1

❖ Creating a Keyspace

➢ A keyspace in Cassandra is a namespace that defines data replication on nodes. A cluster contains one keyspace per node. Given below is the syntax for creating a keyspace using the statement CREATE KEYSPACE.

Syntax
CREATE KEYSPACE <identifier> WITH <properties>

CREATE KEYSPACE "KeySpace Name"
WITH replication = {'class': 'Strategy name', 'replication_factor' : 'No.Of replicas'};

➢ Example
- CREATE KEYSPACE vitb
  WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 2};

➢ We can enter into a keyspace with the following syntax
Use <keyspace name>
Ex:
Use vitb

❖ Altering a Keyspace

➢ ALTER KEYSPACE can be used to alter properties such as the number of replicas and the durable_writes of a KeySpace. Given below is the syntax of this command.

Syntax
ALTER KEYSPACE <identifier> WITH <properties>

ALTER KEYSPACE "KeySpace Name"
WITH replication = {'class': 'Strategy name', 'replication_factor' : 'No.Of replicas'};

➢ Example:
- ALTER KEYSPACE vitb
  WITH replication = {'class':'NetworkTopologyStrategy', 'replication_factor' : 3};

❖ Dropping a keyspace

➢ A KeySpace can be dropped using the command DROP KEYSPACE. Given below is the syntax for dropping a KeySpace.

Syntax

➢ DROP KEYSPACE <identifier>
DROP KEYSPACE "KeySpace name"

➢ Example
DROP KEYSPACE vitb;

Cassandra collections are used to handle tasks. You can store multiple elements in collection. There are three types of collection supported by Cassandra:

- Set
- List
- Map

Set Collection

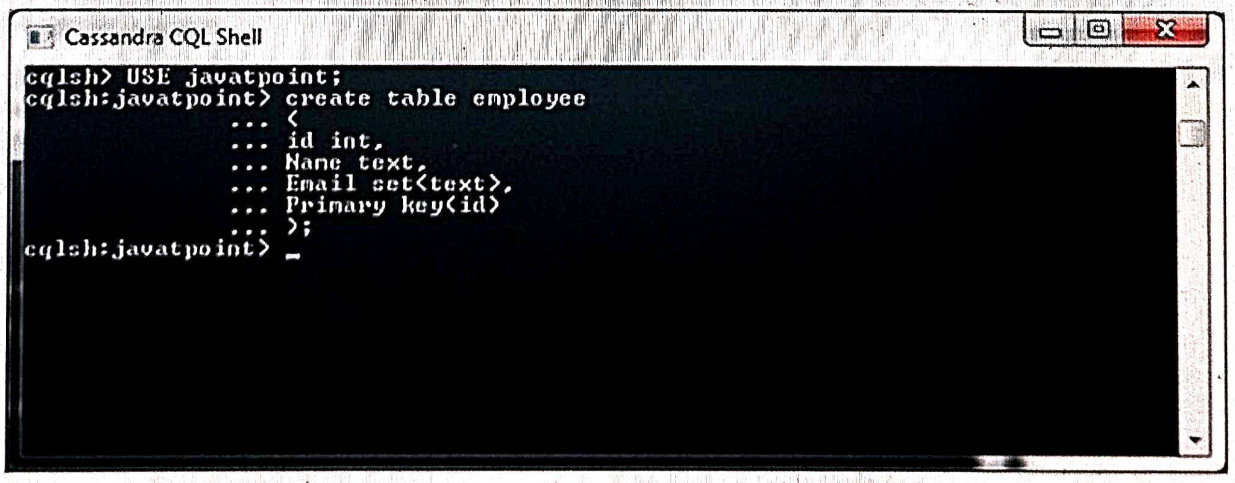
A set collection stores group of elements that returns sorted elements when querying.

Syntax:

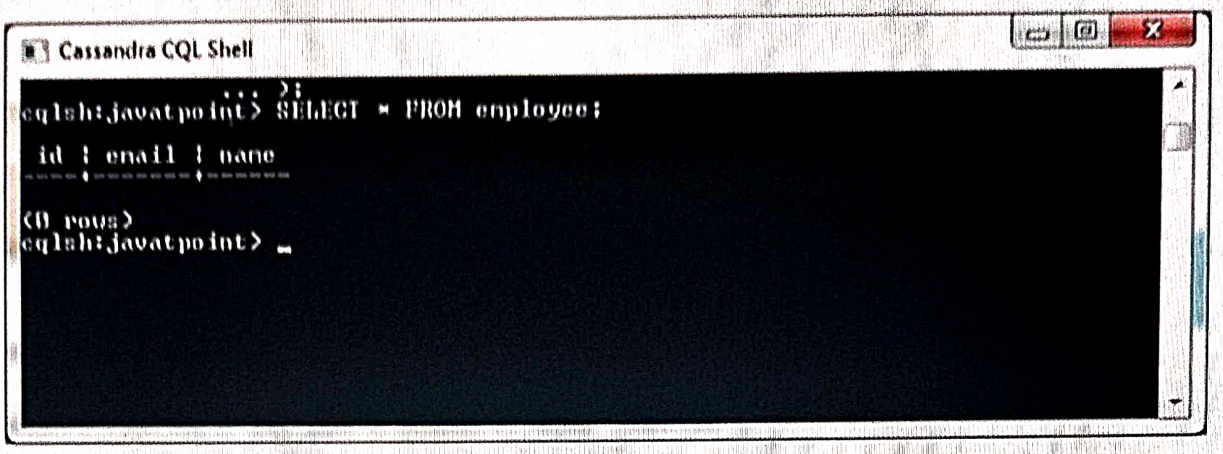
```
1. Create table table_name
2. (
3. id int,
4. Name text,
5. Email set<text>,
6. Primary key(id)
7. );
```

Example:

Let's take an example to demonstrate set collection. Create a table "employee" having the three columns id, name and email.



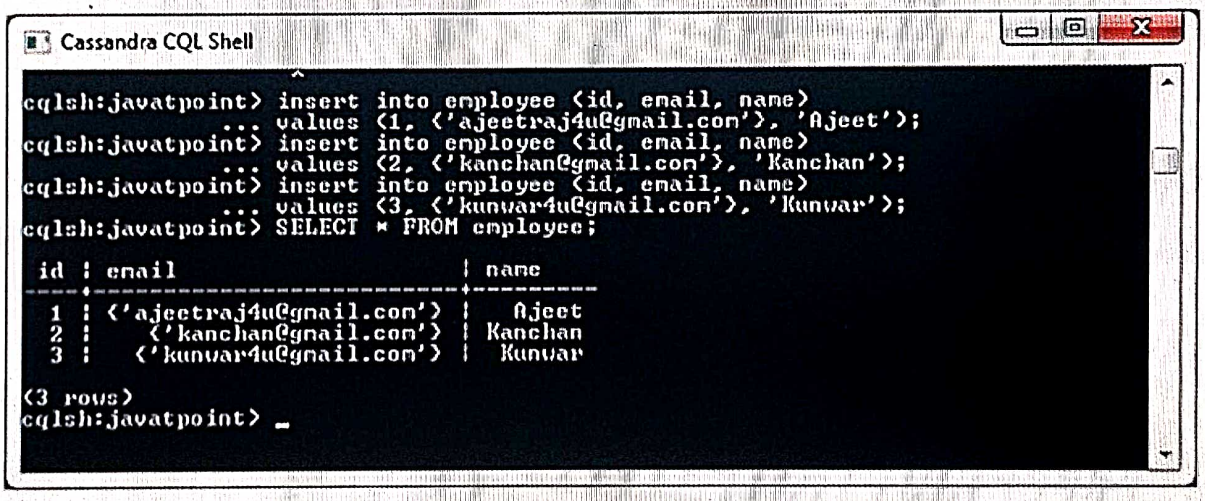
The table is created like this:



Insert values in the table:

1. **INSERT INTO** employee (id, email, **name**)
2. **VALUES**(1, {'ajeetraj4u@gmail.com'}, 'Ajeet');
3. **INSERT INTO** employee (id, email, **name**)
4. **VALUES**(2,{'kanchan@gmail.com'}, 'Kanchan');
5. **INSERT INTO** employee (id, email, **name**)
6. **VALUES**(3, {'kunwar4u@gmail.com'}, 'Kunwar');

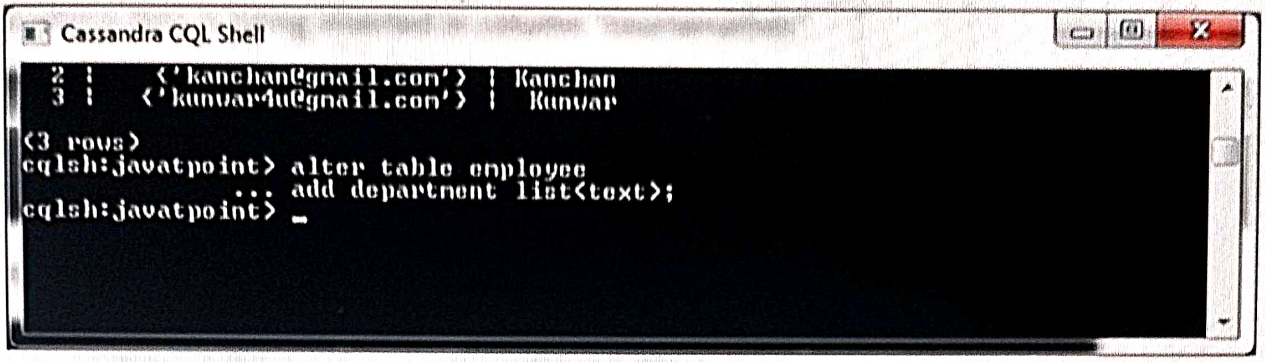
Output:



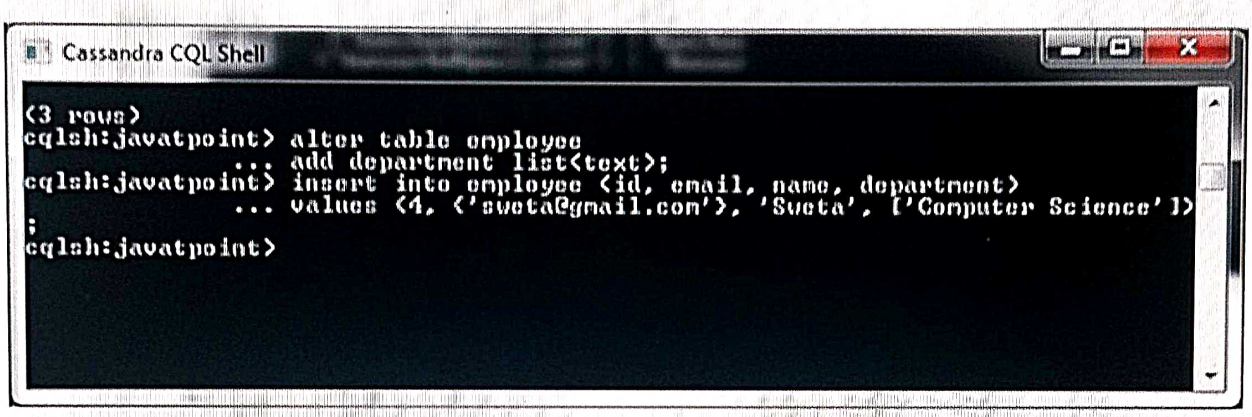
List Collection

The list collection is used when the order of elements matters.

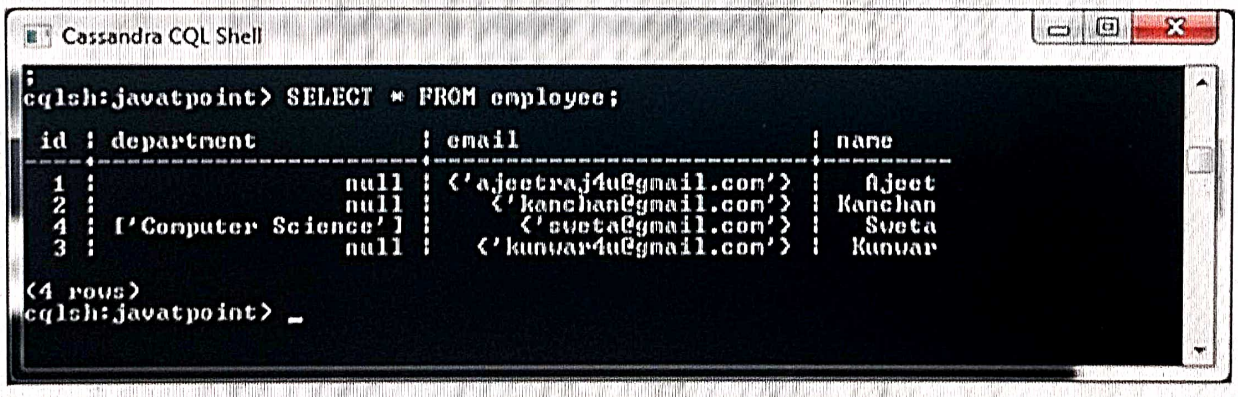
Let's take the above example of "employee" table and a new column name "department" in the table employee.



Now the new column is added. Insert some value in the new column "department".



Output:

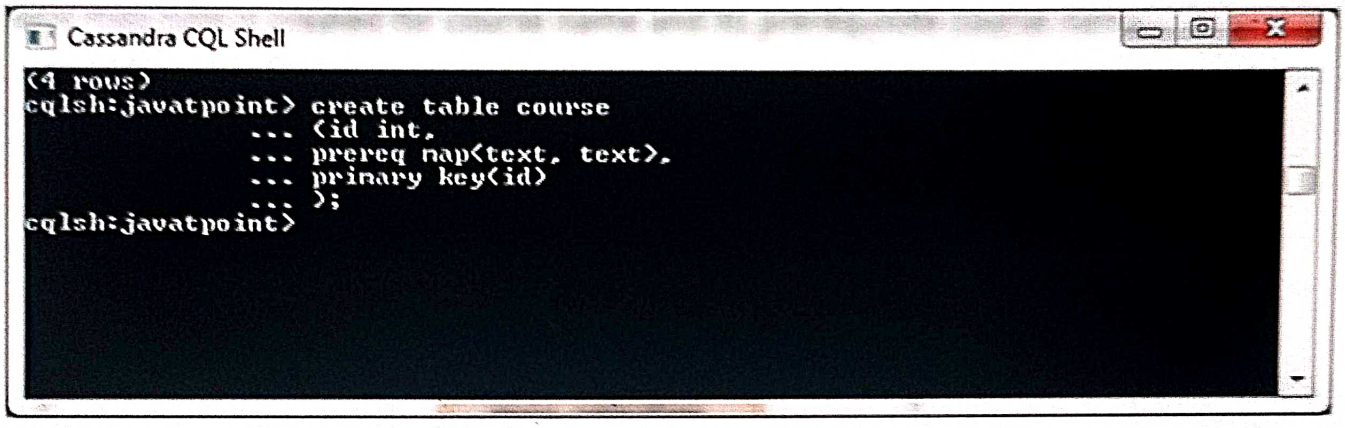


Map Collection

The map collection is used to store key value pairs. It maps one thing to another. For example, if you want to save course name with its prerequisite course name, you can use map collection.

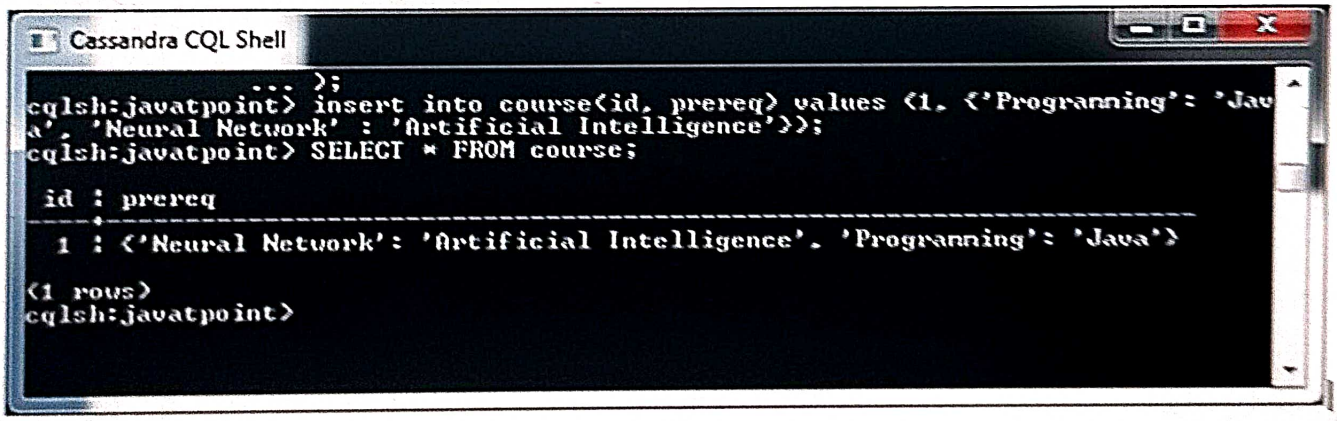
See this example:

Create a table named "course".



Now table is created. Insert some data in map collection type.

Output:



3a

❖ What Is a Document Database?

➢ Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable.

❖ Document Database using MongoDB

➢ MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++.

➢ Collection: A collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

➢ Document: A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

➢ Sample Document: The following example shows the document structure of a blog site, which is simply a comma-separated key-value pair.

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no SQL database',
  by: 'tutorials point',
  URL: 'http://www.tutorialspoint.com',
  tags: ['MongoDB, 'database', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user:'user1',
      message: 'My first comment',
      dateCreated: new Date(2011,1,20,2,15),
      like 0
    },
    {
      user:'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like 5
    }
          ]
}
```

➢ _id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, the next 3 bytes for the machine id, the next 2 bytes for the process id of the MongoDB server, and the remaining 3 bytes are simple incremental VALUE.

Features:

- **Document Type Model:** As we all know data is stored in documents rather than tables or graphs, so it becomes easy to map things in many programming languages.
- **Flexible Schema:** Overall schema is very much flexible to support this statement one must know that not all documents in a collection need to have the same fields.
- **Distributed and Resilient:** Document data models are very much dispersed which is the reason behind horizontal scaling and distribution of data.
- **Manageable Query Language:** These data models are the ones in which query language allows the developers to perform CRUD (Create Read Update Destroy) operations on the data model.

❖ MongoDB Data Modelling

3b

- The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns.
- Flexible Schema
  - Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections, by default, do not require their documents to have the same schema. That is: the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
  - To change the structure of the documents in a collection, such as adding new fields, removing existing fields, or changing the field values to a new type, update the documents to the new structure.

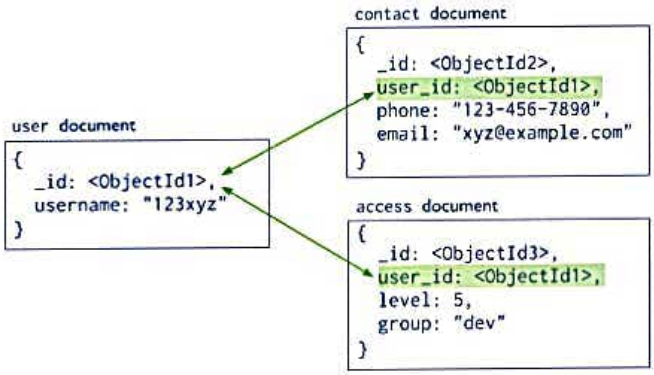
- Document Structure
  - The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. MongoDB allows related data to be embedded within a single document.
- Embedded Data
  - Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These denormalized data models allow applications to retrieve and manipulate related data in a single database operation.

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

Embedded sub-document

- References
  - References store the relationships between data by including links or references from one document to another. Applications can resolve these references to access the related data. Broadly, these are normalized data models.



- Single Document Atomicity
  - In MongoDB, a write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents within a single document.
  - A denormalized data model with embedded data combines all related data in a single document instead of normalizing across multiple documents and collections. This data model facilitates atomic operations.
- Multi-Document Transactions
  - When a single write operation (e.g. db.collection.updateMany()) modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic.
  - When performing multi-document write operations, whether through a single write operation or multiple write operations, other operations may interleave.

❖ MongoDB CRUD Operations

4a

- ➢ CRUD operations create, read, update, and delete documents.
- ➢ Create Operations
  - Create or insert operations and add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.
  - MongoDB provides the following methods to insert documents into a collection:
    - db.collection.insertOne() New in version 3.2
    - db.collection.insertMany() New in version 3.2
  - In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

```
db.users.insertOne(          <-- collection
   {
      name: "sue",           <-- field: value
      age: 26,               <-- field: value      } document
      status: "pending"      <-- field: value
   }
)
```

- ➢ Read Operations
  - Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:
    - db.collection.find()
  - You can specify query filters or criteria that identify the documents to return.

```
db.users.find(                          <-- collection
   { age: { $gt: 18 } },                <-- query criteria
   { name: 1, address: 1 }              <-- projection
).limit(5)                              <-- cursor modifier
```

- ➢ Update Operations
  - Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:
    - db.collection.updateOne() New in version 3.2
    - db.collection.updateMany() New in version 3.2
    - db.collection.replaceOne() New in version 3.2
  - In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
  - You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

```
db.users.updateMany(                          <-- collection
   { age: { $lt: 18 } },                      <-- update filter
   { $set: { status: "reject" } }             <-- update action
)
```

- ➢ Delete Operations
  - Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:
    - db.collection.deleteOne() New in version 3.2
    - db.collection.deleteMany() New in version 3.2
  - In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
  - You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.

```
db.users.deleteMany(                <-- collection
   { status: "reject" }             <-- delete filter
)
```

4b

❖ Suitable Usecases

- ➢ Event Logging
  - Applications have different event logging needs; within the enterprise, there are many different applications that want to log events. Document databases can store all these different types of events and can act as a central data store for event storage. This is especially true when the type of data being captured by the events keeps changing. Events can be sharded by the name of the application where the event originated or by the type of event such as order_processed or customer_logged.
- ➢ Content Management Systems, Blogging Platforms
  - Since document databases have no predefined schemas and usually understand JSON documents, they work well in content management systems or applications for publishing websites, and managing user comments, user registrations, profiles, and web-facing documents.
- ➢ Web Analytics or Real-Time Analytics
  - Document databases can store data for real-time analytics; since parts of the document can be updated, it's very easy to store page views or unique visitors, and new metrics can be easily added without schema changes.
- ➢ E-Commerce Applications
  - E-commerce applications often need to have a flexible schema for products and orders, as well as the ability to evolve their data models without expensive database refactoring or data migration

❖ When Not to Use

- ➢ Complex Transactions Spanning Different Operations
  - If you need to have atomic cross-document operations, then document databases may not be for you. However, there are some document databases that do support these kinds of operations, such as RavenDB.
- ➢ Queries against Varying Aggregate Structure
  - Flexible schema means that the database does not enforce any restrictions on the schema. Data is saved in the form of application entities. If you need to query these entities ad hoc, your queries will be changing (in RDBMS terms, this would mean that as you join criteria between tables, the tables to join keep changing). Since the data is saved as an aggregate, if the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity—basically, you need to normalize the data. In this scenario, document databases may not work.

Neo4j uses a *property graph* database model. A graph data structure consists of **nodes** (discrete objects) that can be connected by **relationships**. Below is the image of a graph with three nodes (the circles) and three relationships (the arrows).

5a

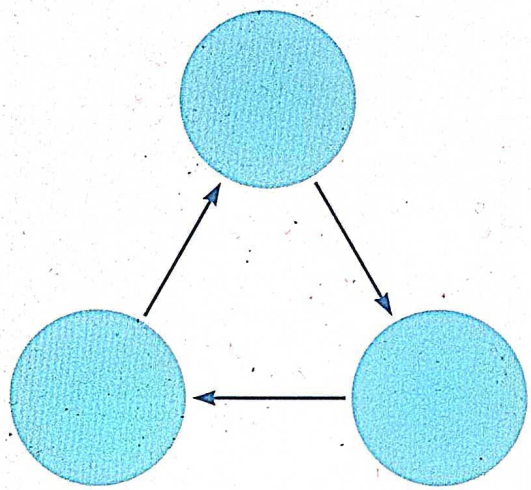


Figure 1. Concept of a graph structure

The Neo4j property graph database model consists of:

- **Nodes** describe entities (discrete objects) of a domain.
- **Nodes** can have zero or more **labels** to define (classify) what kind of nodes they are.
- **Relationships** describe a connection between a *source node* and a *target node*.
- **Relationships** always have a direction (one direction).
- **Relationships** must have a **type** (one type) to define (classify) what type of relationship they are.
- Nodes and relationships can have **properties** (key-value pairs), which further describe them.

Node

Nodes are used to represent *entities* (discrete objects) of a domain.

The simplest possible graph is a single node with no relationships. Consider the following graph, consisting of a single node.

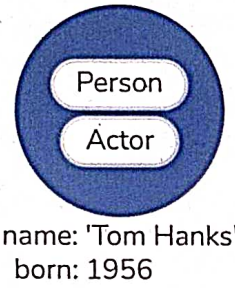


Figure 3. Node

The node labels are:

- `Person`
- `Actor`

The properties are:

- `name: Tom Hanks`
- `born: 1956`

The node can be created with Cypher using the query:

```
CREATE (:Person:Actor {name: 'Tom Hanks', born: 1956})
```

Node labels

Labels shape the domain by grouping (classifying) nodes into sets where all nodes with a certain label belong to the same set.

For example, all nodes representing users could be labeled with the label `User`. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

Since labels can be added and removed during runtime, they can also be used to mark temporary states for nodes. A `Suspended` label could be used to denote bank accounts that are suspended, and a `Seasonal` label can denote vegetables that are currently in season.

A node can have zero to many labels.

In the example graph, the node labels, `Person`, `Actor`, and `Movie`, are used to describe (classify) the nodes. More labels can be added to express different dimensions of the data.

The following graph shows the use of multiple labels.

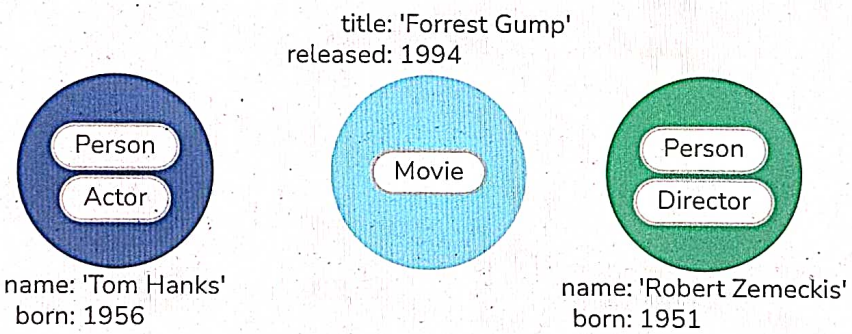


Figure 4. Multiple labels

Relationship

A relationship describes how a connection between a *source node* and a *target node* are related. It is possible for a node to have a relationship to itself.

A relationship:

- Connects a *source node* and a *target node*.
- Has a direction (one direction).
- Must have a **type** (one type) to define (classify) what type of relationship it is.
- Can have properties (key-value pairs), which further describe the relationship.

Relationships organize nodes into structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which may be combined into yet more complex, richly inter-connected structures.

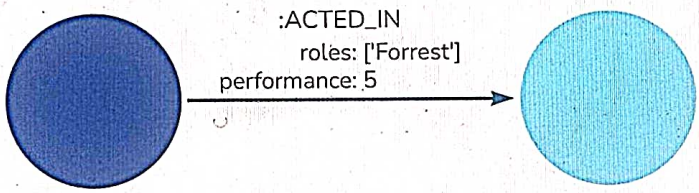


Figure 5. Relationship

The relationship type: `ACTED_IN`

The properties are:

- `roles: ['Forrest']`
- `performance: 5`

The `roles` property has an array value with a single item (`'Forrest'`) in it.

The relationship can be created with Cypher using the query:

```
CREATE ()-[:ACTED_IN {roles: ['Forrest'], performance: 5}]->()
```

> You must create or reference a *source node* and a *target node* to be able to create a relationship.

Relationships always have a direction. However, the direction can be disregarded where it is not useful. This means that there is no need to add duplicate relationships in the opposite direction unless it is needed to describe the data model properly.

A node can have relationships to itself. To express that `Tom Hanks` `KNOWS` himself would be expressed as:

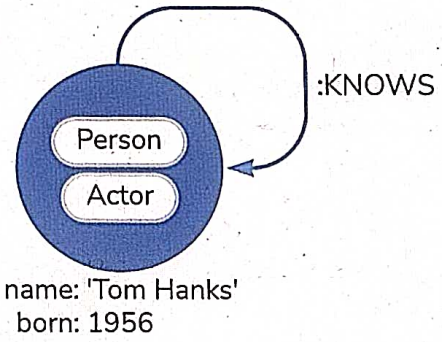


Figure 6. Relationship to a single node

Properties

Properties are key-value pairs that are used for storing data on nodes and relationships.

The value part of a property:

- Can hold different data types, such as `number`, `string`, or `boolean`.
- Can hold a homogeneous list (array) containing, for example, strings, numbers, or boolean values.

Example 1. Number

```
CREATE (:Example {a: 1, b: 3.14})
```

- The property `a` has the type `integer` with the value `1`.
- The property `b` has the type `float` with the value `3.14`.

Example 2. String and boolean

```
CREATE (:Example {c: 'This is an example string', d: true, e: false})
```

- The property `c` has the type `string` with the value `'This is an example string'`.
- The property `d` has the type `boolean` with the value `true`.
- The property `e` has the type `boolean` with the value `false`.

Example 3. Lists

```
CREATE (:Example {f: [1, 2, 3], g: [2.71, 3.14], h: ['abc', 'example'], i: [true, true, false]})
```

- The property `f` contains an array with the value `[1, 2, 3]`.
- The property `g` contains an array with the value `[2.71, 3.14]`.
- The property `h` contains an array with the value `['abc', 'example']`.
- The property `i` contains an array with the value `[true, true, false]`.

5b Using a graph database like Neo4j offers several advantages for representing and querying connected data:

1. **Natural Representation of Relationships**: Graph databases excel at representing relationships between entities, making them ideal for scenarios where relationships are as important as the entities themselves. This natural representation simplifies data modeling and querying, especially in domains such as social networks, recommendation systems, fraud detection, and network analysis.
2. **Flexible Schema**: Graph databases have a flexible schema that allows data models to evolve easily over time without the need for expensive schema migrations. Nodes and relationships can have properties dynamically added or modified, providing agility in adapting to changing data requirements.
3. **High Performance for Connected Data**: Graph databases are optimized for traversing relationships, enabling efficient querying of highly connected data. Unlike relational databases, which may require complex join operations for retrieving related data, graph databases use efficient index-free adjacency to quickly navigate through interconnected nodes and relationships.
4. **Expressive Query Language (Cypher)**: Neo4j provides Cypher, a powerful and intuitive query language designed specifically for graph databases. Cypher allows users to express complex graph patterns and relationships in a concise and readable syntax, making it easier to query and analyze connected data.
5. **Scalability**: Graph databases like Neo4j are designed to scale horizontally to handle large and growing datasets. They can distribute data across multiple nodes in a cluster, providing scalability without sacrificing performance. This scalability is particularly valuable for applications dealing with massive amounts of interconnected data.
6. **Real-Time Insights**: Graph databases enable real-time analysis and insights into connected data. By efficiently traversing relationships and patterns, queries can be executed quickly, allowing for near real-time decision-making and analytics in various domains, including recommendation engines, fraud detection, and network analysis.

6a

Neo4j uses Cypher Query Language (CQL) for interacting with the graph database. Cypher provides various clauses for reading and writing data. Let's discuss some commonly used clauses along with examples:

1. **MATCH Clause (Reading Data)**:

- The MATCH clause is used to retrieve data from the graph database based on specific patterns.
- It allows you to specify patterns of nodes and relationships to match in the graph.
- You can use MATCH to find nodes, relationships, or paths that meet certain criteria.

Example:

```cypher
MATCH (u:User)-[:FRIENDS_WITH]->(friend)
WHERE u.name = 'Alice'
RETURN friend.name
```

This query matches all nodes labeled as "User" that are connected by a "FRIENDS_WITH" relationship to the node with the name 'Alice'. It then returns the names of Alice's friends.

2. **CREATE Clause (Writing Data)**:

- The CREATE clause is used to create nodes and relationships in the graph database.
- It allows you to specify the structure of nodes and relationships to be created.

Example:

```cypher
CREATE (u:User {name: 'Bob', age: 30})
```

This query creates a new node labeled as "User" with the properties 'name' set to 'Bob' and 'age' set to 30.

3. **MERGE Clause (Reading or Writing Data)**:

- The MERGE clause is used to either match existing nodes and relationships or create them if they don't already exist.

- It's commonly used when you want to ensure that a specific pattern exists in the graph, either by finding it or creating it.

Example:

```cypher
MERGE (u:User {name: 'Charlie'})
ON CREATE SET u.created_at = timestamp()
```

This query searches for a node labeled as "User" with the name 'Charlie'. If it exists, it does nothing. If it doesn't exist, it creates a new node with the specified properties and sets the 'created_at' property to the current timestamp.

4. **SET Clause (Updating Data)**:

- The SET clause is used to update the properties of nodes and relationships in the graph.

Example:

```cypher
MATCH (u:User {name: 'Alice'})
SET u.age = 35
```

This query finds the node labeled as "User" with the name 'Alice' and updates her age property to 35.

These are just a few examples of commonly used Cypher clauses for reading and writing data in Neo4j. Cypher provides a rich set of functionalities for interacting with the graph database, allowing for complex queries and updates to be performed efficiently.

6b

❖ Index

- ➢ A database index is a redundant copy of some of the data in the database for the purpose of making searches of related data more efficient. This comes at the cost of additional storage space and slower writes, so deciding what to index and what not to index is an important and often non-trivial task.
- ➢ Once an index has been created, it will be managed and kept up to date by the DBMS. Neo4j will automatically pick up and start using the index once it has been created and brought online.
- ➢ There are multiple index types available:
    - Range index.
    - Lookup index.
    - Text index.
    - Point index.
    - Full-text index.

❖ Constraint

- ➢ In Neo4j, a constraint is used to place restrictions over the data that can be entered against a node or a relationship.
    - ➢ There are two types of constraints in Neo4j:
        - Uniqueness Constraint: It specifies that the property must contain a unique value. (For example: no two nodes with an player label can share a value for the Goals property.)
        - Property Existence Constraint: It makes ensure that a property exists for all nodes with a specific label or for all relationships with a specific type.
        - Create a Uniqueness Constraint
            - CREATE CONSTRAINT ON statement is used to create a uniqueness constraint in Neo4j.
            - CREATE CONSTRAINT ON (Kalam:president) ASSERT Kalam.Name IS UNIQUE
        - Property Existence Constraint
            - Property existence constraint is used to make ensure that all nodes with a certain label have a certain property.
    - ➢ Note: exists property constraint are only available in the Neo4j Enterprise Edition.