MongoDB: A Non-Relational DBMS

65011445 Peeranat Leelawattanapanit

65011693 Soe Moe Htet

Software Engineering Program, KMITL

01286241: Database Systems

Prof. Dr. Suphamit Chittayasothorn

March 31, 2024

Contents

MongoDB: A Non-Relational DBMS

A non-relational database management system (DBMS) is a type of database system that structures data in forms other than the traditional tabular format of the relational database model. Different non-relational DBMS structure and store data in various formats, such as key-value pairs, documents, graphs, or objects. Instead of SQL, they have their own query languages and techniques. As such, non-relational DBMS are commonly described as NoSQL. One popular NoSQL DBMS product is MongoDB. This report aims to discuss the logical data structure, integrity constraints, and data manipulation methods of MongoDB, and how they differ from SQL-based products.

Logical Data Structure

Relational: **president**

```
id  pres_name    birth_yr yrs_serv party
xx  Kennedy J F  1917     2
Democratic
```

MongoDB:

```
Collection: president
      Document 1:
      {
            "_id": ObjectId("xx"),
            "pres_name": "Kennedy J F",
            "birth_yr": 1917,
            "yrs_serv": 2,
            "party": "Democratic"
      }
```

MongoDB is a document-oriented NoSQL DBMS, which stores data in a JSON-like format. A field is an individual key-value pair that is comparable to a table column. A field can have an associated data type, such as string, integer, or boolean. Unlike the relational database model which doesn't allow repeating groups, MongoDB allows array data types. Moving up a level, documents are the JSON-like data structure that groups the individual key-value pairs. More specifically, documents are stored in binary JSON (BSON) format, which means that it encodes the data type and length, resulting in faster traversal than plain JSON. BSON also supports additional data types such as date and binary data. Documents are comparable to the rows of a table. Documents can also contain nested sub-documents, offering further flexibility. In each database, MongoDB organizes its documents into collections, which are similar to tables in the relational database model. The key difference between the two formats is that MongoDB does not enforce a fixed predefined schema. Documents within a collection are allowed to have different structures. This allows for more flexible and dynamic data models, suitable for frequently-changing application requirements.

Integrity Constraints

Since one of the main advantages of MongoDB is data flexibility, it does not enforce the same strict integrity constraints of relational databases. Instead, there are mechanisms in place to help developers implement schema validation, such as ensuring values are in the correct data type and fall in an expected range (e.g. age must be a non-negative integer), or that a value is required and not null. These schema validation rules help guarantee that updates or insertions resulting in an invalid document are rejected. Every MongoDB document must have a unique '_id' field, serving as its primary key. If it is not provided, MongoDB automatically generates a unique 12-byte value to ensure that it is not null.

Denormalized or Normalized Model

One common method for promoting data consistency is to embed related data. This is done by nesting related data in one collection, resulting in a denormalized schema. This is ideal for frequently-accessed data because fewer operations are needed to get query data in the same document, providing better performance for read operations. This is suitable for modeling one-to-one and one-to-many relationships between entities. In the case of one-to-many relations, the array of 'many' child documents can be accessed from their 'one' parent. In practice, a potential drawback is the 16-megabyte limit of document size, so large subdocuments that are likely to be expanded indefinitely should not be nested.

An alternative approach is to normalize data models by using references, which links one document to another. Similar to storing foreign-key attributes in relational databases, a MongoDB document can store the id of another document in a field. This can be used to model complex many-to-many relationships between entities, or large hierarchical data sets. An attempt to query a reference that no longer exists will result in an error. As such, it is important that when an object is deleted, all references to that object must also be deleted. Unlike relational DBMS where foreign key values must match primary key values or be null, MongoDB does not enforce key constraints. To ensure data consistency, referential integrity should be handled by logic in the application layer.

In deciding which approach is best-suited to store related data, it is important to consider the practical nature of the data set. If the data of the related document is usually queried from a parent document, a denormalized approach by embedding is ideal. However, if the data of both related documents are frequently queried on their own and a clear parent-child relationship cannot be established, a normalized approach by referencing is more appropriate.

Database Design

By mapping the president, pres_hobby, and pres_marriage, and election tables directly from the normalized 5NF relational Presidential Database to MongoDB, we get collections with documents in the following format:

Example document in the president collection

```
{
        _id: ObjectId('6607b9d3d34137b70372b756'),
        PRES_NAME: 'Washington G',
        BIRTH_YR: 1732,
        YRS_SERV: 7,
        DEATH_AGE: 67,
        PARTY: 'Federalist',
        STATE_BORN: 'Virginia',
}
```

Example documents in the pres_hobby collection

```
{
        _id: ObjectId('6607b9d3d34137b70372b758'),
        PRES_NAME: 'Washington G',
        HOBBY: 'Fishing'
}
{
        _id: ObjectId('6607b9d3d34137b70372b759'),
        PRES_NAME: 'Washington G',
        HOBBY: 'Riding'
}
```

Example document in the pres_marriage collection

```
{
        _id: ObjectId('6607bcc8d34137b70372b853'),
        PRES_NAME: 'Washington G',
        SPOUSE_NAME: 'Custis M D',
        PR_AGE: 26,
        SP_AGE: 27,
        NR_CHILDREN: 0,
        MAR_YEAR: 1759
}
```

Example document in the election collection:

```
{
        _id: ObjectId('6607bcb6d34137b70372b782'),
        ELECTION_YEAR: 1789,
        CANDIDATE: 'Washington G',
        VOTES: 69,
        WINNER_LOSER_INDIC: 'W'
}
```

It is clear that 3 separate collections for president, pres_hobby, and pres_marriage would not be ideal. This is due to the redundancies, especially in pres_hobby where only the president name and hobby are stored. To promote data integrity, the presidential database shall be denormalized by embedding pres_hobby and pres_marriage as fields in the Presidents collection, as follows:

Example document in the denormalized Presidents collection

```
{
        _id: ObjectId('6607b9d3d34137b70372b756'),
        PRES_NAME: 'Washington G',
        BIRTH_YR: 1732,
        YRS_SERV: 7,
        DEATH_AGE: 67,
        PARTY: 'Federalist',
        STATE_BORN: 'Virginia',
        pres_hobbies: ['Fishing', 'Riding'],
        pres_marriages: [
                {
                        _id: ObjectId('6607bcc8d34137b70372b853'),
                        SPOUSE_NAME: 'Custis M D',
                        PR_AGE: 26,
                        SP_AGE: 27,
                        NR_CHILDREN: 0,
                        MAR_YEAR: 1759
                }
        ]
}
```

Since the election collection contains documents of all presidential candidates, some of whom lost the election, their records are not found in the president table. While it is certainly possible to use the election collection as the main collection and embed president details as child sub-collections, this is not practical. It is highly likely that we would want to query the presidents collection without considering election information. In more advanced usages, we may want to relate the presidents and election collections together. One way that this can be achieved is to add a reference to the _id of the corresponding president, or null in the case where the candidate lost the election.

Example document in the election collection with added reference:

```
{
        _id: ObjectId('6607bcb6d34137b70372b782'),
        ELECTION_YEAR: 1789,
        CANDIDATE: 'Washington G',
        VOTES: 69,
        WINNER_LOSER_INDIC: 'W',
        president_ref: ObjectId('6607b9d3d34137b70372b756')
}
```

Data Definition Language

While MongoDB does not have a proper Data Definition Language (DDL), it achieves DDL operations by executing administrative commands in the MongoDB Shell. These administrative commands are responsible for defining the structure and characteristics of data, such as creating, updating, and deleting collections.

Creating a Database

MongoDB does not have an explicit command for creating databases. Instead, the ‘use’ keyword is used to select a particular database, in this case the PresidentialDatabase, for subsequent commands. If the specified database does not exist, one will be created.

SQL	`CREATE DATABASE PresidentialDatabase`
MongoDB	`use PresidentialDatabase`

Creating a Collection

MongoDB collections are comparable to tables in relational databases. Even though MongoDB is schemaless, integrity constraints can be enforced by using JSON schema validation. If this is not needed, the second parameter (validator) can be omitted completely.

SQL

```
CREATE TABLE  "ELECTION"
   (  "ELECTION_YEAR" NUMBER,
      "CANDIDATE" VARCHAR2(30),
      "VOTES" NUMBER,
      "WINNER_LOSER_INDIC" VARCHAR2(15),
   )
```

MongoDB

```
PresidentialDatabase.createCollection("ELECTION", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: ["ELECTION_YEAR", "CANDIDATE",
            "VOTES", "WINNER_LOSER_INDIC"],
         properties: {
            ELECTION_YEAR: {
               bsonType: "int"
            },
            CANDIDATE: {
               bsonType: "string",
               maxLength: 30
            },
            VOTES: {
               bsonType: "int"
            },
            WINNER_LOSER_INDIC: {
               bsonType: "string",
               maxLength: 15
            }}}}})
```

Dropping a Collection

Dropping a collection removes the entire collection along with all of its documents. This operation is irreversible.

SQL	`DROP TABLE election`
MongoDB	`PresidentialDatabase.election.drop()`

Renaming a Collection

SQL	`ALTER TABLE old_name RENAME TO election`
MongoDB	`PresidentialDatabase.old_name.renameCollection("election")`

Creating an Index

In the following example, SQL creates an index named idx_pres_name on the pres_name column of the president table. In MongoDB, an index is created on the pres_name field of the president collection. There are many types of indexes including single-field, compound, text, etc. In this example, a single-field index is created with the sorting order 1, indicating that the index will be in ascending order. The second parameter sets a custom index name and is optional. If left unspecified, MongoDB automatically names indexes based on the field and sorting order, for example pres_name_1.

SQL	`CREATE INDEX idx_pres_name ON president(pres_name)`
MongoDB	`PresidentialDatabase.president.createIndex({pres_name: 1}, {name: "idx_pres_name"})`

Dropping an Index

When dropping an index, the index property is removed but not the column itself. In this example, the index idx_pres_name is dropped from the president collection.

SQL	`DROP INDEX idx_pres_name ON president`
MongoDB	`PresidentialDatabase.president.dropIndex("idx_pres_name")`

Dropping Many Indexes

Unlike in SQL, MongoDB has an administrative command for dropping all indexes of a collection. When no parameter is passed into the dropIndexes method, all droppable indexes will be dropped. Note that the default '_id' field index cannot be dropped. To drop multiple indexes, an array parameter of index names can be passed into dropIndexes.

MongoDB	`PresidentialDatabase.presidents.dropIndexes()`

Data Manipulation Language

MongoDB Query Language (MQL) is the Data Manipulation Language (DML) for querying MongoDB databases. The following demonstrates the basic syntax for achieving CRUD functionalities.

Insert a Document

MongoDB documents are comparable to rows of a table in a relational database. The following demonstrates inserting a document into the president collection.

SQL

```
INSERT INTO PresidentialDatabase.president (PRES_NAME,
BIRTH_YR, YRS_SERV, DEATH_AGE, PARTY, STATE_BORN)
VALUES ('New President', 2000, 4, 70, 'Democratic', 'New
York')
```

MQL

```
PresidentialDatabase.president.insertOne({
  "PRES_NAME": "New President",
  "BIRTH_YR": 2000,
  "YRS_SERV": 4,
  "DEATH_AGE": 70,
  "PARTY": "Democratic",
  "STATE_BORN": "New York",
});
```

Insert Many Documents

MongoDB allows multiple documents to be inserted at once by supplying an array of documents into the insertMany method. In SQL, multiple separate INSERT INTO statements would be required.

MQL

```
PresidentialDatabase.president.insertMany([
  {
    "PRES_NAME": "President 1",
    "BIRTH_YR": 1950,
    "YRS_SERV": 8,
    "DEATH_AGE": 80,
    "PARTY": "Republican",
    "STATE_BORN": "Texas"
  },
  {
    "PRES_NAME": "President 2",
    "BIRTH_YR": 1960,
    "YRS_SERV": 4,
    "DEATH_AGE": 75,
    "PARTY": "Democratic",
    "STATE_BORN": "California"
  }
]);
```

Updating Field Values

The following demonstrates modifying the yrs_serv and death_age values for presidents with pres_name = ‘New President’:

SQL	`UPDATE PresidentialDatabase.president`
	`SET YRS_SERV = 8, DEATH_AGE = 78`
	`WHERE PRES_NAME = 'New President'`
MQL	`PresidentialDatabase.president.updateMany(`
	`   { "PRES_NAME": "New President" },`
	`   { $set: { "YRS_SERV": 8, "DEATH_AGE": 78 } }`
	`);`

Deleting Documents

The following demonstrates irreversibly dropping all documents of Republican presidents from the president collection:

SQL	`DELETE FROM PresidentialDatabase.president`
	`WHERE PARTY = 'Republican'`
MQL	`db.president.deleteMany({ "PARTY": "Republican" });`

Deleting a Document

In the case requiring only one document to be dropped, the deleteOne method can be used. Much like a relational database where rows have no order, documents in a MongoDB collection are unordered. Without specifying a sort order, the deleted document is one that the database engine retrieves first. There is no guarantee which document will be deleted.

SQL	`DELETE FROM PresidentialDatabase.president`
	`WHERE PARTY = 'Republican'`
MQL	`db.president.deleteOne({ "PARTY": "Republican" });`

Search by a Field Value

The following demonstrates querying details of all presidents born in Texas. The results obtained from the queries are also shown below. Notice that while SQL returns results in a tabular format, MongoDB returns results in JSON (JavaScript Object Notation).

SQL	`SELECT * FROM president WHERE STATE_BORN = 'Texas'`
MQL	`PresidentialDatabase.president.find({STATE_BORN: "Texas"});`

SQL Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	Texas
Eisenhower D D	1890	8	79	Republican	Texas
Johnson L B	1908	5	65	Democratic	Texas

MQL Result:

```
[
  {
    _id: ObjectId('6607b9d3d34137b70372b774'),
    PRES_NAME: 'Roosevelt F D',
    BIRTH_YR: 1882,
    YRS_SERV: 12,
    DEATH_AGE: 63,
    PARTY: 'Democratic',
    STATE_BORN: 'Texas',
  },
  {
    _id: ObjectId('6607b9d3d34137b70372b776'),
    PRES_NAME: 'Eisenhower D D',
    BIRTH_YR: 1890,
    YRS_SERV: 8,
    DEATH_AGE: 79,
    PARTY: 'Republican',
    STATE_BORN: 'Texas',
  },
  {
    _id: ObjectId('6607b9d3d34137b70372b778'),
    PRES_NAME: 'Johnson L B',
    BIRTH_YR: 1908,
    YRS_SERV: 5,
    DEATH_AGE: 65,
    PARTY: 'Democratic',
    STATE_BORN: 'Texas',
  }
]
```

Filtering a collection

```
PresidentialDatabase.president.find(
      // Query criteria
      {
            // Specify filter conditions
            // Example:
            // FIELD_NAME: VALUE,
            // FIELD_NAME: { OPERATOR: VALUE }
      },
      // Projection (optional)
      {
            // Specify fields to include/exclude
            // Example:
            // FIELD_NAME: 1, // Include this field
            // FIELD_NAME: 0  // Exclude this field
      }
);
```

Replace FIELD_NAME with the field name to filter on, and VALUE with the value to filter by. Query operators like $eq, $gt, $lt, $in, $regex, etc., can be used for more complex filtering. Additionally, it can include a projection parameter to specify which fields to include or exclude in the query results.

For example:

```
PresidentialDatabase.president.find(
      // Query criteria: Only presidents in the Republican PARTY
      {
            PARTY: "Republican"
      },
      // Projection: Include only PRES_NAME and PARTY fields
      {
            PRES_NAME: 1,
            PARTY: 1,
            _id: 0  // Exclude the _id field
      }
);
```

This query will return documents from the presidents collection where the PARTY field is equal to "Republican". It will include only the PRES_NAME and PARTY fields in the results and exclude the _id field.

1. **Filtering by equality**

   This query will return documents where the PARTY field equals "Democratic".

SQL	`SELECT * FROM president WHERE PARTY = 'Democratic'`
MQL	`PresidentialDatabase.president.find({PARTY: "Democratic"});`

2. **Filtering by greater than**

   This query will return documents where the BIRTH_YR field is greater than 1800.

SQL	`SELECT * FROM president WHERE BIRTH_YR > 1800`
MQL	`PresidentialDatabase.president.find({BIRTH_YR: {$gt: 1800}});`

3. **Filtering by less than**

   This query will return documents where the YRS_SERV field is less than 4.

SQL	`SELECT * FROM president WHERE YRS_SERV < 4`
MQL	`PresidentialDatabase.president.find({YRS_SERV: {$lt: 4}});`

In addition to $gt and $lt, $gte and $lte can also be used for ‘greater than or equal to’ and ‘less than or equal to’, respectively.

Projecting a collection

1. **Return specific fields**

   This query will return documents where the PARTY field equals "Democratic", and it will include only the PRES_NAME and BIRTH_YR fields in the result.

SQL	`SELECT PRES_NAME, BIRTH_YR`
	`FROM president`
	`WHERE PARTY = ‘Democratic’`

MQL	`PresidentialDatabase.president.find({PARTY: “Democratic”},`
	`{PRES_NAME: 1, BIRTH_YR: 1 });`

2. **Exclude specific fields**

   This query will return documents where the YRS_SERV field is greater than 4, and it will exclude the PARTY and STATE_BORN fields from the result. In SQL, this can only be achieved by listing the remaining rows to include.

MQL	`PresidentialDatabase.president.find({YRS_SERV: { $gt: 4}},`
	`{PARTY: 0, STATE_BORN: 0 });`

3. **Include nested fields**

   The following query is done on the denormalized presidents table. This query will return documents where the PRES_NAME field equals "Adams J", and it will include only the SPOUSE_NAME field from the pres_marriages array. To access nested fields in an array of subdocuments, the dot notation field.subdocument_field is used.

MQL	`PresidentialDatabase.presidents.find({PRES_NAME: “Adams J”},`
	`{“pres_marriages.SPOUSE_NAME: 1 });`

__Sorting a collection__

1. **Sort by a Single Field in Ascending Order:**
   This query will return all documents from the presidents collection sorted by the BIRTH_YR field in ascending order.

SQL	`SELECT * FROM president ORDER BY BIRTH_YR`
MQL	`PresidentialDatabase.president.find().sort({ BIRTH_YR: 1});`

2. **Sort by a Single Field in Descending Order:**
   This query will return all documents from the presidents collection sorted by the DEATH_AGE field in descending order.

SQL	`SELECT * FROM president ORDER BY DEATH_AGE DESC`
MQL	`PresidentialDatabase.president.find().sort({ DEATH_AGE: -1});`

3. **Sort by multiple fields:**
   This query will return all documents from the presidents collection sorted first by the PARTY field in ascending order and then by the BIRTH_YR field in descending order.

SQL	`SELECT * FROM president ORDER BY PARTY, BIRTH_YR DESC`
MQL	`PresidentialDatabase.president.find() .sort({ PARTY: 1, BIRTH_YR: -1 });`

Limiting a collection

1. **Limit the Number of Documents Returned:**
   This query will return the first 5 documents from the president collection.

SQL	`SELECT *`
	`FROM president`
	`LIMIT 5`

MQL	`PresidentialDatabase.president.find().limit(5);`

2. **Combining Limit with Sorting:**
   This query will first sort all documents in the presidents collection by the BIRTH_YR field in descending order and then return the top 10 documents.

SQL	`SELECT *`
	`FROM president`
	`ORDER BY BIRTH_YR DESC`
	`LIMIT 10`

MQL	`PresidentialDatabase.president.find().sort({BIRTH_YR: -1 })`
	`.limit(10);`

3. **Limit with Projection:**
   This query will return only the PRES_NAME and BIRTH_YR fields for the first 3 documents from the presidents collection, excluding the _id field.

SQL	`SELECT PRES_NAME, BIRTH_YR`
	`FROM president`
	`LIMIT 3`

MQL	`PresidentialDatabase.president.find({},`
	`{PRES_NAME: 1, BIRTH_YR: 1, _id: 0 }).limit(3);`

Counting a collection

1. **Counting Documents Based on a Filter:**
   This query will return the count of documents in the presidents collection where the PARTY field is equal to "Democratic".

SQL	**SELECT** COUNT(*) **FROM** president **WHERE** PARTY = ‘Democratic’
MQL	PresidentialDatabase.president.**count**({PARTY: “Democratic” });

2. **Counting Documents with Conditions:**
   This query will return the count of documents in the presidents collection where the BIRTH_YR field is greater than or equal to 1800.

SQL	**SELECT** COUNT(*) **FROM** president **WHERE** BIRTH_YR >= 1800
MQL	PresidentialDatabase.president.**count**({BIRTH_YR:{$gte: 1800}});

3. **Count Documents with Multiple Conditions**
   This query will return the count of documents in the presidents collection where the PARTY field equals "Republican" and the BIRTH_YR field is greater than or equal to 1800.

SQL	**SELECT** COUNT(*) **FROM** president **WHERE** PARTY = ‘Republican’ **AND** BIRTH_YR >= 1800
MQL	PresidentialDatabase.president.**count**({PARTY: “Republican”, BIRTH_YR : {$gte: 1800}});

Example Queries

1. List the president name and death age of the president who died the youngest.

SQL

```sql
SELECT PRES_NAME, DEATH_AGE
FROM president
WHERE DEATH_AGE = (SELECT MIN(DEATH_AGE) FROM president)
```

MQL

```javascript
PresidentialDatabase.president.aggregate([
  // Filter out presidents with unknown death age
  {
    $match: {
      DEATH_AGE: { $exists: true, $ne: null }
    }
  },
  // Sort by death age in ascending order
  {
    $sort: { DEATH_AGE: 1 }
  },
  // Limit to the first result (the president who died youngest)
  {
    $limit: 1
  },
  // Project to include only PRES_NAME and DEATH_AGE fields
  {
    $project: {
      _id: 0,
      PRES_NAME: 1,
      DEATH_AGE: 1
    }
  }
]);
```

Result:

```
[ { PRES_NAME: 'Kennedy J F', DEATH_AGE: 46 } ]
```

2. List details of presidents who belonged to the party which has the highest number of presidents in Ohio.

SQL	
	```
	SELECT *
	FROM president
	WHERE party = (SELECT PARTY
	               FROM president
	               WHERE STATE_BORN = 'Ohio'
	               GROUP BY PARTY
	               HAVING COUNT(*) >= ALL (SELECT COUNT(*)
	                                       FROM president
	                                       WHERE STATE_BORN = 'Ohio'
	                                       GROUP BY PARTY))
	```

MQL	
	```
	PresidentialDatabase.president.aggregate([
	  // Match presidents from Ohio
	  {
	    $match: {
	      STATE_BORN: "Ohio"
	    }
	  },
	  // Group by party and count the number of presidents in each party
	  {
	    $group: {
	      _id: "$PARTY",
	      count: { $sum: 1 }
	    }
	  },
	  // Sort by count in descending order
	  {
	    $sort: { count: -1 }
	  },
	  // Limit to the party with the highest count
	  {
	    $limit: 1
	  },
	  // Lookup presidents belonging to the party with the highest count
	  {
	    $lookup: {
	      from: "president",
	      localField: "_id",
	      foreignField: "PARTY",
	      as: "president"
	    }
	  },
	  // Unwind the president array
	  { $unwind: "$president" },
	  // Project to reshape the output
	  {
	    $project: {
	      _id: 0,
	      PRES_NAME: "$president.PRES_NAME",
	      PARTY: "$president.PARTY",
	      STATE_BORN: "$president.STATE_BORN"
	    }}]);
	```

3. List the names, numbers of votes, state of the presidents who have the highest votes from each state.

SQL

```
SELECT
    p.STATE_BORN AS state,
    p.PRES_NAME AS president,
    e.VOTES AS maxVotes
FROM
    president p
JOIN
    election e ON p.PRES_NAME = e.CANDIDATE
WHERE
    (p.STATE_BORN, e.VOTES) IN (
        SELECT
            p.STATE_BORN,
            MAX(e.VOTES)
        FROM
            president p
        JOIN
            election e ON p.PRES_NAME = e.CANDIDATE
        GROUP BY
            p.STATE_BORN
    );
```

MQL

```
PresidentialDatabase.president.aggregate([
  // Lookup the election data for each president
  {
    $lookup: {
      from: "election",
      localField: "PRES_NAME",
      foreignField: "CANDIDATE",
      as: "electionData"
    }
  },
  // Unwind the election data array
  { $unwind: "$electionData" },
  // Group by birth state and find the president with the highest votes
  {
    $group: {
      _id: "$STATE_BORN",
      president: { $first: "$PRES_NAME" },
      maxVotes: { $max: "$electionData.VOTES" }
    }
  },
  // Project to reshape the output
  {
    $project: {
      _id: 0,
      state: "$_id",
      president: 1,
      maxVotes: 1
    }
  }
]);
```

Querying Ability Comparison

Querying capabilities between SQL and MQL, reveals their strengths and weaknesses. SQL follows the fixed structure of relational databases with clearly defined schemas, often requiring joins and subqueries. However, MQL is more flexible, which supplements MongoDB's document-oriented nature. MQL has a more diverse range of query methods, such as aggregate, find, and count, to accommodate the schemaless documents with varying structures. MQL also offers stages of aggregation pipelines with $match, $group, and $project. This versatility allows complex queries and computation to be executed in a more structured manner. SQL is more limited to a few built-in functions, such as COUNT, SUM, and AVERAGE, so more advanced query techniques may be needed to achieve the same result. This can oftentimes lead to ambiguous or difficult to understand queries. SQL is well-suited to handle relational queries between tables, while MQL is more optimal for handling nested structures. This is consistent with their distinct schema designs, with MongoDB favoring embedding related data.

Conclusion

In summary, while SQL databases excel in handling structured data and complex transactions with strong consistency guarantees, NoSQL databases like MongoDB offer greater flexibility, scalability, and performance for diverse data models and query requirements. The integrity constraints imposed by relational DBMS provide data consistency and isolation, preventing issues arising from data redundancies. However, for MongoDB, data duplication is a common and accepted practice. Therefore, much of the responsibility to ensure data reliability falls to the developers who must add logic at the application layer to prevent errors. While relational databases are ideal for structured data with fixed well-defined schemas, MongoDB's document-oriented approach, flexible schema, and powerful aggregation framework make it well-suited for modern applications dealing with frequently evolving and diverse unstructured or semi-structured data. NoSQL databases are also designed to be horizontally scalable. This means that data can be easily distributed across multiple servers, ideal for dealing with large volumes of data without partitioning. It is not possible to conclude whether a relational or NoSQL database is better. They serve different purposes and have their different strengths and weaknesses. Ultimately, the choice of DBMS depends largely on the program requirements and use cases.

References

Anderson, B., Nicholson, B. (2022, June 12). *SQL vs. NoSQL Databases: What's the Difference?* IBM. https://www.ibm.com/blog/sql-vs-nosql/

Ashtari, H. (2022, October 18). *What Is NoSQL? Features, Types, and Examples.* Spiceworks. https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-nosql/

Ellingwood, J. (n.d.). *How to query and filter documents in MongoDB.* Prisma. https://www.prisma.io/dataguide/mongodb/querying-documents

GeeksForGeeks. (2023, December 1). *MongoDB Advantages & Disadvantages*. https://www.geeksforgeeks.org/mongodb-advantages-disadvantages/

MongoDB Manual. (n.d.). *Data Modeling.* MongoDB. https://www.mongodb.com/docs/manual/data-modeling/

MongoDB Manual. (n.d.). *SQL to MongoDB Mapping Chart.* MongoDB. https://www.mongodb.com/docs/manual/reference/sql-comparison/