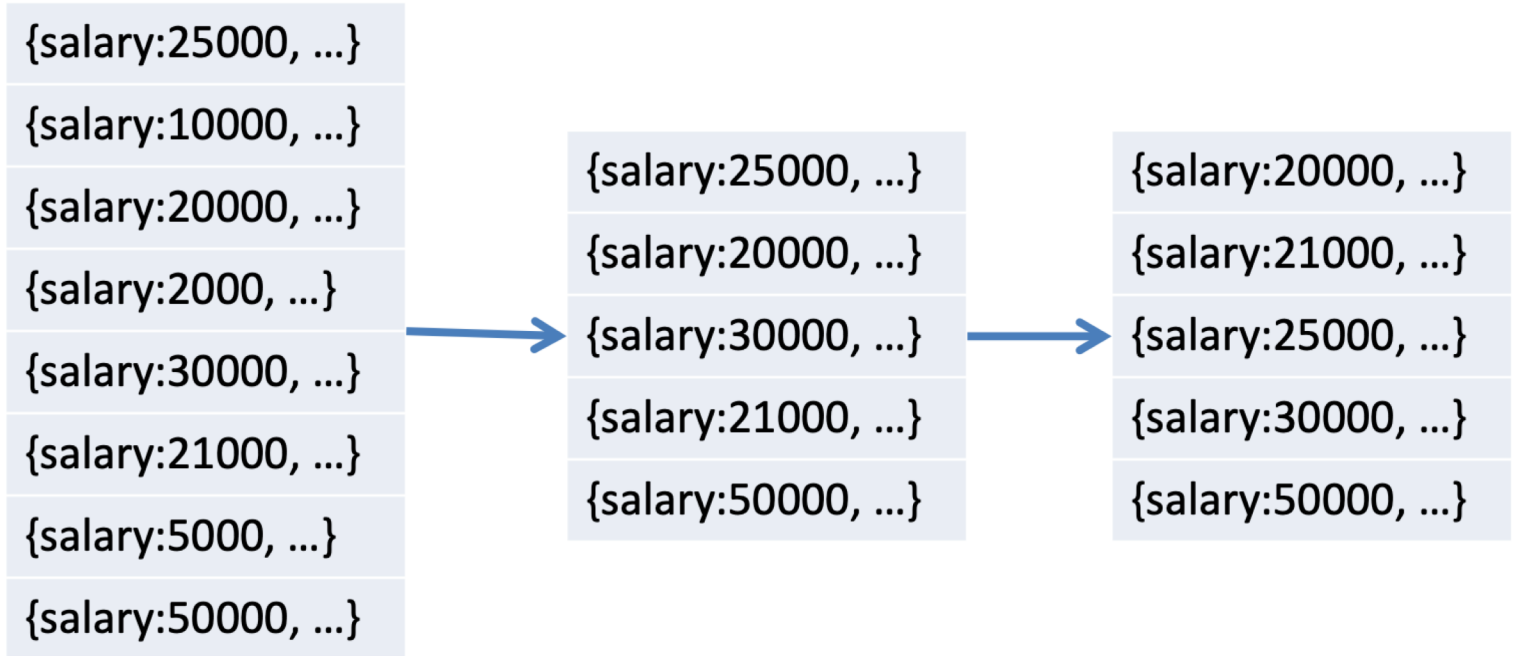
```
db.collection('users').find({}, { name: 1, email: 1 })  // Only returns the 'name' and 'email' fields
```

Query Example – find()

- Query all employee names with salary greater than 18000 sorted in ascending order

```
db.users.find({salary:{$gt:18000}, {name:1}}).sort({salary:1})
```

Collection | Condition | Projection | Modifier



Query Example – insert()

- Insert a row entry for new employee Sally

```
db.users.insert({
        name: “sally”,
        salary: 15000,
        designation: “MTS”,
        teams: [ “cluster-management” ]
        })
```

Query Example – Update()

- All employees with salary greater than 18000 get a designation of Executive

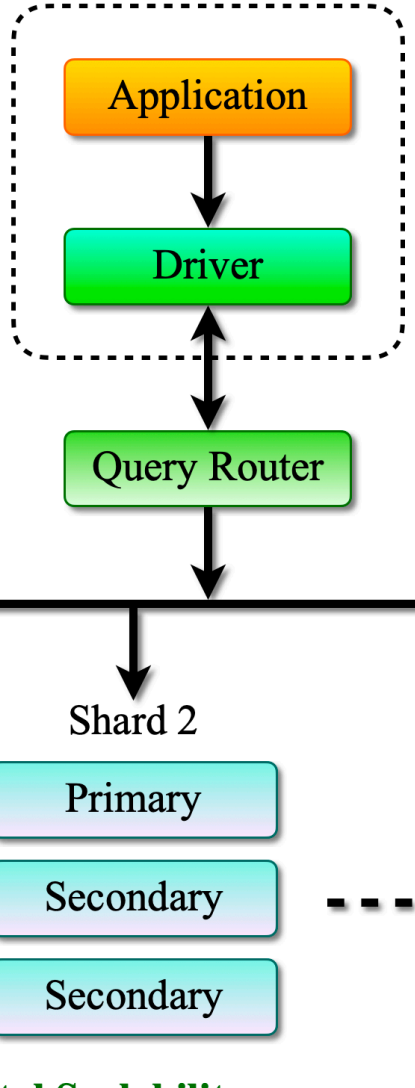
```
                    db.users.update(
Update Criteria         {salary:{$gt:18000}},
Update Action           {$set: {designation: "Manager"}},
Update Option           {multi: true}
                    )
```

- Multi option allows multiple document update

MongoDB Architecture

2. Native language drivers

```
db.customer.insert({…})
db.customer.find({
      name: “Raj Kumar”})
```



1. Dynamic Document Schema

```
{ name: “Raj Kumar”,
  date: “2011-07-18”,
  address: “2nd Floor.”,
  phone: [
     { home: 1234567890},
     { mobile: 1234567890} ]
}
```

3. High availability

- Replica sets

4. High Performance

- Data locality
- Rich Indexes
- RAM

5. Horizontal Scalability

- Sharding

MongoDB Architecture

- **Cluster Architecture**: MongoDB can be run in a clustered environment that offers high availability and horizontal scalability. The cluster architecture includes MongoDB instances running as a part of replica sets or sharded clusters.

- **Replica Set**: A replica set in MongoDB is a group of MongoDB servers that maintain the same data set. Replica sets provide redundancy and high availability, which are achieved by replicating data across multiple servers

```js
// Configuring a replica set
mongod --port 27017 --dbpath /data/db --replSet rs0
```

- **Sharding**: Sharding is a method for distributing data across multiple machines.

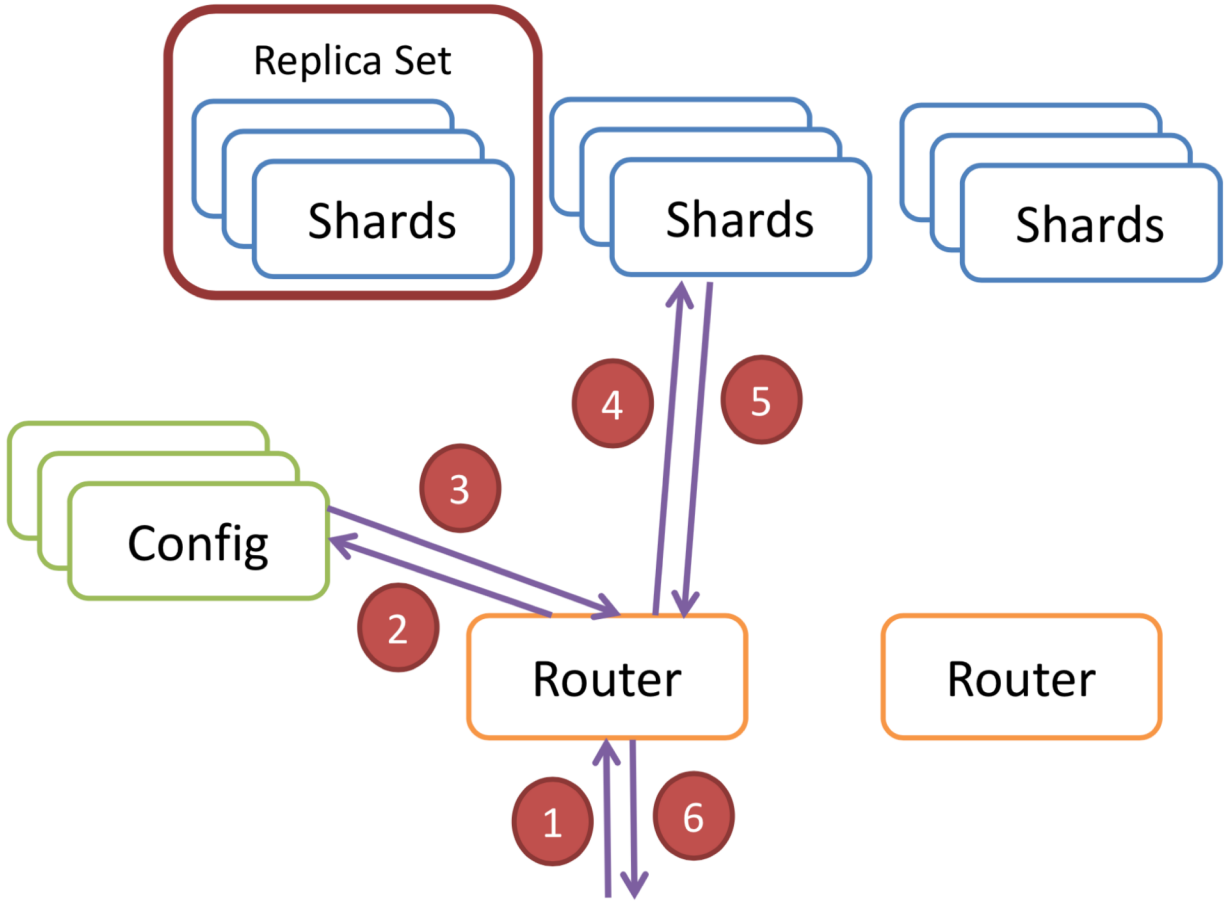
Replica Set

- A Replica Set in MongoDB is a group of MongoDB instances that host the same data set.
- In a replica set, one node is a primary node that receives all write operations while other nodes, known as secondary nodes, replicate the primary's oplog and apply the operations to their data sets.

- **Components of Replica Set**:
  - The primary node: Processes all write operations.
  - Secondary nodes: Mirror the primary node to provide redundancy and backup.
  - Arbiter node (optional): Participates in elections to help avoid a tie.
- **Advantages of Replica Set**:
  - High data availability
  - Data redundancy
  - Read scalability

Sharding

- Sharding is a database architecture that partitions, or shards, data across multiple servers, providing a method for horizontal scaling.



- Shards: mongod servers store the data
- Multiple shard servers form a *replica set*
- Replica set maintain same replica of data
- Routers: mongos interfaces with clients and routers operations to appropriate shards
- Config: Stores collection level metadata.

MongoDB indexing and Aggregation

MongoDB Indexing and Aggregation

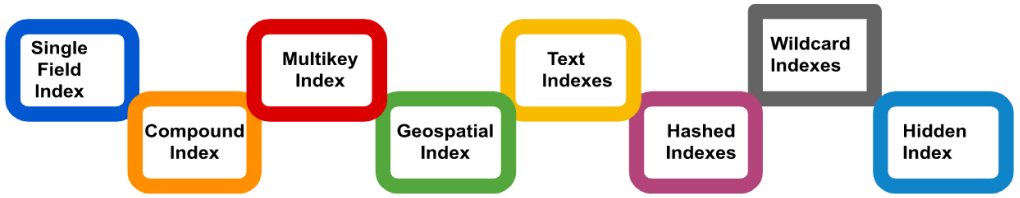
- **Indexing Basics:**
  - Introduction to the concept and role of indexing in MongoDB.
- **Benefits of Indexing:**
  - Dive into how indexing enhances query performance and write speeds.
- **Creating and Using Indexes:**
  - Learn practical skills for creating, using, and managing indexes.
- **Introduction to MongoDB Aggregation Framework:**
  - Understand how MongoDB allows sophisticated data manipulation beyond basic CRUD operations.

Indexing

- **Indexing is like a database 'table of contents'.**
  - Swiftly locates data, eliminating the need for full table scans.
- **Why Indexing in MongoDB?**
  - Crucial for handling vast volumes of data.
  - Significantly improves performance by limiting search space.
- **Types of Indexes in MongoDB:**
  - **Single Field**
  - **Compound Index**
  - **Multikey Index**
  - **Text Index**.
  - **2D Index**

Types of Indexes in MongoDB

- **Single Field Index**
  - Indexing on a single field, including embedded or top-level fields.
  - Enhances efficiency for searches on that field.
- **Compound Index**
  - Indexing on multiple fields.
  - Field order is essential; influences sorting and querying.
- **Multikey Index**
  - Ideal for fields containing arrays of values.
  - Each element of the array is indexed, facilitating specific value searches.
- **Text Index**
  - Suitable for working with text strings.
  - Enables specific text searches within large text blocks
- **Geospatial (2D) Index**
  - Suitable for geospatial queries involving 2D geometri
  - Enhances efficiency of geographical searches.



Single Field Index

- **Single Field Index in MongoDB**
  - Quick access to data based on one field in the document collection.
  - Analogous to a book's index leading to specific page numbers.
- **Creating a Single Field Index**
  - Use createIndex() function.
  - Specify the field to be indexed.
  - Example: db.users.createIndex({name: 1}).
    - '1' indicates ascending order, '-1' for descending.
- **Use Cases & Performance Benefits**
  - Ideal for queries searching values on a specific field.
  - Reduces the number of documents MongoDB scans, improving performance.
  - Considerations: Indexes require storage space.

Compound Index

- **Compound Indexes in MongoDB**
  - - Index multiple fields together for quicker multi-field search.
  - - Supports queries matching multiple fields, enhancing read operation speeds.
- **Creating a Compound Index**
  - - Use `createIndex()` function.
  - - Specify multiple fields.
  - - Example: `db.users.createIndex({lastName: 1, firstName: 1})`.
  - - '1' indicates ascending order for both fields.
- **Use Cases & Performance Considerations**
  - - Ideal for queries matching on multiple fields, supports sorting of data.
  - - The field order in the index determines the sort order of results.
  - - Considerations: Compound indexes require more storage space.

Multikey Indexes

- **Multikey Indexes in MongoDB**
  - Indexes fields with array of values.
  - Each array element is indexed separately, enabling efficient searches over array contents.
- **Creating a Multikey Index**
  - Similar to creating a single field index.
  - Example: db.blogPosts.createIndex({tags: 1}).
    - If 'tags' is an array, MongoDB automatically creates a multikey index.
- **Use Cases & Performance Considerations**
  - Ideal for queries finding documents based on elements within an array field.
  - Considerations: Require more storage due to individual array element indexing. Use judiciously based on application needs.

Text Indexes

- **Text Indexes in MongoDB**
  - Efficiently search for string content within collections.
  - Supports querying of text content.
- **Creating a Text Index**
  - Use createIndex() function.
  - Different parameter for text indexing.
  - Example: db.blogPosts.createIndex({content: "text"}).
- **Use Cases & Text Search Operations**
  - Ideal for searches for words or phrases within text content.
  - MongoDB provides operators for phrase search, word exclusion, case-sensitive searches, and more.

Geospatial Index

- **2D Indexes in MongoDB**
  - Designed to handle geospatial data, supporting queries on planar geometry.
  - Ideal for data representing points on a map.
- **Creating a 2D Index**
  - Similar to creating other indexes.
  - Example: db.locations.createIndex({coordinates: "2d"}).
    - 'coordinates' field holds an array of [x, y] coordinates.
- **Use Cases & Performance Considerations**
  - Useful for location-based queries, e.g., finding locations within a certain distance from a point.
  - Considerations: Require additional storage. Use only when necessary to support application's requirements.

Benefits of indexing

- **Improved Query Performance:**
  - Reduces the number of documents MongoDB needs to scan.
  - Enhances speed and efficiency of queries.
  - Analogy: Easier to find a book in a well-organized library.
- **Faster Write Operations:**
  - While known for impact on read operations, also improves write operations.
  - Helps MongoDB locate documents faster for updates and inserts.

Creating and Managing Indexes

- **Creating Indexes:**
  - Use createIndex() function.
  - Example: db.users.createIndex({name: 1}), '1' indicates ascending order.
- **Indexing Strategies:**
  - Understand application's data access patterns.
  - Identify frequently queried fields for indexing.
  - Field order in compound indexes impacts sort order and match conditions.
  - Consider read-write ratio; excessive indexes could slow down heavy-write app.
- **Index Management:**
  - Use db.collection.getIndexes() to list all indexes.
  - Remove unnecessary indexes with db.collection.dropIndex().
  - Leverage MongoDB Management Service (MMS) for automatic index suggestions.

Aggregation Framework

- **Concept of Data Aggregation:**
  - Transforming input data into grouped, summarized output.
  - Allows calculations like averages, sums, max, min, etc.
- **Aggregation Pipeline:**
  - Processes data records and returns computed results.
  - Similar to Unix pipeline; output of one operation is input for the next.
  - Can utilize indexes to boost performance.
- **Map-Reduce:**
  - Data processing paradigm for condensing large volumes of data.
- **Single Purpose Aggregation Methods:**
  - Convenience methods like count(), distinct(), and group().
  - Ideal for simple aggregations, but lack flexibility of Pipeline and Map-Reduce.

Aggregation Pipeline

- **Understanding Aggregation Pipeline:**
  - A framework for data transformation and analysis.
  - Processes data records and yields computed results.
  - Works much like a Unix pipeline.
- **Pipeline Operators:**
  - Work on data and modify it.
  - Examples include $match, $group, $sort, and more.
- **Pipeline Execution:**
  - Pipelines run in sequence, with the output of one stage serving as input for the next
  - Allows for efficient data processing within MongoDB.
-

Map-Reduce Function

- **Definition of Map-Reduce:**
  - A technique for processing large data sets.
  - Composed of a 'Map' operation that filters data, and a 'Reduce' operation that performs a summary.
- **Functionality of Map-Reduce:**
  - 'Map' step: Each input document turns into more than one key-value pairs.
  - 'Reduce' step: Key-value pair data is condensed into a single output.
- **Usage Scenarios for Map-Reduce:**
  - Ideal for complex data processing tasks.
  - Use when Aggregation Pipeline doesn't provide required functionality.

Single Purpose Aggregation Methods

- **Count:**
  - Returns the count of documents that match a query.
  - Example usage: db.collection.count(query).
- **Distinct:**
  - Finds the distinct values for a specified field across a single collection.
  - Example usage: db.collection.distinct(field, query).
- **Group:**
  - Groups documents by a specified identifier expression and applies the accumulator expression(s) to each group.
  - Example usage: db.collection.group({ key, reduce, initial}).

Group Method

- **Definition of 'Group':**
  - MongoDB operation that groups documents by a specified identifier.
  - Applies accumulator expressions to each group.
- **Functionality of 'Group':**
  - Allows aggregation tasks like summing field values, averaging, etc.
  - Example usage: db.collection.group({ key, reduce, initial }).
- **Usage Scenario for 'Group':**
  - Ideal for grouping data by certain criteria and performing calculations.
  - For example, finding the total sales by each salesperson.

Summary

- **Review of Key Concepts:**
  - Indexing in MongoDB enhances query performance.
  - We learned about different types of indexes (Single Field, Compound, Multikey, Text, 2D, etc.)
  - We explored MongoDB's Aggregation Framework, including the Aggregation Pipeline, Map-Reduce, and Single Purpose Aggregation Methods.
- **Applying the Knowledge:**
  - We will now apply these concepts in the following lab session.
  - Practical implementation of these techniques will solidify your understanding.

Lab

MongoDB indexing and Aggregation