CS232L – Database Management System

LAB#10

PostgreSQL VS MongoDB

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS232L – Database Management system Lab

Lab 10– PostgreSQL and MongoDB

Objective

The objective of this session is to

1. Introduction of PostgreSQL and MongoDB
2. Why use MongoDB?
3. Differences between both RDBMS and MongoDb
4. Examples
5. Comparing the mongodb query language to SQL

Instructions

- Open the handout/lab manual in front of a computer in the lab during the session.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering problems, indicate the commands you entered and the output displayed.

Introduction

One of the most important parts of the function of any company is a secure database. With phishing attacks, malware, and other threats on the rise, it is essential that you make the right choice to keep your data safe and process it effectively. However, it can be extremely difficult to choose from the wide variety of database solutions on the market today.
Two commonly used options are **MongoDB** and **PostgreSQL**.

Here are the key points from our Mongo DB vs. PostgreSQL comparison:

- MongoDB is an open-source non-relational database system that falls under the NoSQL category.
- PostgreSQL is a relational database management system.

1. PostgreSQL

Relational databases are great at running complex queries and data-based reporting in cases where the data structure doesn't change frequently.
Open-source databases like PostgreSQL is a highly stable database management system.
PostgreSQL stores the data in the **tabular format** and uses procedures and views as the core components.

Use cases for PostgreSQL include bank systems, risk assessment, multi-app data repository, BI (business intelligence), manufacturing, and powering various business applications. It is ideal for transactional workflows. Also, PostgreSQL has fail-safes and redundancies that make its storage particularly reliable. This means that it is perfect for important industries like healthcare and manufacturing.

Architecture:

PostgreSQL has a monolithic architecture, meaning that the components are completely united. This also means that the database can only scale as much as the machine running it.

2. MongoDB?

MongoDB is a schema-free,
general purpose,
document high-performance database.

It is distributed database built for modern application developers.
Example: For high-volume websites like eBay, Amazon, Twitter, or Facebook.

Common use cases for MongoDB include customer analytics, content management, business transactions, and product data. Thanks to its ability to scale, the database is also ideal for mobile solutions that need to be scaled to millions of users.

Architecture:

The database features a distributed architecture, meaning that components function across multiple platforms in collaboration with one another. This also means that MongoDB has nearly unlimited scalability since it can be scaled across more than one platform as needed. That is one of the many factors that **differentiate MongoDB from a relational database.**

Differences

MongoDB	PostgreSQL
Schema-free	SQL-based but supports various NoSQL features
Document database	Relational database
Uses JSON	Uses SQL
Distributed architecture	Monolithic architecture
Uses collections	Uses tables
Uses documents to obtain data	Uses rows to obtain data
Does not support foreign key constraints	Supports foreign key constraints
Uses the aggregation pipeline for running queries	Uses GROUP_BY
Uses indexes	Uses joins

Which is best and why?

PostgreSQL is best when: you want a relational database that will run complex SQL queries and work with lots of existing applications based on a tabular, relational data model, PostgreSQL will do the job.

MongoDB is best when: you are at the beginning of a development project and are seeking to figure out your needs and data model by using an agile development process, MongoDB will shine because developers can reshape the data on their own, when they need to. MongoDB enables you to manage data of any structure, not just tabular structures defined in advance.

Environment Setup

Complete installation of MongoDB is given in the link below:
https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/

Comparing the MongoDB Query Language to SQL

The relational database model that PostgreSQL uses relies on storing data in tables and then using Structured Query Language (SQL) for database access.

To make this work, in PostgreSQL and all other SQL databases, the database schema must be created and data relationships established before the database is populated with data. Related information may be stored in separate tables, but associated through the use of Foreign Keys and JOINs.

The challenge of using a relational database is the need to define its structure in advance. Changing structure after loading data is often very difficult, requiring multiple teams across development, DBA, and Ops to tightly coordinate changes.

Now in the document database world of MongoDB, the structure of the data doesn't have to be planned up front in the database and it is much easier to change. Developers can decide what's needed in the application and change it in the database accordingly.

Query language map

Both PostgreSQL and MongoDB have a rich query language. Below are a few examples of SQL statements and how they map to MongoDB.

1. **Creating table:**

SQL	MongoDB
`CREATE TABLE users ( user_id VARCHAR(20) NOT NULL, age INTEGER NOT NULL, status VARCHAR(10));`	Implicitly created on first `insertOne()` or `insertMany()` operation. The primary key `_id` is automatically added if `_id` field is not specified. However, you can also explicitly create a collection: `db.createCollection("users")`

2. **Insertion:**

```sql
INSERT INTO users(user_id, age, status)
VALUES ('bcd001', 45,"A");
```

```javascript
db.users.insertOne({
  user_id: "bcd001",
  age: 45,
  status: "A"
})
// see note below table.
```

3. **Retrieving data:**

```sql
SELECT *
FROM users;
```

```javascript
db.users.find()
```

4. **Updating a record:**

```sql
UPDATE users
SET status = 'C'
WHERE age &gt; 25;
```

```javascript
db.users.updateOne(
    { age: { $gt: 25 } },
    { $set: { status: "C" } },
    { multi: true }
)
// see note below table.
```

Note: db.collection.insertMany() can be used for more than one document.

Details and some more Examples and comparisons:

The following table presents the various SQL statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume a table named `people`.
- The MongoDB examples assume a collection named `people` that contain documents of the following prototype:

```
{
_id: ObjectId("509a8fb2f3f4948bd2f983a0"),
user_id: "abc123",
age: 55,
status: 'A'
}
```

1. Insert

The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL Schema Statements	MongoDB Schema Statements

`CREATE TABLE people ( id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id) )`	Implicitly created on first `insertOne()` or `insertMany()` operation. The primary key `_id` is automatically added if `_id` field is not specified.
	`db.people.insertOne( { user_id: "abc123", age: 55, status: "A" } )`
	However, you can also explicitly create a collection:
	`db.createCollection("people")`

SQL	MongoDB
`ALTER TABLE people ADD join_date DATETIME`	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level. However, at the document level, `updateMany()` operations can add fields to existing documents using the `$set` operator. `db.people.updateMany( { }, { $set: { join_date: new Date() } } )`
`ALTER TABLE people DROP COLUMN join_date`	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level. However, at the document level, `updateMany()` operations can remove fields from documents using the `$unset` operator. `db.people.updateMany( { }, { $unset: { "join_date": "" } } )`
`DROP TABLE people`	`db.people.drop()`

2.Insert

The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL INSERT Statements	MongoDB insertOne() Statements
`INSERT INTO people(user_id, age, status) VALUES ("bcd001", 45, "A")`	`db.people.insertOne( { user_id: "bcd001", age: 45, status: "A" } )`

Insert Multiple Documents

`db.collection.insertMany()` can insert *multiple* documents into a collection. Pass an array of documents to the method.

The following example inserts three new documents into the `inventory` collection. If the documents do not specify an `_id` field, MongoDB adds the `_id` field with an ObjectId value to each document. See Insert Behavior.

```
db.inventory.insertMany([
{ item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
{ item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
{ item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }
])
```

`insertMany()` returns a document that includes the newly inserted documents `_id` field values.

To retrieve the inserted documents, query the collection:

```
db.inventory.find( {} )
```

Insert Behavior

Collection Creation

If the collection does not currently exist, insert operations will create the collection.

`_id` Field

In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the `_id` field, the MongoDB driver automatically generates an ObjectId for the `_id` field.

3. Select

The following table presents the various SQL statements related to reading records from tables and the corresponding MongoDB statements.

SQL SELECT Statements	MongoDB find() Statements
SELECT * FROM people	db.people.find()
SELECT id, user_id, status FROM people	db.people.find({ }, { user_id: 1, status: 1 })
SELECT user_id, status FROM people	db.people.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM people WHERE status = "A"	db.people.find({ status: "A" })
SELECT user_id, status FROM people WHERE status = "A"	db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM people WHERE status != "A"	db.people.find({ status: { $ne: "A" } })
SELECT * FROM people WHERE status = "A" AND age = 50	db.people.find({ status: "A", age: 50 })
SELECT * FROM people WHERE age > 25	db.people.find({ age: { $gt: 25 } })
SELECT * FROM people WHERE user_id like "%bc%"	db.people.find({ user_id: /bc/ })

SELECT * FROM people WHERE status = "A" ORDER BY user_id ASC	db.people.find({ status: "A" }).sort({ user_id: 1 })
SELECT * FROM people WHERE status = "A" ORDER BY user_id DESC	db.people.find({ status: "A" }).sort({ user_id: -1 })
SELECT COUNT(*) FROM people	db.people.count() *or* db.people.find().count()
SELECT COUNT(user_id) FROM people	db.people.count({ user_id: { $exists: true } })

4. Update Records

The following table presents the various SQL statements related to updating existing records in tables and the corresponding MongoDB statements.

UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" } })
SET age = age + 3 WHERE status = "A"	db.people.updateMany({ status: "A" } , { $inc: { age: 3 } })

5. Delete Records

The following table presents the various SQL statements related to deleting records from tables and the corresponding MongoDB statements.

SQL Delete Statements	MongoDB deleteMany() Statements
DELETE FROM people WHERE status = "D"	db.people.deleteMany({ status: "D" })
DELETE FROM people	db.people.deleteMany({})

SQL to Aggregation Mapping Chart

The aggregation pipeline allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation operators:

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match`
GROUP BY	`$group`
HAVING	`$match`
ORDER BY	`$sort`
SUM()	`$sum`
COUNT()	`$sum` `$sortByCount`
join	`$lookup`

Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

SQL Example	MongoDB Example	Description

SELECT COUNT(*) AS count
FROM orders
	db.orders.aggregate([
	{
	$group: {
	_id: null,
	count: { $sum: 1 }
	}
	}
	])
		Count all records from `orders`

SELECT SUM(price) AS total
FROM orders
	db.orders.aggregate([
	{
	$group: {
	_id: null,
	total: { $sum: "$price" }
	}
	}
	])
		Sum the `price` field from `orders`

SELECT cust_id,
SUM(price) AS total
FROM orders
GROUP BY cust_id
	db.orders.aggregate([
	{
	$group: {
	_id: "$cust_id",
	total: { $sum: "$price" }
	}
		For each unique `cust_id`, sum the `price` field.

```
}
])
```

SQL	MongoDB	Description

SELECT cust_id,
SUM(price) AS total
FROM orders
GROUP BY cust_id
ORDER BY total
	db.orders.aggregate([
	{
	$group: {
	_id: "$cust_id",
	total: { $sum: "$price" }
	}
	},
	{ $sort: { total: 1 } }
	])
		For each unique `cust_id`, sum the `price` field, results sorted by sum.

SQL	MongoDB	Description

SELECT cust_id,
ord_date,
SUM(price) AS total
FROM orders
GROUP BY cust_id,
ord_date
	db.orders.aggregate([
	{
	$group: {
	_id: {
	cust_id: "$cust_id",
	ord_date: { $dateToString: {
	format: "%Y-%m-%d",
	date: "$ord_date"
	}}
	},
	total: { $sum: "$price" }
	}
	}
	])
		For each unique `cust_id`, `ord_date` grouping, sum the `price` field. Excludes the time portion of the date.

SQL	MongoDB	Description

SELECT cust_id,
SUM(li.qty) as qty
FROM orders o,
order_lineitem li
WHERE li.order_id = o.id
GROUP BY cust_id
	db.orders.aggregate([
	{ $unwind: "$items" },
	{
	$group: {
	_id: "$cust_id",
	qty: { $sum: "$items.qty" }
	}
	}
	])
		For each unique `cust_id`, sum the corresponding line item `qty` fields associated with the orders.

```sql
SELECT COUNT(*)
FROM (SELECT cust_id,
             ord_date
      FROM orders
      GROUP BY cust_id,
               ord_date)
      as DerivedTable
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: {
           cust_id: "$cust_id",
           ord_date: { $dateToSt
              format: "%Y-%m-%d"
              date: "$ord_date"
           }}
        }
     }
   },
   {
     $group: {
        _id: null,
        count: { $sum: 1 }
     }
   }
] )
```

Count the number of distinct `cust_id`, `ord_date` groupings. Excludes the time portion of the date.