

Introduction to NoSQL (4360704)

Laboratory Manual

Semester-6

Diploma in Computer Engineering

Enrolment Number	
Name	
Branch	Computer Engineering
Academic Term	2025-26 (Even Term)
Institute	AVPTI, Rajkot



**Directorate Of Technical Education
Gandhinagar - Gujarat**

DTE's Vision:

To facilitate quality technical and professional education having relevance for both industry and society, with moral and ethical values, giving equal opportunity and access, aiming to prepare globally competent technocrats.

DTE's Mission:

- To provide globally competitive technical education;
- Remove geographical imbalances and inconsistencies;
- Develop student friendly resources with a special focus on girls' education and support to weaker sections;
- Develop programs relevant to industry and create a vibrant pool of technical professionals.

Institute's Vision:

To cater skilled engineers having potential to convert global challenges into opportunities through embedded values and quality technical education.

Institute's Mission:

- Impart quality technical education and prepare diploma engineering professionals to meet the need of industries and society.
- Adopt latest tools and technologies for promoting systematic problem solving skills to promote innovation and entrepreneurship
- Emphasize individual development of students by inculcating moral, ethical and life skills.

Department's Vision:

Develop globally competent Computer Engineering Professionals to achieve excellence in an environment conducive for technical knowledge, skills, moral values and ethical values with a focus to serve the society

Department's Mission:

- To provide state of the art infrastructure and facilities for imparting quality education and computer engineering skills for societal benefit.
- Adopt industry-oriented curriculum with an exposure to technologies for building systems & application in computer engineering
- To provide quality technical professional as per the industry and societal needs, encourage entrepreneurship, nurture innovation and life skills in consonance with latest interdisciplinary trends.

Certificate

This is to certify that Mr./Ms.

Enrolment No. of Semester: 6th of Diploma in
Computer Engineering of Institute AVPTI, Rajkot (GTU Code: 602) has
satisfactorily completed the term work in course Introduction to NoSQL
(4360704) for the Academic Year: 2023-24 (Even Term) as prescribed in the
GTU curriculum.

Place: Rajkot

Date:

Faculty Signature

Programme Outcomes (POs):

Following Programme outcomes are expected to be achieved through the practical of the course:

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the engineering problems.
2. **Problem analysis:** Identify and analyse well-defined engineering problems using codified standard methods.
3. **Design/development of solutions:** Design solutions for engineering well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern engineering tools and appropriate technique to conduct standard tests and measurements.
5. **Engineering practices for society, sustainability and environment:** Apply appropriate technology in context of society, sustainability, environment and ethical practices.
6. **Project Management:** Use engineering management principles individually, as a team member or a leader to manage projects and effectively communicate about well-defined engineering activities.
7. **Life-long learning:** Ability to analyses individual needs and engage in updating in the context of technological changes in field of engineering.

Program Specific Outcomes (PSOs)

- Able to apply the knowledge gained from Mathematics, Basic Sciences in general and all computer science courses in particular to identify, formulate and solve real life complex engineering problems faced in industries and society.
- The ability to employ modern computer languages, environments and platforms in creating innovative career paths in Hardware, Networking and Software Development technologies.

Practical Outcome - Course Outcome matrix

Course Outcomes (COs):

- CO1. Analyze the impact of the CAP theorem on various NoSQL databases, highlighting the trade-offs between consistency, availability, and partition tolerance in database systems.
- CO2. Apply MongoDB's features and basic CRUD operations to design and manipulate data structures effectively, demonstrating proficiency in utilizing a document-oriented database.
- CO3. Demonstrate Cassandra's data model and query language (CQL), showcasing the ability to create and manage distributed data tables efficiently.
- CO4. Identify the significance of graph databases, illustrating their practical applications in solving complex relationship-oriented problems.
- CO5. Utilize Redis data structures and functionalities to implement efficient caching strategies, showcasing the role of Redis in enhancing data retrieval performance.

Sr. No.	Experiment/Practical Outcome	CO1	CO2	CO3	CO4	CO5
1.	Introduction and Types of NoSQL Databases	✓	-	-	-	-
2.	Introduction and Installation of MongoDB	-	✓	-	-	-
3.	Basic CRUD Operations with MongoDB	-	✓	-	-	-
4.	Introduction and Setup of Cassandra	-	-	✓	-	-
5.	Data Modeling and Simple Queries with Cassandra	-	-	✓	-	-
6.	Introduction to Neo4j Graph Databases	-	-	-	✓	-
7.	Basic Graph Queries and Implementations with Neo4j	-	-	-	✓	-
8.	Redis Basics: Introduction and Key-Value Operations	-	-	-	-	✓

Progressive Assessment Sheet / Index

Sr. No	Experiment/Practical Outcome	Date	Marks (25)	Sign
1.	Introduction and Types of NoSQL Databases			
2.	Introduction and Installation of MongoDB			
3.	Basic CRUD Operations with MongoDB			
4.	Introduction and Setup of Cassandra			
5.	Data Modeling and Simple Queries with Cassandra			
6.	Introduction to Neo4j Graph Databases			
7.	Basic Graph Queries and Implementations with Neo4j			
8.	Redis Basics: Introduction and Key-Value Operations			

Rubrics for Continuous Assessment- CA (25 Marks)

Component	Criteria	Marks	Assessment
Laboratory Work and Questionnaire	Excellent	(23-25)	Demonstrates exceptional proficiency in both laboratory work and questionnaire assessments, consistently applying skills and understanding effectively.
	Proficient	(18-22)	Shows a strong command of both laboratory work and questionnaire assessments, with minor areas for improvement.
	Satisfactory	(13-17)	Achieves a satisfactory level of performance in laboratory work and questionnaire assessments, with room for improvement in some areas.
	Needs Improvement	(8-12)	Demonstrates limited proficiency in both laboratory work and questionnaire assessments, with significant areas for improvement.
	Inadequate	(0-7)	Fails to meet acceptable standards in both laboratory work and questionnaire assessments; significant improvement is required.

Practical 1. Introduction and Types of NoSQL Databases

A. Objective

To provide students with a comprehensive understanding of NoSQL databases, their underlying principles, and the various types available. It focuses on the fundamental CAP theorem in distributed systems and delves into the unique characteristics of various NoSQL databases. Ultimately, the practical aims to empower students to make informed decisions in selecting a database solution aligned with specific project needs.

B. Relevant Program Outcomes (POs)

1. Application of Fundamental Knowledge:

- Apply foundational knowledge in computer science and information technology to comprehend the principles and types of NoSQL databases.

2. Problem Analysis in Database Context:

- Identify and analyze specific engineering challenges related to NoSQL databases using established methodologies and principles.

3. Designing Solutions for NoSQL Issues:

- Develop solutions for well-defined technical problems in the context of NoSQL databases, contributing to the design of effective data management systems.

4. Utilization of Engineering Tools for Database Testing:

- Apply modern engineering tools and testing techniques to assess and validate the performance of NoSQL databases, ensuring adherence to standard practices.

5. Engineering Practices in NoSQL Context:

- Integrate appropriate NoSQL technologies considering societal impact, sustainability, environmental aspects, and ethical considerations in database design and management.

6. Project Management in NoSQL Implementation:

- Utilize engineering management principles to coordinate and communicate effectively within a team while managing projects related to NoSQL database development.

7. Commitment to Continuous Learning in NoSQL Technology:

- Demonstrate the ability to analyze personal learning needs and engage in lifelong learning, particularly in the evolving landscape of NoSQL databases and related technologies.

C. Competency and Practical Skills

1. NoSQL Fundamentals:

- Develop a strong foundation in NoSQL databases, covering principles, architecture, and essential characteristics.

2. CAP Theorem Insight:

- Understand the implications of the CAP theorem on distributed systems, emphasizing the trade-offs between consistency, availability, and partition tolerance.

3. NoSQL Types Proficiency:

- Differentiate between document-oriented, key-value, column-family, and graph databases, discerning their specific applications.

4. Decision-Making and Application:

- Apply theoretical knowledge to practical scenarios, making informed decisions on NoSQL database selection based on project requirements.

D. Relevant Course Outcomes (Cos)

Analyze the impact of the CAP theorem on various NoSQL databases, highlighting the trade-offs between consistency, availability, and partition tolerance in database systems.

E. Practical Outcome (PRo)

- Attain a thorough grasp of NoSQL databases, covering principles, CAP theorem insights, and diverse database features.

- Develop skills to make informed choices by exploring real-world use cases, understanding factors influencing database selection, and conducting comparative analyses.

- Apply theoretical knowledge practically, ensuring students can implement NoSQL concepts and make effective decisions in various project scenarios.

F. Expected Affective Domain Outcome (ADos)

1. Enhanced Interest and Engagement:

- Foster increased interest and engagement among students, promoting a positive attitude towards the study of NoSQL databases.

2. Improved Decision-Making Confidence:

- Strengthen students' confidence in making well-informed decisions regarding database choices through practical insights and hands-on experience.

G. Prerequisite Theory

Introduction to NoSQL Databases

NoSQL databases, or "Not Only SQL," represent a diverse category of database management systems designed to address the limitations of traditional relational databases.

NoSQL databases are defined as non-tabular databases that handle data storage differently as compared to relational tables. Unlike SQL databases, NoSQL databases are schema-less and provide a more flexible data model, allowing for the storage and retrieval of large volumes of unstructured or semi-structured data. They are particularly well-suited for handling dynamic and rapidly evolving data in distributed and scalable environments.

NoSQL databases are gaining traction for real-time cloud, web, and big data applications due to their key features:

1. Multiple Data Model Compatibility:

- NoSQL supports various data models, allowing flexibility in handling unstructured, semi-structured, and structured data.
- Ideal for Agile development, as it accommodates different data models without the need for separate databases.

2. Enhanced Scalability and Availability:

- NoSQL offers simplified scalability with a serverless, peer-to-peer architecture, allowing for horizontal scaling and improved performance.
- Sharding enables efficient handling of massive data volumes, and auto-replication ensures high availability in case of failures.

3. Global Data Distribution:

- Advanced NoSQL databases facilitate global data distribution by operating across multiple cloud regions and data centers.
- Contrastingly, relational databases often rely on centralized applications, while NoSQL minimizes wait times by distributing data globally.

4. Minimal Downtime:

- NoSQL databases feature robustness and minimal downtime.
- Serverless architecture and data replication across nodes ensure business continuity, with alternative nodes providing access in case of a malfunction.

Can NoSQL completely replace relational databases?

No, NoSQL cannot entirely replace relational databases. The choice between the two depends on the specific needs of an organization. NoSQL is preferred for handling diverse data types in cloud and web applications with a broad and rapidly growing user base. Its flexibility, multi-modality, scalability, availability, and global distribution make it ideal for certain use cases. However, both database types can coexist and complement each other, allowing organizations to leverage the strengths of each based on their requirements.

Types of NoSQL Databases

There are four main types of NoSQL databases, each catering to specific use cases:

1. Document-oriented Databases:

- Store data in flexible, JSON-like documents.
- Examples: MongoDB, CouchDB.

2. Key-Value Stores:

- Simplest NoSQL model, storing data as key-value pairs.
- Examples: Redis, DynamoDB.

3. Column-family Stores:

- Organize data into columns rather than rows.
- Examples: Apache Cassandra, HBase.

4. Graph Databases:

- Designed for managing and querying interconnected data.
- Examples: Neo4j, Amazon Neptune.

CAP Theorem

The CAP theorem, originally known as the CAP principle, illuminates the inherent trade-offs in designing distributed systems with replication. It highlights the challenge of simultaneously achieving consistency, availability, and partition tolerance in such systems.

1. Consistency (C):

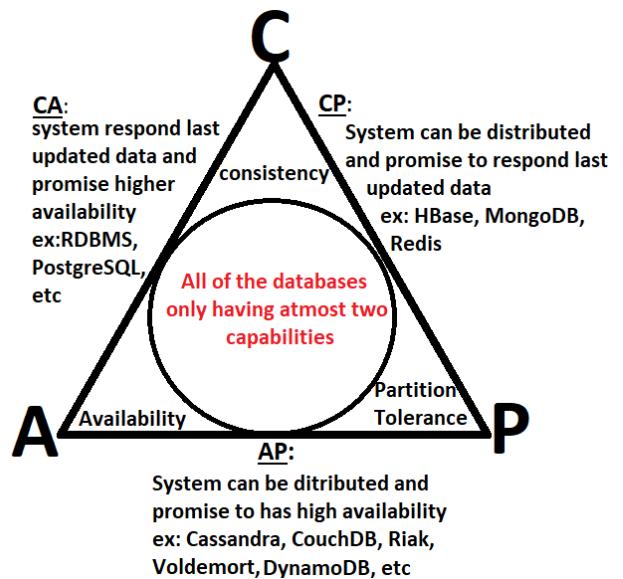
- All nodes in the system see the same data at the same time.

2. Availability (A):

- Every request receives a response, without guarantee that it contains the most recent version of the information.

3. Partition Tolerance (P):

- The system continues to operate despite network partitions that may cause communication failures between nodes.



The CAP theorem asserts that distributed databases can prioritize at most two of these three properties. Database systems fall into categories based on their priorities:

- CA (Consistency and Availability): Prioritizing availability over consistency, responding with possibly stale data. Examples include Cassandra, CouchDB, Riak, and Voldemort.
- AP (Availability and Partition Tolerance): Emphasizing availability over consistency, possibly responding with stale data. Distributed across nodes, designed to operate reliably despite network partitions. Examples include Amazon DynamoDB and Google Cloud Spanner.
- CP (Consistency and Partition Tolerance): Focusing on consistency over availability, responding with the latest updated data. Distributed across nodes, designed to operate reliably in the face of network partitions. Examples include Apache HBase, MongoDB, and Redis.

It's crucial to note that database systems may exhibit different behaviors based on configurations and settings, influencing their consistency, availability, and partition tolerance. Even in the case of specialized databases like Neo4j, which prioritizes consistency and partition tolerance over availability, the CAP theorem remains relevant. In situations of network partition or failure, Neo4j sacrifices availability to maintain consistency.

System architects must make trade-offs based on the application's requirements and the characteristics of the underlying NoSQL database.

Comparisons between MongoDB, Cassandra, Neo4j, and Redis

Criteria	MongoDB	Cassandra	Neo4j	Redis
Data Model	Document-oriented (BSON)	Column-family	Graph	Key-value store
Query Language	MongoDB Query Language (MQL)	CQL (Cassandra Query Language)	Cypher Query Language	N/A (Command-based)
Schema	Dynamic schema	Dynamic schema	Schema-based (flexible)	Schema less
Scalability	Horizontal scaling through sharding	Horizontal scaling	Horizontal scaling	Horizontal scaling
Consistency	Tunable consistency (Eventual to Strong)	Tunable consistency (Eventual to Strong)	Strong consistency	Eventual consistency
Partition Tolerance	Yes	Yes	Yes	Yes
Use Case Focus	General-purpose, document storage	Wide-column store, time-series data	Graph-based, relationships	Cache, real-time analytics
Indexing	Multiple types, including compound and geo-spatial indexes	Secondary indexes	Indexing based on relationships	Limited indexing options
Query Performance	Good	Excellent	Excellent for graph queries	Very fast (in-memory storage)
ACID Transactions	Supports ACID transactions	Limited support for transactions	Supports ACID transactions	Supports transactions through multi commands
Community Support	Large and active community	Active community	Active community	Active community
Use Cases	Content management systems, e-commerce, real-time analytics	Time-series data, logging, IoT	Social networks, fraud detection	Caching, message broker

Factors Influencing Choice of Database

- Data Model: Choose based on whether document-oriented, key-value, column-family, or graph data models are most suitable for the application.
- Scalability Requirements: Consider the need for horizontal scalability and how well the database can distribute data across multiple nodes.
- Consistency and Availability Needs: Assess the trade-offs between consistency and availability based on the application's requirements.
- Use Case Characteristics: Tailor the choice of database to the specific demands of the application, such as read and write patterns, data complexity, and relationships.
- Development and Operational Considerations: Evaluate factors like ease of development, community support, and operational overhead when choosing a NoSQL database for a project.

In conclusion, understanding the nuances of NoSQL databases, their types, trade-offs, and specific use cases is crucial for making informed decisions in selecting the right database solution for diverse applications.

H. Resources/Equipment Required

1. Internet Connectivity:

- Reliable internet connectivity is essential for accessing online resources, databases, and relevant documentation.

2. Documentation and Learning Materials:

- Provide comprehensive documentation, tutorials, and learning materials to guide students through the practical exercises.

Practical 2. Introduction and Installation of MongoDB

A. Objective

Familiarize students with MongoDB through an introduction to its features and advantages, followed by hands-on experience installing and connecting to the database.

B. Relevant Program Outcomes (POs)

1. Basic and Discipline-Specific Knowledge:

- Apply fundamental engineering knowledge to understand MongoDB's role and significance in modern database systems.

2. Problem Analysis:

- Identify and analyze engineering challenges related to MongoDB installation and connectivity using established methods.

3. Design/Development of Solutions:

- Design solutions for well-defined technical problems encountered during the installation and connection processes in MongoDB.

4. Engineering Tools, Experimentation, and Testing:

- Utilize appropriate engineering tools to conduct experiments and tests during the MongoDB installation, ensuring proficiency in modern database technologies.

5. Engineering Practices for Society, Sustainability, and Environment:

- Apply MongoDB installation practices considering societal, sustainability, and environmental aspects, adhering to ethical considerations.

6. Project Management:

- Demonstrate project management skills while handling the installation of MongoDB, either individually or as part of a team, and effectively communicate progress.

7. Life-Long Learning:

- Exhibit the ability to adapt and engage in continuous learning, updating skills to match technological changes in the field of database management, particularly MongoDB.

C. Competency and Practical Skills

1. Technical Proficiency:

- Demonstrate competence in installing MongoDB, understanding its features, and navigating its environment.

2. Problem-Solving:

- Apply analytical skills to troubleshoot and resolve issues related to MongoDB installation and connectivity.

3. Documentation and Communication:

- Effectively document the installation process and communicate findings, demonstrating clarity and precision in conveying technical information.

4. Adaptability:

- Exhibit adaptability by quickly grasping MongoDB's installation procedures and staying abreast of any updates or changes in the installation process.

D. Relevant Course Outcomes (Cos)

Apply MongoDB's features and basic CRUD operations to design and manipulate data structures effectively, demonstrating proficiency in utilizing a document-oriented database.

E. Practical Outcome (PRo)

Proficiently install MongoDB, configure, and establish successful connectivity to designated instance

F. Expected Affective Domain Outcome (ADos)

Develop a positive attitude towards MongoDB installation, fostering confidence and competence.

G. Prerequisite Theory

Overview of MongoDB

MongoDB is a widely-used, open-source NoSQL database that embraces a document-oriented data model. Unlike traditional relational databases, MongoDB stores data in flexible, JSON-like documents, allowing for dynamic schemas and easy scalability. It offers high performance, horizontal scalability, and is particularly well-suited for applications with rapidly evolving schemas and large amounts of unstructured or semi-structured data.

NoSQL databases vs. relational databases

Relational Database	NoSQL
It is used to handle data coming in low velocity.	It is used to handle data coming in high velocity.
It gives only read scalability.	It gives both read and write scalability.
It manages structured data.	It manages all type of data.
Data arrives from one or few locations.	Data arrives from many locations.

It supports complex transactions.	It supports simple transactions.
It has single point of failure.	No single point of failure.
It handles data in less volume.	It handles data in high volume.
Transactions written in one location.	Transactions written in many locations.
support ACID properties compliance	doesn't support ACID properties
It's difficult to make changes in database once it is defined	Enables easy and frequent changes to database
schema is mandatory to store the data	schema design is not required
Deployed in vertical fashion.	Deployed in Horizontal fashion.

BSON (Binary JSON) data format

BSON is a binary encoded JavaScript Object Notation (JSON)—a textual object notation widely used to transmit and store data across web based applications. JSON is easier to understand as it is human-readable, but compared to BSON, it supports fewer data types.

This format optimizes data storage and retrieval, ensuring efficiency in MongoDB's operations. Key aspects of BSON include:

1. Binary Encoding:

BSON employs binary encoding to represent JSON-like documents in a compact and efficient manner. This binary format facilitates faster data processing and reduces storage space compared to traditional text-based JSON.

2. Data Types:

BSON supports a rich set of data types, including strings, integers, floating-point numbers, arrays, and nested documents. This versatility allows MongoDB to handle diverse and complex data structures, accommodating the dynamic nature of modern applications.

3. Efficient Storage:

The binary nature of BSON enhances data storage efficiency by eliminating the need for textual parsing. This results in smaller document sizes, reducing I/O operations and improving overall performance, especially in scenarios with large datasets.

4. Extended Data Types:

BSON introduces additional data types not found in standard JSON, such as Date and Binary. These extensions enhance MongoDB's ability to represent diverse information, including date and time values and binary data like images or multimedia content.

5. Support for Querying and Indexing:

BSON's structure aligns with MongoDB's query language and indexing capabilities, enabling efficient searching and retrieval of data. MongoDB can leverage BSON's binary format to accelerate query execution and optimize the performance of read and write operations.

Understanding BSON is essential for MongoDB developers, as it forms the foundation for storing and processing data within the database. Its binary representation contributes to MongoDB's speed, flexibility, and scalability, making it a key component in the success of MongoDB as a NoSQL database solution.

Use cases and scenarios where MongoDB excels

1. Content Management Systems (CMS):

MongoDB is well-suited for CMS applications where content structures can vary widely. Its flexible document-oriented model allows developers to adapt to changing content requirements without compromising performance. MongoDB's ability to handle large volumes of unstructured data makes it an ideal choice for storing and retrieving diverse content types efficiently.

2. Real-time Analytics:

In scenarios requiring real-time data analysis, MongoDB excels by providing fast read and write operations. Its document-oriented data model allows for the storage of complex and varied data, facilitating quick querying and analysis. MongoDB's support for horizontal scaling ensures responsiveness even with high volumes of real-time data.

3. Internet of Things (IoT) Applications:

MongoDB is a preferred database for IoT projects where massive amounts of sensor data are generated. Its ability to handle diverse data types and scale horizontally makes it suitable for storing and retrieving sensor readings, device information, and other IoT-related data. MongoDB's flexibility accommodates the dynamic nature of IoT environments.

4. E-commerce Platforms:

MongoDB is well-suited for e-commerce applications handling a vast range of product information, customer data, and transaction histories. Its ability to model complex relationships between products and categories, coupled with horizontal scaling capabilities, ensures high performance and scalability for e-commerce platforms, especially during peak times.

5. Log and Event Tracking:

MongoDB excels in log and event tracking applications where fast insertion and retrieval of log data are crucial. Its support for indexing and efficient querying enables the analysis of logs and events in real-time. MongoDB's horizontal scalability ensures that log and event tracking systems can handle increasing data volumes seamlessly.

6. Mobile App Backend Services:

MongoDB serves as an effective backend database for mobile applications, providing a JSON-like data model that aligns well with the data structures commonly used in mobile development. Its automatic sharding and replication features ensure high availability and reliability for mobile app backend services.

7. Geospatial Applications:

MongoDB's geospatial indexing capabilities make it a suitable choice for applications that involve location-based data, such as mapping and geolocation services. It can efficiently store and retrieve geospatial information, supporting queries based on proximity, distance, and other location-based criteria.

8. Dynamic and Agile Development Environments:

MongoDB is particularly advantageous in development environments that prioritize flexibility and adaptability. Its schema-less design allows developers to iterate quickly, accommodating changes to application requirements without the need for extensive schema modifications.

Understanding these use cases showcases MongoDB's versatility and applicability across a wide range of scenarios, making it a robust choice for developers and organizations seeking a flexible and scalable NoSQL database solution.

MongoDB Features and Advantages

1. Document-Oriented Data Model:

MongoDB employs a flexible, document-oriented data model where data is stored in BSON (Binary JSON) documents. This schema-less approach allows developers to work with dynamic and evolving data structures, accommodating changes without the need for a predefined schema.

2. Dynamic Schema:

MongoDB's dynamic schema enables developers to add fields to documents on the fly. This flexibility is particularly beneficial in scenarios where data structures evolve over time, facilitating agile development and reducing the complexity associated with rigid, predefined schemas.

3. High Performance:

MongoDB is designed for high performance, offering fast read and write operations. Its indexing and query optimization features, combined with the ability to horizontally scale by sharding, make it suitable for handling large volumes of data and supporting applications with demanding performance requirements.

4. Horizontal Scalability:

MongoDB excels in scalable architectures through horizontal scaling. By distributing data across multiple servers via sharding, MongoDB can handle increased data loads and provide seamless scalability without sacrificing performance.

5. Indexing and Query Optimization:

MongoDB supports various indexing techniques, enhancing query performance. Developers can create indexes on fields to speed up data retrieval, and the query optimizer optimizes execution plans for efficient searches, ensuring optimal performance even with extensive datasets.

6. Aggregation Framework:

MongoDB's powerful aggregation framework allows for complex data transformations and analysis within the database. It supports pipeline-style aggregations, enabling developers to perform tasks such as filtering, grouping, and projecting data directly within the database.

7. Replication for High Availability:

MongoDB provides built-in replication features to ensure high availability. By maintaining multiple copies of data across replica sets, MongoDB can continue to operate even in the event of a server failure. Automatic failover mechanisms further enhance the system's reliability.

8. Geospatial Indexing:

MongoDB includes geospatial indexing, making it well-suited for location-based applications. This feature enables efficient storage and retrieval of geospatial data, supporting queries based on proximity, distance, and other location-specific criteria.

9. GridFS for Large File Storage:

MongoDB includes GridFS, a specification for storing and retrieving large files such as images, videos, and audio files. This feature allows developers to seamlessly integrate large file storage within the database, eliminating the need for a separate file storage system.

10. Community and Ecosystem:

MongoDB benefits from a vibrant and active community, providing a rich ecosystem of tools, drivers, and extensions. This community support enhances the development experience and ensures access to a wealth of resources for troubleshooting and optimization.

Understanding and leveraging these features make MongoDB a robust choice for modern applications, offering developers the flexibility, scalability, and performance required to address diverse and evolving data challenges.

Installing MongoDB

System Requirements and Compatibility:

MongoDB is a cross-platform NoSQL database, compatible with various operating systems. Before installation, ensure your system meets the following requirements:

- Operating Systems: Windows, Linux, macOS
- Processor Architecture: 64-bit recommended
- Memory (RAM): Allocate sufficient RAM for optimal performance.
- Disk Space: Plan for data storage, indexes, and system files.
- File System: Choose a supported file system like ext4 (Linux) or NTFS (Windows).
- Network: Ensure robust and low-latency network connectivity.

Downloading the MongoDB Installer:

1. Visit the official MongoDB website (<https://www.mongodb.com/try/download/community>) to access the download page.

MongoDB Atlas

MongoDB Enterprise
Advanced

MongoDB Community
Edition

**MongoDB Community
Server**

MongoDB Community
Kubernetes Operator

Tools

Atlas SQL Interface

Mobile & Edge



MONGODB COMMUNITY SERVER

MongoDB Community Server Download

Community Server

The Community version of our distributed database offers a flexible document data model along with support for:

- Ad-hoc queries
- Secondary indexing
- Real-time aggregations to provide powerful ways to access and analyze your data

[Select package](#)

Try MongoDB Atlas

The database is also offered as a fully-managed service with [MongoDB Atlas](#). Get access to advanced functionality such as:

- Auto-scaling
- Serverless instances
- Full-text search, and data distribution across regions and clouds
- Multi-region and multi-cloud support

[Sign up here](#) or get started from your terminal:

[Homebrew](#) [More](#)

```
$ brew install mongodb-atlas
$ atlas setup
```

2. Select the appropriate version for your operating system (Windows, Linux, or macOS).
3. Choose the desired edition (Community) and click the download button.
4. Follow the prompts to save the installer file to your local machine.

[Solutions](#) [Company](#) [Pricing](#)

Version
7.0.4 (current)

Platform
Windows x64

Package
msi

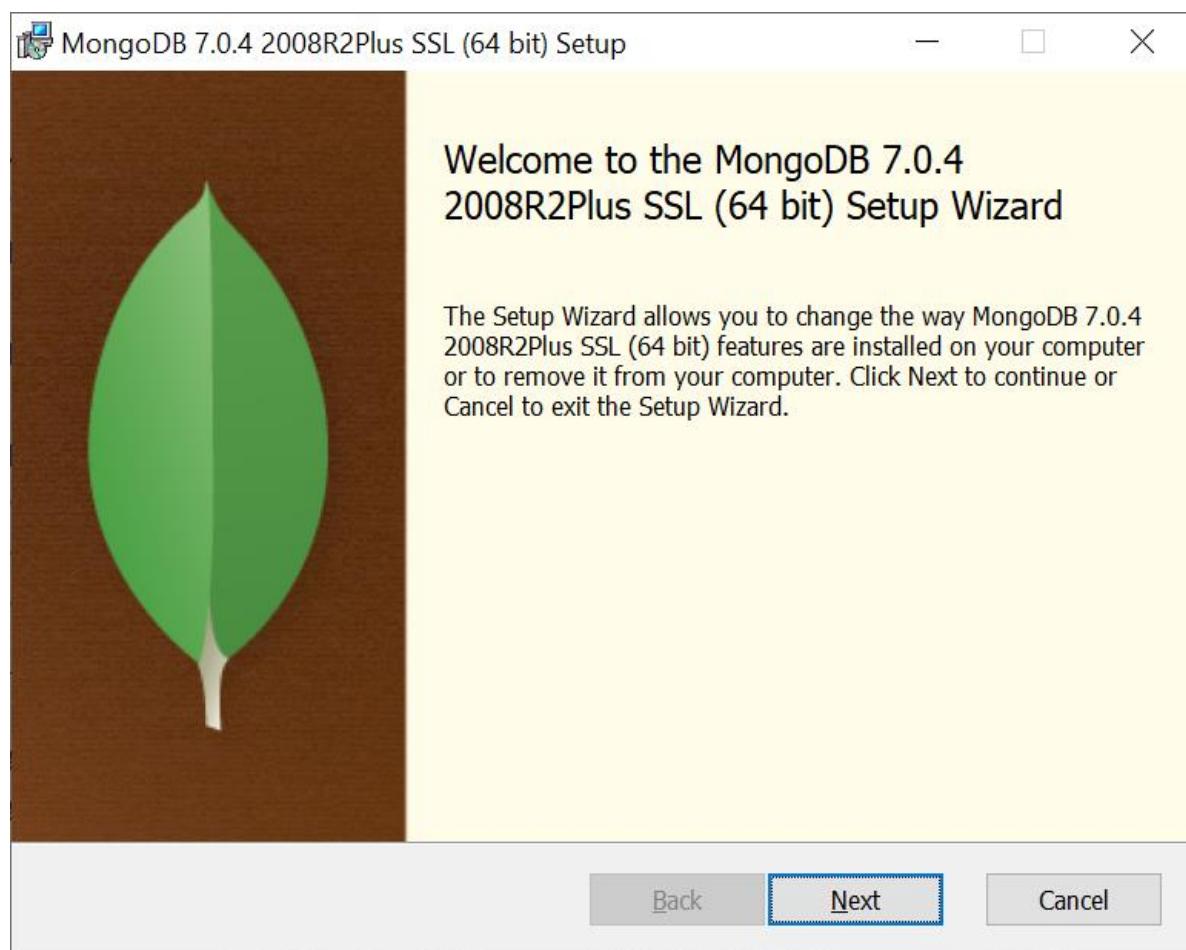
[Download](#)[Copy link](#)[More Options](#)

...

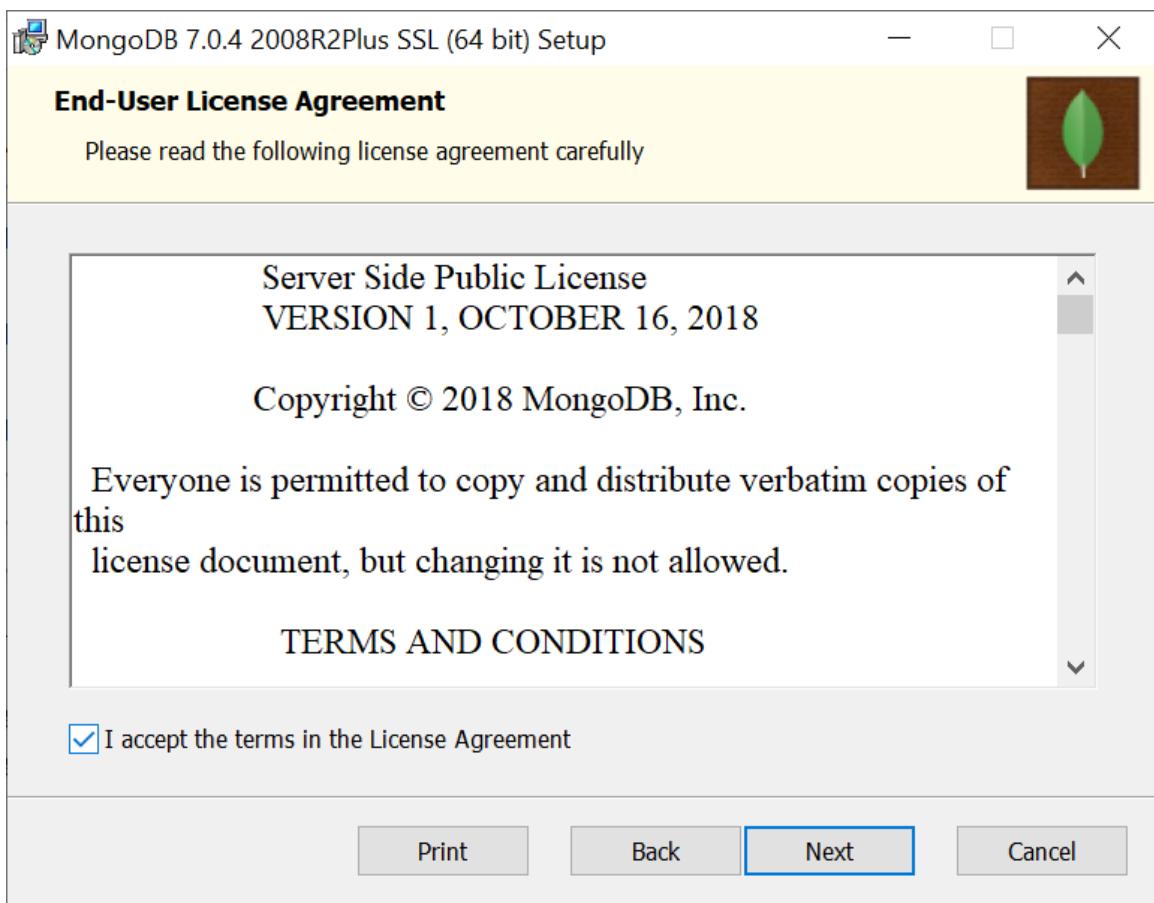
Installation on Windows, Linux, and macOS:

- Windows:

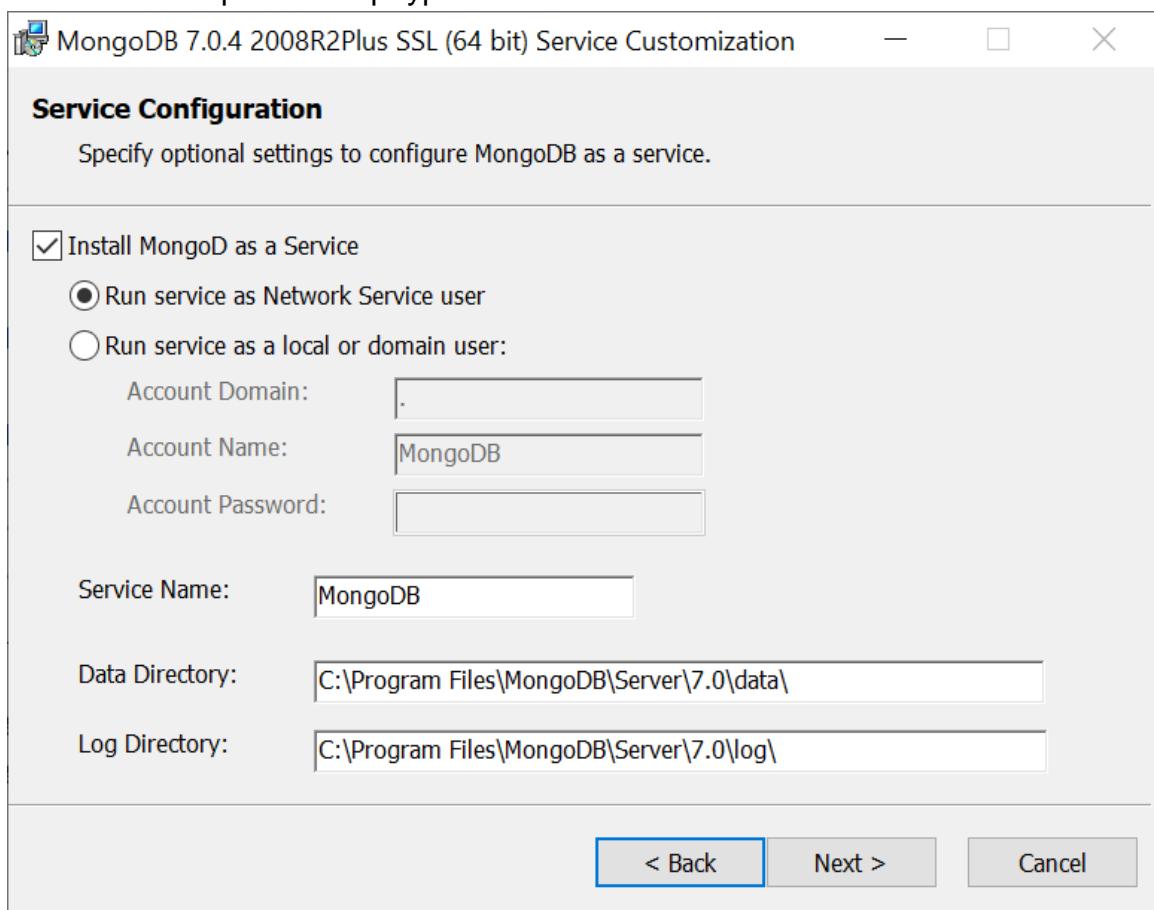
- Run the downloaded installer (.msi file).



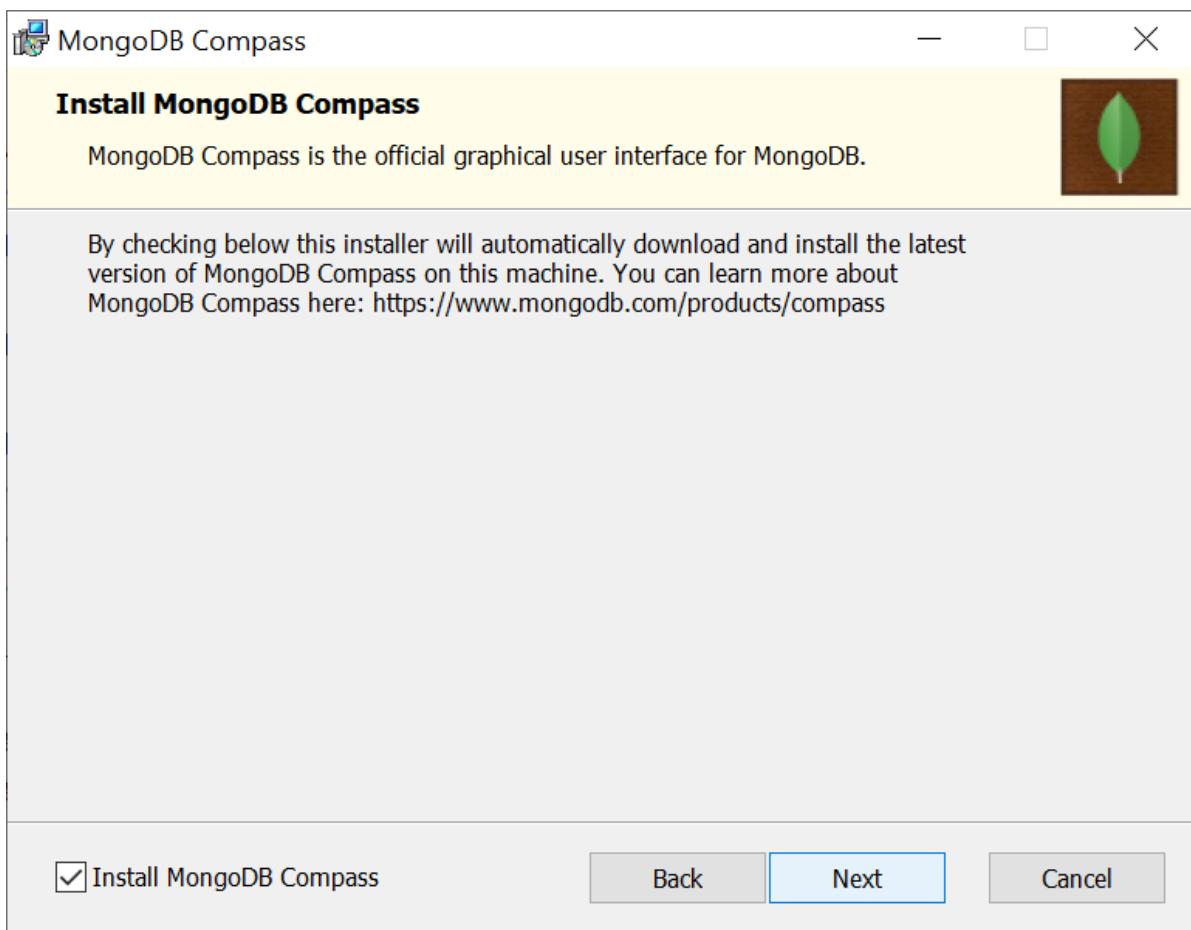
- Follow the installation wizard, accepting the license agreement.



- Choose the "Complete" setup type for a standard installation.



- MongoDB will be installed as a Windows service.



- Linux:

- Extract the downloaded archive (.tgz file).
- Move the extracted files to a desired location.
- Optionally, create a symbolic link to the `bin` directory for easy access.
- MongoDB is now ready for configuration.

- macOS:

- Install MongoDB using Homebrew: `brew tap mongodb/brew && brew install mongodb/brew/mongodb-community`.
- Alternatively, download the .tgz file and follow similar steps as Linux.
- MongoDB is installed and ready for configuration.

Configuration Options During Installation:

- Basic Configuration:
 - Choose the installation directory.
 - Configure whether MongoDB should run as a service (Windows).
- Network Configuration:
 - Specify the port for MongoDB to listen on (default: 27017).
 - Configure network binding options for access control.
- Data and Log Paths:
 - Set the data directory where MongoDB will store its databases.
 - Configure the log path for MongoDB server logs.

Verifying the Installation:

- Open a command prompt or terminal window.
- Run the MongoDB shell: `mongod` .
- Verify the MongoDB server version and connection.

Starting and stopping the MongoDB Server:

- Windows:
 - MongoDB as a service starts automatically. Use `net start MongoDB` and `net stop MongoDB` to manage the service.
- Linux and macOS:
 - Start MongoDB: `mongod` or `sudo service mongod start` .
 - Stop MongoDB: `mongod --shutdown` or `sudo service mongod stop` .

Checking Server Status and Logs:

- Use the following commands to check server status and view logs:
 - `mongod --version`: Display MongoDB version.
 - `mongod --fork --logpath /var/log/mongodb/mongod.log`: Start MongoDB as a background process with logging.
 - `mongo admin --eval "db.runCommand({whatsmyuri: 1})"`: Check the server status and connection URI.

Following these steps ensures a smooth MongoDB installation, configuration, and verification process, allowing you to start working with MongoDB databases on your chosen platform.

Connecting to MongoDB

MongoDB Connection String:

A MongoDB connection string is a URI-like string that specifies how to connect to a MongoDB instance. It includes information such as the host, port, authentication details, and other parameters. The general format is `mongodb://username:password@host:port/database`.

Connecting via MongoDB Shell:

1. Open a terminal or command prompt.
2. Run the `mongo` command.
3. Connect to a MongoDB instance using the connection string: `mongo "mongodb://username:password@host:port/database"

Connecting through a MongoDB Driver (e.g., Python, JavaScript, Java):

Using a MongoDB driver in your preferred programming language:

- Python (PyMongo):

```
from pymongo import MongoClient
client = MongoClient("mongodb://username:password@host:port/database")
```

- JavaScript (Node.js):

```
const MongoClient = require('mongodb').MongoClient;
const client = new MongoClient("mongodb://username:password@host:port/database", {
  useNewUrlParser: true });
```

- Java:

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
MongoClient client =
MongoClients.create("mongodb://username:password@host:port/database");
```

Authentication and User Management:

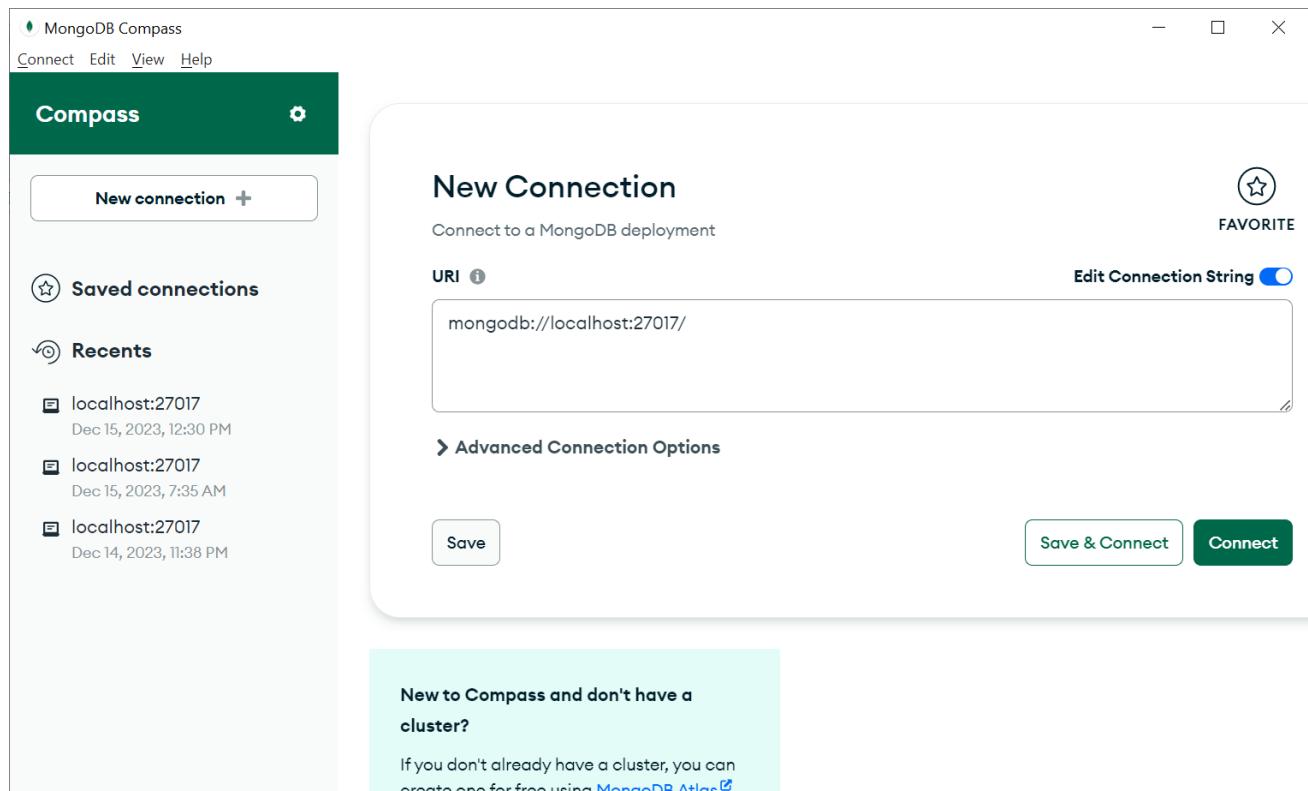
- Create a new user:

```
db.createUser({
  user: "username",
  pwd: "password",
  roles: ["readWrite", "dbAdmin"]
});
```

- Authenticate in the MongoDB shell:

```
db.auth("username", "password");
```

Overview of MongoDB Compass (GUI tool for MongoDB):



MongoDB Compass is a graphical user interface (GUI) tool for MongoDB:

- Features:

- Visual exploration of data.
- Index management and query optimization.
- Real-time server statistics.
- Schema analysis and visualization.
- Query performance profiling.

MongoDB Shell (mongosh)

The MongoDB Shell, mongosh, is a JavaScript and Node.js REPL environment for interacting with MongoDB deployments in Atlas, locally, or on another remote host. Use the MongoDB Shell to test queries and interact with the data in your MongoDB database.

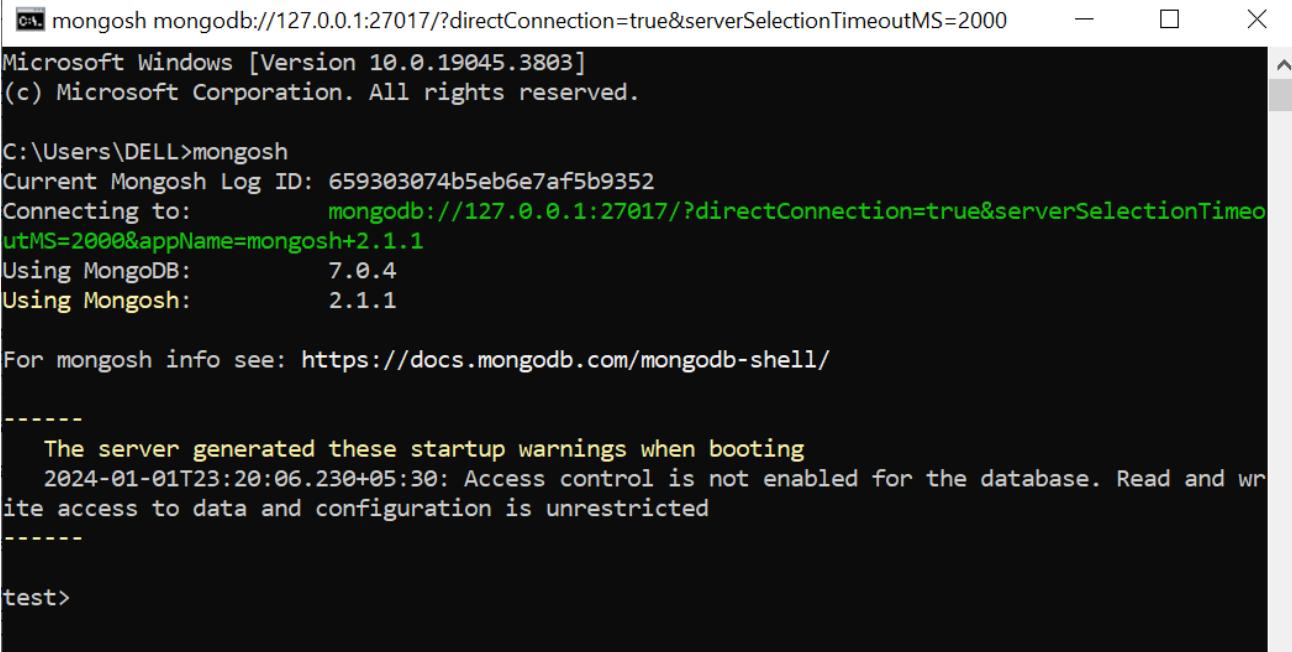
To install mongosh, you can use the MongoDB Community Edition, which includes the MongoDB server and tools, including the mongosh shell. The installation steps vary depending on your operating system.

For Windows:

1. Download the MongoDB Community Edition installer from the official MongoDB website: [MongoDB Community Download](#).
2. Run the installer and follow the installation instructions.

3. Once installed, you can open the mongosh shell from the command prompt or PowerShell.

After installation, you can launch mongosh by typing mongosh in your terminal or command prompt.



```

mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Microsoft Windows [Version 10.0.19045.3803]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>mongosh
Current Mongosh Log ID: 659303074b5eb6e7af5b9352
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeo
utMS=2000&appName=mongosh+2.1.1
Using MongoDB:      7.0.4
Using Mongosh:      2.1.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2024-01-01T23:20:06.230+05:30: Access control is not enabled for the database. Read and wr
ite access to data and configuration is unrestricted
-----
test>

```

Remember to start the MongoDB server (mongod) before using mongosh. The installation process may differ slightly based on updates or changes, so refer to the MongoDB documentation for the most current information: [MongoDB Installation Guides](#).

Helpers

Show Databases

```
show dbs
db // prints the current database
```

Switch Database

```
use <database_name>
```

Show Collections

```
show collections
```

Run JavaScript File

```
load("myScript.js")
```

Connecting to a Remote MongoDB Instance:

When connecting to a MongoDB instance on a remote server:

- Modify the connection string with the remote server's details.

- Ensure the server allows remote connections (bind to all IP addresses or specific IP).

Securing the MongoDB Deployment:

- Encryption:

- Use SSL/TLS for encrypted communication between clients and servers.

- Authentication:

- Require authentication for all connections.

- Create user accounts with minimal necessary privileges.

- Firewall Rules:

- Restrict incoming connections to trusted IP addresses.

- Utilize network security groups to control access.

- Role-Based Access Control (RBAC):

- Assign roles to users based on their responsibilities.

- Regularly review and update user roles.

- Audit Logging:

- Enable MongoDB's audit log to track user activities.

- Monitor and analyze the audit log for security insights.

- Updates and Patching:

- Keep MongoDB and system software up to date with the latest security patches.

- Regularly check for MongoDB updates and apply them promptly.

By following these guidelines, you can establish a secure MongoDB deployment, ensuring that data is protected and access is restricted to authorized users and applications.

Additional Topics

Backup and Restore Strategies:

1. mongodump and mongorestore:

- Use `mongodump` to create a binary export of the database.
- For restoration, employ `mongorestore` to rebuild the database from the dump.

2. File System Snapshots:

- Take snapshots of the underlying file system for point-in-time recovery.
- Ensure MongoDB is in a consistent state during snapshot creation.

3. Continuous Backup Services:

- Leverage third-party backup services or MongoDB Atlas Backup to automate regular backups.
- Enable incremental backups for efficient storage usage.

4. Backup to Remote Location:

- Store backups in a remote location to protect against data center failures.
- Consider encryption for data in transit and at rest

Introduction to MongoDB Atlas (MongoDB's Cloud Database Service):

1. Managed Cloud Database:

- MongoDB Atlas is a fully-managed cloud database service provided by MongoDB, Inc.
- It handles administrative tasks like backups, upgrades, and scaling automatically.

2. Multi-Cloud Availability:

- MongoDB Atlas supports multiple cloud providers, including AWS, Azure, and Google Cloud.
- Enables flexibility in choosing the cloud infrastructure that best suits your needs.

3. Security and Compliance:

- Provides built-in security features such as encryption at rest and in transit.
- Complies with industry standards and certifications.

4. Automated Scaling:

- Allows for automatic horizontal scaling by adjusting the number of nodes in a cluster based on demand.
- Ensures performance optimization without manual intervention.

Introduction to MongoDB Stitch (Backend as a Service):

1. Serverless Backend:

- MongoDB Stitch is a serverless platform that eliminates the need for traditional server management.
- Developers focus on writing application logic without managing infrastructure.

2. Integration with Atlas:

- Seamlessly integrates with MongoDB Atlas for database services.
- Provides a unified platform for backend development and database management.

3. Authentication and Authorization:

- Handles user authentication and authorization with ease.
- Supports various authentication providers, including email/password, Google, and Facebook.

4. HTTP Services and Triggers:

- Allows the creation of HTTP services for serverless functions.
- Supports triggers for automatic execution of functions based on events.

5. Real-Time Updates:

- Enables real-time data synchronization using MongoDB's change streams.
- Applications receive instant updates when data changes in the database.

By adhering to these strategies, practices, and leveraging MongoDB's cloud services, developers can ensure the robustness, scalability, and efficiency of MongoDB deployments, whether in traditional environments or cloud-based solutions.

Practical 3. Basic CRUD Operations with MongoDB

A. Objective

The primary objective of this practical is to familiarize students with the fundamental CRUD (Create, Read, Update, Delete) operations in MongoDB, a NoSQL database. Students will gain hands-on experience in performing these operations using the MongoDB shell and understand the underlying principles of data manipulation in a MongoDB environment.

B. Relevant Program Outcomes (POs)

1. Application of Basic Mathematics, Science, and Engineering Fundamentals:

Apply fundamental mathematical and engineering concepts in implementing CRUD operations with MongoDB.

2. Problem Analysis:

Utilize codified standard methods to identify and analyze well-defined engineering problems related to MongoDB data manipulation.

3. Design/Development of Solutions:

Design effective solutions for well-defined technical problems in MongoDB, emphasizing CRUD operations and database structure.

4. Engineering Tools, Experimentation, and Testing:

Apply modern engineering tools to interact with MongoDB databases and conduct tests and measurements, ensuring data integrity.

5. Engineering Practices for Society, Sustainability, and Environment:

Integrate appropriate technology, considering societal needs, sustainability, and environmental impact when implementing CRUD operations in MongoDB.

6. Project Management:

Use engineering management principles for effective communication and collaborative project management during MongoDB activities.

7. Life-Long Learning:

Demonstrate a commitment to life-long learning by analyzing individual needs and updating skills in response to technological changes in MongoDB and NoSQL databases.

C. Competency and Practical Skills

1. Competency in MongoDB CRUD Operations:

- Demonstrate proficiency in creating, reading, updating, and deleting data in MongoDB, showcasing a solid understanding of basic CRUD operations.

2. Practical Skills in Data Modeling:

- Apply practical skills in designing and implementing MongoDB document structures, ensuring efficiency and adherence to best practices for data modeling.

3. Proficiency in MongoDB Shell:

- Develop practical skills in using the MongoDB shell to navigate databases, execute CRUD operations, and verify the success of data manipulations.

4. Problem-Solving and Error Handling:

- Exhibit competency in identifying and resolving common errors during CRUD operations, showcasing problem-solving skills in the MongoDB environment.

D. Relevant Course Outcomes (Cos)

Apply MongoDB's features and basic CRUD operations to design and manipulate data structures effectively, demonstrating proficiency in utilizing a document-oriented database.

E. Practical Outcome (PRo)

Students will acquire practical proficiency in MongoDB's CRUD operations, mastering document structure design, shell navigation, and error handling. This ensures a foundational understanding for real-world data manipulation scenarios in MongoDB.

F. Expected Affective Domain Outcome (ADos)

Students will cultivate a strong appreciation for proficient database management, showcasing enhanced confidence in executing MongoDB CRUD operations. They will adopt a positive attitude towards addressing real-world data challenges, expressing a sense of accomplishment and motivation. Recognizing the importance of their MongoDB skills, students will be motivated to apply these acquired competencies in future engineering applications, fostering a positive affective response towards database manipulation and an eagerness to implement MongoDB concepts in their professional endeavors.

G. Prerequisite Theory

What is CRUD in MongoDB?

CRUD operations define a user interface's conventions, enabling users to view, search, and modify database components.

To modify MongoDB documents, one connects to a server, queries the relevant documents, adjusts the desired properties, and sends the data back to update the database. CRUD follows data-oriented standards aligned with HTTP action verbs.

Breaking down individual CRUD operations:

- Create Operation: Inserts new documents into the MongoDB database.
- Read Operation: Queries a document in the database.
- Update Operation: Modifies existing documents in the database.
- Delete Operation: Removes documents from the database.

Create or Switch Database

To create or switch a database in MongoDB, use the `use` command:

```
use your_database_name
```

Replace `your_database_name` with the desired name. MongoDB creates the database on-demand when data is inserted.

MongoDB switches to it if it exists; otherwise, it creates the database when data is inserted.

Checking the Current Database:

Verify the current database with:

```
db
```

This displays the current database name.

Dropping a Database

To delete a database, use:

```
db.dropDatabase()
```

Ensure you are in the intended database before executing this command.

- Lazy Creation: MongoDB creates databases when data is inserted, optimizing resource usage.
- Switching Context: The `use` command facilitates seamless transitions between databases.
- Database Naming Rules: Case-sensitive names with allowed characters [a-z], [A-Z], [0-9], period (.), underscore (_), and dollar sign (\$).

Example:

Creating and using a database:

```
use mydatabase
```

Dropping a database:

```
db.dropDatabase()
```

Managing Collections in MongoDB

To create a collection in MongoDB, you can use the `createCollection` method:

```
db.createCollection("your_collection_name")
```

Replace `your_collection_name` with the desired name for your collection.

To drop (delete) a collection, use the `drop` method:

```
db.your_collection_name.drop()
```

Replace `your_collection_name` with the name of the collection you want to drop.

- Collection Naming Rules:

- Follow similar naming rules as databases.
- Case-sensitive names with allowed characters [a-z], [A-Z], [0-9], period (.), underscore (_), and dollar sign (\$).

Example:

Creating a collection:

```
db.createCollection("products")
```

Dropping a collection:

```
db.products.drop()
```

Managing Data in MongoDB

Inserting Documents:

To add documents to a collection in MongoDB, use the `insertOne` or `insertMany` method:

```
db.collection_name.insertOne({ key1: value1, key2: value2, ... })
```

For multiple documents:

```
db.collection_name.insertMany([
  { key1: value1, key2: value2, ... },
  { key1: value1, key2: value2, ... },
  ...
])
```

Reading Documents:

To retrieve documents, use the `find` method:

```
db.collection_name.find()
```

You can add criteria to filter the results:

```
db.collection_name.find({ key: value })
```

Updating Documents:

To update documents, use the `updateOne` or `updateMany` method:

```
db.collection_name.updateOne({ filter_criteria }, { $set: { key: new_value } })
```

For multiple documents:

```
db.collection_name.updateMany({ filter_criteria }, { $set: { key: new_value } })
```

Deleting Documents:

To delete documents, use the `deleteOne` or `deleteMany` method:

```
db.collection_name.deleteOne({ filter_criteria })
```

For multiple documents:

```
db.collection_name.deleteMany({ filter_criteria })
```

- Ensure documents match the desired structure for the collection.
- Be specific when using criteria to avoid unintended updates or deletions.
- Utilize `'\$set` and other update operators for precise modifications.

Examples:

Inserting documents:

```
db.products.insertOne({ name: "Product A", price: 19.99 })
```

Reading documents:

```
db.products.find()
```

Updating documents:

```
db.products.updateOne({ name: "Product A" }, { $set: { price: 24.99 } })
```

Deleting documents:

```
db.products.deleteOne({ name: "Product A" })
```

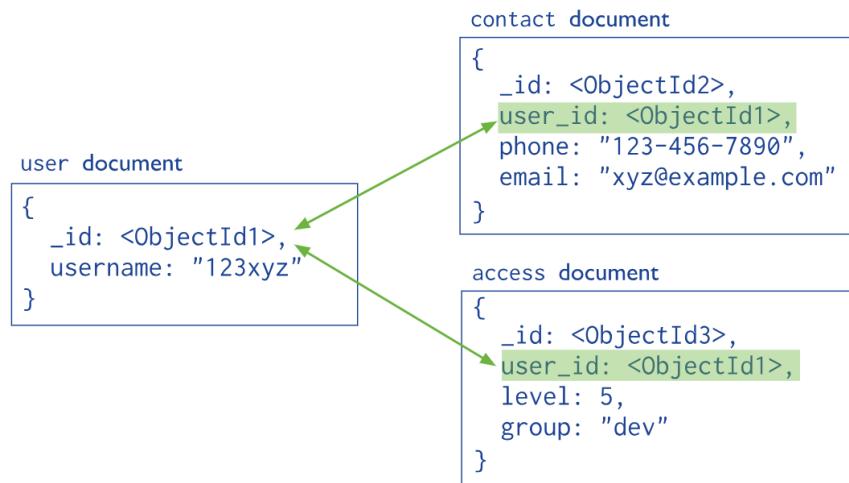
Data modeling in MongoDB

In MongoDB, the way data is organized is very flexible. Unlike in traditional databases, you don't have to decide and declare how the data should look beforehand. Each piece of data (document) can be unique, but usually, they follow a similar pattern. The challenge is finding a good balance between what the application needs, how fast the database works, and how you want to get the data back. When designing how data should look, think about how the data will be used in the application and how the data naturally fits together.

Document Structure in MongoDB:

Designing MongoDB data models involves crucial decisions on document structure and representing data relationships. Two approaches are commonly used: references and embedded documents.

References: Relationships are established through links or references between documents, allowing applications to access related data by resolving these references. This approach follows a normalized data model.



Embedded Data: Relationships are captured by storing related data within a single document structure. MongoDB supports embedding document structures within fields or arrays, enabling retrieval and manipulation of related data in a single database operation. This approach follows denormalized data models.



Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

```
{
  _id: ,
  Emp_ID: "10025AE336"
  Personal_details:{
    First_Name: "Radhika",
    Last_Name: "Sharma",
    Date_Of_Birth: "1995-09-26"
  },
  Contact: {
    e-mail: "radhika_sharma.123@gmail.com",
    phone: "9848022338"
  },
  Address: {
    city: "Hyderabad",
    Area: "Madapur",
    State: "Telangana"
  }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
  _id: <ObjectId101>,
  Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
  _id: <ObjectId102>,
  empDocID: " ObjectId101",
  First_Name: "Radhika",
  Last_Name: "Sharma",
  Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
  _id: <ObjectId103>,
  empDocID: " ObjectId101",
  e-mail: "radhika_sharma.123@gmail.com",
  phone: "9848022338"
}
```

Address:

```
{
  _id: <ObjectId104>,
  empDocID: " ObjectId101",
  city: "Hyderabad",
  Area: "Madapur",
  State: "Telangana"
}
```

Considerations while designing Schema in MongoDB

- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Indexing and Query Optimization

Indexing in MongoDB

Indexing is a fundamental optimization technique in MongoDB that significantly improves the efficiency of data retrieval operations. By creating indexes on specific fields, MongoDB accelerates query performance, enabling faster access to relevant documents within a collection.

Key Concepts:

1. Index Types:

- MongoDB supports various index types, including single-field, compound, multikey, and text indexes.
- Each index type caters to specific query patterns and enhances data access.

2. Index Creation:

- Indexes can be created using the `createIndex` method.
- Syntax: `db.collection_name.createIndex({ field_name: 1 })` for an ascending order index.

3. Default Index on `_id`:

- MongoDB automatically creates an index on the `_id` field for each collection.
- Additional indexes can be strategically created based on the application's query requirements.

Managing Indexes:

1. Listing Existing Indexes:

- Use the `getIndexes` method to view existing indexes on a collection.
- Syntax: `db.collection_name.getIndexes()`.

2. Dropping an Index:

- Unnecessary indexes can be dropped using the `dropIndex` method.
- Syntax: `db.collection_name.dropIndex({ field_name: 1 })` for dropping the index on the "field_name."

Best Practices:

1. Selective Indexing:

- Identify fields frequently used in queries for selective indexing.
- Avoid over-indexing to maintain a balance between read and write performance.

2. Regular Monitoring:

- Monitor index usage regularly to assess their effectiveness.
- Utilize tools like MongoDB Compass or the Database Profiler for comprehensive monitoring.

3. Index Types for Different Scenarios:

- Choose index types based on specific query patterns and application requirements.

Practical Implementation:

Creating an index on the "name" field in the "products" collection

```
db.products.createIndex({ name: 1 })
```

Listing existing indexes in the "products" collection

```
db.products.getIndexes()
```

Dropping an index on the "name" field in the "products" collection

```
db.products.dropIndex({ name: 1 })
```

Query Optimization in MongoDB

Query optimization is a critical aspect of MongoDB database performance tuning, focusing on enhancing the efficiency of data retrieval operations. Optimized queries result in faster response times and improved overall performance.

Key Strategies:

1. Covered Queries:

- Aim for covered queries where the index alone satisfies the query without loading documents from the collection.
- Covered queries minimize the need to fetch documents, improving performance.

2. Query Execution Analysis:

- Utilize the `explain` method to analyze query execution plans and identify areas for optimization.

- Syntax: `db.collection_name.find({ field_name: "value" }).explain("executionStats")`.

3. Index Hinting:

- Use the `hint` method to explicitly specify which index MongoDB should use for a query.
- Syntax: `db.collection_name.find({ field_name: "value" }).hint({ field_name: 1 })`.

Best Practices:

1. Query Specificity:

- Craft queries that are specific and target only the necessary data.
- Minimize the use of wildcard operators to narrow down the dataset.

2. Avoiding Collection Scans:

- Ensure queries utilize indexes to avoid collection scans, which can be resource-intensive.
- Collection scans involve examining every document in a collection, impacting performance.

Practical Implementation:

1. Explaining a Query:

- Use the `explain` method to analyze the execution plan of a query.
- Example: `db.collection_name.find({ field_name: "value" }).explain("executionStats")`.

2. Index Hinting:

- Explicitly specify the index to be used for a query using the `hint` method.
- Example: `db.collection_name.find({ field_name: "value" }).hint({ field_name: 1 })`.

Example:

Explaining the execution plan of a query on the "products" collection

```
db.products.find({ price: { $gte: 20 } }).explain("executionStats")
```

Using index hinting for a query on the "products" collection

```
db.products.find({ name: "Product A" }).hint({ name: 1 })
```

Aggregation Framework

Aggregation in MongoDB is a powerful framework that allows for the transformation and processing of documents within a collection. It enables users to perform advanced data manipulations, combining and transforming data in various ways to derive meaningful insights. Aggregation is especially useful for complex analytics and reporting tasks.

Key Concepts:

1. Pipeline Stages:

- Aggregation operations are structured as a pipeline, where each stage performs a specific transformation on the data.
- Common stages include `'\$match` , `'\$group` , `'\$project` , `'\$sort` , and `'\$unwind` .

2. Expression Operators:

- Aggregation expressions allow the manipulation of data within the pipeline stages.
- Examples include arithmetic operators (`\$add` , `'\$subtract`), array operators (`\$push` , `'\$addToSet`), and conditional operators (`'\$ifNull` , `'\$cond`).

3. Grouping and Summarization:

- The `'\$group` stage is fundamental for grouping documents based on specified criteria.
- Aggregation functions like `'\$sum` , `'\$avg` , `'\$min` , and `'\$max` facilitate summarization within groups.

Practical Implementation:

1. Basic Aggregation:

- Example pipeline for calculating the average price of products:

```
db.products.aggregate([
  { $group: { _id: null, avg_price: { $avg: "$price" } } }
])
```

2. Grouping by a Field:

- Example pipeline for calculating the average price per category:

```
db.products.aggregate([
```

```
{ $group: { _id: "$category", avg_price: { $avg: "$price" } } }
])
```

3. Text Search and Matching:

- Example pipeline for performing a text search on product names:

```
db.products.aggregate([
  { $match: { $text: { $search: "keyword" } } }
])
```

Best Practices:

1. Optimizing Pipeline:

- Design the pipeline efficiently to minimize unnecessary stages.
- Leverage indexes for better performance in aggregation.

2. Understanding the Data:

- Familiarize yourself with the data structure to craft effective aggregation pipelines.
- Utilize expressions to transform and manipulate data as needed.

Example:

Aggregation pipeline to find the total sales for each product category

```
db.sales.aggregate([
  { $group: { _id: "$product.category", total_sales: { $sum: "$quantity" } } },
  { $sort: { total_sales: -1 } }
])
```

Replica Sets

A replica set in MongoDB is a distributed system that provides high availability and fault tolerance by maintaining multiple copies of data across multiple servers. This ensures data redundancy and allows for automatic failover in case of server failures. Replica sets are a fundamental component for building resilient MongoDB deployments.

Key Concepts:

1. Primary and Secondaries:

- A replica set consists of multiple servers, with one designated as the primary and the others as secondaries.
- The primary handles all write operations, while secondaries replicate data from the primary.

2. Automatic Failover:

- In the event of a primary node failure, replica sets automatically elect a new primary from the available secondaries.

- This ensures continuous operation and minimal downtime.

3. Data Redundancy:

- Each member of the replica set contains a full copy of the data, providing redundancy and data availability.

- Data consistency is maintained through the replication process.

Configuration and Implementation:

1. Setting Up a Replica Set:

- Initialize a replica set using the `rs.initiate()` command on the primary node.
- Add secondary nodes to the replica set using `rs.add()` .

2. Read Preference:

- Clients can specify their read preferences to direct read operations to the primary or secondary nodes.

- This allows for load distribution and optimal performance.

3. Monitoring and Maintenance:

- MongoDB provides tools like `rs.status()` for monitoring the health of a replica set.
- Regularly check the status and perform maintenance tasks such as adding or removing nodes.

Practical Considerations:

1. Quorum and Elections:

- Replica sets require a majority (quorum) of members to be available for elections.
- Ensure an odd number of nodes to avoid split-brain scenarios.

2. Arbiter Nodes:

- Arbiter nodes are lightweight nodes used to break ties in elections.
- They do not store data but participate in the election process.

3. Data Safety:

- Regularly backup data and monitor the replica set to ensure data safety.
- Perform routine maintenance tasks to address issues promptly.

Example:

Initializing a replica set with three nodes

```
rs.initiate(
{
  _id: "myReplicaSet",
  members: [
    { _id: 0, host: "mongo1:27017" },
    { _id: 1, host: "mongo2:27017" },
    { _id: 2, host: "mongo3:27017" }
  ]
})
```

Sharding in MongoDB

Sharding is a horizontal scaling technique in MongoDB designed to distribute data across multiple servers, or shards. This enables MongoDB to handle large datasets and high write and read traffic, ensuring scalability and improved performance. Sharding is a key feature for organizations with growing data demands.

Key Concepts:

1. Shard:

- A shard is an individual server or a replica set that stores a subset of the data.
- Shards collectively manage the entire dataset.

2. Shard Key:

- The shard key is a field chosen to distribute data across shards.
- Proper selection of the shard key is crucial for efficient sharding.

3. Balancer:

- MongoDB's balancer redistributes data across shards to maintain an even distribution.
- Balancing ensures optimal performance and resource utilization.

Configuration and Implementation:

1. Setting Up Sharding:

- Enable sharding on a MongoDB instance using the `shardCollection` command.
- Select a database and specify a shard key for sharding.

2. Choosing a Shard Key:

- The choice of shard key significantly impacts sharding efficiency.

- Consider query patterns, data distribution, and growth when selecting a shard key.

3. Scaling Shards:

- Add additional shards as data and traffic increase.
- MongoDB dynamically redistributes data to maintain an even load.

Practical Considerations:

1. Query Isolation:

- Ensure that queries align with the chosen shard key to take advantage of sharding benefits.
- Queries that span multiple shards may result in performance overhead.

2. Monitoring and Balancing:

- Regularly monitor the health and performance of shards using tools like `sh.status()` and `mongostat`.
- Allow the balancer to operate for optimal data distribution.

3. Backup and Restore:

- Backup strategies should consider sharded environments.
- Restoring data may involve additional steps due to sharding.

Example:

Enabling sharding on a database

```
sh.enableSharding("myDatabase")
```

Sharding a collection using a specific shard key

```
sh.shardCollection("myDatabase.myCollection", { "shardKeyField": 1 })
```

H. Resources/Equipment Required

1. Computers/Laptops:

- Ensure students have access to devices with pre-installed MongoDB for hands-on practice.

2. Internet Connection:

- Provide a stable internet connection for MongoDB installation and accessing online documentation.

3. Learning Materials:

- Distribute guides and tutorials to aid students during the practical session.

4. Sample Datasets:

- Supply datasets for students to apply CRUD operations, enhancing their practical understanding.

5. Technical Support:

- Offer assistance for installation or operational issues, ensuring a smooth learning experience.

I. Practical related Questions

1. Create `Products` collections and perform following queries on it.

```
{
  "_id": ObjectId("5f8a4c261c9d4400001a4af4"),
  "name": "Laptop",
  "price": 1200.99,
  "category": "Electronics",
  "stock": 50
}
```

- Create a database named `OnlineStore` and a collection named `Products` with at least 4 fields.
- Insert at least 10 records into the `Products` collection with various product details.
- Retrieve and display all the documents inserted into the `Products` collection.
- Retrieve and display the documents in the `Products` collection, sorted by price in ascending order.
- Update the price of a product in the `Products` collection based on a given condition.
- Delete a record from the `Products` collection based on a given condition.

2. Create `Books` collections and perform following queries on it.

```
{  
    "_id": ObjectId("5f8a4c261c9d4400001a4af7"),  
    "title": "The Great Gatsby",  
    "author": "F. Scott Fitzgerald",  
    "genre": "Classic",  
    "year": 1925  
}
```

- Create a database named `Library` and a collection named `Books` with at least 4 fields
- Insert at least 10 records into the `Books` collection with details about various books.
- Retrieve and display all the documents inserted into the `Books` collection.
- Update the publication year of a book in the `Books` collection based on a given condition.
- Find and display records in the `Books` collection based on a specific criterion, e.g., genre.
- Retrieve and display the documents in the `Books` collection, sorted by publication year in ascending order.
- Delete a record from the `Books` collection based on a given condition.

Practical 4. Introduction and Setup of Cassandra

A. Objective

To provide students with a comprehensive introduction to Apache Cassandra, emphasizing its schema-agnostic data model, the role of keyspace, and basic concepts of the Cassandra Query Language (CQL). Students will gain hands-on experience by setting up a Cassandra cluster, configuring essential parameters, and executing basic CQL commands.

B. Relevant Program Outcomes (POs)

1. Basic Knowledge Application: Apply fundamental engineering knowledge to set up an Apache Cassandra cluster for effective distributed database management problem-solving.
2. Problem Analysis Skills: Identify and analyze engineering problems in Cassandra cluster setup, employing standard methods for troubleshooting and resolution.
3. Design Solutions Capability: Design effective solutions for technical challenges in Apache Cassandra configuration, ensuring optimal distributed database performance.
4. Engineering Tools Proficiency: Apply modern engineering tools and techniques to conduct tests and measurements, validating the successful setup of the Cassandra cluster.
5. Ethical Engineering Practices: Apply appropriate technology considering societal needs, sustainability, and ethics in the context of Apache Cassandra setup.
6. Project Management Skills: Utilize engineering management principles for collaborative and effective communication, leading or contributing to the successful Cassandra cluster setup.
7. Life-long Learning Mindset: Analyze individual learning needs and adapt to technological changes, fostering continuous development in the field of distributed database management with Apache Cassandra.

C. Competency and Practical Skills

1. Technical Proficiency: Students will demonstrate proficiency in setting up an Apache Cassandra cluster, configuring parameters, and launching the distributed database system.
2. Problem Solving Skills: Through troubleshooting common issues, students enhance their problem-solving abilities in distributed database management with Apache Cassandra.
3. System Design and Implementation: Competency is developed in designing and implementing solutions for technical challenges, ensuring efficient performance of Apache Cassandra.
4. Tool Proficiency and Validation: Application of modern tools for tests and measurements validates the successful setup of the Cassandra cluster, honing practical skills in tool utilization.

D. Relevant Course Outcomes (Cos)

Demonstrate Cassandra's data model and query language (CQL), showcasing the ability to create and manage distributed data tables efficiently.

E. Practical Outcome (PRo)

Students will showcase their competence by successfully configuring an Apache Cassandra cluster, translating theoretical knowledge into hands-on skills for distributed database management. Additionally, a detailed troubleshooting report will highlight their proficiency in identifying and resolving common issues encountered during the Cassandra cluster setup, demonstrating practical problem-solving abilities.

F. Expected Affective Domain Outcome (ADos)

Students will gain a practical understanding of distributed database management by setting up an Apache Cassandra cluster, boosting confidence and self-efficacy. Troubleshooting common issues will enhance resilience and problem-solving skills, fostering emotional engagement. Exposure to system design will promote responsibility and ethical awareness. These practical aims to cultivate a positive shift in attitudes, fostering enthusiasm and continued interest in the dynamic field of distributed databases.

G. Prerequisite Theory

Overview of Cassandra: A Distributed NoSQL Database

Cassandra is a highly scalable, distributed NoSQL database designed for handling large amounts of data across multiple commodity servers without any single point of failure. Developed by Apache Software Foundation, it offers high availability, fault tolerance, and seamless horizontal scaling, making it a popular choice for applications with demanding data requirements.

History of Cassandra

- Cassandra was developed at Facebook for inbox search.
- It was open-sourced by Facebook in July 2008.
- Cassandra was accepted into Apache Incubator in March 2009.
- It was made an Apache top-level project since February 2010.

Apache Cassandra is used by smaller organizations while Datastax enterprise is used by the larger organization for storing huge amount of data. Apache Cassandra is managed by Apache.

Key Features:

1. **Distributed Architecture:** Cassandra's peer-to-peer architecture enables data distribution across nodes, ensuring even load distribution and fault tolerance.

2. No Single Point of Failure: With a decentralized design, Cassandra eliminates the risk of a single point of failure, enhancing reliability and system resilience.
3. High Availability: Data is replicated across multiple nodes, allowing for uninterrupted access even in the event of node failures or network issues.
4. Scalability: Cassandra's linear scalability allows for the addition of nodes to accommodate growing data needs, making it suitable for applications experiencing rapid expansion.
5. Flexible Data Model: Cassandra supports a schema-agnostic model, allowing developers to store and retrieve data in a flexible, dynamic manner, accommodating changing application requirements.
6. Query Language (CQL): Cassandra Query Language (CQL) is similar to SQL, making it easier for developers familiar with relational databases to transition to Cassandra.
7. Tunable Consistency: Users can configure the level of data consistency based on their application's requirements, striking a balance between performance and data integrity.

Use Cases:

1. Big Data Analytics: Cassandra excels in handling vast amounts of data, making it a preferred choice for big data analytics applications.
2. IoT (Internet of Things): Its ability to scale horizontally makes Cassandra well-suited for managing large volumes of data generated by IoT devices.
3. Real-time Applications: Cassandra's low-latency architecture is ideal for real-time applications, such as financial systems and online gaming platforms.
4. Time-Series Data: The database's efficient handling of time-series data makes it valuable for applications like monitoring systems and logging.

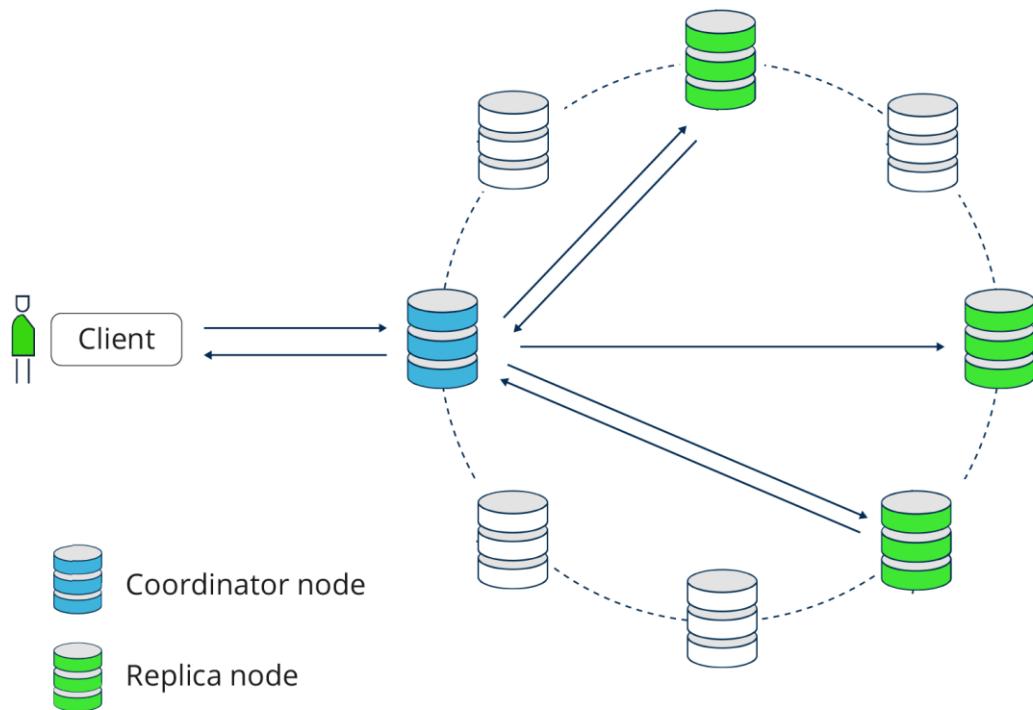
Challenges:

1. Learning Curve: The decentralized nature of Cassandra may pose a learning curve for those accustomed to traditional relational databases.
2. Consistency Trade-offs: Achieving high availability often involves trade-offs in terms of data consistency, requiring careful configuration based on application needs.

Cassandra offers a robust solution for organizations seeking a highly scalable, fault-tolerant, and flexible NoSQL database, particularly well-suited for applications with demanding data requirements.

Cassandra's architecture is tailored for handling extensive big data workloads across a cluster of interconnected nodes, eliminating a single point of failure. Operating on a peer-to-peer distributed system, each independent node can seamlessly handle both read and write requests, irrespective of the data's physical location in the cluster. In the event of a node failure, requests can be redirected to other nodes, ensuring continuous service availability.

Data replication in Cassandra involves nodes acting as replicas, safeguarding data against failures. In case of discrepancies, Cassandra prioritizes returning the most recent value to the client and initiates background read repairs to update outdated values.



Key components include nodes (data storage units), data centers (collections of related nodes), and clusters (containing one or more data centers). The commit log serves as a crash-recovery mechanism, capturing every write operation. Data progresses to the mem-table, a memory-resident structure, before being flushed to SSTables—disk files—upon reaching a threshold. Bloom filters, quick algorithms for set membership testing, function as a special cache accessed after each query.

This architecture ensures robustness, fault tolerance, and scalability in managing distributed data across a Cassandra cluster.

Cassandra Query Language

Users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database (Keyspace) as a container of tables. Programmers use cqlsh: a prompt to work with CQL or separate application language drivers.

Client's approach any of the nodes for their read-write operations. That node (coordinator) plays a proxy between the client and the nodes holding the data.

Write Operations

Every write activity of nodes is captured by the commit logs written in the nodes. Later the data will be captured and stored in the mem-table. Whenever the mem-table is full, data will be written into the SSTable data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

Read Operations

During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.

Cassandra Data Model

Data model in Cassandra is totally different from normally we see in RDBMS. Let's see how Cassandra stores its data.

Cluster

Cassandra database is distributed over several machines that are operated together. The outermost container is known as the Cluster which contains different nodes. Every node contains a replica, and in case of a failure, the replica takes charge. Cassandra arranges the nodes in a cluster, in a ring format, and assigns data to them.

Keyspace

Keyspace is the outermost container for data in Cassandra. Following are the basic attributes of Keyspace in Cassandra:

Replication factor: It specifies the number of machine in the cluster that will receive copies of the same data.

Replica placement Strategy: It is a strategy which species how to place replicas in the ring. There are three types of strategies such as:

- 1) Simple strategy (rack-aware strategy)
- 2) old network topology strategy (rack-aware strategy)
- 3) network topology strategy (datacenter-shared strategy)

Column families: column families are placed under keyspace. A keyspace is a container for a list of one or more column families while a column family is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

In Cassandra, a well data model is very important because a bad data model can degrade performance, especially when you try to implement the RDBMS concepts on Cassandra.

Cassandra data Models Rules

- > Cassandra doesn't support JOINS, GROUP BY, OR clause, aggregation etc. So you have to store data in a way that it should be retrieved whenever you want.
- > Cassandra is optimized for high write performances so you should maximize your writes for better read performance and data availability. There is a tradeoff between data write and data read. So, optimize you data read performance by maximizing the number of data writes.
- > Maximize data duplication because Cassandra is a distributed database and data duplication provides instant availability without a single point of failure.

Installing and Configuring Cassandra

Visit the official Apache Cassandra website and download the latest stable release.

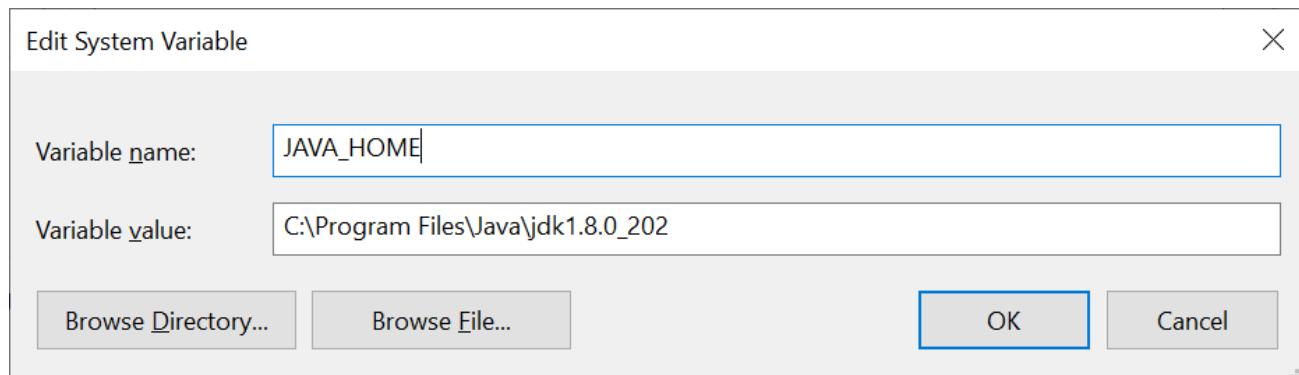
https://cassandra.apache.org/_/download.html

Follow the installation instructions provided for your operating system (Windows, Linux, or macOS).

- Ensure that Java Development Kit (JDK) is installed on your system, as Cassandra relies on Java.

Configure Java Environment for Cassandra:

- Set the `JAVA_HOME` environment variable to the installation path of your JDK.

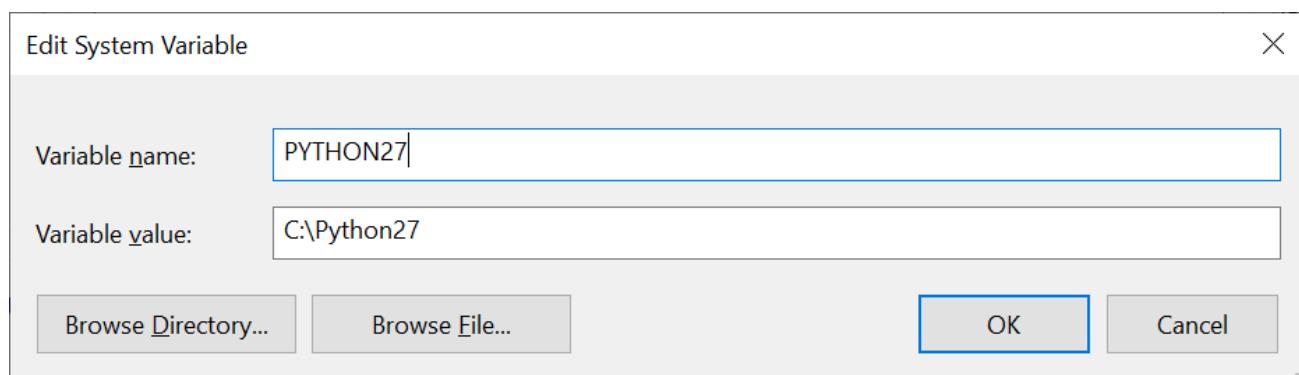


- Add the `bin` directory of the JDK to your system's `PATH` variable.
- Verify your Java installation by running `java -version` in the command line.

Install the latest version of Java 8 the Oracle Java Standard Edition 8 (OpenJDK 8). To verify that you have the correct version of java installed, type `java -version`.

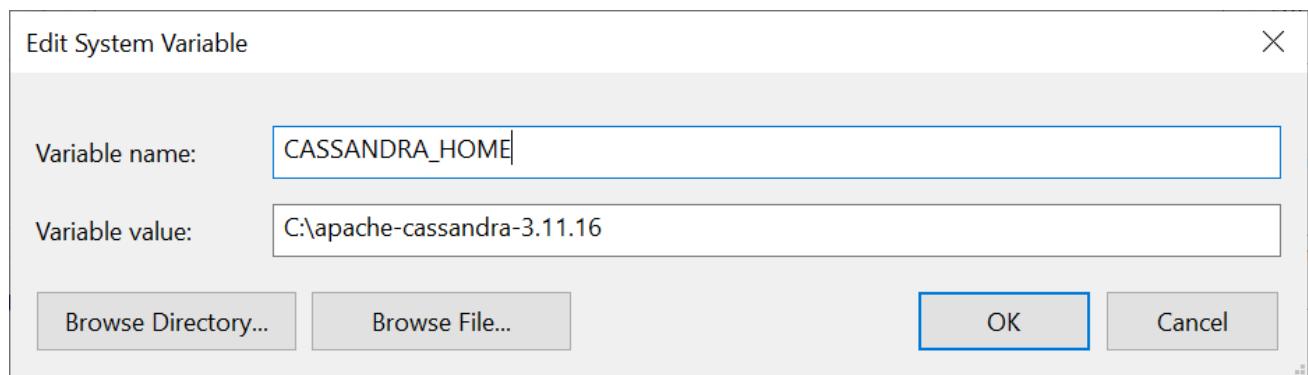
Configure Python for CQL

Users interact with the Cassandra database by utilizing the `cqlsh` bash shell. You need to install Python 2.7 for `cqlsh` to handle user requests properly.



For using `cqlsh`, the latest version of Python 2.7 (support deprecated). To verify that you have the correct version of Python installed, type `python --version`

Download the Apache Cassandra installation file, extract it to the folder named 'apache-cassandra-3.11.16', and relocate this folder to a directory of your preference, for example, 'C:\apache-cassandra-3.11.16'.



Enable unrestricted PowerShell execution Policy

In order to run Cassandra with fully featured functionalities, we need to unrestrict Powershell execution policy. It will look like below on running Cassandra.

- In Windows search bar, type powershell, then select 'Windows Powershell' and run as administrator. You will come to the location 'C:\WINDOWS\system32'.
- Type command : Set-ExecutionPolicy Unrestricted

```

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\WINDOWS\system32> Get-ExecutionPolicy -List

Scope ExecutionPolicy
-----
MachinePolicy      Undefined
UserPolicy         Undefined
Process           Undefined
CurrentUser        Undefined
LocalMachine       AllSigned

PS C:\WINDOWS\system32> Set-ExecutionPolicy Unrestricted

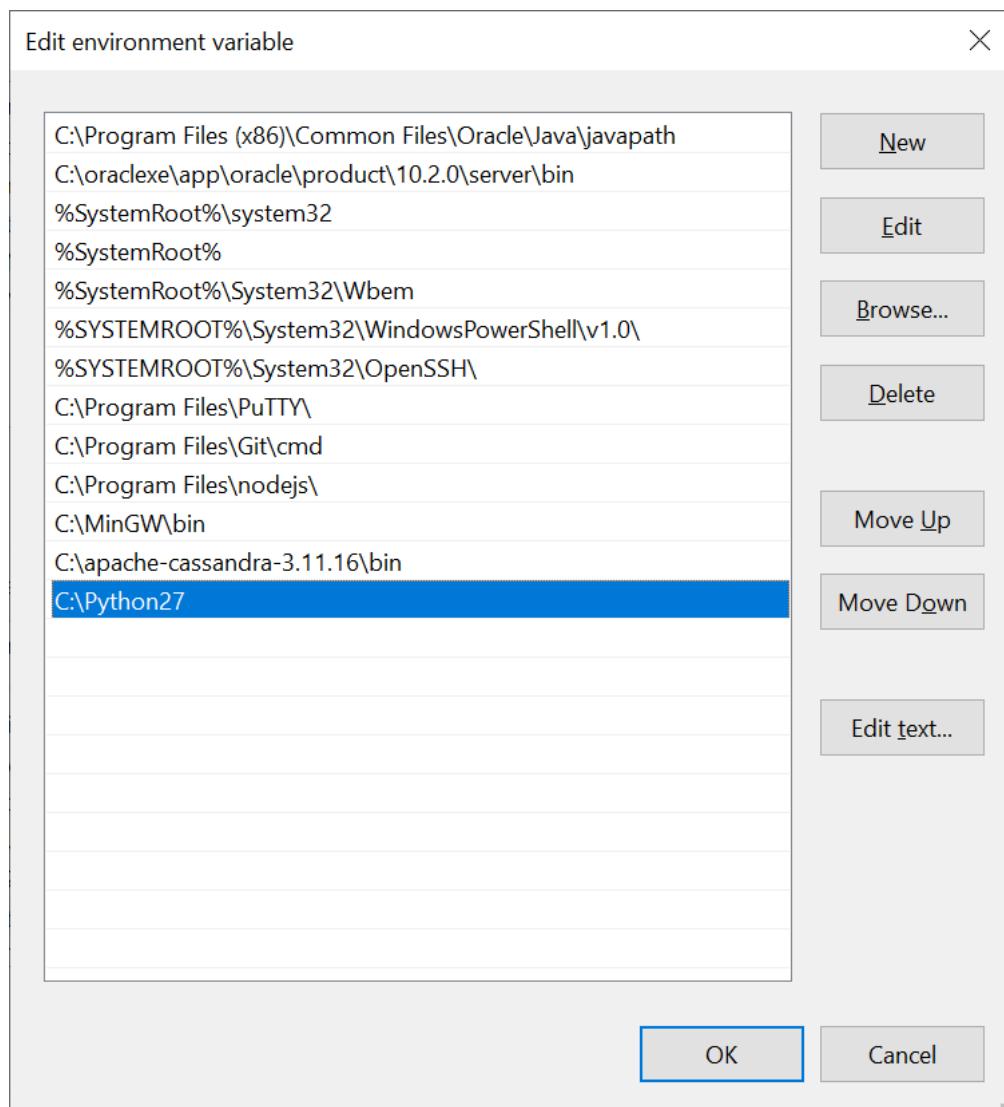
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy may expose you to the security risks described in the about_Execution_Policies help topic at https://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): Y
PS C:\WINDOWS\system32> Get-ExecutionPolicy -List

Scope ExecutionPolicy
-----
MachinePolicy      Undefined
UserPolicy         Undefined
Process           Undefined
CurrentUser        Undefined
LocalMachine       Unrestricted

```

Verifying Cassandra Installation:

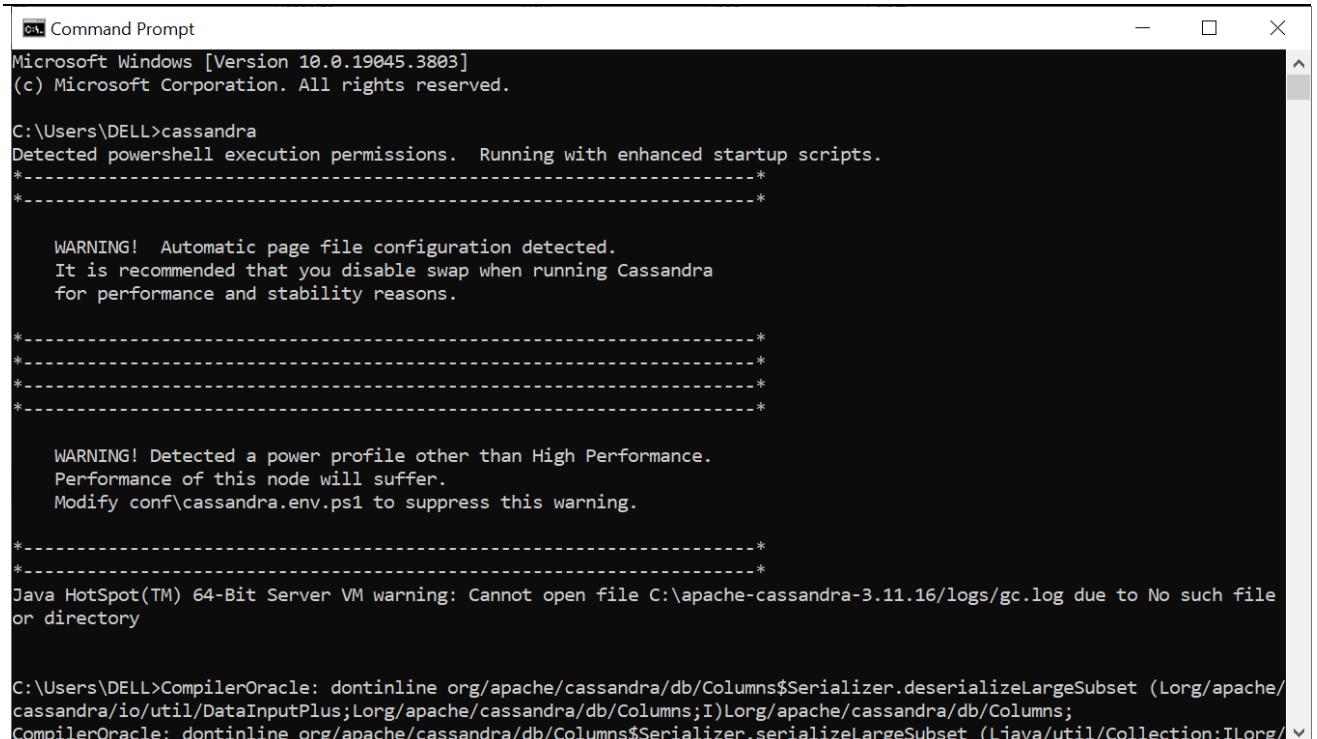
- Go to the Environment Variables section, select 'Path' under System variables, and click 'Edit.'
- Add a new entry, ' C:\apache-cassandra-3.11.16\bin'.



Start Cassandra from Windows CMD

Type the following command to start the Cassandra server:

```
cassandra
```



```
c:\ Command Prompt
Microsoft Windows [Version 10.0.19045.3803]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>cassandra
Detected powershell execution permissions. Running with enhanced startup scripts.
*-----*
*-----*

WARNING! Automatic page file configuration detected.
It is recommended that you disable swap when running Cassandra
for performance and stability reasons.

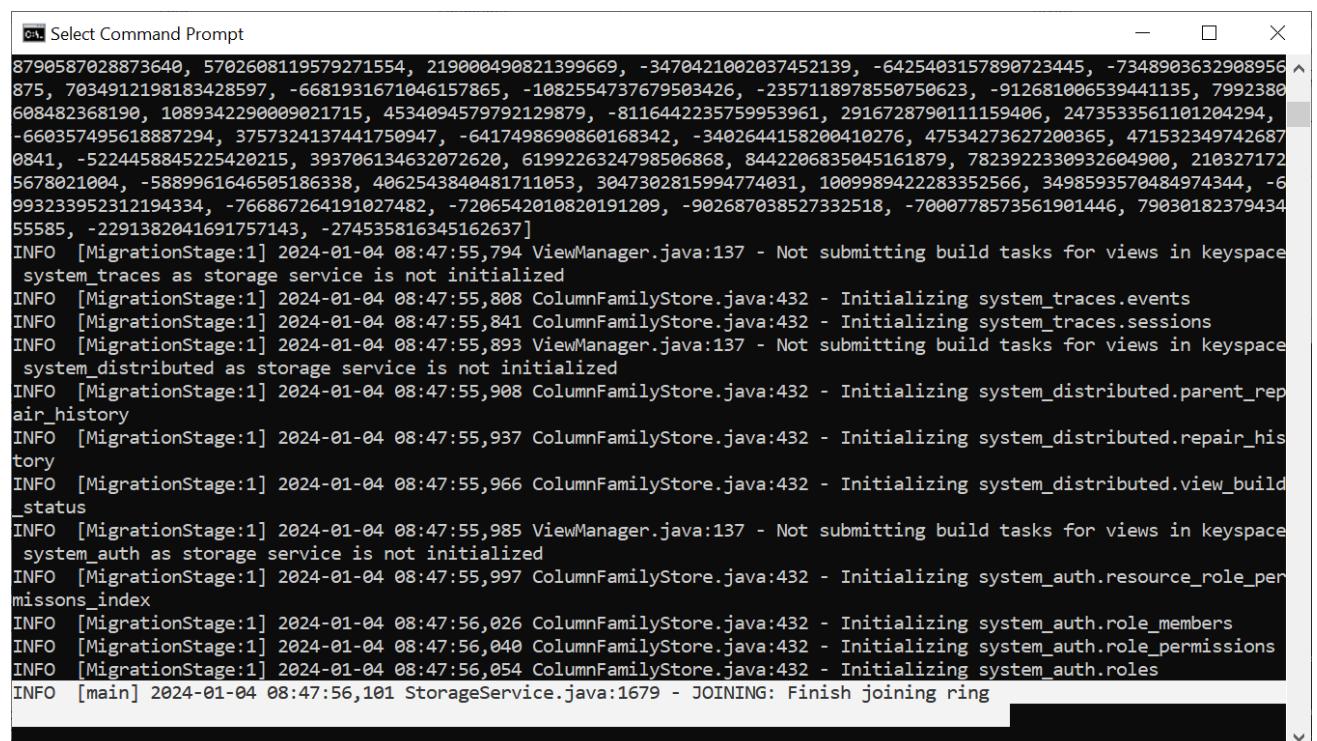
*-----*
*-----*
*-----*
*-----*

WARNING! Detected a power profile other than High Performance.
Performance of this node will suffer.
Modify conf\cassandra.env.ps1 to suppress this warning.

*-----*
*-----*
Java HotSpot(TM) 64-Bit Server VM warning: Cannot open file C:\apache-cassandra-3.11.16/logs/gc.log due to No such file
or directory

C:\Users\DELL>CompilerOracle: dontinline org/apache/cassandra/db/Columns$Serializer.deserializeLargeSubset (Lorg/apache/
cassandra/io/util/DataInputPlus;Lorg/apache/cassandra/db/Columns;I)Lorg/apache/cassandra/db/Columns;
CompilerOracle: dontinline org/apache/cassandra/db/Columns$Serializer.serializeLargeSubset (Ljava/util/Collection;ILorg/
```

The system proceeds to start the Cassandra Server.



```
c:\ Select Command Prompt
8790587028873640, 5702608119579271554, 219000490821399669, -3470421002037452139, -6425403157890723445, -7348903632908956 ^
875, 7034912198183428597, -6681931671046157865, -1082554737679503426, -2357118978550750623, -912681006539441135, 7992380
608482368190, 1089342290009021715, 4534094579792129879, -8116442235759953961, 2916728790111159406, 2473533561101204294,
-6680357495618887294, 3757324137441750947, -6417498690860168342, -3402644158200410276, 47534273627200365, 471532349742687
0841, -5224458845225420215, 393706134632072620, 6199226324798506868, 8442206835045161879, 7823922330932604900, 210327172
5678021004, -5889961646505186338, 4062543840481711053, 3047302815994774031, 1009989422283352566, 3498593570484974344, -6
993233952312194334, -766867264191027482, -7206542010820191209, -902687038527332518, -7000778573561901446, 79030182379434
55585, -2291382041691757143, -274535816345162637]
INFO [MigrationStage:1] 2024-01-04 08:47:55,794 ViewManager.java:137 - Not submitting build tasks for views in keyspace
system_traces as storage service is not initialized
INFO [MigrationStage:1] 2024-01-04 08:47:55,808 ColumnFamilyStore.java:432 - Initializing system_traces.events
INFO [MigrationStage:1] 2024-01-04 08:47:55,841 ColumnFamilyStore.java:432 - Initializing system_traces.sessions
INFO [MigrationStage:1] 2024-01-04 08:47:55,893 ViewManager.java:137 - Not submitting build tasks for views in keyspace
system_distributed as storage service is not initialized
INFO [MigrationStage:1] 2024-01-04 08:47:55,908 ColumnFamilyStore.java:432 - Initializing system_distributed.parent_repair_history
INFO [MigrationStage:1] 2024-01-04 08:47:55,937 ColumnFamilyStore.java:432 - Initializing system_distributed.repair_history
INFO [MigrationStage:1] 2024-01-04 08:47:55,966 ColumnFamilyStore.java:432 - Initializing system_distributed.view_build_status
INFO [MigrationStage:1] 2024-01-04 08:47:55,985 ViewManager.java:137 - Not submitting build tasks for views in keyspace
system_auth as storage service is not initialized
INFO [MigrationStage:1] 2024-01-04 08:47:55,997 ColumnFamilyStore.java:432 - Initializing system_auth.resource_role_permissions_index
INFO [MigrationStage:1] 2024-01-04 08:47:56,026 ColumnFamilyStore.java:432 - Initializing system_auth.role_members
INFO [MigrationStage:1] 2024-01-04 08:47:56,040 ColumnFamilyStore.java:432 - Initializing system_auth.role_permissions
INFO [MigrationStage:1] 2024-01-04 08:47:56,054 ColumnFamilyStore.java:432 - Initializing system_auth.roles
INFO [main] 2024-01-04 08:47:56,101 StorageService.java:1679 - JOINING: Finish joining ring
```

Access Cassandra cqlsh from Windows CMD

While the initial command prompt is still running open a new command line prompt from the same bin folder. Enter the following command to access the Cassandra cqlsh bash shell:

```
cqlsh
```

You now have access to the Cassandra shell and can proceed to issue basic database commands to your Cassandra server.

```
c:\ Command Prompt - cqlsh
Microsoft Windows [Version 10.0.19045.3803]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>cqlsh

WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.16 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cqlsh>
```

Type help to get list of cql shell commands.

```
c:\ Command Prompt - cqlsh
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cqlsh> help

Documented shell commands:
=====
CAPTURE    CLS          COPY        DESCRIBE   EXPAND    LOGIN     SERIAL    SOURCE    UNICODE
CLEAR      CONSISTENCY DESC        EXIT       HELP      PAGING    SHOW     TRACING

CQL help topics:
=====
AGGREGATES      CREATE_KEYSPACE    DROP_TRIGGER    TEXT
ALTER_KEYSPACE   CREATE_MATERIALIZED_VIEW DROP_TYPE    TIME
ALTER_MATERIALIZED_VIEW CREATE_ROLE    DROP_USER    TIMESTAMP
ALTER_TABLE      CREATE_TABLE     FUNCTIONS    TRUNCATE
ALTER_TYPE       CREATE_TRIGGER   GRANT      TYPES
ALTER_USER       CREATE_TYPE     INSERT      UPDATE
APPLY           CREATE_USER    INSERT_JSON USE
ASCII            DATE          INT        UUID
BATCH           DELETE        JSON
BEGIN           DROP_AGGREGATE KEYWORDS
BLOB            DROP_COLUMNFAMILY LIST_PERMISSIONS
BOOLEAN         DROP_FUNCTION  LIST_ROLES
COUNTER         DROP_INDEX    LIST_USERS
CREATE_AGGREGATE CREATE_KEYSPACE PERMISSIONS
CREATE_COLUMNFAMILY DROP_MATERIALIZED_VIEW REVOKE
CREATE_FUNCTION  DROP_ROLE     SELECT
CREATE_INDEX    DROP_TABLE    SELECT_JSON

cqlsh> _
```

Create a Keyspace

In the CQL shell, create a keyspace using the CREATE KEYSPACE statement. Replace 'your_keyspace' with the desired keyspace name and set the replication strategy and factor based on your requirements.

```
CREATE KEYSPACE your_keyspace
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```

Use the Keyspace

Switch to the newly created keyspace using the USE statement.

```
USE your_keyspace;
```

```
cqlsh> CREATE KEYSPACE BLOG
...           WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

Warnings :
Your replication factor 3 for keyspace blog is higher than the number of nodes 1

cqlsh> USE BLOG
...
...
cqlsh:blog>
```

Shutting Down Cassandra

- To stop Cassandra, use the following command:

```
bin/nodetool stopdaemon
```

- Ensure that Cassandra is gracefully shut down to prevent data corruption.

By following these steps, you can successfully install, configure, and start working with Apache Cassandra, setting the foundation for building robust and scalable distributed database systems.

H. Resources/Equipment Required

1. Computing Devices: Students will need individual computing devices such as laptops or desktops with sufficient processing power and memory to support the installation and configuration of Apache Cassandra.
2. Internet Connectivity: A stable internet connection is essential for downloading the necessary software packages, updates, and documentation related to Apache Cassandra.
3. Apache Cassandra Software: Students must have access to the latest version of the Apache Cassandra software. This includes the installation files and documentation available for download from the official Apache Cassandra website.
4. Operating System: Ensure that students' devices are running a compatible operating system. Apache Cassandra is compatible with various operating systems, including Linux, Windows, and macOS.
5. Java Development Kit (JDK): Apache Cassandra requires Java to run. Students should have a compatible version of the JDK installed on their devices before setting up the Cassandra cluster.

Practical 5. Data Modeling and Simple Queries with Cassandra

A. Objective

Students will explore data modeling principles in Apache Cassandra, emphasizing the creation of efficient schemas. The goal is to strengthen understanding of Cassandra's query language (CQL) through hands-on experience with simple queries. Additionally, students will perform monitoring, troubleshooting, and learn performance tuning and optimization techniques. Furthermore, they will implement compaction strategies, ensuring a comprehensive skill set for leveraging Cassandra in practical scenarios.

B. Relevant Program Outcomes (POs)

1. Application of Fundamental Knowledge:

- Apply basic mathematics, science, and engineering principles to formulate efficient data models and perform fundamental operations in Cassandra.

2. Problem Analysis:

- Identify and analyze well-defined engineering challenges related to data modeling, simple queries, and basic operations using standardized methods in Cassandra.

3. Solution Design/Development:

- Design effective solutions for engineering problems associated with data modeling, simple queries, and fundamental operations in Cassandra, contributing to the creation of optimal schemas.

4. Engineering Tools and Testing:

- Utilize modern engineering tools and techniques for conducting standard tests and measurements, ensuring the reliability of Cassandra data models and operations.

5. Engineering Practices for Sustainability:

- Apply appropriate technologies within the societal context, emphasizing sustainability, environmental considerations, and ethical practices in Cassandra database design and operations.

6. Project Management:

- Demonstrate proficiency in engineering management principles, both individually and as part of a team, to effectively manage projects and communicate about Cassandra-related engineering activities, including basic operations.

7. Lifelong Learning:

- Exhibit the ability to analyze individual learning needs and engage in continuous updating within the evolving technological landscape of data modeling, simple queries, and basic operations in the field of engineering.

C. Competency and Practical Skills

1. Data Modeling Proficiency:

- Develop competency in crafting effective data models in Cassandra, showcasing the ability to design schemas that align with engineering requirements.

2. Query Execution Skills:

- Acquire practical skills in executing simple queries using Cassandra Query Language (CQL), demonstrating proficiency in retrieving and manipulating data within the Cassandra database.

3. Troubleshooting and Optimization Techniques:

- Gain hands-on experience in identifying and resolving issues related to data modeling and query performance, while implementing optimization strategies for enhanced efficiency.

D. Relevant Course Outcomes (Cos)

Demonstrate Cassandra's data model and query language (CQL), showcasing the ability to create and manage distributed data tables efficiently.

E. Practical Outcome (PRo)

Students will have cultivated expertise in crafting efficient data models within Apache Cassandra and executing simple queries using Cassandra Query Language (CQL). The practical will empower them with hands-on skills to design optimal schemas, retrieve and manipulate data effectively, troubleshoot potential issues, and implement optimization techniques. This comprehensive practical outcome ensures students are well-prepared to apply their knowledge in real-world scenarios involving data modeling and basic querying in Cassandra.

F. Expected Affective Domain Outcome (ADos)

1. Enhanced Confidence and Engagement:

- Students will develop increased confidence in navigating and applying data modeling concepts in Cassandra, fostering a heightened sense of engagement and enthusiasm for working with distributed databases.

2. Improved Problem-Solving Mindset:

- The practical aims to instill a problem-solving mindset in students, encouraging them to approach challenges associated with data modeling and simple queries in Cassandra with resilience and innovative thinking. This outcome contributes to a positive shift in attitudes towards overcoming real-world engineering complexities.

G. Prerequisite Theory

Basic operations and maintenance

In Cassandra, the management of keyspaces involves creating, altering, and dropping. Here are examples of how you can perform these actions using the Cassandra Query Language (CQL):

Create Keyspace:

To create a keyspace, use the `CREATE KEYSPACE` statement. Replace 'your_keyspace' with the desired keyspace name and set the replication strategy and factor based on your requirements.

```
CREATE KEYSPACE your_keyspace
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```

Alter Keyspace:

To alter an existing keyspace, use the `ALTER KEYSPACE` statement. This can include modifying the replication strategy, replication factor, or other configuration options.

```
ALTER KEYSPACE your_keyspace
    WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': 3, 'DC2': 2};
```

Drop Keyspace:

To drop (delete) a keyspace and all its data, use the `DROP KEYSPACE` statement. Be cautious, as this operation is irreversible.

```
DROP KEYSPACE your_keyspace;
```

These statements provide the basic syntax for creating, altering, and dropping keyspaces in Cassandra. Customize the keyspace names, replication strategies, and other parameters based on your specific use case and requirements.

In Cassandra, you can manage tables using the Cassandra Query Language (CQL). Below are examples of creating, altering, and dropping tables:

Create Table:

To create a table, use the `CREATE TABLE` statement. Replace 'your_table' with the desired table name and specify the columns along with their data types.

```
CREATE TABLE your_table (
    id UUID PRIMARY KEY,
    name TEXT,
    age INT,
    email TEXT
);
```

Alter Table:

To alter an existing table, you can use the `ALTER TABLE` statement. This allows you to add or drop columns, change the data type of a column, or modify other table properties.

```
-- Add a new column
ALTER TABLE your_table ADD phone_number TEXT;
-- Drop a column
ALTER TABLE your_table DROP age;
-- Modify column data type
ALTER TABLE your_table ALTER email TYPE VARCHAR;
```

Drop Table:

To drop (delete) a table, use the `DROP TABLE` statement. Be cautious, as this operation permanently removes the table and its data.

```
DROP TABLE your_table;
```

These statements provide the basic syntax for creating, altering, and dropping tables in Cassandra. Customize the table names, column definitions, and other parameters based on your specific use case and requirements.

In Cassandra, the `TRUNCATE` statement is used to remove all data from a table while keeping the table structure intact. This operation is similar to deleting all rows from the table but more efficient, as it does not involve the same overhead.

Truncate Table:

To truncate a table, use the following `TRUNCATE` statement:

```
TRUNCATE your_table;
```

Replace 'your_table' with the name of the table you want to truncate. This statement will remove all rows from the specified table, but the table structure, including column definitions and indexes, will remain unchanged.

It's important to note that truncating a table is a non-reversible operation, and it permanently removes all data from the table. Ensure you have a backup or are certain about this action before executing it, especially in a production environment.

In Cassandra, you can create and drop indexes using the Cassandra Query Language (CQL). Below are examples of creating and dropping an index on a table:

Create Index:

To create an index on a column, use the `CREATE INDEX` statement. This allows you to create an index to improve the performance of queries based on that column.

```
CREATE INDEX your_index_name ON your_table (your_column);
```

Replace 'your_index_name' with the desired name for the index, 'your_table' with the table name, and 'your_column' with the column on which you want to create the index.

Drop Index:

To drop (delete) an index, use the `DROP INDEX` statement. This removes the specified index from the table.

```
DROP INDEX your_index_name;
```

Replace 'your_index_name' with the name of the index you want to drop. This statement removes the index from the table but does not affect the table structure or data.

Ensure that you carefully consider the implications of adding and removing indexes, as it can impact the performance and storage requirements of your Cassandra database.

In Cassandra, the `BATCH` statement is used to group multiple CQL statements (queries, updates, or deletes) into a single atomic operation. This ensures that either all the statements in the batch are executed successfully or none of them are. Batches are useful when you need to maintain consistency across multiple write operations.

Basic Batch Syntax:

```
BEGIN BATCH
    // CQL statements to be executed
    APPLY BATCH;
```

Here is a more detailed example:

```
BEGIN BATCH
    INSERT INTO your_table (id, name, age) VALUES (uuid(), 'John', 25);
    UPDATE your_table SET email = 'john@example.com' WHERE id = <some_id>;
    DELETE FROM another_table WHERE id = <another_id>;
    APPLY BATCH;
```

- `BEGIN BATCH`: Indicates the beginning of the batch.

- `APPLY BATCH`: Indicates the end of the batch.

CRUD Operations

In Cassandra, you can perform basic data manipulation operations such as insert, select, update, and delete using the Cassandra Query Language (CQL). Here are examples of each operation:

Insert:

To insert data into a table, use the `INSERT INTO` statement:

```
INSERT INTO your_table (id, name, age, email) VALUES (uuid(), 'John Doe', 25, 'john.doe@example.com');
```

Replace 'your_table' with the name of your table and adjust the values accordingly.

Select:

To retrieve data from a table, use the `SELECT` statement:

```
SELECT * FROM your_table WHERE id = <some_id>;
```

Replace 'your_table' with the table name and `<some_id>` with the specific identifier.

Update:

To update existing data in a table, use the `UPDATE` statement:

```
UPDATE your_table SET email = 'new.email@example.com' WHERE id = <some_id>;
```

Replace 'your_table' with the table name, `<some_id>` with the specific identifier, and adjust the values accordingly.

Delete:

To delete data from a table, use the `DELETE` statement:

```
DELETE FROM your_table WHERE id = <some_id>;
```

Replace 'your_table' with the table name and `<some_id>` with the specific identifier.

Remember to replace placeholders such as 'your_table' and '<some_id>' with your actual table name and identifier. Additionally, ensure that your data model and application requirements guide the structure of these queries for optimal performance and scalability.

Cassandra Collections

Cassandra collections are used to handle tasks. You can store multiple elements in collection.

In Cassandra, `SET`, `LIST`, and `MAP` are collection types that allow you to store multiple values within a single column. These collections can be useful when you need to handle scenarios where a column contains multiple items. Here's a brief overview of each:

SET:

A `SET` is an unordered collection of unique elements. Each element in the set must be of the same data type. Duplicate values are not allowed.

Example:

```
CREATE TABLE example_set (
    id UUID PRIMARY KEY,
    tags SET<TEXT>
```

```
 );
INSERT INTO example_set (id, tags) VALUES (uuid(), {'tag1', 'tag2', 'tag3'});
```

LIST:

A `LIST` is an ordered collection of elements where duplicates are allowed. Elements in the list can be of different data types.

Example:

```
CREATE TABLE example_list (
    id UUID PRIMARY KEY,
    comments LIST<TEXT>
);
INSERT INTO example_list (id, comments) VALUES (uuid(), ['Comment 1', 'Comment 2',
    'Comment 3']);
```

MAP:

A `MAP` is a collection of key-value pairs where each key is associated with a specific value. Keys and values can have different data types.

Example:

```
CREATE TABLE example_map (
    id UUID PRIMARY KEY,
    properties MAP<TEXT, TEXT>
);
INSERT INTO example_map (id, properties) VALUES (uuid(), {'key1': 'value1', 'key2':
    'value2'});
```

In these examples, `SET`, `LIST`, and `MAP` are used to store collections of tags, comments, and properties, respectively. It's important to note that while these collection types offer flexibility, their usage should align with your specific data modeling needs and query patterns. Additionally, consider the impact on performance and scalability when working with large collections.

Monitoring and troubleshooting

Monitoring a Cassandra cluster involves using tools and techniques to assess its health, performance, and other relevant metrics.

`nodetool` is a command-line utility in Apache Cassandra that provides various operations and management tasks for interacting with and monitoring Cassandra nodes. It is a powerful tool for system administrators and developers to perform actions on a Cassandra cluster. Here are some commonly used `nodetool` commands:

1. Status and Information:

- Node Status:

```
nodetool status
```

Displays the status of each node in the cluster, including their state (UN - Up, DN - Down), load, and tokens.

- Cluster Information:

```
nodetool info
```

Provides information about the cluster, including the Cassandra version, data center, and Rack.

2. Performance and Metrics:

- Compaction Stats:

```
nodetool compactionstats
```

Displays information about ongoing compactions.

- Thread Pool Stats:

```
nodetool tpstats
```

Shows statistics for thread pools, helping identify performance bottlenecks.

- Latency Information:

```
nodetool cfstats
```

Displays column family statistics, including read and write latencies.

3. Data Management:

- Table Snapshot:

```
nodetool snapshot <keyspace> <table>
```

Takes a snapshot of a specific table in a keyspace. Useful for backup purposes.

- Compact Tables:

```
nodetool compact <keyspace> <table>
```

Forces compaction on a specific table to reclaim disk space.

- Flush Tables:

```
nodetool flush <keyspace> <table>
```

Flushes data from memtables to disk for a specific table.

4. Ring Management:

- Token Ring:

```
nodetool ring
```

Displays the token ring information, showing the distribution of tokens across the cluster.

- Move Node:

```
nodetool move <new_token>
```

Moves a node to a new token in the ring.

- Decommission Node:

```
nodetool decommission
```

Decommissions a node from the Cassandra cluster.

5. Repair and Maintenance:

- Repair Node:

```
nodetool repair
```

Initiates a repair operation to ensure data consistency.

- Cleanup Node:

```
nodetool cleanup
```

Performs cleanup by removing obsolete data on a node.

- Garbage Collection (GC) Grace Period:

```
nodetool setcompactionthroughput <value>
```

Adjusts the compaction throughput, affecting the rate of garbage collection.

These are just a few examples of the numerous `nodetool` commands available. Running `nodetool` without any arguments provides a list of available commands and their descriptions. Always refer to the official documentation for the specific version of Cassandra you are using for the most accurate and up-to-date information.

Performance tuning and optimization

Performance tuning and optimization in Apache Cassandra involve configuring and adjusting various settings to ensure the cluster operates efficiently and meets the desired performance goals.

1. Memory Settings:

```
# Memory allocation settings for the Java Virtual Machine (JVM)
# -Xms: Initial heap size
# -Xmx: Maximum heap size
# -Xmn: Young generation size
# Adjust values based on your system's RAM and workload
```

heap_size_options:

- "-Xms4G"
- "-Xmx4G"
- "-Xmn800M"

2. File Cache Size:

```
# Size of the file cache to utilize OS page cache
# Adjust based on available system memory
file_cache_size_in_mb: 512
```

3. Disk Access Mode:

```
# Method for Cassandra to access data on disk
# Options: mmap, standard, or mmap_index_only
disk_access_mode: mmap
```

4. Commit Log Settings:

```
# Commit log synchronization settings
# periodic: Periodically flushes the commit log to disk
# commitlog_sync_period_in_ms: Time interval for periodic commit log flush
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
```

5. Concurrent Reads and Writes:

```
# Number of concurrent read and write operations per node
# Adjust based on workload and system capacity
concurrent_reads: 32
concurrent_writes: 32
```

6. Native Transport Settings:

```
# Native transport (CQL) settings
# native_transport_max_threads: Maximum number of threads to process native transport
(CQL) requests
native_transport_max_threads: 2048
```

8. Endpoint Snitch:

```
# Strategy for determining proximity and network topology
# Choose an appropriate snitch for your deployment
endpoint_snitch: GossipingPropertyFileSnitch
```

7. Read and Write Timeouts:

```
# Timeout settings for read and write operations
# Adjust based on application requirements
read_request_timeout_in_ms: 5000
write_request_timeout_in_ms: 2000
```

8. Consistency Levels:

```
# Consistency levels for read and write operations
# Adjust based on application requirements
# Use LOCAL_QUORUM for better performance in multi-data center setups
read_consistency_level: ONE
write_consistency_level: LOCAL_QUORUM
```

9. Additional JVM and GC Tuning:

```
# Additional Java Virtual Machine (JVM) options and garbage collection tuning
# Adjust based on your specific JVM version and garbage collection strategy
# Monitor GC logs to optimize settings
jvm_options:
- "-XX:+UseG1GC"
- "-XX:MaxGCPauseMillis=500"
```

Always refer to the official Cassandra documentation for version 3.11 for detailed information on these settings.

Compaction Strategy

Compaction is the process of merging multiple SSTables (Sorted String Tables) into a smaller number of SSTables, reducing storage space and improving read performance. Cassandra provides different compaction strategies, each with its own advantages and use cases. Here are some common compaction strategies in Cassandra:

1. SizeTieredCompactionStrategy (STCS):

- Description: Segments SSTables based on size and compacts smaller SSTables into larger ones.
- Use Case: Suitable for write-intensive workloads with uniform data distribution.

```
compaction:
enabled: true
default_compaction_strategy: SizeTieredCompactionStrategy
```

2. LeveledCompactionStrategy (LCS):

- Description: Divides SSTables into levels, each with a fixed size. Compacts SSTables within the same level, then promotes them to the next level.
- Use Case: Suitable for read-heavy workloads, provides more predictable and tunable compaction.

```
compaction:
enabled: true
default_compaction_strategy: LeveledCompactionStrategy
```

3. TimeWindowCompactionStrategy (TWCS):

- Description: Groups SSTables based on time intervals, compacts data within each time window.

- Use Case: Suitable for time-series data where older data can be compacted separately from newer data.

```
compaction:
  enabled: true
  default_compaction_strategy: TimeWindowCompactionStrategy
```

4. DateTieredCompactionStrategy (DTCS):

- Description: Similar to TimeWindowCompactionStrategy but uses a more flexible time window definition.

- Use Case: Suitable for time-series data with varying write rates.

```
compaction:
  enabled: true
  default_compaction_strategy: DateTieredCompactionStrategy
```

5. SizeTieredCompactionStrategy with STCSIngestTTL:

- Description: Extension of SizeTieredCompactionStrategy optimized for Time-To-Live (TTL) data.

- Use Case: Suitable for write-intensive workloads with TTL-enabled data.

```
compaction:
  enabled: true
  default_compaction_strategy: SizeTieredCompactionStrategy
  compaction_strategy_options:
    STCSIngestTTL: true
```

Choose the compaction strategy based on your specific use case, workload characteristics, and performance requirements. Always monitor and test different strategies in a controlled environment to determine the most effective one for your Cassandra deployment.

H. Resources/Equipment Required

1. Computing Devices:

- Students should have access to individual computing devices, preferably laptops, with the necessary hardware specifications to run Apache Cassandra and related tools smoothly.

2. Software and Tools:

- Installation of Apache Cassandra: Ensure students have the latest version of Apache Cassandra installed on their devices, configured and ready for use in a controlled environment.
- Integrated Development Environment (IDE): Provide guidance on using an appropriate IDE that supports Cassandra Query Language (CQL) for developing and executing simple queries.

3. Documentation and Tutorials:

- Comprehensive documentation and tutorials covering data modeling principles, basic operations, and simple queries in Cassandra to facilitate learning and hands-on practice.

4. Internet Connectivity:

- Stable internet connectivity is essential for accessing additional learning resources, troubleshooting guides, and potential updates or patches for Cassandra.

I. Practical related Questions

1. Create a keyspace named "ecommerce" with a replication strategy of 'NetworkTopologyStrategy' and replication factor of 3, placing replicas in 'DC1' and 'DC2'.
2. Alter the keyspace "ecommerce" to change the replication strategy to 'SimpleStrategy' with a replication factor of 2.
3. Create a table named "products" with columns: product_id (UUID, primary key), product_name (TEXT), price (DOUBLE), and stock_quantity (INT).
4. Alter the table "products" to add a new column "manufacturer" of type TEXT.
5. Create an index named "idx_product_name" on the "products" table for the "product_name" column.
6. Insert a new product into the "products" table with the following values: (uuid(), 'Laptop', 1200.0, 50, 'Dell').
7. Select all products from the "products" table.
8. Update the stock_quantity of the product with product_id 'some_id' to 40.
9. Delete the product with product_id 'some_id' from the "products" table.
10. Truncate the "products" table.
11. Drop the index "idx_product_name."
12. Drop the table "products."
13. Drop the keyspace "ecommerce."

Practical 6. Introduction to Neo4j Graph Databases

A. Objective

To introduce students to Neo4j Graph Databases by covering the basics, emphasizing the property graph model and graph theory fundamentals. It explores use cases in areas like social networks and fraud detection while guiding students through the installation process of Neo4j for hands-on experience.

B. Relevant Program Outcomes (POs)

1. Basic and Discipline-Specific Knowledge: Apply fundamental knowledge of mathematics, science, and engineering to address engineering challenges, with a focus on utilizing Neo4j Graph Databases.
2. Problem Analysis: Identify and analyze specific engineering problems, employing standardized methods for comprehensive problem understanding within the context of Neo4j Graph Databases.
3. Design/Development of Solutions: Formulate design solutions for well-defined technical issues, contributing to the design of systems components or processes aligned with the requirements of Neo4j Graph Databases.
4. Engineering Tools, Experimentation, and Testing: Apply contemporary engineering tools and techniques to conduct standard tests and measurements, particularly in the realm of Neo4j Graph Databases.
5. Engineering Practices for Society, Sustainability, and Environment: Integrate appropriate technology with consideration for societal impact, sustainability, environmental factors, and ethical practices, especially within the Neo4j Graph Databases domain.
6. Project Management: Utilize engineering management principles effectively, whether working individually, as a team member, or in a leadership role, to manage projects and communicate about engineering activities related to Neo4j Graph Databases.
7. Life-Long Learning: Demonstrate the ability to analyze individual learning needs and actively engage in continuous learning, adapting to technological changes specifically within the field of Neo4j Graph Databases.

C. Competency and Practical Skills

1. Graph Database Modeling: Develop proficiency in constructing and understanding graph database models, with a focus on the property graph model used in Neo4j.
2. Installation and Setup: Demonstrate competence in installing and configuring Neo4j, ensuring a clear understanding of system requirements and successful setup, including access to the Neo4j Browser.

3. Problem-Solving with Graph Theory: Apply graph theory fundamentals to solve practical engineering problems, showcasing the practical application of graph databases in scenarios like social networks, fraud detection, and recommendation engines.

D. Relevant Course Outcomes (Cos)

Identify the significance of graph databases, illustrating their practical applications in solving complex relationship-oriented problems.

E. Practical Outcome (PRo)

Students will have a comprehensive understanding of Neo4j Graph Databases, focusing on key aspects such as the basics of graph databases, graph theory fundamentals, use cases for graph databases, and the installation process of Neo4j. This practical outcome ensures that students are well-versed in fundamental graph concepts, acquainted with real-world applications, and proficient in setting up Neo4j for practical implementation.

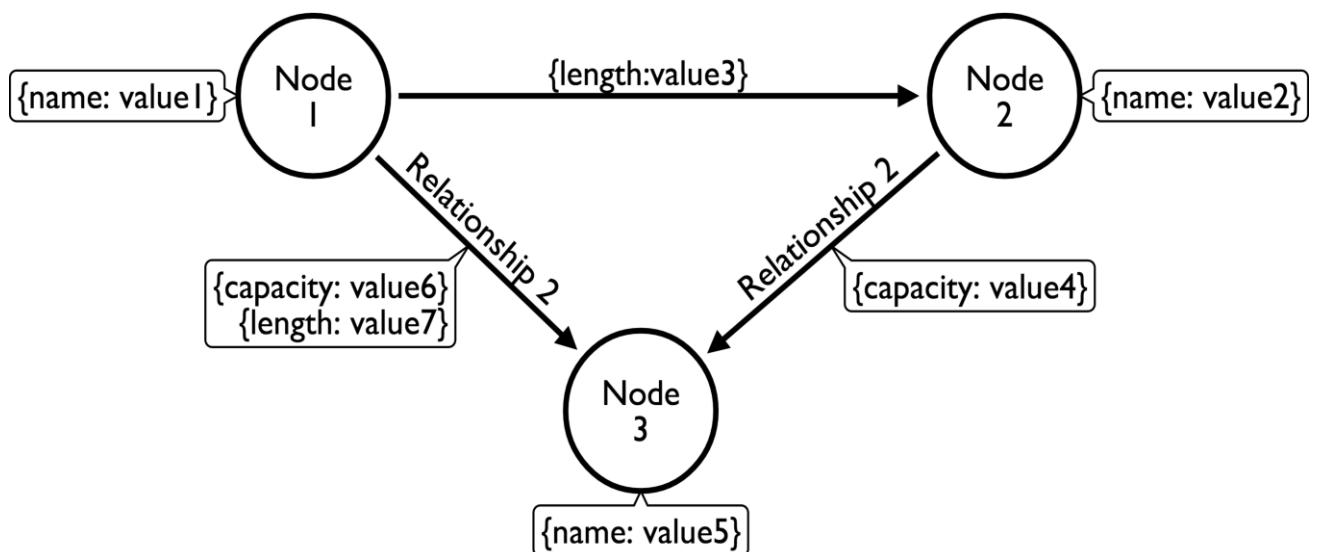
F. Expected Affective Domain Outcome (ADos)

1. Appreciation for Graph Database Concepts: Develop a heightened appreciation for the conceptual foundations of graph databases, recognizing their significance in representing and querying complex relationships.
2. Analytical Thinking: Cultivate analytical thinking skills by understanding and critically evaluating graph theory fundamentals, enabling a deeper comprehension of nodes, relationships, and edges within the database context.
3. Application Awareness: Gain awareness of practical applications of graph databases, fostering the ability to identify scenarios where graph structures excel, such as in social networks, fraud detection, and recommendation engines.
4. Technical Proficiency: Acquire technical proficiency in installing Neo4j, demonstrating competency in configuring the system environment and ensuring successful setup for subsequent practical exercises.

G. Prerequisite Theory

Basics of graph databases

Graph databases are a type of NoSQL database that uses graph structures with nodes, edges, and properties to represent and store data. They are particularly well-suited for scenarios where relationships between entities are important and need to be efficiently queried.



- Node: Represents an entity in the graph.
- Edge: Represents a relationship between two nodes.
- Property: Key-value pairs associated with nodes and edges, providing additional information.
 - Graph databases often use the Cypher query language for data manipulation and retrieval. Cypher is designed to be expressive and intuitive for querying graph data.
 - Graph databases excel at traversing relationships between nodes. You can easily follow edges to navigate through the graph and discover connections between nodes.
 - Graph databases are designed to efficiently handle queries involving relationships, making them well-suited for applications that involve complex and interconnected data.

Examples of Graph Databases:

- Neo4j: A popular open-source graph database.
- Amazon Neptune: A managed graph database service by AWS.
- Microsoft Azure Cosmos DB: Supports graph data models along with other NoSQL models.

Use Cases:

- Graph databases are suitable for scenarios where relationships are as important as the data itself, such as social networks, fraud detection, recommendation engines, network analysis, and knowledge graphs.

1. Social Networks:

- Modeling users as nodes and relationships (friendship, following) as edges allows for efficient representation and traversal of social networks, enabling features like friend recommendations.

2. Recommendation Engines:

- Nodes representing users and items (products, content) with edges capturing interactions help build personalized recommendation systems by analyzing the graph structure.

3. Fraud Detection:

- Identifying unusual patterns and connections in financial transactions or user activities by traversing the graph can efficiently detect fraud and security breaches.

4. Knowledge Graphs:

- Modeling information as nodes and relationships facilitates the creation of knowledge graphs, aiding in organizing and navigating interconnected data for insights and discovery.

5. Biological and Medical Research:

- Graph databases are employed to represent and analyze relationships between biological entities (genes, proteins) in genomics research, drug discovery, and systems biology.

6. IT Operations and Dependency Mapping:

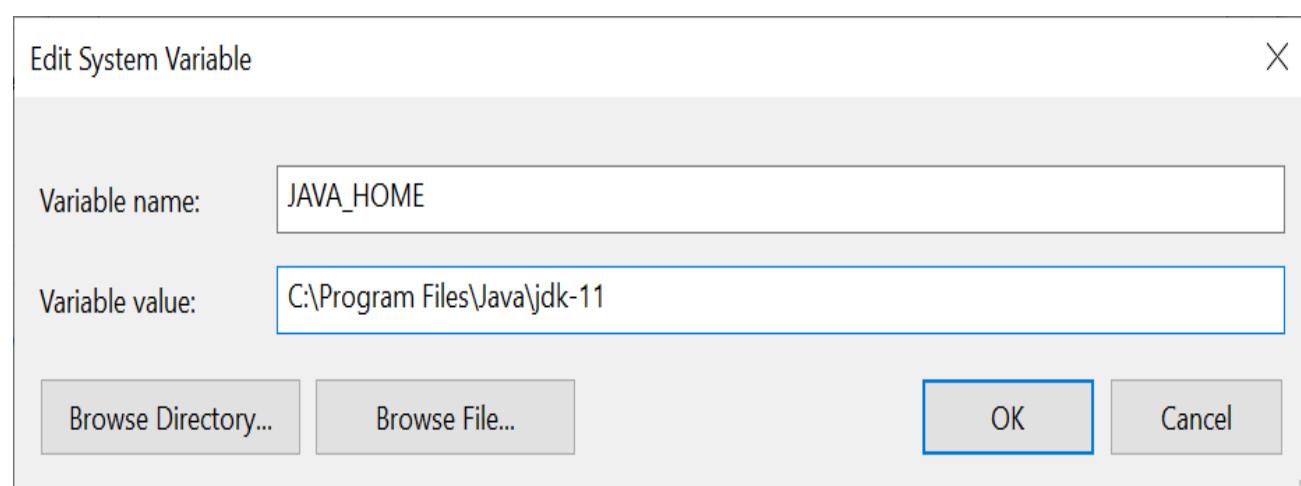
- Representing IT components and their dependencies as nodes and edges helps visualize and manage complex IT infrastructures, optimizing operations and troubleshooting.

Installing Neo4j on Windows

Follow these instructions to install Neo4j on your server:

1. Install Java:

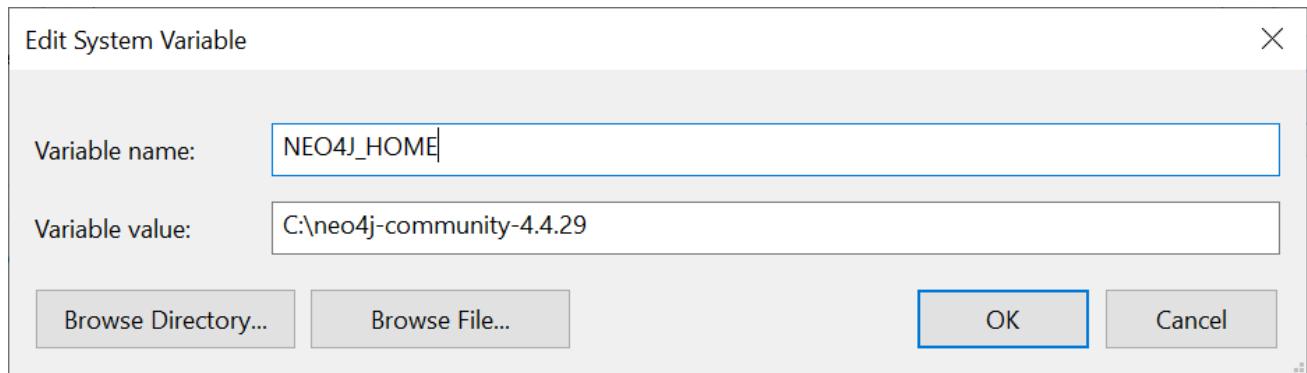
- If OpenJDK 11 or Oracle Java 11 is not already installed, obtain and install it on your system.
- Set the `JAVA_HOME` environment variable to the installation path of your JDK.



2. Download Neo4j:

- Visit the Neo4j Download Center and download the latest release.

- <https://neo4j.com/deployment-center/>
- Always download Neo4j from the Neo4j Download Center.
- Verify the integrity by checking the SHA hash. Click on SHA-256 below your downloaded file on the Download Center and compare it with the calculated SHA-256 hash using platform-specific commands.
- Right-click the downloaded ZIP file and select "Extract All."
- Choose a permanent location for the extracted files, for example, D:\neo4j\ (referred to as NEO4J_HOME).



3. Run Neo4j:

- To run Neo4j as a console application, use: `<NEO4J_HOME>\bin\neo4j console`.

```
C:\Windows\System32\cmd.exe
C:\neo4j-community-4.4.29\bin>neo4j
Missing required subcommand
Usage: Neo4j <COMMAND>
Neo4j database server CLI.

Commands:
  console          Start server in console.
  start            Start server as a daemon.
  stop             Stop the server daemon.
  restart          Restart the server daemon.
  status           Get the status of the server.
  install-service  Install the Windows service.
  uninstall-service Uninstall the Windows service.
  update-service   Update the Windows service.
  version, --version Print version information and exit.
  help, --help      Displays help information about the specified command

C:\neo4j-community-4.4.29\bin>
```

- To install Neo4j as a service, use: `<NEO4J_HOME>\bin\neo4j install-service`.

```
C:\neo4j-community-4.4.29\bin>neo4j install-service
```

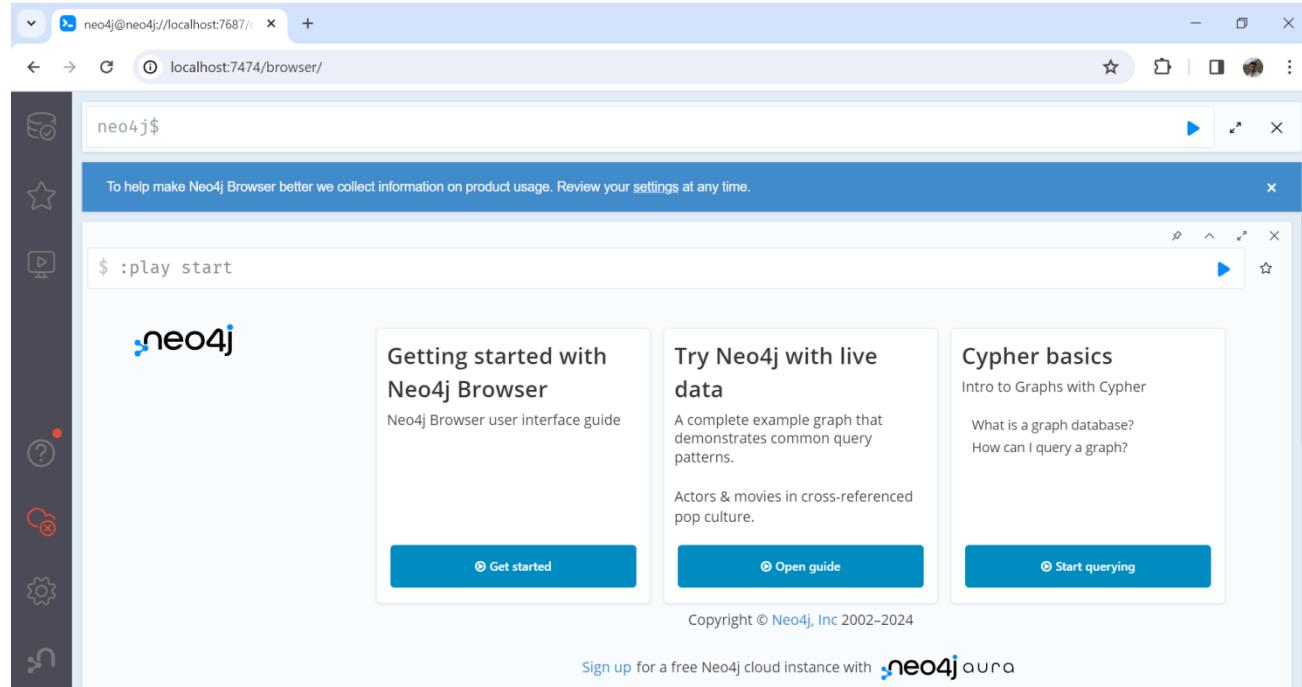
Start neo4j with command neo4j start

```
c:\ Select C:\Windows\System32\cmd.exe
C:\neo4j-community-4.4.29\bin>neo4j install-service
Neo4j service is already installed

C:\neo4j-community-4.4.29\bin>neo4j start
Directories in use:
home:          C:\neo4j-community-4.4.29
config:         C:\neo4j-community-4.4.29\conf
logs:          C:\neo4j-community-4.4.29\logs
plugins:        C:\neo4j-community-4.4.29\plugins
import:         C:\neo4j-community-4.4.29\import
data:           C:\neo4j-community-4.4.29\data
certificates:  C:\neo4j-community-4.4.29\certificates
licenses:       C:\neo4j-community-4.4.29\licenses
run:            C:\neo4j-community-4.4.29\run
Starting Neo4j.
Started neo4j. It is available at http://localhost:7474
There may be a short delay until the server is ready.
```

4. Access Neo4j Browser:

- Visit <http://localhost:7474> in your web browser.
- Connect using the username 'neo4j' with the default password 'neo4j'.
- You will be prompted to change the password.



- Stop the server by typing Ctrl-C in the console.

H. Resources/Equipment Required

1. Computers or Laptops: Students need access to individual computers or laptops capable of running Neo4j software.

2. Internet Connection: A stable internet connection is required for software downloads and accessing online resources.
 3. Neo4j Software: Ensure the latest version of Neo4j is available for download and installation on student machines.
 4. Documentation and Guides: Provide instructional materials covering graph database basics, graph theory, use cases, and Neo4j installation steps.
 5. Technical Support: Have support available to assist students with any software or installation issues during the practical session.
-
2. The Cypher query language is specifically designed for querying relational databases.
 3. Nodes in a graph database represent entities, while edges represent properties.
 4. Amazon Neptune is an example of a document database.
 5. Graph databases are not efficient in handling queries involving relationships.
 6. To install Neo4j as a service, the command is `<NEO4J_HOME>\bin\neo4j run-service`.

Practical 7. Basic Graph Queries and Implementations with Neo4j

A. Objective

Students will acquire expertise in the Cypher Query Language, enabling them to execute fundamental graph operations, explore graph algorithms and their applications. The practical will additionally cover Neo4j optimization techniques and real-world graph database scenarios, enhancing students' skills in graph querying and implementations.

B. Relevant Program Outcomes (POs)

1. Basic and Discipline specific knowledge: Apply foundational knowledge of mathematics, science, and engineering principles, along with specialized engineering expertise, to address engineering challenges encountered in the practical implementation of basic graph queries and solutions with Neo4j.
2. Problem analysis: Demonstrate the ability to identify and analyze specific engineering problems related to graph databases, utilizing established methods to formulate solutions and address challenges in graph query implementations.
3. Design/development of solutions: Design effective solutions for engineering problems involving graph databases, contributing to the development of technical strategies for implementing graph queries within the Neo4j environment.
4. Engineering Tools, Experimentation and Testing: Apply contemporary engineering tools and methodologies to conduct standardized tests and measurements, specifically in the context of basic graph queries and implementations using Neo4j.
5. Engineering practices for society, sustainability, and environment: Apply engineering practices that align with societal needs, sustainability principles, and environmental considerations while implementing basic graph queries in the Neo4j graph database.
6. Project Management: Apply engineering management principles, whether working individually, as part of a team, or in a leadership role, to effectively manage projects involving the implementation of basic graph queries and solutions using Neo4j.
7. Life-long learning: Demonstrate the ability to analyze individual learning needs and engage in ongoing updates and learning activities, particularly in response to technological advancements in the field of graph databases and Neo4j implementations.

C. Competency and Practical Skills

1. Cypher Query Proficiency: Execute fundamental graph queries using the Cypher Query Language in Neo4j, showcasing adeptness in retrieving and manipulating graph data.
2. Graph Operations Mastery: Apply essential graph operations, including node and relationship creation, updating, and deletion, to address practical scenarios within the Neo4j environment.

3. Neo4j Optimization Techniques: Implement optimization techniques within Neo4j to enhance query performance and improve system efficiency, particularly in managing large-scale graph databases.
4. Real-world Scenario Resolution: Apply acquired skills to solve real-world graph database scenarios, engaging in hands-on experiences related to query optimization, troubleshooting, and effective decision-making in graph-related problem-solving.

D. Relevant Course Outcomes (Cos)

Identify the significance of graph databases, illustrating their practical applications in solving complex relationship-oriented problems.

E. Practical Outcome (PRo)

Students will achieve a comprehensive understanding of basic graph queries and implementations using Neo4j. The practical outcome includes hands-on proficiency in the Cypher Query Language, mastery of essential graph operations, application of graph algorithms, utilization of Neo4j optimization techniques, and successful resolution of real-world graph database scenarios.

F. Expected Affective Domain Outcome (ADos)

1. Increased Appreciation for Graph Database Concepts: Develop a heightened appreciation for the relevance and significance of graph database concepts, recognizing their role in solving real-world engineering challenges.
2. Enhanced Problem-Solving Attitude: Cultivate a problem-solving mindset specific to graph databases, fostering the ability to identify, analyze, and address engineering issues through effective graph query implementations with Neo4j.
3. Confidence in Cypher Query Language: Build confidence and proficiency in using the Cypher Query Language, instilling a sense of accomplishment and mastery in performing basic graph queries and implementations.
4. Application of Optimization Techniques: Acquire the capability to strategically apply Neo4j optimization techniques, demonstrating a nuanced understanding of how to enhance the performance of graph database queries in practical scenarios.

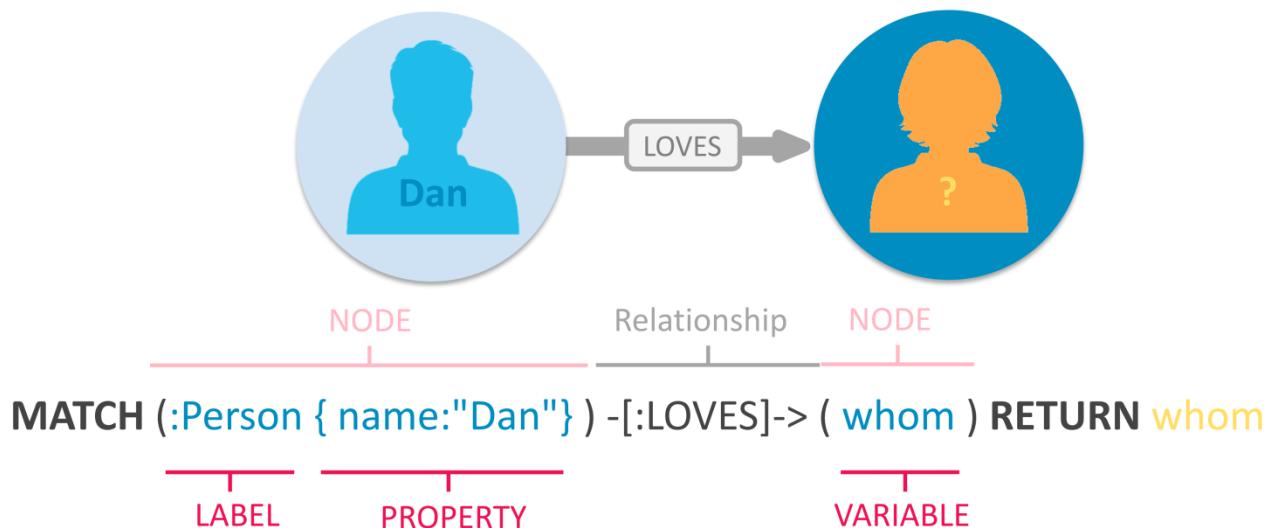
G. Prerequisite Theory

Cypher Query Language

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. It is like SQL for graphs, and was inspired by SQL so it lets you focus on what data you want out of the graph (not how to go get it). It is the easiest graph language to learn by far because of its similarity to other languages and intuitiveness.

Cypher is unique because it provides a visual way of matching patterns and relationships. Cypher was inspired by an ASCII-art type of syntax where (nodes)-[:ARE_CONNECTED_TO]->(otherNodes) using rounded brackets for circular (nodes), and -[:ARROWS]-> for relationships. When you write a query, you draw a graph pattern through your data.

Neo4j users use Cypher to construct expressive and efficient queries to do any kind of create, read, update, or delete (CRUD) on their graph, and Cypher is the primary interface for Neo4j.



Once you start neo4j, you can use the :play cypher command inside of Neo4j Browser to get started.

\$:play cypher

CREATE

Create a node

Let's use Cypher to generate a small social graph.

CREATE (ee:Person { name: "Emil", from: "Sweden", klout: 99 })

- **CREATE** clause to create data
- **()** parenthesis to indicate a node
- **ee:Person** a variable 'ee' and label 'Person' for the new node
- **{}** brackets to add properties to the node

2 / 9 < • • • • • • • • • • >

Neo4j's developer pages cover the basics of the language, which you can explore by topic area below, starting with basic material, and building up towards more complex material.

Cypher provides first class support for a number of data types. These fall into several categories which will be described in detail in the following subsections:

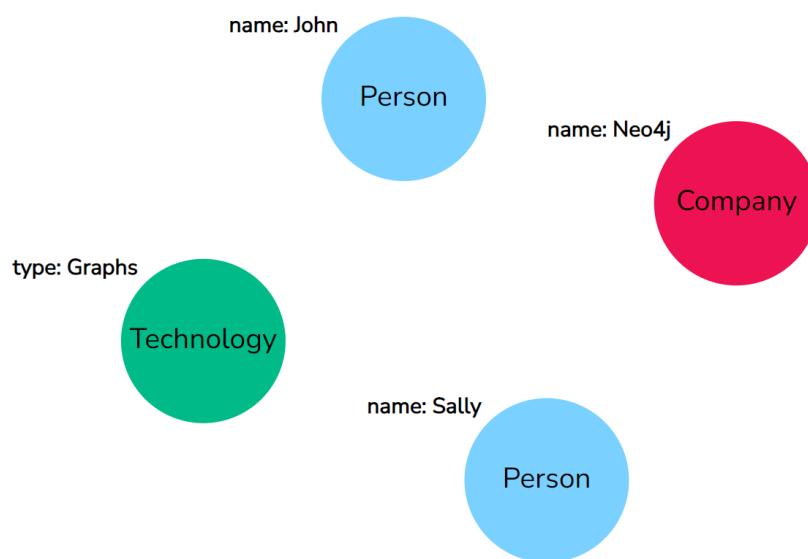
Property types: Integer, Float, String, Boolean, Point, Date, Time, LocalTime, DateTime, LocalDateTime, and Duration.

Structural types: Node, Relationship, and Path.

Composite types: List and Map

In Cypher, comments are added by starting a line with // and writing text after the slashes. Using two forward slashes designates the entire line as a comment, explaining syntax or query functionality.

In Cypher, representing nodes involves enclosing them in parentheses, mirroring the visual representation of circles used for nodes in the graph model. Nodes, which signify data entities, are identified by finding nouns or objects in the data model. For instance, in the example (Sally), (John), (Graphs), and (Neo4j) are nodes.



1. Node Variables:

- Nodes in Cypher can be assigned variables, such as (p) for person or (t) for thing. These variables function similarly to programming language variables, allowing you to reference the nodes by the assigned name later in the query.

2. Node Labels:

- Node labels in Cypher, similar to tags in the property graph data model, help group similar nodes together. Labels, like Person, Technology, and Company, act as identifiers, aiding in specifying certain types of entities to look for or create. Using node labels in queries helps Cypher optimize execution and distinguish between different entities.

```

()          //anonymous node (no label or variable) can refer to any node in the database
(p:Person)    //using variable p and label Person
(:Technology) //no variable, label Technology
(work:Company) //using variable work and label Company

```

In Cypher, relationships are denoted by arrows (--) or (<-->) between nodes, resembling the visual representation of connecting lines. Relationship types and properties can be specified in square brackets within the arrow. Directed relationships use arrows, while undirected relationships use double dashes (--) allowing flexible traversal in either direction without specifying the physical orientation in queries.

```

//data stored with this direction
CREATE (p:Person)-[:LIKES]->(t:Technology)

//query relationship backwards will not return results
MATCH (p:Person)<-[:LIKES]-(t:Technology)

//better to query with undirected relationship unless sure of direction
MATCH (p:Person)-[:LIKES]-(t:Technology)

```

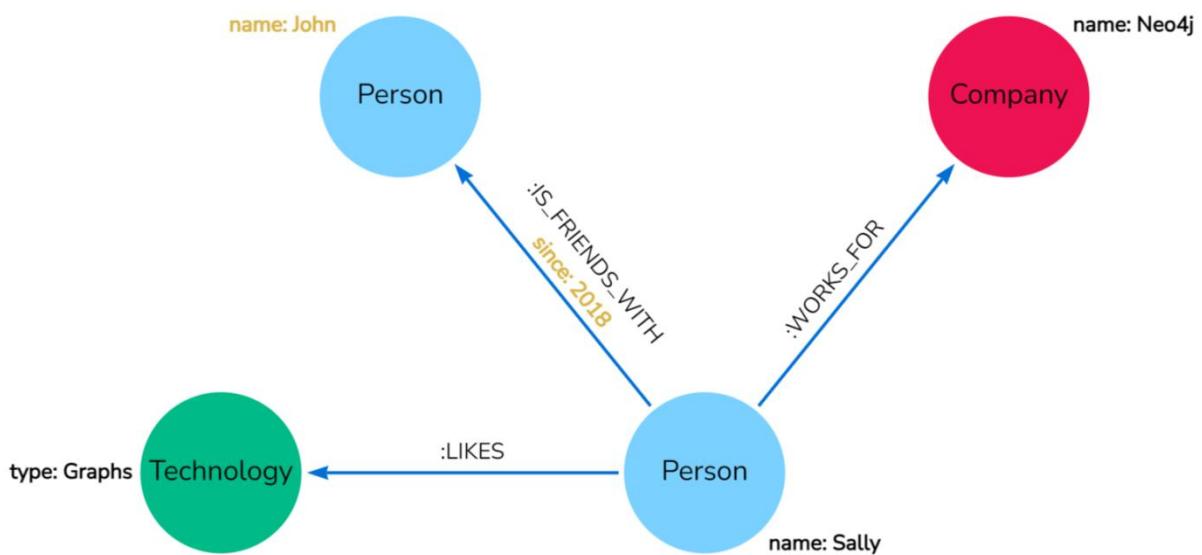
1. Relationship Types:

- Relationship types in Cypher categorize connections between nodes, providing meaning to the relationships similar to how labels group nodes. Good naming conventions using verbs or actions are recommended for clarity and readability in Cypher queries.

2. Relationship Variables:

- Like nodes, relationships in Cypher can be assigned variables such as [r] or [rel]. These variables, whether short or expressive like [likes] or [knows], allow referencing the relationship later in a query. Anonymous relationships can be specified with two dashes (--, ->, <--) if they are not needed for reference.

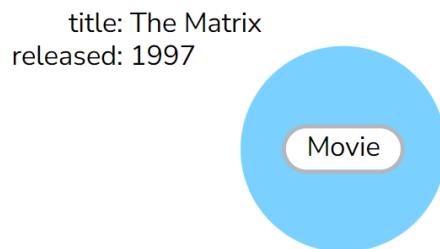
In Cypher, node and relationship properties are represented using curly braces within the parentheses for nodes and brackets for relationships. For example, a node property is expressed as `(p:Person {name: 'Sally'})`, and a relationship property is denoted as `-[rel:IS_FRIENDS_WITH {since: 2018}]->`.



In Cypher, patterns are composed of nodes and relationships, expressing the fundamental structure of graph data. Patterns can range from simple to intricate, and in Cypher, they are articulated by combining node and relationship syntax, such as `(p:Person {name: "Sally"})-[rel:LIKES]-(g:Technology {type: "Graphs"})`.

Creating Data with CREATE Clause:

- In Cypher, the `CREATE` clause is used to add data by specifying patterns representing graph structures, labels, and properties. For example, `CREATE (:Movie {title: 'The Matrix', released: 1997})` creates a movie node with specified properties.



- To return created data, the `RETURN` clause is added, referencing variables assigned to pattern elements. For instance, `CREATE (p:Person {name: 'Keanu Reeves'}) RETURN p` creates a person node and returns it in the result.

Matching Patterns with MATCH Clause:

- The `MATCH` clause is used for finding patterns in the graph. It enables specifying patterns similar to `CREATE` but focuses on identifying existing data. For example, `MATCH (m:Movie) RETURN m` finds all movie nodes.



- The `MERGE` clause combines elements of `MATCH` and `CREATE`, ensuring uniqueness by checking for existing data before creating. It's useful for creating or matching nodes and relationships. For instance, `MERGE (m:Movie {title: 'Cloud Atlas'}) ON CREATE SET m.released = 2012 RETURN m` merges or creates a movie node and returns it.

- Return values can be aliased for better readability using the `AS` keyword. For example, `RETURN tom.name AS name, tom.born AS 'Year Born'` provides cleaner and more informative result labels.

Filtering results

Explore result refinement in Cypher by using the WHERE clause to filter and retrieve specific subsets of data based on boolean expressions, predicates, and comparisons, including logical operators like AND, OR, XOR, and NOT.

```
MATCH (m:Movie)
WHERE m.title = 'The Matrix'
RETURN m
```

1. Inserting Data:

- Use the `CREATE` keyword to add nodes and relationships to Neo4j.
- Patterns can be created blindly, but using `MATCH` before `CREATE` ensures uniqueness.

```
CREATE (j:Person {name: 'Jennifer'})-[rel:IS_FRIENDS_WITH]->(m:Person {name: 'Mark'})
```

2. Updating Data:

- Modify node properties using `SET` after a `MATCH` statement.
- Update relationship properties similarly by specifying the relationship in the `MATCH` clause.

```
MATCH (p:Person {name: 'Jennifer'})
SET p.birthdate = date('1980-01-01')
RETURN p
```

3. Deleting Data:

- Delete relationships with `DELETE` after a `MATCH` specifying the relationship.

```
MATCH (j:Person {name: 'Jennifer'})-[r:IS_FRIENDS_WITH]->(m:Person {name: 'Mark'})
DELETE r
```

- Delete nodes without relationships using `DELETE` after a `MATCH` specifying the node.

```
MATCH (m:Person {name: 'Mark'})
DELETE m
```

- Use `DETACH DELETE` to delete a node along with its relationships.

```

MATCH (m:Person {name: 'Mark'})
DETACH DELETE m

```

4. Deleting Properties:

- Remove properties using `REMOVE` or set them to `null` with `SET` for nodes.

```

//delete property using REMOVE keyword
MATCH (n:Person {name: 'Jennifer'})
REMOVE n.birthdate

```

```

//delete property with SET to null value
MATCH (n:Person {name: 'Jennifer'})
SET n.birthdate = null

```

5. Avoiding Duplicate Data with MERGE:

- Use `MERGE` to perform a "select-or-insert" operation for nodes and relationships.
- `MERGE` checks for the entire pattern's existence and creates it if not found.

```

MERGE (mark:Person {name: 'Mark'})
RETURN mark

```

6. Handling MERGE Criteria:

- Utilize `ON CREATE SET` and `ON MATCH SET` to specify actions during node or relationship creation or matching.
- This helps initialize properties when creating and update properties when matching.

```

MATCH (j:Person {name: 'Jennifer'})
MATCH (m:Person {name: 'Mark'})
MERGE (j)-[r:IS_FRIENDS_WITH]->(m)
RETURN j, r, m

```

Graph algorithms and their applications

Graph algorithms provide one of the most potent approaches to analyzing connected data because their mathematical calculations are specifically built to operate on relationships. They describe steps to be taken to process a graph to discover its general qualities or specific quantities.

The library contains implementations for the following types of algorithms:

- Path Finding - these algorithms help find the shortest path or evaluate the availability and quality of routes
- Centrality - these algorithms determine the importance of distinct nodes in a network
- Community Detection - these algorithms evaluate how a group is clustered or partitioned, as well as its tendency to strengthen or break apart
- Similarity - these algorithms help calculate the similarity of nodes
- Topological link prediction - these algorithms determine the closeness of pairs of nodes

- Node Embeddings - these algorithms compute vector representations of nodes in a graph.
- Node Classification - this algorithm uses machine learning to predict the classification of nodes.
- Link prediction - these algorithms use machine learning to predict new links between pairs of nodes

Neo4j optimization techniques

Memory Configuration Guidelines:

OS Memory Sizing:

- Reserve around 1GB for non-Neo4j server activities.
- Avoid exceeding available RAM to prevent OS swapping, which impacts performance.

Page Cache Sizing:

- Utilize the page cache to cache Neo4j data stored on disk.
- Estimate page cache size by summing the sizes of relevant database files and adding a growth factor.
- Configure the page cache size in `neo4j.conf` (default is 50% of available RAM).

Heap Sizing:

- Configure a sufficiently large heap space for concurrent operations (8G to 16G is often adequate).
- Adjust heap size using parameters `dbms.memory.heap.initial_size` and `dbms.memory.heap.max_size` in `neo4j.conf`.
- Set these parameters to the same size for optimal performance.

(Refer to the Neo4j Operations Manual for detailed discussions on heap memory configuration, distribution, and garbage collection tuning.)

Logical Logs:

- Logical transaction logs are crucial for recovery after an unclean shutdown and incremental backups.
- Log files are rotated after reaching a specified size (e.g., 25 MB).
- Configure log retention policy using the `dbms.tx_log.rotation.retention_policy` parameter (recommended: 7 days).

Number of Open Files:

- The default open file limit of 1024 may be insufficient, especially with multiple indexes or high connection volumes.

- Increase the limit to a practical value (e.g., 40000) based on usage patterns.
- Adjust system-wide open file limit following platform-specific instructions (ulimit command for current session).

Real-world graph database scenarios

Example #1: Using Neo4j to determine customer preferences

Suppose we need to learn preferences of our customers to create a promotional offer for a specific product category, such as notebooks. First, Neo4j allows us to quickly obtain a list of notebooks that customers have viewed or added to their wish lists. We can use this code to select all such notebooks:

```

MATCH (:Customer)-[:ADDED_TO_WISH_LIST|:VIEWED]->(notebook:Product)-
[:IS_IN]->(:Category {title: 'Notebooks'})

RETURN notebook;

```

Now that we have a list of notebooks, we can easily include them in a promotional offer. Let's make a few modifications to the code above:

```

CREATE(offer:PromotionalOffer {type: 'discount_offer', content: 'Notebooks discount
offer...'})

WITH offer

MATCH (:Customer)-[:ADDED_TO_WISH_LIST|:VIEWED]->(notebook:Product)-
[:IS_IN]->(:Category {title: 'Notebooks'})

MERGE(offer)-[:USED_TO_PROMOTE]->(notebook);

```

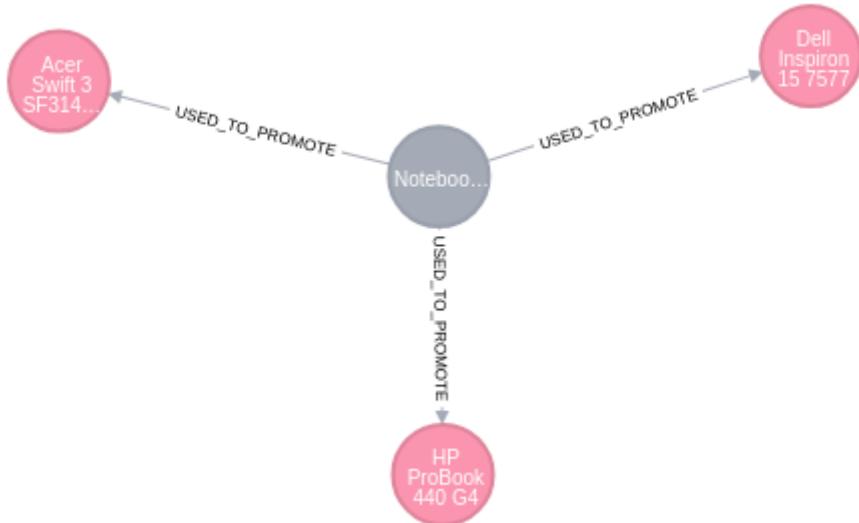
We can track the changes in the graph with the following query:

```

MATCH (offer:PromotionalOffer)-[:USED_TO_PROMOTE]->(product:Product)

RETURN offer, product;

```



Linking a promotional offer with specific customers makes no sense, as the structure of graphs allows you to access any node easily. We can collect emails for a newsletter by analyzing the products in our promotional offer.

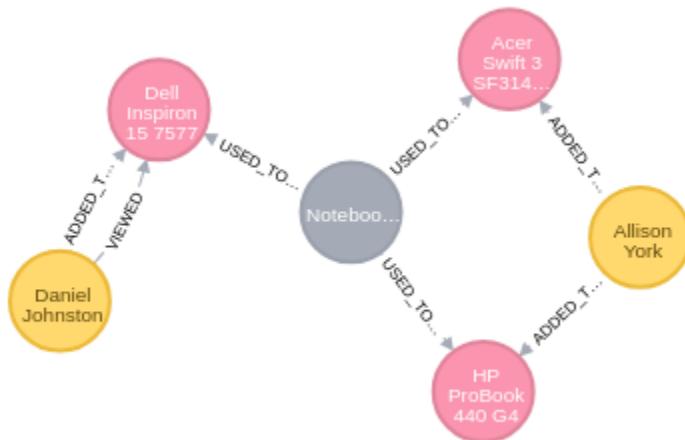
When creating a promotional offer, it's important to know what products customers have viewed or added to their wish lists. We can find out with this query:

```

MATCH (offer:PromotionalOffer {type: 'discount_offer'})-[:USED_TO_PROMOTE]-
>(product:Product)<-[:ADDED_TO_WISH_LIST|:VIEWED]-(customer:Customer)

RETURN offer, product, customer;

```



This example is simple, and we could have implemented the same functionality in a relational database. But our goal is to show the intuitiveness of Cypher and to demonstrate how simple it is to write queries in Neo4j.

Example #2: Using Neo4j to devise promotional offers

Now let's imagine that we need to develop a more efficient promotional campaign. To increase conversion rates, we should offer alternative products to our customers. For

example, if a customer shows interest in a certain product but doesn't buy it, we can create a promotional offer that contains alternative products.

To show how this works, let's create a promotional offer for a specific customer:

```

MATCH (alex:Customer {name: 'Alex McGyver'})

MATCH (free_product:Product)

WHERE NOT ((alex)-->(free_product))

MATCH (product:Product)

WHERE ((alex)-->(product))

MATCH (free_product)-[:IS_IN]->()-<[:IS_IN]-(product)

WHERE ((product.price - product.price * 0.20) >= free_product.price <= (product.price + product.price * 0.20))

RETURN free_product;

```

This query searches for products that don't have either ADDED_TO_WISH_LIST, VIEWED, or BOUGHT relationships with a client named Alex McGyver. Next, we perform an opposite query that finds all products that Alex McGyver has viewed, added to his wish list, or bought. Also, it's crucial to narrow down recommendations, so we should make sure that these two queries select products in the same categories. Finally, we specify that only products that cost 20 percent more or less than a specific item should be recommended to the customer.

Now let's check if this query works correctly.

The product variable is supposed to contain the following items:

Xiaomi Mi Mix 2 (price: \$420.87). Price range for recommendations: from \$336.70 to \$505.04.

Sony Xperia XA1 Dual G3112 (price: \$229.50). Price range for recommendations: from \$183.60 to \$275.40.

The free_product variable is expected to have these items:

Apple iPhone 8 Plus 64GB (price: \$874.20)

Huawei P8 Lite (price: \$191.00)

Samsung Galaxy S8 (price: \$784.00)

Sony Xperia Z22 (price: \$765.00)

Note that both product and free_product variables contain items that belong to the same category, which means that the [:IS_IN]->()-<[:IS_IN] constraint has worked.

As you can see, none of the products except for the Huawei P8 Lite fits in the price range for recommendations, so only the P8 Lite will be shown on the recommendations list after the query is executed.

Now we can create our promotional offer. It's going to be different from the previous one (personal_replacement_offer instead of discount_offer), and this time we're going to store a customer's email as a property of the USED_TO_PROMOTE relationship as the products contained in the free_product variable aren't connected to specific customers. Here's the full code for the promotional offer:

```

MATCH (alex:Customer {name: 'Alex McGyver'})

MATCH (free_product:Product)

WHERE NOT ((alex)-->(free_product))

MATCH (product:Product)

WHERE ((alex)-->(product))

MATCH (free_product)-[:IS_IN]->()-[:IS_IN]-(product)

WHERE ((product.price - product.price * 0.20) >= free_product.price <= (product.price + product.price * 0.20))

CREATE(offer:PromotionalOffer {type: 'personal_replacement_offer', content: 'Personal replacement offer for ' + alex.name})

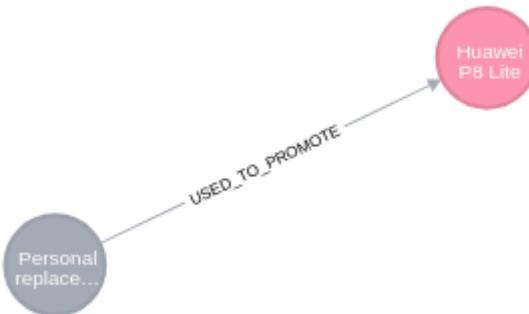
WITH offer, free_product, alex

MERGE(offer)-[rel:USED_TO_PROMOTE {email: alex.email}]->(free_product)

RETURN offer, free_product, rel;
```

Let's take a look at the result of this query:

- In the form of a graph



- In the form of a table

offer	free_product	ref
<pre>{ "type": "personal_replacement_offer", "content": "Personal replacement offer for Alex McGyver" }</pre>	<pre>{ "availability": true, "title": "Huawei P8 Lite", "shippability": true, "price": 191.0 }</pre>	<pre>{ "email": "mcgalex@example.com" }</pre>

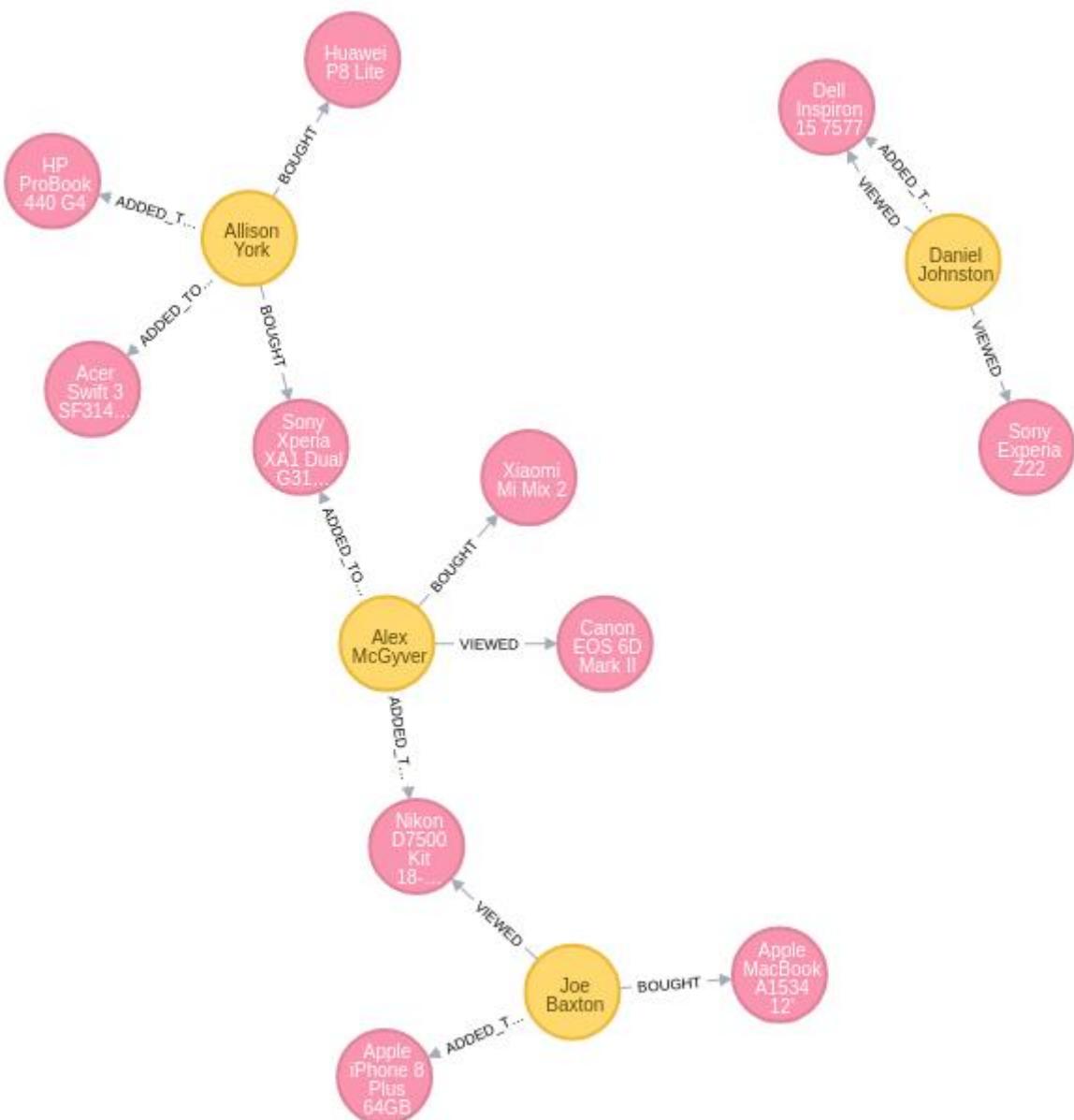
Example #3: Building a recommendation system with Neo4j

The Neo4j database proves useful for building a recommendation system.

Imagine we want to recommend products to Alex McGyver according to his interests. Neo4j allows us to easily track the products Alex is interested in and find other customers who also have expressed interest in these products. Afterward, we can check out these customers' preferences and suggest new products to Alex.

First, let's take a look at all customers and the products they've viewed, added to their wish lists, and bought:

```
MATCH (customer:Customer)-->(product:Product)  
  
RETURN customer, product;
```



As you can see, Alex has two touch points with other customers: the Sony Xperia XA1 Dual G3112 (purchased by Allison York) and the Nikon D7500 Kit 18–105mm VR (viewed by Joe Baxton). Therefore, in this particular case, our product recommendation system should offer to Alex those products that Allison and Joe are interested in (but not the products Alex is also interested in). We can implement this simple recommendation system with the help of the following query:

```

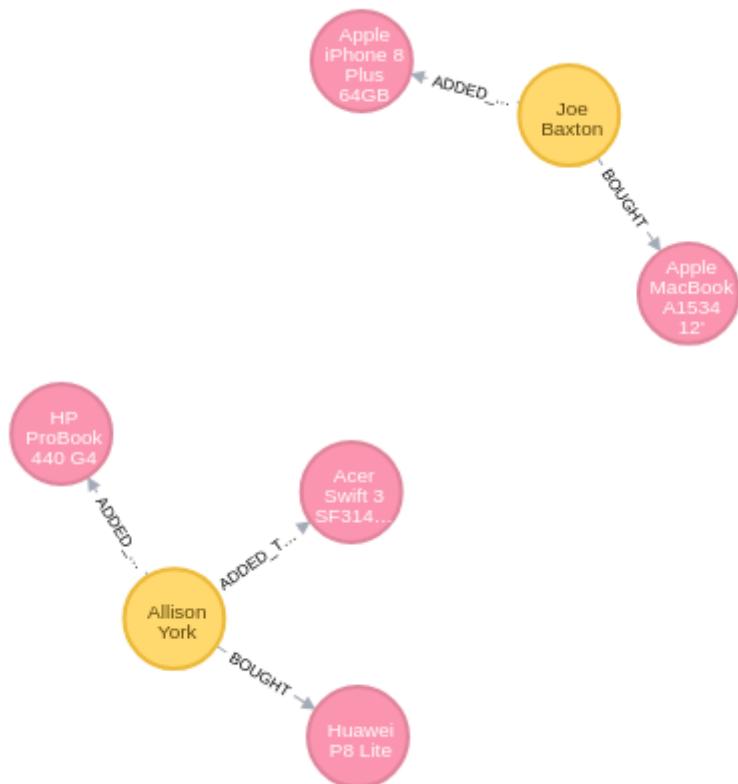
MATCH (:Customer {name: 'Alex McGyver'})-->(product:Product)<--(customer:Customer)

MATCH (customer)-->(customer_product:Product)

WHERE (customer_product <> product)

RETURN customer, customer_product;

```



We can further improve this recommendation system by adding new conditions, but the takeaway is that Neo4j helps you build such systems quickly and easily.

H. Resources/Equipment Required

1. Neo4j Database: Access to a functioning Neo4j database is essential for hands-on experience with basic graph queries.
2. Computing Device: Students need personal computers or laptops meeting the system requirements for Neo4j installation and execution.
3. Internet Connection: A stable internet connection is necessary for accessing Neo4j resources, documentation, and additional learning materials during the practical.
4. Cypher Query Language Guide: Students should have access to a guide or documentation for the Cypher Query Language to assist in formulating and executing queries.

Write Cypher queries for following

1. to add a new node representing a movie with the title "Inception" and release year 2010. Write a Cypher query to achieve this.
2. Retrieve the names of all actors who have acted in movies released after the year 2000. Write a Cypher query for this.
3. Change the release year of the movie "The Matrix" to 1999. Write a Cypher query to update this information.
4. Delete a relationship between a person node (representing a user) and a movie node (representing a favorite movie). Write a Cypher query to perform this deletion.
5. Retrieve the names of directors who directed movies with a rating higher than 8. Write a Cypher query with the WHERE clause.
6. Create a new node representing an actor named "Tom Hanks" and assign the label "Actor" to the node. Write a Cypher query for this.
7. Find all pairs of actors who have acted together in the same movie. Write a Cypher query to retrieve this information.
8. Retrieve the names of users who have rated a movie with a rating greater than 4. Write a Cypher query with the MATCH and RETURN clauses.
9. Connect a person node representing "Alice" to a movie node representing "The Shawshank Redemption" with a relationship indicating that Alice has rated the movie. Write a Cypher query for this.
- 10 Retrieve the titles of movies released between 2010 and 2020. Write a Cypher query with conditional operators.

Practical 8. Redis Basics: Introduction and Key-Value Operations

A. Objective

To provide a comprehensive understanding of Redis, covering its overview, key data structures, real-world applications, essential commands, and advanced features, integrated with diverse technologies. This approach ensures students acquire hands-on proficiency in employing Redis for optimal data operations and exploring its broader utility in contemporary tech ecosystems.

B. Relevant Program Outcomes (POs)

1. Application of Basic Knowledge:

- Apply fundamental knowledge of mathematics, science, and engineering to address engineering challenges encountered in Redis Basics, ensuring a solid foundation for problem-solving.

2. Problem Analysis:

- Utilize codified standard methods to identify and analyze well-defined engineering problems within Redis, fostering a systematic approach to problem-solving.

3. Design and Development:

- Design effective solutions for well-defined technical issues in Redis, contributing to the development of systems components or processes aligned with specified requirements.

4. Engineering Tools and Testing:

- Apply contemporary engineering tools and techniques to conduct standard tests and measurements, enhancing proficiency in utilizing engineering tools within the context of Redis.

5. Engineering Practices and Sustainability:

- Incorporate appropriate technology, considering societal, sustainability, and environmental factors, while adhering to ethical practices in the context of Redis operations.

6. Project Management:

- Demonstrate the application of engineering management principles for effective project management, whether working individually, as a team member, or in a leadership role, while communicating engineering activities clearly.

7. Life-long Learning:

- Exhibit the ability to assess individual learning needs and engage in continuous learning, adapting to technological changes in the field of engineering, particularly in the context of Redis Basics.

C. Competency and Practical Skills

1. Key-Value Operations Proficiency:

- Showcase mastery in executing key-value operations, managing data efficiently in Redis.

2. Data Structure Manipulation:

- Demonstrate practical skills in manipulating various Redis data structures for effective information storage and retrieval.

3. Advanced Features Utilization:

- Apply competencies in utilizing advanced Redis features, including transactions, Pub/Sub, and geospatial indexes, for diverse engineering applications.

4. Real-World Application Proficiency:

- Apply Redis to address real-world engineering challenges, showcasing practical skills in developing solutions aligned with industry needs.

D. Relevant Course Outcomes (Cos)

Utilize Redis data structures and functionalities to implement efficient caching strategies, showcasing the role of Redis in enhancing data retrieval performance.

E. Practical Outcome (PRo)

Students will gain hands-on proficiency in executing key-value operations, manipulating diverse data structures, and solving engineering problems using Redis. The practical outcome emphasizes competency in executing essential Redis commands, showcasing the integration of Redis with other technologies, and applying this knowledge to real-world scenarios. This comprehensive approach ensures students acquire practical skills essential for effective data management and engineering applications in various contexts.

F. Expected Affective Domain Outcome (ADos)

1. Enhanced Confidence:

- Students are expected to develop increased confidence in working with Redis, gaining assurance in executing key-value operations and utilizing various data structures for practical problem-solving.

2. Increased Appreciation for Real-World Applications:

- The practical is designed to foster an appreciation for the real-world applications of Redis, allowing students to understand how key-value operations and data structures align with solving tangible engineering challenges.

3. Improved Problem-Solving Skills:

- Through hands-on exercises and application of Redis in engineering scenarios, students are expected to enhance their problem-solving skills, cultivating a practical mindset in addressing challenges using Redis functionalities.

4. Heightened Adaptability to Technological Tools:

- The practical is anticipated to contribute to students' adaptability to technological tools by exposing them to essential Redis commands and advanced features, preparing them to efficiently integrate Redis into diverse technological environments.

G. Prerequisite Theory

Overview of Redis

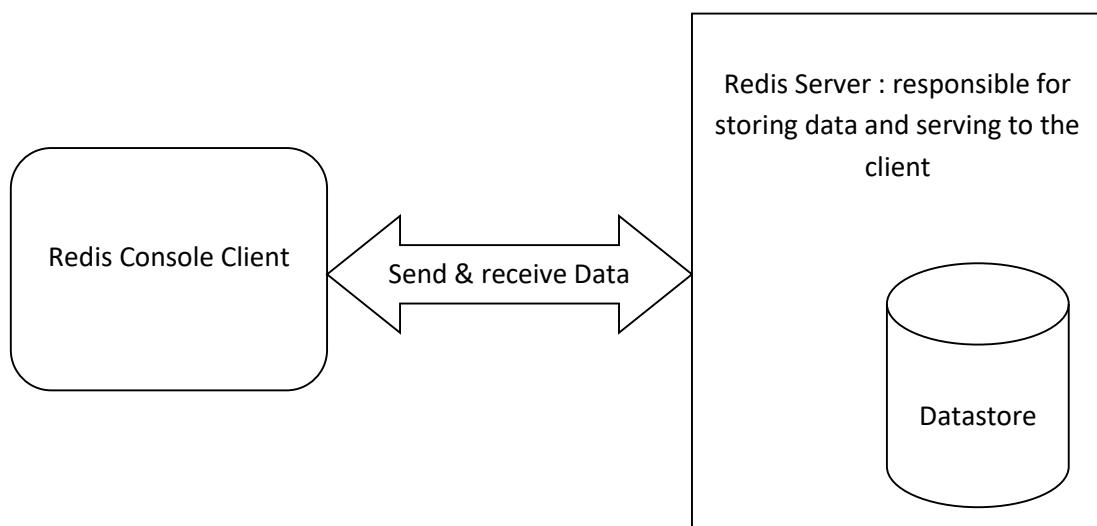
Redis is a NoSQL database which follows the principle of key-value store. The key-value store provides ability to store some data called a value, inside a key. You can receive this data later only if you know the exact key used to store it.

Redis, an open-source (BSD licensed) in-memory data structure store, serves as a versatile solution functioning as a database, cache, and message broker. Classified as a NoSQL database, Redis offers the flexibility to store substantial volumes of data without the constraints typically associated with relational databases.

Distinguishing itself through support for an array of data structures, Redis accommodates strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs, and geospatial indexes featuring radius queries. This diverse set of capabilities positions Redis as a powerful tool for managing varied data types and enables its application across a wide spectrum of use cases.

Redis Architecture:

Redis follows a client-server architecture and is known for its simplicity and efficiency. The key components in Redis architecture are the Redis client and the Redis server.



1. Redis Client:

- The Redis client is any application or program that interacts with the Redis server to perform read and write operations.
- Clients can be written in various programming languages, and Redis provides official client libraries for several languages, including Python, Java, Ruby, and more.
- Clients communicate with the Redis server using the Redis protocol, a lightweight binary protocol optimized for performance.

2. Redis Server:

- The Redis server is the core component responsible for storing and managing data in-memory.
- It operates as a daemon process, running independently on a server, and listens for client connections on a specified port (default is 6379).
- The server manages various data structures, processes commands, and responds to client requests.
- Redis can be configured to run in different modes, such as as a standalone instance, as a master in a replication setup, or as a node in a clustered environment.

Key Characteristics of Redis Architecture:

- Redis stores all data in RAM, providing fast read and write operations. This makes it ideal for use cases where low-latency access to data is crucial.
- Redis is single-threaded, meaning it processes one command at a time. While this may seem like a limitation, it simplifies the design and allows for better predictability and consistency.
- While Redis is an in-memory database, it provides options for persistence. Data can be periodically saved to disk or written to an append-only file, ensuring data durability.
- Redis supports master-slave replication, allowing data to be replicated across multiple nodes. This enhances data availability and provides fault tolerance.
- Redis can be horizontally scaled through partitioning, enabling large datasets to be distributed across multiple nodes.
- Redis processes commands atomically, ensuring that operations either succeed completely or fail, maintaining data consistency.

Redis data structures

Redis supports a variety of data structures, each designed to serve specific use cases. Here's an overview of the main data structures supported by Redis:

1. Strings:

- Strings are the simplest and most basic data type in Redis. They can contain any kind of data, such as text, binary data, or serialized objects.
- Redis provides various operations on strings, including set, get, append, increment, decrement, and more.

2. Hashes:

- Hashes are maps between string field names and string values, so they are useful for representing objects with multiple attributes.
- Hashes in Redis are particularly suitable for storing and retrieving objects with a small number of fields.

3. Lists:

- Lists are collections of ordered elements, where each element can be any data type.
- Redis provides powerful operations on lists, such as push, pop, index-based access, range retrieval, and blocking pop operations.

4. Sets:

- Sets are collections of unique elements with no specific order.
- Set operations include adding, removing, and checking for the existence of members. Redis also supports set operations like union, intersection, and difference between sets.

5. Sorted Sets:

- Sorted sets are similar to sets, but each element in a sorted set is associated with a score.
- Elements in a sorted set are kept sorted based on their scores, allowing for efficient range queries and ranking of elements.

Redis features efficient Bitmaps for binary flag representation, HyperLogLogs for estimating unique element cardinality with minimal memory usage, and Geospatial Indexes for location-based storage and retrieval. These data structures, coupled with Redis's in-memory storage and atomic operations, make it a powerful tool for applications like caching, real-time analytics, and messaging systems. Understanding the distinct characteristics of each data structure is vital for maximizing Redis's effectiveness in various scenarios.

Use cases for Redis

1. Caching:

- Redis is widely used for caching frequently accessed data, reducing the load on backend databases and improving overall application performance.
- Example: Large-scale e-commerce websites often use Redis to cache product details, reducing the load on the primary database and providing users with faster access to frequently viewed items.

2. Real-Time Analytics:

- Redis facilitates real-time analytics by providing low-latency access to data, making it suitable for monitoring and analyzing dynamic information.
- Example: Online gaming platforms utilize Redis for real-time analytics to monitor player activities, track in-game events, and provide instantaneous feedback on player performance.

3. Message Queues:

- Redis's publish/subscribe mechanism makes it a popular choice for building scalable message queues and real-time communication systems.
- Example: Popular messaging platforms leverage Redis as a message queue to ensure seamless and real-time communication between users, allowing for instant message delivery.

4. Session Storage:

- In web applications, Redis serves as an efficient session store, managing user session data with quick read and write operations.
- Example: Social media platforms use Redis as a session store to manage user sessions, ensuring a smooth and responsive experience as users navigate through various features and pages.

5. Leaderboards and Counting:

- Redis's sorted sets are utilized for implementing leaderboards and efficiently managing scores associated with various entities.
- Example: Gaming applications implement Redis sorted sets to create dynamic leaderboards that showcase top scores in real-time, promoting competition and engagement among players.

6. Geospatial Applications:

- Redis supports geospatial data types, making it valuable for location-based services, such as storing and querying items based on geographic coordinates.
- Example: Ride-sharing apps use Redis for geospatial indexing to efficiently locate and match drivers with passengers based on their real-time geographic coordinates, optimizing service delivery.

These examples illustrate how Redis is applied in practical scenarios across different industries, emphasizing its widespread adoption for improving performance, scalability, and real-time capabilities.

Install Redis on Ubuntu

Note: Redis is not officially supported on Windows. To install Redis on Windows, you'll first need to enable WSL2 (Windows Subsystem for Linux). WSL2 lets you run Linux binaries

natively on Windows. For this method to work, you'll need to be running Windows 10 version 2004 and higher or Windows 11.

To install Redis on Ubuntu, go to the terminal and type the following commands –

```
$sudo apt-get update  
$sudo apt-get install redis-server
```

This will install Redis on your machine.

Start Redis

```
$redis-server
```

Check If Redis is Working

```
$redis-cli
```

This will open a redis prompt.

```
redis 127.0.0.1:6379>
```

In the above prompt, **127.0.0.1** is your machine's IP address and **6379** is the port on which Redis server is running. Now type the following **PING** command.

```
redis 127.0.0.1:6379> ping  
PONG
```

This shows that Redis is successfully installed on your machine.

Redis - Configuration

In Redis, there is a configuration file (redis.conf) available at the root directory of Redis. Although you can get and set all Redis configurations by Redis CONFIG command.

Syntax

Following is the basic syntax of Redis CONFIG command.

```
redis 127.0.0.1:6379> CONFIG GET CONFIG_SETTING_NAME
```

Example

```
redis 127.0.0.1:6379> CONFIG GET loglevel  
1) "loglevel"  
2) "notice"
```

To get all configuration settings, use * in place of CONFIG_SETTING_NAME

```
redis 127.0.0.1:6379> CONFIG GET *
```

Edit Configuration

To update configuration, you can edit redis.conf file directly or you can update configurations via CONFIG set command.

```
redis 127.0.0.1:6379> CONFIG SET CONFIG_SETTING_NAME  
NEW_CONFIG_VALUE
```

Example

```
redis 127.0.0.1:6379> CONFIG SET loglevel "notice"
```

Basic commands and operations

Redis commands are used to perform some operations on Redis server.

To run commands on Redis server, you need a Redis client. Redis client is available in Redis package, which we have installed earlier.

Redis provides a rich set of commands and operations to interact with its various data structures. Here are some basic Redis commands and operations:

1. Key-Value Operations:

Indeed, these key-value operations are fundamental to working with Redis. Let's delve into these commands:

1. SET key value:

- This command sets the value associated with a given key. If the key already exists, it updates the existing value; otherwise, it creates a new key-value pair.

```
SET mykey "Hello, Redis!"
```

2. GET key:

- Used to retrieve the value associated with a specific key. It is a fundamental read operation in Redis.

```
GET mykey
```

This would return `"Hello, Redis!"` if the previous SET command was executed.

3. DEL key:

- Deletes a key and its associated value from the Redis database.

```
DEL mykey
```

After executing this command, the key "mykey" and its value will be removed from the database.

These key-value operations form the basis for many Redis use cases, allowing efficient storage and retrieval of data. They are commonly used for caching, session management, and various other scenarios where quick and direct access to data is essential.

2. String Operations:

Certainly, let's delve into these string operations in Redis:

1. APPEND key value:

- This command appends the specified value to the existing value of a key. If the key does not exist, a new key is created with the provided value.

```
SET mystring "Hello"
APPEND mystring ", Redis!"
```

After executing these commands, the value of the key "mystring" would be "Hello, Redis!".

2. STRLEN key:

- Retrieves the length of the string value stored at the specified key.

```
STRLEN mystring
```

This would return the length of the string in the key "mystring".

3. INCR key / DECR key:

- INCR increments the integer value stored at the specified key by 1, while DECR decrements it by 1. If the key does not exist, a new key is created with the value set to 1.

```
SET mycounter 10
INCR mycounter
```

After these commands, the value of "mycounter" would be 11.

```
DECR mycounter
```

Subsequently, the value of "mycounter" would be 10 again.

These string operations are useful for manipulating and analyzing text-based data as well as managing counters or numerical values in Redis.

3. Hash Operations:

Certainly, let's explore these hash operations in Redis:

1. HSET key field value:

- Sets the value of a field within a hash. If the hash does not exist, a new hash is created with the specified key.

```
HSET user:1001 username "john_doe"
```

This command sets the username field to "john_doe" within the hash associated with the key "user:1001".

2. HGET key field:

- Retrieves the value of a specified field within a hash.

```
HGET user:1001 username
```

This command would return the value "john_doe" associated with the username field in the hash.

3. HGETALL key:

- Retrieves all fields and their corresponding values in a hash.

```
HGETALL user:1001
```

This command returns a list of field-value pairs for the hash associated with the key "user:1001".

These hash operations are particularly useful for representing and managing structured data within Redis. Hashes can be employed to store and retrieve information related to entities, making them a valuable choice for scenarios where data needs to be organized into fields and subfields.

4. List Operations:

Certainly, let's explore these list operations in Redis:

1. LPUSH key value1 value2 ...:

- Inserts one or more values at the beginning of a list.

```
LPUSH mylist "apple" "banana" "cherry"
```

This command inserts the values "apple," "banana," and "cherry" at the beginning of the list associated with the key "mylist."

2. RPUSH key value1 value2 ...:

- Inserts one or more values at the end of a list.

```
RPUSH mylist "date" "fig"
```

This command appends the values "date" and "fig" to the end of the list associated with the key "mylist."

3. LRANGE key start stop:

- Retrieves a range of elements from a list. The range is specified by the start and stop indices.

```
LRANGE mylist 0 -1
```

This command retrieves all elements from the list associated with the key "mylist."

These list operations in Redis are beneficial for scenarios where data needs to be stored in an ordered sequence. Lists are commonly used for implementing queues, managing job queues, and storing logs where the order of events is crucial. The ability to insert and retrieve elements from both ends of the list makes Redis lists versatile for various use cases.

5. Set Operations:

1. SADD key member1 member2 ...:

- Adds one or more members to a set. If the set does not exist, a new set is created.

```
SADD myset "apple" "banana" "orange"
```

This command adds the members "apple," "banana," and "orange" to the set associated with the key "myset."

2. SMEMBERS key:

- Retrieves all members of a set.

```
SMEMBERS myset
```

This command returns all members of the set associated with the key "myset."

3. SINTER key1 key2 ...:

- Retrieves the intersection of multiple sets.

```
SADD set1 "apple" "banana" "orange"  
SADD set2 "banana" "cherry" "orange"  
SINTER set1 set2
```

In this example, the last command returns the members "banana" and "orange," which are common to both sets.

Redis sets are useful for scenarios where you need to represent a collection of unique elements. Set operations like union, intersection, and difference provide powerful tools for analyzing and manipulating sets in various applications.

6. Sorted Set Operations:

Certainly, let's explore these sorted set operations in Redis:

1. ZADD key score1 member1 score2 member2 ...:

- Adds members with associated scores to a sorted set. If a member already exists, its score is updated.

```
ZADD highscores 100 "PlayerA" 150 "PlayerB" 80 "PlayerC"
```

This command adds players to the "highscores" sorted set with corresponding scores.

2.ZRANGE key start stop:

- Retrieves a range of elements from a sorted set based on their scores.

```
ZRANGE highscores 0 -1 WITHSCORES
```

This command retrieves all members and their scores from the "highscores" sorted set.

Sorted sets in Redis are particularly useful when you need to maintain a ranking or leaderboard based on scores. These operations allow you to efficiently retrieve and update elements based on their scores, making sorted sets suitable for scenarios such as gaming leaderboards or any application involving ordered data.

7. Bitwise Operations (Bitmaps):

Certainly, let's explore these bitwise operations with bitmaps in Redis:

1. SETBIT key offset value:

- Sets or clears the bit at a specified offset in the string value stored at a key. The value can be 0 or 1.

```
SETBIT mybitmap 5 1
```

This command sets the bit at offset 5 in the bitmap associated with the key "mybitmap" to 1.

2. GETBIT key offset:

- Retrieves the bit value at a specified offset in the string value stored at a key.

```
GETBIT mybitmap 5
```

This command returns the bit value at offset 5 in the bitmap associated with the key "mybitmap."

Bitwise operations in Redis are often used for scenarios where you need to efficiently represent and manipulate sets of binary flags or indicators. Bitmaps can be employed for tasks such as tracking user preferences, monitoring system states, or implementing compact data structures that rely on binary representation.

8. Pub/Sub (Publish/Subscribe):

Certainly, let's explore these Pub/Sub operations in Redis:

1. PUBLISH channel message:

- Publishes a message to a specified channel. Any clients subscribed to that channel will receive the message.

```
PUBLISH news_channel "Breaking News: Redis 7.0 Released!"
```

This command publishes the message "Breaking News: Redis 7.0 Released!" to the "news_channel."

2. SUBSCRIBE channel:

- Subscribes the client to the specified channel. The client will receive messages published to that channel.

```
SUBSCRIBE news_channel
```

This command subscribes the client to the "news_channel," allowing it to receive messages published to that channel.

Redis Pub/Sub is commonly used for building real-time messaging systems, chat applications, and event notification systems. Publishers broadcast messages to specific channels, and subscribers receive messages from channels they are interested in, facilitating efficient communication between different parts of an application.

9. Transactions:

Certainly, let's explore these transaction-related commands in Redis:

1. MULTI:

- Marks the start of a transaction block. Subsequent commands are queued up for execution as part of the transaction.

```
MULTI
SET key1 "value1"
SET key2 "value2"
```

This command initiates a transaction block, and the subsequent SET commands will be executed atomically.

2. EXEC:

- Executes all previously queued commands in a transaction. If the transaction is successful, the changes are committed; otherwise, they are rolled back.

```
EXEC
```

This command executes the queued commands within the transaction block. If successful, the changes are committed.

3. DISCARD:

- Discards all commands in the current transaction, effectively canceling the transaction.

```
DISCARD
```

This command discards all commands queued in the current transaction block, undoing any changes made so far.

These transaction commands in Redis provide a way to group multiple commands into a single atomic operation. This ensures that either all commands in the transaction are executed, or none of them are, maintaining data consistency. Transactions are particularly useful in scenarios where it's crucial to perform a series of operations atomically.

10. Key Expiry:

Certainly, let's explore these commands related to key expiration in Redis:

1. EXPIRE key seconds:

- Sets the time to live (TTL) of a key, indicating how long the key should be retained before it is automatically deleted.

```
EXPIRE mykey 60
```

This command sets the key "mykey" to expire in 60 seconds. After this time elapses, the key will be automatically removed from the database.

2. TTL key:

- Retrieves the remaining time to live of a key in seconds.

```
TTL mykey
```

This command returns the remaining time to live of the key "mykey." If the key is persistent (does not have a set expiration), TTL returns -1. If the key does not exist or has expired, TTL returns -2.

These commands are valuable for managing the lifecycle of keys in Redis. Setting an expiration time is useful for scenarios such as caching, where you want to automatically refresh data after a certain period. The TTL command provides a convenient way to check the remaining time before a key expires.

These are just a few examples, and Redis provides a comprehensive set of commands to perform a wide range of operations on different data structures. Understanding these basic commands is essential for effectively working with Redis in various applications.

Advanced features of Redis

Redis offers several advanced features that contribute to its versatility and wide range of use cases. Here are some notable advanced features of Redis:

1. Persistence:

- Redis supports different mechanisms for persistence, allowing data to be saved to disk. This ensures that data is not lost when Redis is restarted. Options include RDB snapshots and AOF (Append-Only File) logs.

2. Replication:

- Redis supports master-slave replication, allowing data to be asynchronously replicated from one Redis server (master) to one or more Redis servers (slaves). This provides data redundancy, high availability, and scalability.

3. Partitioning:

- Redis allows horizontal scaling through partitioning, where data is distributed across multiple Redis instances. This helps handle large datasets and improves performance by leveraging the capabilities of multiple servers.

4. Lua Scripting:

- Redis supports Lua scripting, allowing users to write custom scripts that can be executed on the server. This feature enables complex operations and transactions to be executed atomically on the server side.

5. Transactions:

- Redis supports transactions using the MULTI, EXEC, and DISCARD commands. Multiple commands can be grouped together in a transaction, ensuring they are executed atomically. This helps maintain data consistency.

6. Keyspace Notifications:

- Redis provides the ability to subscribe to notifications for specific keyspace events. Clients can be notified when certain events, such as key expirations or modifications, occur in the Redis dataset.

7. Bitmap Operations:

- Redis supports advanced bitmap operations, allowing efficient manipulation of sets of bits. This is useful for scenarios such as tracking user behavior, handling flags, and implementing efficient data structures.

8. HyperLogLogs:

- HyperLogLogs provide approximate cardinality estimation for sets of unique elements with minimal memory usage. This feature is useful for counting distinct items in large datasets with reduced memory requirements.

9. Geospatial Indexing:

- Redis supports geospatial data types and provides commands for storing, querying, and manipulating data based on geographic location. This is valuable for location-based services and applications.

10. Cluster Mode:

- Redis Cluster is a distributed implementation of Redis that provides high availability and horizontal scaling. It divides the dataset into multiple partitions across nodes, ensuring data is distributed and replicated for fault tolerance.

11. Security Features:

- Redis supports authentication through passwords and provides access control mechanisms to restrict client access based on IP addresses. This helps enhance the security of Redis deployments.

These advanced features contribute to Redis's appeal in various use cases, including real-time analytics, caching, messaging systems, and more. Understanding and leveraging these features allows developers and system administrators to optimize Redis for specific requirements and achieve better performance and reliability.

Using Redis in real-world scenarios

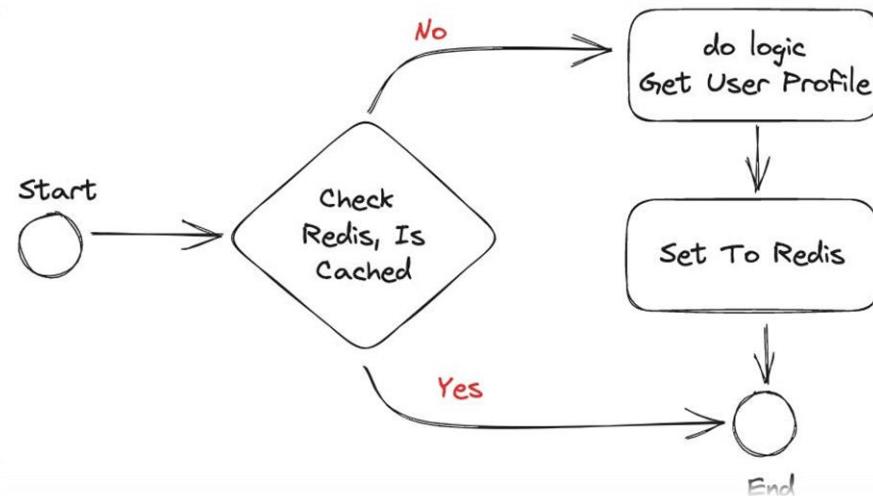
Example-1: User Profile Cache (HTTP API Cache)

Redis Commands: GET, SET, EXPIRE

HTTP API cache refers to the caching of HTTP responses from an API (Application Programming Interface). It involves storing the response of an API request in a cache, which can be subsequently used to serve future requests without the need to re-fetch the data from the API server.

One of the common use cases for API caching is storing user profiles. In an authentication system, usually only the user ID or email (something unique) is stored in the authentication

token. However, the frontend often requires the user's name and profile picture on every page. Retrieving the user profile repeatedly can become redundant, so api caching with Redis can greatly enhance scalability.



```

function GetUserProfile(user_id) {
    var cache_response = redis.Do(`GET user:{user_id}`)
    if(cache_response) {
        return json.Unmarshal(cache_response)
    }

    var response = getProfile(user_id)
    var marshalled_resp = json.Marshal(response)
    redis.Do(`SET user:{user_id} {marshalled_resp} EX 600`)
    return response
}
  
```

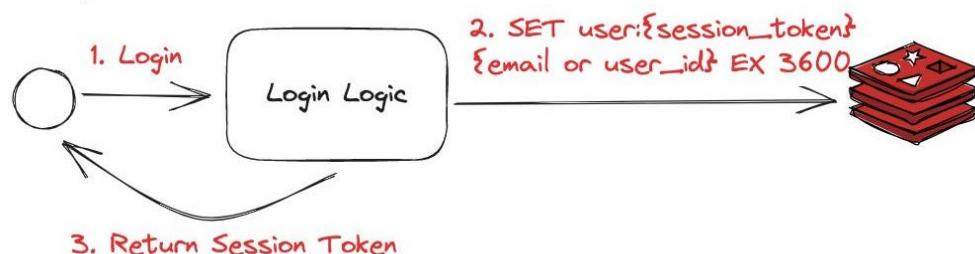
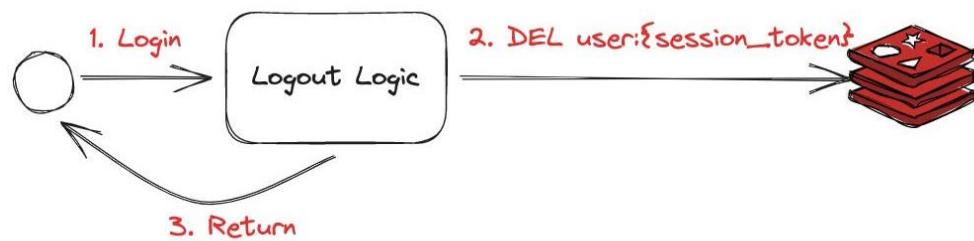
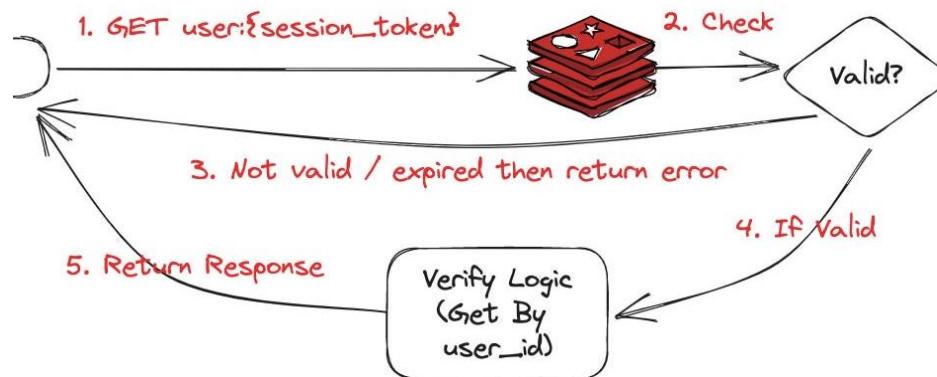
HTTP REST APIs typically return JSON responses. If we want to utilize Redis to cache the responses, we can store JSON string as the cached value. For subsequent requests, we can simply unmarshalled the string and return the cached JSON object.

Example-2: Redis for Session Storage (Login & Logout)

Related Commands: GET, SET, EXPIRE, DEL

Session-based authentication is a widely used approach for user authentication in web applications. It involves generating a session token after login, which is then used to keep track of the authenticated user.

Similar to the issue with user profiles, the session token needs to be checked every time a user performs an action that requires authentication. Consequently, querying the database can easily become a bottleneck in high-traffic use cases. Therefore, using Redis for session storage is considered one of the best solutions.

Login**Logout****Verify Token**

```

function Login(email, password) {
    var session_token = loginLogic()
    redis.Do(`SET user:session:{session_token} {email} EX 3600`)
    return session_token
}

function Logout(session_token) {
    redis.Do(`DEL user:session:{session_token}`)
}

function VerifyToken(session_token) {
    var email = redis.Do(`GET user:session:{session_token}`)
    if(!email) {
        return null, errors('token is expired or not exists')
    }
    var user_detail = getUserDetail(email)
    return user_detail, null
}
  
```

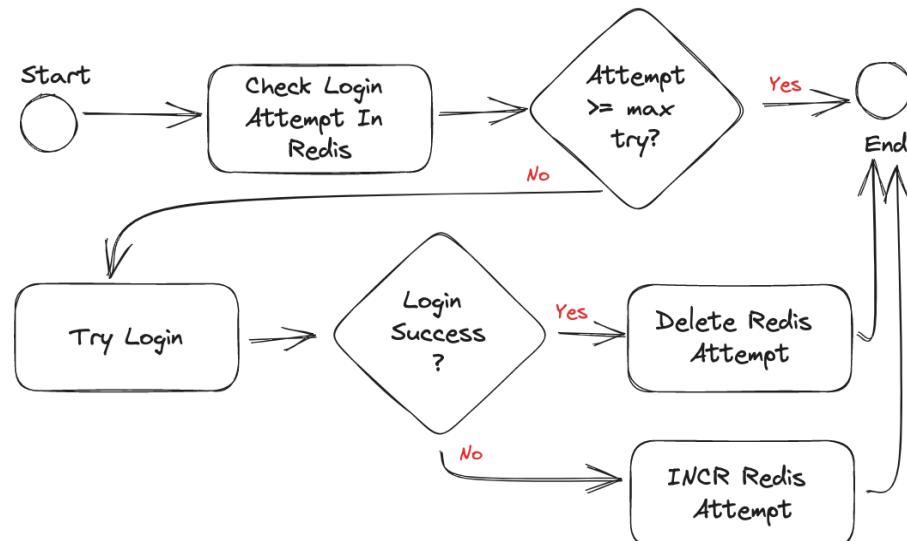
Example-3: Log In Throttling

Related Command: INCRBY, EXPIRE, TTL

Authentication has many interesting use cases, one of which is preventing users from hacking other users through brute force attacks. One of the easiest ways to achieve this is by implementing throttling. Throttling refers to the practice of intentionally limiting the rate or speed of a process or service. You might have seen messages such:

“There have been too many login failures. Try again in x seconds”

This is an implementation of throttling for login attempts to prevent brute force attacks. In a simple manner, throttling involves storing the number of login attempts made by a user. However, storing this data in databases like MySQL or PostgreSQL may seem overly engineering. Additionally, databases like MySQL and PostgreSQL do not have built-in time-based expiration, unlike Redis. Using Redis to implement throttling is a straightforward solution. Please refer to the system design below:



```

function Login(context, username, password){
    var guest_session = context.GuestSession
    var max_attempt = 5
    var current_attempt = redis.Do('GET login_attempt:{guest_session}')
    if(current_attempt >= max_attempt){
        var expire_lock = redis.Do('TTL login_attempt:{guest_session}')
        return null, errors(`There have been too many login failures. Try again in {expire_lock} seconds`)
    }

    var is_login_success, auth_token = login(username, password)
    if(!is_login_success) {
        redis.Do('INCR login_attempt:{guest_session}')
    }
}
  
```

```

redis.Do(`EXPIRE login_attempt:{guest_session} 300`)
return null, errors("user not found / invalid password")
}
redis.Do(`DEL login_attempt:{guest_session}`)
return auth_token, null
}

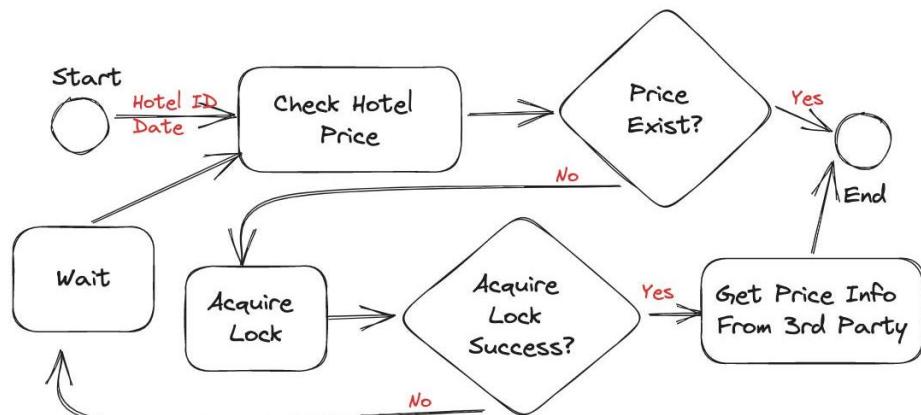
```

Example-4: Dynamic Pricing in Hotel Industry (Redis Distributed Locking)

Related Command: GET, SET NX EX

When working with horizontal scaling or even microservices, it becomes necessary to implement distributed locking at the application level to address issues such as race conditions or thundering herd problems. Redis can be leveraged for distributed locking, offering several advantages including high scalability, fault tolerance, and strong consistency guarantees.

Consider a scenario where you are developing a hotel booking platform that fetches dynamic prices from a third-party API. In a situation where multiple users are requesting hotel information for the same date, you may want to avoid overwhelming the third-party API. Here, you can utilize Redis distributed locking to resolve this issue effectively.



```

function CheckHotelPrice(hotel_id, book_date) {
    var price_detail = null
    while(true) {
        price_detail = redis.Do(`GET hotel_price:{hotel_id}:{book_date}`)
        if(price_detail != null){
            break
        }

        var current_machine_name = os.GetHostname()
        var acquire_lock = redis.Do(`SET hotel_price:{hotel_id}:{book_date}:lock
{current_machine_name} NX EX 5`)

        if(!acquire_lock){
            sleep(250) // waiting 250 ms
            continue
        }
    }
}

```

```

        price_detail = getHotelPriceFromThirdParty(hotel_id, booking_at)
        redis.Do(`SET hotel_price:{hotel_id}:{book_date} {price_detail} EX 600`)
        break
    }

    return price_detail
}

```

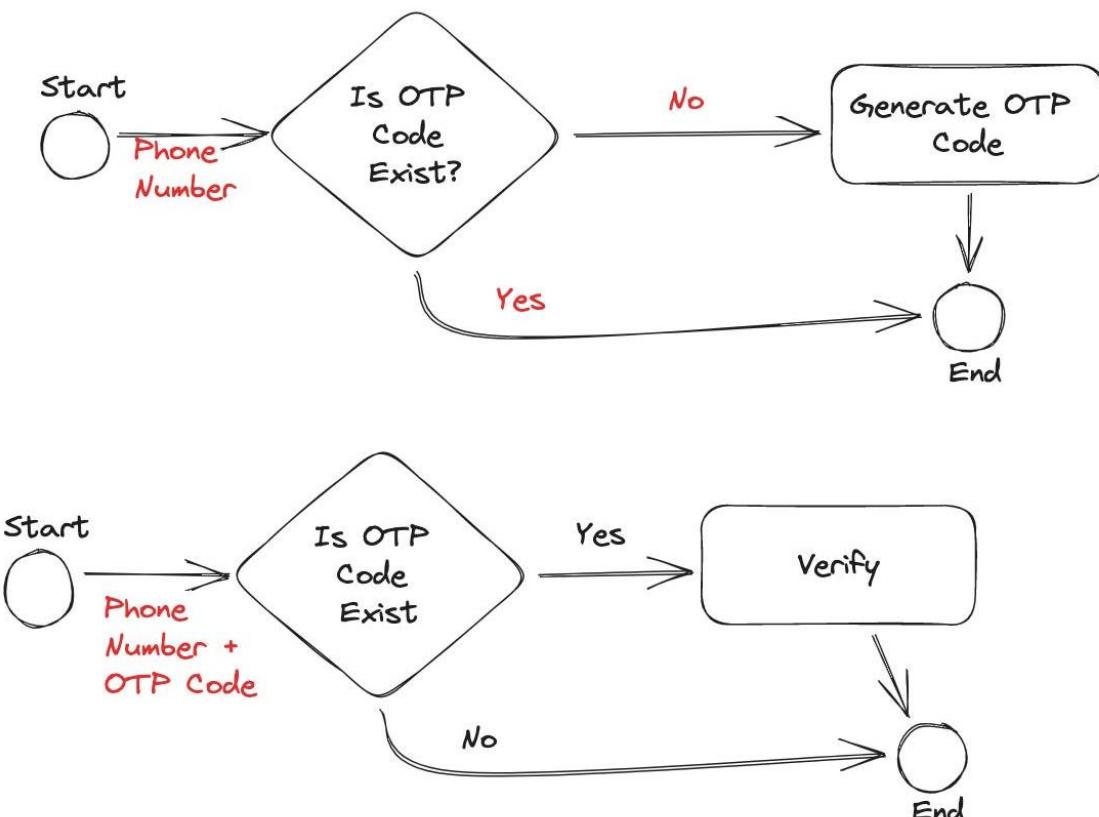
Example-5: OTP (One Time Password) using Redis

Related Command: SET NX EX, GET, DEL

"OTP" stands for "One-Time Password." It is a security measure used to authenticate users and verify their identity during online transactions or account logins. A one-time password is a unique and temporary code that is typically valid for a short period of time, usually for a single login session or transaction. Since Redis has time-based expiration using the EXPIRE command, it is well-suited for building an OTP (One-Time Password) system.

The OTP system consists of two functions:

- Generate OTP
- Verify OTP



Redis can be leveraged to build an OTP system. Below is pseudocode that demonstrates how Redis can be used to develop an OTP system:

```

function GenerateOTP(phone_number) {
    var otp_code = generate_otp_code(6)
    var is_success = redis.Do(`SET otp:{phone_number} otp_code NX EX 600`)
    if(!is_success){
        var ttl = redis.Do(`TTL otp:{phone_number}`)
        return errors("otp has been sent, try again in {ttl} minutes")
    }
    sendOtpViaSMS(phone_number, otp_code)
    return null
}

function VerifyOTP(phone_number, otp_code) {
    var stored_otp = redis.Do(`GET otp:{phone_number}`)
    if(!stored_otp){
        return errors(`Code has been expired`)
    } else if(stored_otp != otp_code){
        return errors(`Invalid OTP Code`)
    }

    redis.Do(`DEL otp:{phone_number}`)
    return null
}

```

Example-6: Waiting List in Help Desk System (Redis for Job Queue)

Related Commands: LPUSH, RPOPLPUSH, LREM, LPOS

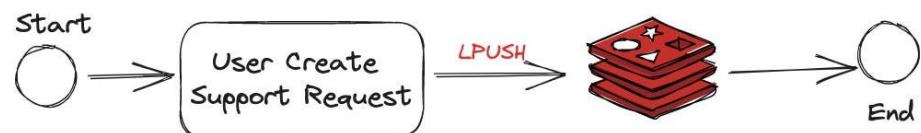
A help desk is a system designed to manage and streamline customer support and issue resolution processes within an organization. When the number of support request much bigger than the number of customer support it will generate a problem.

Imagine you only have 3 customer support officer and at one time there are 100 request to have chat support, how to solve it?

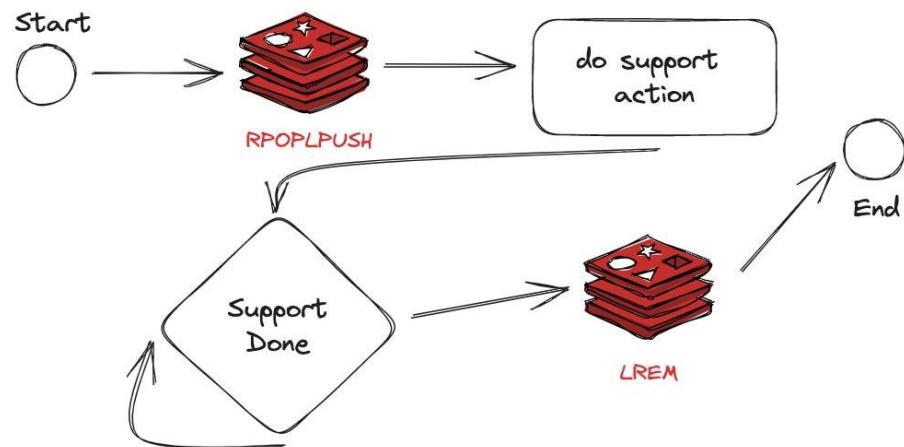
It's a common where help desk system has waiting list mechanism. In technical, waiting list is similar to JOB QUEUE in data structure. Redis has a powerfull command to solve job queue problems. In a help desk system, there are at least two functions:

- Create Request (by User)
- Handle Support Request (by Customer Support)

Create Support Request



Handle Support Request



```

function CreateRequest(payload) {
    var request_id = createSupportRequest(payload)
    redis.Do(`LPUSH helpdesk {request_id}`)
}

function GetQueuePos(request_id){
    var my_queue_number = redis.Do(`LPOS helpdesk {request_id} RANK -1`)
    // Notes: "RANK -1" means you reverse the sort
    return my_queue_number + 1
}

function HandleSupportRequest(cs_id){
    while (true) {
        var request_id = redis.Do(`RPOPLPUSH helpdesk helpdesk:{cs_id}`)
        if(request_id == null) {
            sleep(60)
            continue
        }
        var request_detail = getSupportRequest(request_id)
        connect(cs_id, request_detail.user_id)
        redis.Do(`LREM helpdesk:{customer_support_id} 1 {request_id}`)
    }
}
  
```

Resources/Equipment Required

1. Technical Setup:

- Ensure that students have access to individual computers or virtual machines with Redis installed, creating a conducive environment for hands-on practice.

- Redis is not officially supported on Windows. To install Redis on Windows, you'll first need to enable WSL2 (Windows Subsystem for Linux). WSL2 lets you run Linux binaries natively on Windows. For this method to work, you'll need to be running Windows 10 version 2004 and higher or Windows 11.

2. Educational Resources:

- Provide comprehensive documentation on Redis basics, key-value operations, and essential commands for students to reference during the practical session.

3. Training Environment:

- Set up a simulated environment with Redis configured to mimic real-world scenarios, allowing students to apply key-value operations and commands in a controlled setting.

4. Support Structure:

- Establish a support structure, including knowledgeable facilitators and assistance, to help students navigate technical issues and ensure a smooth learning experience during the Redis Basics practical.

I. Practical related Questions

Write Redis queries for following

1. Write a query to set the value of a key named "username" to "john_doe".
2. Write a query to retrieve the length of the string stored in the key "message".
3. Write a query to add the values "apple," "orange," and "banana" to a set with the key "fruits".
4. Write a query to retrieve all fields and values from a hash with the key "user:1001".

Write Redis queries for following

1. Using only Redis commands, demonstrate how you can create a new key named "counter" and set its value to 100, but only if the key doesn't already exist. If the key exists, increment its value by 10.
2. Write a Redis query to append the string ", Redis is powerful!" to the existing value of the key "description" only if the length of the current value is less than 50 characters.
3. Suppose you have a hash named "user:101" representing a user profile. Add a new field "age" with a value of 25, but only if the field doesn't exist. If the field exists, increment its value by 1.
4. Create a list named "logs" and insert the values "log_entry_1," "log_entry_2," and "log_entry_3" at the beginning of the list. After that, trim the list to keep only the first two elements.
5. Implement a scenario where you have two sets, "setA" and "setB." Write a query to add the elements present in both sets to a new set named "common_elements."
6. Assume you have a sorted set named "scores" with members "playerA," "playerB," and "playerC" having scores 100, 150, and 80, respectively. Write a query to update the score of "playerB" to 120.
7. Create a bitmap named "user_flags" with a length of 8 bits. Set the bits at positions 2 and 5 to 1, and then retrieve the value of the bitmap.
8. Create a key "temporary_data" with a value "sensitive_info" and set it to expire in 60 seconds. Write a query to retrieve the value of the key and check its time to live after 70 seconds.

