

Experiment No. 9

Title: Demonstration No-SQL database

Batch:SY-IT(B3)

Roll No.:16010423076

Experiment No.:9

Aim: To implement NOSQL database using MongoDB and PostgreSQL.

Resources needed: MongoDB, PostgreSQL

Theory:**MongoDB:**

MongoDB is a general-purpose document database designed for modern application development and for the cloud. Its scale-out architecture allows you to meet the increasing demand for your system by adding more nodes to share the load

MongoDB is having following key concepts,

- **Documents:** The Records in a Document Database
MongoDB stores data as JSON documents. The document data model maps naturally to objects in application code, making it simple for developers to learn and use. The fields in a JSON document can vary from document to document. Documents can be nested to express hierarchical relationships and to store structures such as arrays. The document model provides flexibility to work with complex, fast-changing, messy data from numerous sources. It enables developers to quickly deliver new application functionality. For faster access internally and to support more data types, MongoDB converts documents into a format called Binary JSON or BSON. But from a developer perspective, MongoDB is a JSON database.
- **Collections:** Grouping Documents
In MongoDB, a collection is a group of documents. Collection can be seen as tables, but collections in MongoDB are far more flexible. Collections do not enforce a schema, and documents in the same collection can have different fields. Each collection is associated with one MongoDB database
- **Replica Sets:** For High Availability
In MongoDB, high availability is built right into the design. When a database is created in MongoDB, the system automatically creates at least two more copies of the data, referred to as a replica set. A replica set is a group of at least three MongoDB instances that continuously replicate data between them, offering redundancy and protection against downtime in the face of a system failure or planned maintenance.
- **Sharding:** For Scalability to Handle Massive Data Growth
A modern data platform needs to be able to handle very fast queries and massive datasets using ever bigger clusters of small machines. Sharding is the term for distributing data intelligently across multiple machines. MongoDB shards data at the collection level, distributing documents in a collection across the shards in a cluster. The result is a scale-out architecture that supports even the largest applications.
- **Aggregation Pipelines:** For Fast Data Flows
MongoDB offers a flexible framework for creating data processing pipelines called aggregation pipelines. It features dozens of stages and over 150 operators and expressions, enabling you to process, transform, and analyze data of any structure at

scale. One recent addition is the Union stage, which flexibly aggregate results from multiple collections.

Besides this MongoDB provides,

- variety of indexing strategies for speeding up the queries along with the Performance Advisor, which analyses queries and suggests indexes that would improve query performance
- Support for different programming languages which includes Node.js, C, C++, C#, Go, Java, Perl, PHP, Python, Ruby, Rust, Scala, and Swift with actively maintained library updated with newly added features.
- Various tools and utilities for monitoring MongoDB.
- Cloud services

PostgreSQL:

PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. The origins of PostgreSQL date back to 1986 as part of the POSTGRES project at the University of California at Berkeley and has more than 30 years of active development on the core platform.

PostgreSQL comes with many features aimed to help developers build applications, administrators to protect data integrity and build fault-tolerant environments, and help you manage your data no matter how big or small the dataset. In addition to being free and open source, PostgreSQL is highly extensible. For example, you can define your own data types, build out custom functions, and even write code from different programming languages without recompiling your database.

Some of the features of PostgreSQL are as follows,

- **Data Types**
 - Primitives: Integer, Numeric, String, Boolean
 - Structured: Date/Time, Array, Range / Multirange, UUID
 - Document: JSON/JSONB, XML, Key-value (Hstore)
 - Geometry: Point, Line, Circle, Polygon
 - Customizations: Composite, Custom Types
- **Data Integrity**
 - UNIQUE, NOT NULL
 - Primary Keys
 - Foreign Keys
 - Exclusion Constraints
 - Explicit Locks, Advisory Locks
- **Concurrency, Performance**
 - Indexing: B-tree, Multicolumn, Expressions, Partial
 - Advanced Indexing: GiST, SP-Gist, KNN Gist, GIN, BRIN, Covering indexes, Bloom filters
 - Sophisticated query planner / optimizer, index-only scans, multicolumn statistics
 - Transactions, Nested Transactions (via savepoints)
 - Multi-Version concurrency Control (MVCC)
 - Parallelization of read queries and building B-tree indexes
 - Table partitioning
 - All transaction isolation levels defined in the SQL standard, including Serializable
 - Just-in-time (JIT) compilation of expressions
- **Reliability, Disaster Recovery**
 - Write-ahead Logging (WAL)
 - Replication: Asynchronous, Synchronous, Logical

- o Point-in-time-recovery (PITR), active standbys
- o Tablespaces
- **Security**
- **Extensibility**
- **Internationalisation, Text Search**

PostgreSQL types for NOSQL:

JSON data types are for storing JSON (JavaScript Object Notation) data. Such data can also be stored as text, but the JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules. There are also assorted JSON-specific functions and operators available for data stored in these data types.

PostgreSQL offers two types for storing JSON data: **json** and **jsonb**. To implement efficient query mechanisms for these data types PostgreSQL also provides the **jsonpath** data type

The **json** and **jsonb** data types accept *almost* identical sets of values as input. The major practical difference is one of efficiency. The **json** data type stores an exact copy of the input text, which processing functions must reparse on each execution; while **jsonb** data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. **jsonb** also supports indexing, which can be a significant advantage.

Procedure:

1. Create a repository of documents containing six family member of yours (including yourself), with minimum seven attributes each, in POSTGRES
2. Perform selection and projection queries with different criterias on the created relation
3. Export the relation to json document
4. Import the document to MongoDB
5. Perform Insert, Search, Update, and Delete operations on the collection using
 - i. MongoDB Compass
 - ii. MongoDB Shell
 Use link
<https://drive.google.com/drive/folders/12QEkVpHVRgWtgKdyxx-x1v9sznykKnEv>
6. Demonstrate pipeline in MongoDB with minimum three (03) stages.

Results: *(Queries depicting the above said activity performed individually and snapshots of the results (if any))*

USE COURIER NEW FONT WITH SIZE = 10 FOR QUERY STATEMENTS

1)

```
CREATE TABLE family (
  member_id SERIAL PRIMARY KEY,
  name TEXT,
  age INTEGER,
  relation TEXT,
  occupation TEXT,
```

Page No.:

```

city TEXT,
phone_number TEXT,
email TEXT
);

```

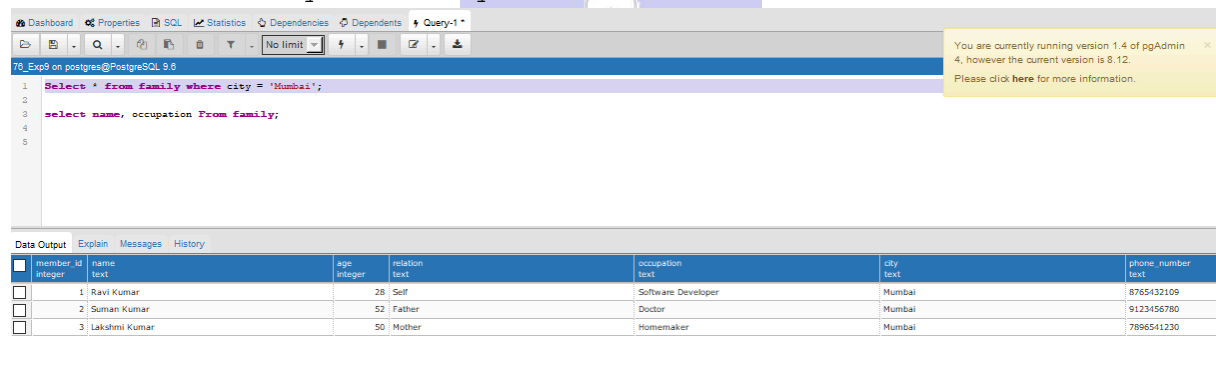
```

INSERT INTO family (name, age, relation, occupation, city, phone_number,
email)
VALUES
('Ravi Kumar', 28, 'Self', 'Software Developer', 'Mumbai',
'8765432109', 'ravi.kumar@email.com'),
('Suman Kumar', 52, 'Father', 'Doctor', 'Mumbai', '9123456780',
'suman.kumar@email.com'),
('Lakshmi Kumar', 50, 'Mother', 'Homemaker', 'Mumbai', '7896541230',
'lakshmi.kumar@email.com'),
('Anjali Singh', 23, 'Sister', 'Designer', 'Delhi', '6655443322',
'anjali.singh@email.com'),
('Rohit Verma', 34, 'Uncle', 'Engineer', 'Bangalore', '9988776655',
'rohit.verma@email.com'),
('Sneha Verma', 30, 'Aunt', 'Teacher', 'Chennai', '5566778899',
'sneha.verma@email.com');

```

2)

Select * from family where city = 'Mumbai';



The screenshot shows the pgAdmin 4 interface. The SQL query editor contains the following query:

```

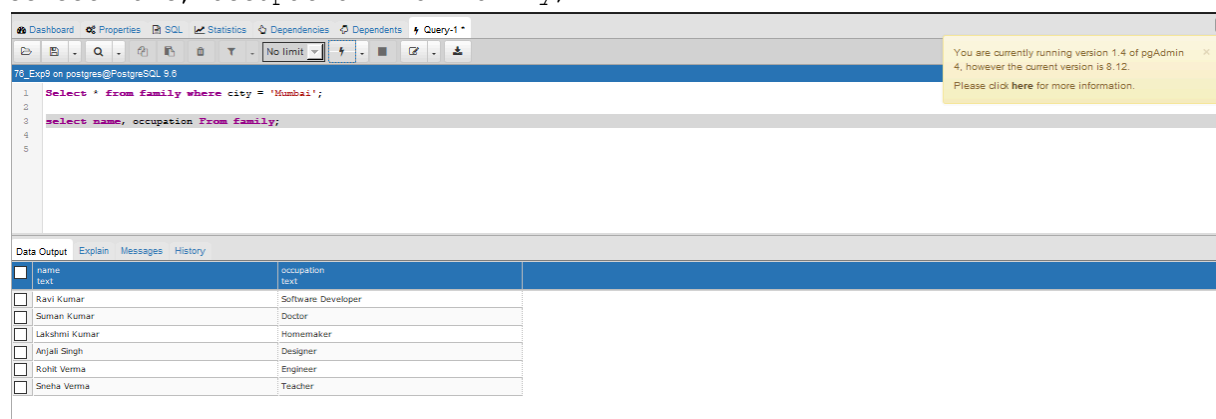
1 Select * from family where city = 'Mumbai';
2
3 select name, occupation From family;
4
5

```

The query results are displayed in the Data Output tab, showing three rows of data:

member_id	name	age	relation	occupation	city	phone_number
1	Ravi Kumar	28	Self	Software Developer	Mumbai	8765432109
2	Suman Kumar	52	Father	Doctor	Mumbai	9123456780
3	Lakshmi Kumar	50	Mother	Homemaker	Mumbai	7896541230

select name, occupation From family;



The screenshot shows the pgAdmin 4 interface. The SQL query editor contains the following query:

```

1 Select * from family where city = 'Mumbai';
2
3 select name, occupation From family;
4
5

```

The query results are displayed in the Data Output tab, showing three rows of data:

name	occupation
Ravi Kumar	Software Developer
Suman Kumar	Doctor
Lakshmi Kumar	Homemaker
Anjali Singh	Designer
Rohit Verma	Engineer
Sneha Verma	Teacher

3)

Then download the json file.

Name	Date modified	Type	Size
family.json	10/25/2024 12:00 PM	JSON Source File	1 KB

4)

MongoDB

Import JSON file into the MongoDB collection.

5)

```
db.family.insertOne({
  "name": "Karan Mehta",
  "age": 22,
  "relation": "Cousin",
  "occupation": "Analyst",
  "city": "Ahmedabad",
  "phone_number": "6677889900",
  "email": "karan.mehta@email.com"
});
```

```
> db.family.insertOne({
  "name": "Karan Mehta",
  "age": 22,
  "relation": "Cousin",
  "occupation": "Analyst",
  "city": "Ahmedabad",
  "phone_number": "6677889900",
  "email": "karan.mehta@email.com"
});
< {
  acknowledged: true,
  insertedId: ObjectId('671b412067ee345522c3b669')
}
```

```
db.family.updateOne(
  { "name": "Ravi Kumar" },
  { $set: { "occupation": "Project Manager" } }
);
```

```
> db.family.updateOne(
  { "name": "Ravi Kumar" },
  { $set: { "occupation": "Project Manager" } }
);
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

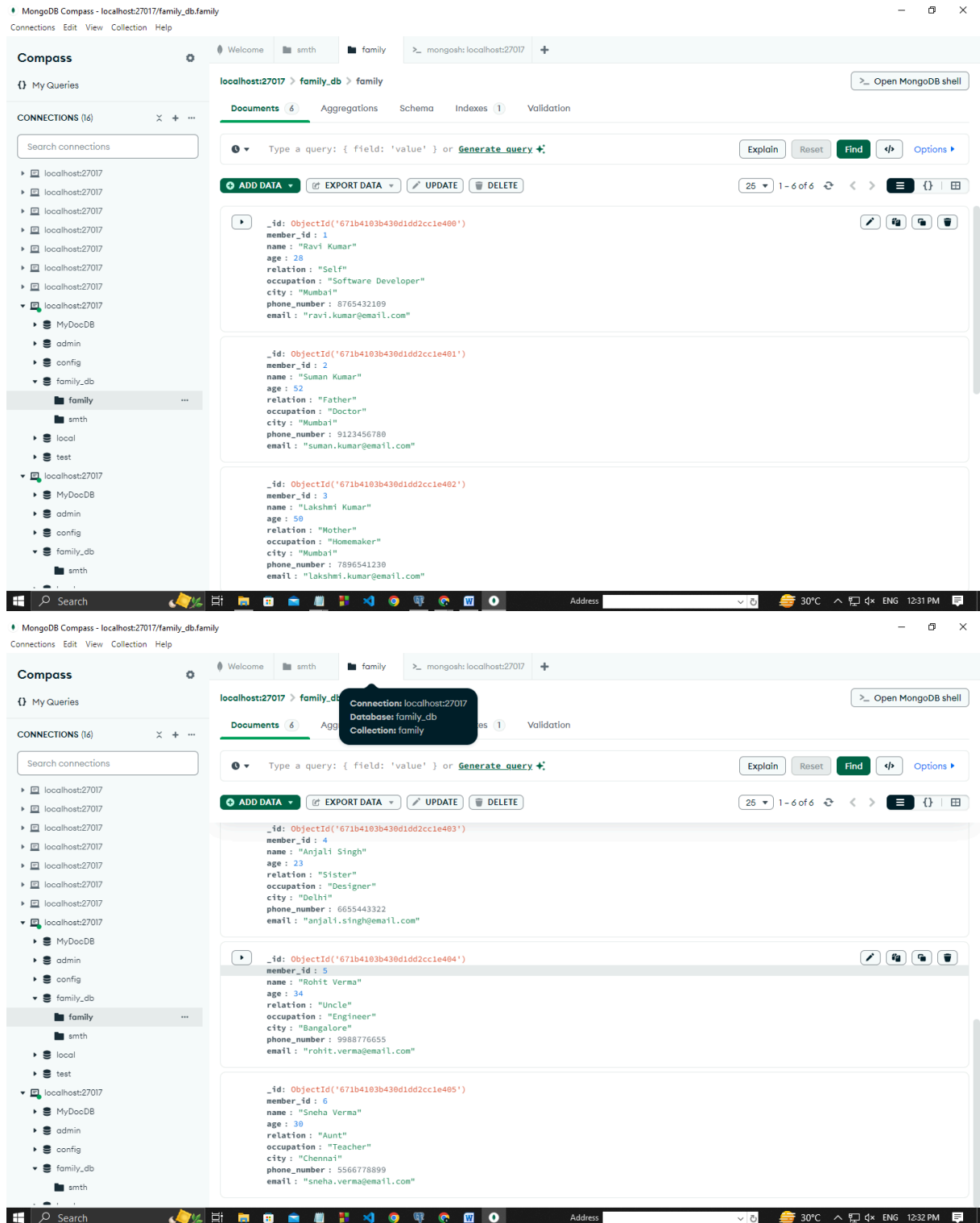
```
db.family.deleteOne({ "name": "Anjali Singh" });
```

```
> db.family.deleteOne({ "name": "Anjali Singh" });
< {
  acknowledged: true,
  deletedCount: 1
}
```

```
db.family.aggregate([
  { $group: { _id: "$city", average_age: { $avg: "$age" } } },
  { $sort: { average_age: -1 } },
  { $project: { _id: 0, city: "$_id", average_age: 1 } }
]);
```

```
> db.family.aggregate([
  { $group: { _id: "$city", average_age: { $avg: "$age" } } },
  { $sort: { average_age: -1 } },
  { $project: { _id: 0, city: "$_id", average_age: 1 } }
]);
< {
  average_age: 43.333333333333336,
  city: 'Mumbai'
}
{
  average_age: 34,
  city: 'Bangalore'
}
{
  average_age: 30,
  city: 'Chennai'
}
{
  average_age: 22,
  city: 'Ahmedabad'
}
```

Final Collection :



Questions:

Explain with query implementation on relation created by you

1. Any five jsonb specific operators in PostgreSQL

- @> (Contains):

This operator checks if the left JSONB object contains the right JSONB object.

Example: `SELECT '{"name": "John", "age": 30}::jsonb @> '{"name": "John"}' AS contains_name;`

Output: true

- **<@ (Is Contained By):**
This operator checks if the left JSONB object is contained within the right JSONB object.
Example: `SELECT '{"name": "John"}'::jsonb <@ '{"name": "John", "age": 30}'::jsonb AS is_contained;`
Output: true

- **? (Key Exists):**
This operator checks if a specified key exists in the JSONB object.
Example: `SELECT '{"name": "John", "age": 30}'::jsonb ? 'name' AS key_exists;`
Output: true

- **#> (Get JSON Object at Path):**
This operator retrieves the JSON object at the specified path.
Example: `SELECT '{"a": {"b": {"c": "value"}}}'::jsonb #> '{a,b}' AS value;`
Output: `{"c": "value"}`

- **-> (Get JSON Object Field):**
This operator retrieves a JSON object field as JSON.
Example: `SELECT '{"name": "John", "age": 30}'::jsonb -> 'name' AS name_field;`
Output: "John"

2. Any five collection methods in MongoDB (Besides `db.collection.insertOne`, `db.collection.deleteOne`, `db.collection.updateOne`, `db.collection.find`)

- **db.collection.findOne():**
This method retrieves a single document from a collection.
Example: `db.collection.findOne({ name: "John" });`
- **db.collection.countDocuments():**
This method counts the number of documents that match a specified filter.
Example: `db.collection.countDocuments({ age: { $gte: 30 } });`
- **db.collection.aggregate():**
This method performs aggregation operations on the collection.
Example: `db.collection.aggregate([{ $match: { age: { $gte: 30 } } }, { $group: { _id: "$age", count: { $sum: 1 } } }]);`
- **db.collection.findAndModify():**
This method atomically finds a document and modifies it.
Example: `db.collection.findAndModify({ query: { name: "John" }, update: { $set: { age: 31 } }, new: true });`
- **db.collection.createIndex():**
This method creates an index on a specified field to improve query performance.
Example: `db.collection.createIndex({ name: 1 });`

Outcomes:

CO3. Illustrate the concept of security, query processing, indexing and normalization for relational databases.

Conclusion: (Conclusion to be based on outcomes achieved)

From this experiment, I gained a deeper understanding of both relational and non-relational database management systems. By creating a family table in PostgreSQL, I was able to implement various SQL queries, demonstrating how to extract and manipulate data efficiently. Transitioning to MongoDB, I practiced importing JSON data, performing insertions, updates, and aggregations, which showcased the strengths of document-oriented databases.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

1. <https://www.mongodb.com/basics>
2. <https://www.postgresql.org/about/>
3. <https://www.postgresql.org/docs/13/datatype-json.html>

