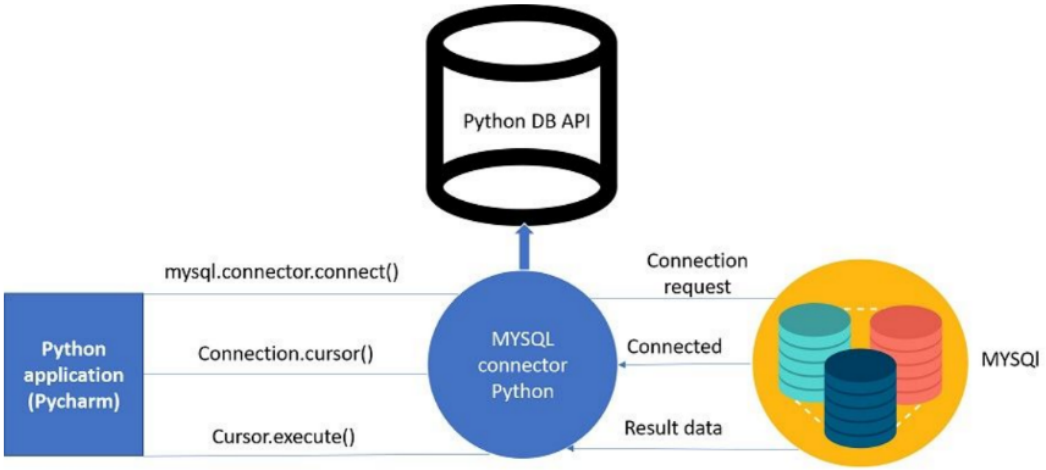
Unit -4

Python in MySQL

Python support relational Database and SQL Cursors

MySQL Connector
Mysql connector python is a module or library available in python to communicate with MySQL.




```
pip install mysql-connector-python
```

Establishing connection between MySQL and python

- Create a python file called main.py
- Type this code out in main.py file

```
import mysql.connector

mydb = mysql.connector.connect{host = "localhost" ,user ="Ary
c=mydb.cursor()
def fetch_student():
    query="SELECT * FROM table student"
    c.execute(query)
    result=c.fetchall()
def insert_student():
    c.execute("INSERT INTO student(student_id,student_name) V
    c.commit()
c.close()
mydb.close()
```

A cursor is used to execute SQL queries and fetch results from the database.

Close the cursor and the connection

💡 **connect - cursor - execute - fetch - close**

💡 **If insert/update is there even do commit**

Dynamic Operation

Select, insert, update and delete operations are done using `cursor.execute(query,values)`

`%s` is used to add the value and after adding `db.commit()` is done

```
def add_data(dealer_id, dealer_name, dealer_city):
    c.execute('INSERT INTO DEALER(dealer_id, dealer_name, dealer_city) VALUES (%s,%s,%s)',
              (dealer_id, dealer_name, dealer_city))
    mydb.commit()
```

```
cursor = db.connection.cursor()
cursor.execute("INSERT INTO table_ (employee_capacity, employee_booking) VALUES (%s, %s)", (employee_capacity, employee_booking))
db.connection.commit()
cursor.close()
```

`c.execute(“INSERT INTO <table_name> (columns in table name) VALUES (%s)” , (<column_name>))`

`db.commit()`

- `fetchone()`: Retrieves the next row of a query result set and returns it as a tuple or `None` if no more rows are available.
- `fetchall()`: Fetches all rows from the result set. The rows are returned as a list of tuples.

To insert multiple VALUES

use `cursor.executemany`

```
cursorObject = dataBase.cursor()

sql = "INSERT INTO STUDENT (NAME, BRANCH, ROLL, SECTION, AGE)\
VALUES (%s, %s, %s, %s, %s)"
val = [("Nikhil", "CSE", "98", "A", "18"),
       ("Nisha", "CSE", "99", "A", "18"),
       ("Rohan", "MAE", "43", "B", "20"),
       ("Amit", "ECE", "24", "A", "21"),
       ("Anil", "MAE", "45", "B", "20"),
       ("Megha", "ECE", "55", "A", "22"),
       ("Sita", "CSE", "95", "A", "19")]

cursorObject.executemany(sql, val)
dataBase.commit()
```

Why use XML instead of HTML

1. **XML for Structured and Machine-Readable Data:**
    - XML is preferred over HTML when the primary goal is to structure and exchange data in a standardized and machine-readable format.
    - It provides a consistent and standardized way to describe data, enhancing its readability and interpretability by machines.

2. **Custom Data Structures in XML:**
    - Unlike HTML, which is focused on web page presentation, XML allows users to define custom data structures using elements and attributes.
    - This flexibility enables the tailoring of data structures to specific needs, accommodating a wide range of data complexities.
3. **Semantic Clarity with XML:**
    - XML's use of semantic tag names contributes to the clarity and understanding of data.
    - Unlike HTML's predefined tags that mainly describe how content should be displayed, XML's focus on semantic meaning enhances the communication of data intent.
4. **Standardized Structure and Extensibility:**
    - XML's standardized structure and extensibility make it well-suited for scenarios where data needs to be reliably exchanged between diverse applications and systems.
    - The support for data validation through schemas adds an additional layer of reliability to data exchange processes.

Types of Data

Structured Data:

- **Definition:** Structured data is information that adheres to a predefined and organized format, typically stored in a relational database.
- **Example:** Records in a relational database table, like those in the COMPANY database, follow a consistent format.
- **Design Requirement:** Structured data requires careful database schema design to establish and define its structure systematically.
- Less Flexible

Semistructured Data:

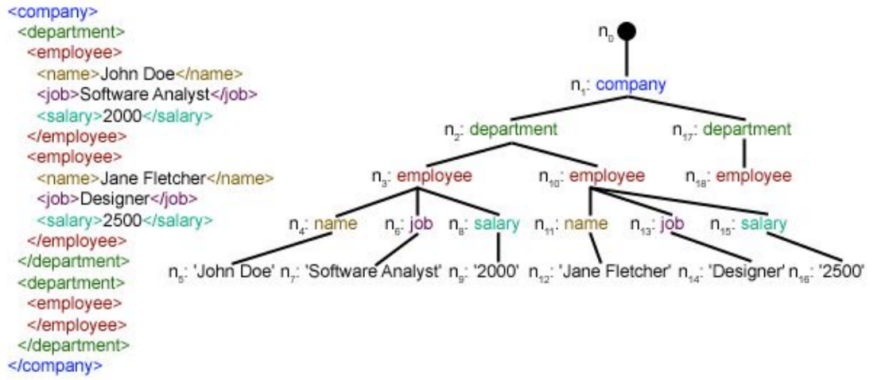
- **Definition:** Semistructured data lacks a rigid, predefined structure like structured data but still exhibits some level of organization or hierarchy.
- **Storage:** Unlike structured data stored in relational databases, semistructured data allows flexibility in terms of data formats and attributes.

- **Representation:** Semistructured data can be displayed as a directed graph, with labels on edges representing schema names and internal nodes representing objects or composite attributes.
- Schema handling is bad

Unstructured Data:

- **Definition:** Unstructured data refers to information lacking a predefined structure or format.
- **Characteristics:** Unstructured data, such as audio, text documents, or web pages, lacks a clear and organized schema, posing challenges for analysis and processing.
- **Example:** HTML text documents, lacking schema information, are challenging for computer programs to interpret automatically.

XML Hierarchical (Tree) Data Model



In the tree representation, internal nodes represent complex elements, whereas leaf nodes represent simple elements. That is why the XML model is called a **tree model** or a **hierarchical model.**

Types of XML Documents

●**Data-centric XML documents**: These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over the Web. These usually follow a predefined schema that defines the tag names.

●

Document-centric XML documents: These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.

●**Hybrid XML documents**: These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

SQL	NOSQL
Relational Database management system	Distributed Database management system
Vertically Scalable	Horizontally Scalable
Fixed or predifined Schema	Dynamic Schema
Not suitable for hierarchical data storage	Best suitable for hierarchical data storage
Can be used for complex queries	Not good for complex queries

No SQL Database

NoSQL Databases Overview:

1. **Definition and Types:** NoSQL databases, standing for "not only SQL," differ from relational tables. They come in various types based on their data model, including document, key-value, wide-column, and graph databases.
2. **Cost and Storage:** NoSQL databases emerged as storage costs decreased, allowing applications to handle vast amounts of data in different formats

(structured, semi-structured, polymorphic) without needing a predefined schema.

Characteristics of NoSQL Systems:

Related to Distributed Databases and Systems:

1. **Scalability:** NoSQL systems employ horizontal scalability to distribute data among new nodes without interrupting operations.
2. **Availability, Replication, and Eventual Consistency:** Focus on continuous availability through data replication, with eventual consistency.
3. **Replication Models:** Master-slave and master-master replication models are used to enhance availability and performance.
4. **Sharding:** Sharding of file records is often employed to distribute load and improve load balancing and data availability.
5. **High-Performance Data Access:** Techniques like hashing or range partitioning are used to achieve high-performance data access.

Related to Data Models and Query Languages:

1. **Not Requiring a Schema:** NoSQL systems often allow semi-structured, self-describing data, eliminating the need for a predefined schema.
2. **Less Powerful Query Languages:** NoSQL systems may not require powerful query languages like SQL, as they often provide a set of functions and operations as a programming API.
3. **Versioning:** Some NoSQL systems support storage of multiple versions of data items with timestamps.

Types of NoSQL Databases:

1. **Document-based NoSQL Systems:** Examples include CouchDB and MongoDB, storing data in document formats like JSON.
2. **Key-Value Stores:** Examples include DynamoDB and Redis, with a simple data model based on fast access by keys.
3. **Column-based/Wide Column NoSQL Systems:** Examples include Cassandra and HBase, partitioning tables by column into column families.
4. **Graph-based NoSQL Systems:** Examples include Neo4j, representing data as graphs and allowing traversal through path expressions.

Need for NoSQL

- A structured relational SQL system may not be appropriate because:
  (1) SQL systems offer too many services (**powerful query language, concurrency control, etc.**), which this application may not need
  (2) a structured data model such the traditional relational model may be too restrictive.

- Apache Hbase is an open source NOSQL
- Google developed a NOSQL system known as BigTable(column-based or wide column stores)
- Amazon developed a NOSQL system called DynamoDB
- Facebook developed a NOSQL system called Apache Cassandra
- graph-based NOSQL systems,or graph databases; these include Neo4J and GraphBase.
- Document Based - MongoDB
- Voldemort has been used by LinkedIn for data storage.
- Redis key-value cache and store: caches its data in main memory

TO REMEMBER

- **AMAZON - KEY - DYNAMO**
- **MONGO -DOCUMENT -JSON**
- **GRAPH - NEO4J**
- **FACEBOOK - CASSANDRA -COLUMN**

DATA MODELS AND QUERY LANGUAGE && DISTRUBUTED DATABASES

NoSQL Characteristics Related to Distributed Databases and Systems:

1. **Scalability:**

- **Description:** NoSQL systems emphasize horizontal scalability, allowing the addition of new nodes while the system is operational.
- **Requirement:** Techniques for distributing existing data among new nodes without interrupting system operation are essential.

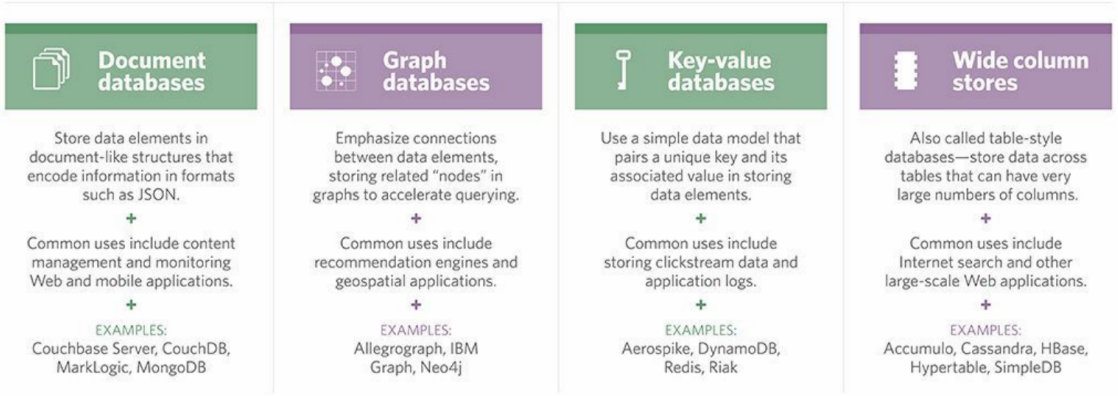
2. **Availability, Replication, and Eventual Consistency:**
   - **Description:** Continuous system availability is a key requirement for many NoSQL applications.
   - **Implementation:** Data replication across multiple nodes is employed transparently, ensuring that if one node fails, data remains accessible on other nodes.
   - **Benefits:** Replication enhances data availability and can improve read performance, as read requests can be serviced from any replicated data node.
3. **Replication Models:**
   - **Types:** Two major replication models in NoSQL systems are master-slave and master-master replication.
   - **Master-Slave:** Requires one master copy for write operations, propagated to slave copies with eventual consistency.
   - **Master-Master:** Allows reads and writes at any replica, but may not guarantee consistency across nodes storing different copies.
4. **Sharding:**
   - **Purpose:** Sharding of file records is often used in NoSQL systems to distribute the load of accessing file records across multiple nodes.
   - **Effectiveness:** Combining sharding and replicating shards improves load balancing and data availability.
5. **High-Performance Data Access:**
   - **Requirement:** NoSQL applications often require efficient retrieval of individual records or objects from large datasets.
   - **Techniques:**
     - **Hashing:** A hash function is applied to the key, determining the object's location based on the hash value.

    - **Range Partitioning:** Location is determined by a range of key values, with a partition holding objects within a specified key range

NoSQL Characteristics Related to Data Models and Query Languages:

1. **Not Requiring a Schema:**
    - **Flexibility:** Many NoSQL systems do not require a predefined schema.
    - **Implementation:** This flexibility is achieved by allowing semi-structured, self-describing data.
2. **Less Powerful Query Languages:**
    - **Query Needs:** NoSQL applications may not require powerful query languages like SQL.
    - **Typical Usage:** Search queries in NoSQL systems often locate single objects in a file based on their keys.
    - **API Approach:** NoSQL systems typically provide a programming API, where reading and writing data objects are accomplished by calling appropriate operations through the API.
3. **Versioning:**
    - **Feature:** Some NoSQL systems support storage of multiple versions of data items.
    - **Implementation:** Timestamps indicate when a data version was created.

NoSQL Data Models:



1. **Document-based NoSQL Systems:**
    - **Representation:** Data is stored in document form using well-known formats like JSON.
    - **Examples:** CouchDB, MongoDB.

2. **NoSQL Key-Value Stores:**
    - **Model:** Simple data model based on fast access by key to associated values.
    - **Values:** Values can be records, objects, documents, or more complex data structures.
    - **Examples:** DynamoDB, Redis.

3. **Column-based / Wide Column NoSQL Systems:**
    - **Model Partitioning:** Tables are partitioned by column into column families (vertical partitioning).
    - **Storage:** Each column family is stored in its own files.
    - **Examples:** Cassandra, HBase.

4. **Graph-based NoSQL Systems:**
    - **Representation:** Data is represented as graphs, and relationships between nodes are explored through path expressions.
    - **Example:** Neo4j.

Consistency - Availibility - Paritioning (CAP) Theorem

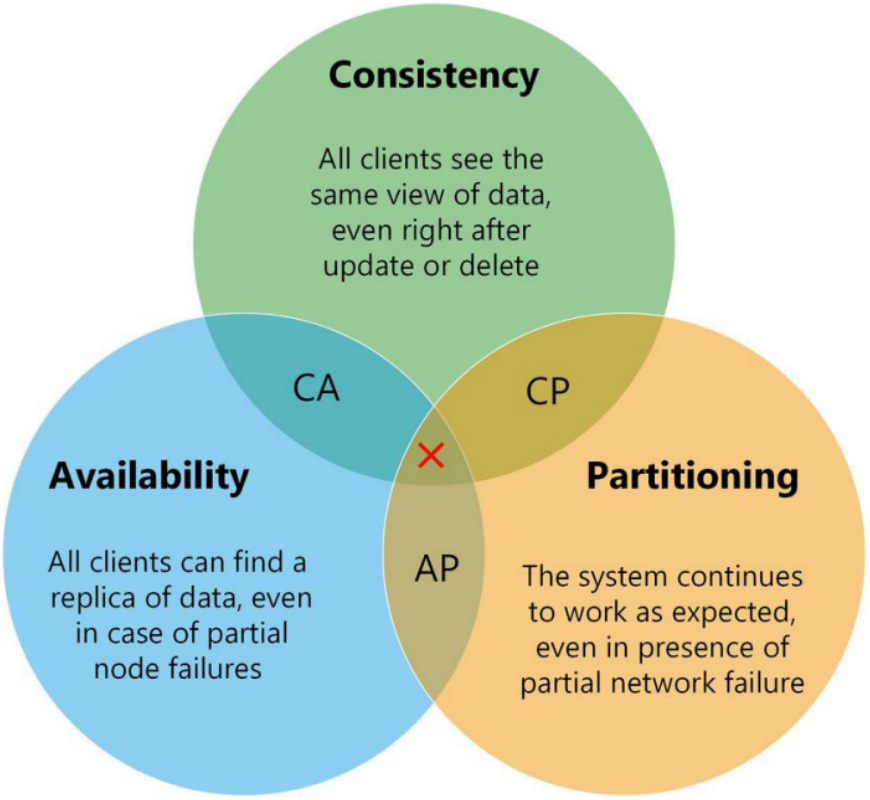
> "It is not possible to guarantee all three of the desirable properties— consistency, availability, and partition tolerance—at the same time in a distributed system with data replication".

Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions.

Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.

Partition tolerance means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.

Consistency is given up



1. Which of the following is a key-value database?

(a) CouchDB (b) MongoDB **(c) DynamoDB** (d) neo4j

2. The A in CAP theorem stands for?

(a) Atomicity **(b) Availability** (c) Adaptability (d) Active

3. Consider the following statements:

(i) NOSQL systems allow a dynamic schema

(ii) Vertical Scalability is employed in NOSQL systems

(iii) Sharding distributes the load of accessing the file records to multiple nodes.

(iv) Neo4j is a column based NOSQL system

The correct statements are:

(a) (i),(iv) (b) (ii), (iii), (iv) (c) (ii),(iv) **(d) (i),(iii)**

Document Based NoSQL

Store data as collections of similar documents sometimes known as document stores.

No requirement to specify a schema—rather, the documents are specified as self-describing data.

Although the **documents in a collection should be similar,** they can have different data elements (attributes), and new documents can have new data elements that do not exist in any of the current documents in the collection.

MongoDB

- It is a document database, which allows you to store objects nested to any depth and the nested data can be queried in an ad-hoc fashion. - **scalable database**
- It enforces **no schema,** so documents can contain fields or types that no other document in the collection contains.
- MongoDB documents are stored in **BSON (Binary JSON) format**
- Documents are organized in a structure called **collection.** A collection does not have a schema.
- **Document can be considered as a row in a table and collection can be considered as an entire table.**
- `db.createCollection(“project”, { capped : true, size : 1310720, max : 500 } )`
- create a collection called project to hold PROJECT objects from the COMPANY database
- Each document in a collection has a unique **ObjectId field, called _id**, which is automatically indexed
- System-generated ObjectIds have a specific format, which combines the timestamp
  when the object is created (4 bytes, in an internal MongoDB format), the node id
  (3 bytes), the process id (2 bytes), and a counter (3 bytes) into a 16-byte Id value

ObjectId (16-byte Id value)

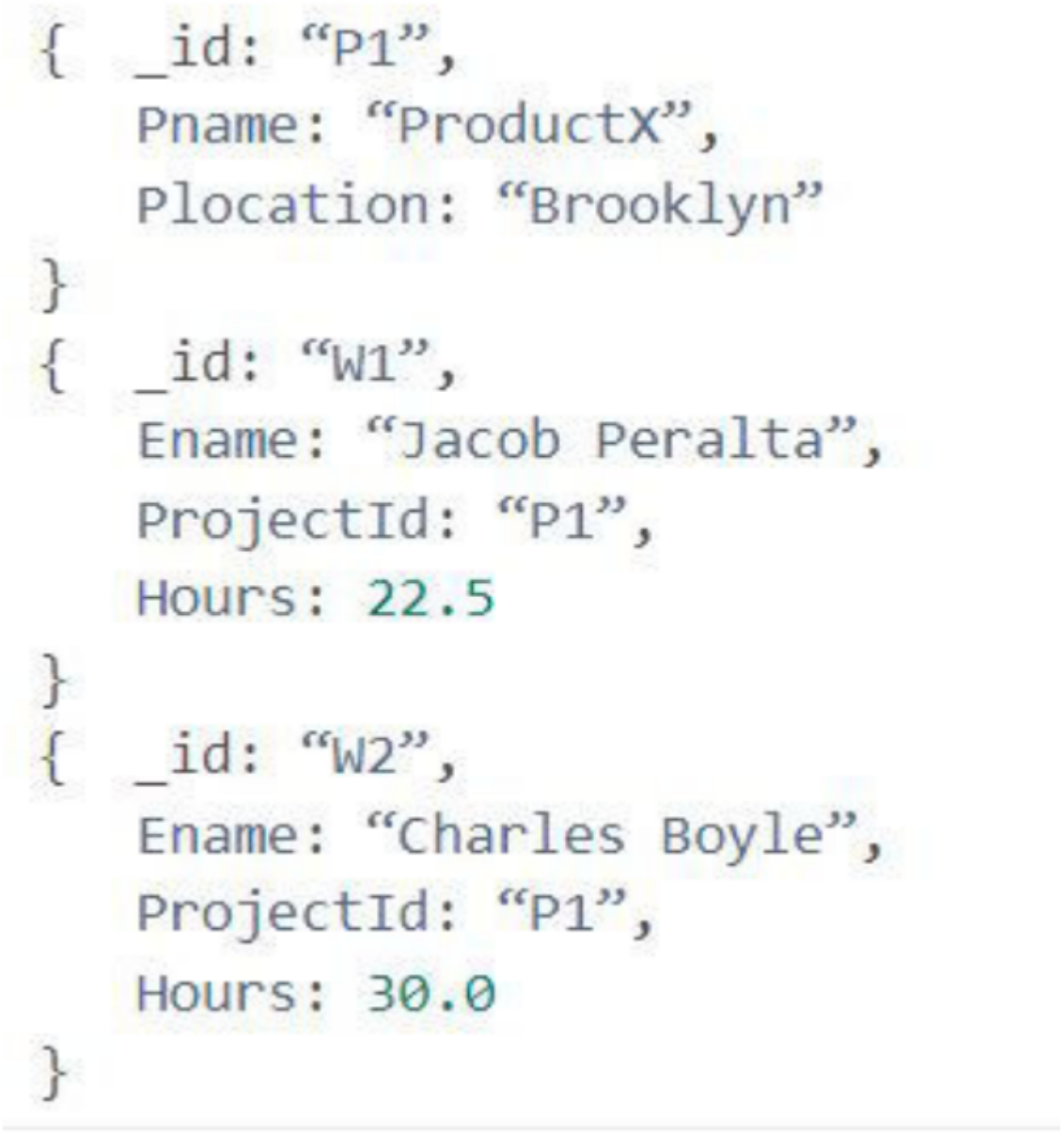
timestamp (4 bytes)

node id(3 bytes)

process id (2 bytes)

counter (3 bytes)

```
{
_id: “P1”,
 Pname: “ProductX”,
 Plocation: “Brooklyn”,
 WorkerIds: [ “W1”, “W2” ]
}
{ _id: “W1”,
  Ename: “Jacob Peralta”,
  Hours: 22.5
}
{ _id: “W2”,
  Ename: “Charles Boyle”,
  Hours: 30.0
}
```



The documents whose _id starts with W (for worker) will be stored in the “worker” collection.

In this example, Worker references are embedded in the project document, but the worker documents themselves are stored in a separate “worker” collection.

MongoDB vs RDMS

RDBMS	MongoDB
It is a relational database.	It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage.	Suitable for hierarchical data storage.
It is vertically scalable i.e increasing RAM.	It is horizontally scalable i.e we can add more servers.
It has a predefined schema.	It has a dynamic schema.

It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).	It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).
It is row-based.	It is document-based.
It is slower in comparison with MongoDB.	It is almost 100 times faster than RDBMS.
Supports complex joins.	No support for complex joins.

Create , Read , Update and Delete (CRUD) Operations in MongoDB

Create()

Format:

```
db.createCollection(name, options)
```

Example:

```
db.createCollection("myCollection")
```

Inserting documents into collections

```
db.<collection_name>.insert(<document(s)>)
```

Example:

```
db.project.insert( { _id: “P1”, Pname: “ProductX”, Plocation: “Brooklyn” } )
```

Read() - find()

```
db.<collection_name>.find(<condition>)
```

lt -> less than

gt -> greater than

lte -> less than or equal to

gte -> greater than or equal to

```
//and -> , or $and [{condition 1},{condition 2}]
//or -> $or:[{condition 1},{condition 2}]
```

- `$and`: Returns documents where both queries match
- `$or`: Returns documents where either query matches
- `$nor`: Returns documents where both queries fail to match
- `$not`: Returns documents where the query does not match

Example

Query: Display the first document in the collection

```
db.employee.findOne()
```

Query: Display the document of employee with empid=2

```
db.employee.find({“empid”:2})
```

Query: Display the document related to employees whose age is less than 30

```
db.employee.find({“age”:{$lt:”30”}})
```

Query: Return the employees born after ‘1995-08-07’ from employee collection

```
db.employee.find( { “birth”: { $gt: new Date('1995-08-07') } } )
```

Query : Return all the employees who have Java as a skill

```
db.employee.find({“skill”:”Java”})
```

Query: Return all the employees skilled in Java and salary of 75000

```
db.employee.find({“skill”:”Java”,”salary”:”75000”})
```

Query: Return all employees either skilled in Java or salary of 80000

```
db.employee.find({$or:[{“skill”:”Java”},{”salary”:”75000”}])
```

Query: Return documents where DOB is between 1997-09-02 and 2001-11-09

```
db.employees.find( { dob: { $gt: new Date('1997-09-02'), $lt: new Date('2001-11-09') } } )
```

Query: Return documents where employees are skilled in “Java” and either with salary 80000 or 95000

```
db.employee.find({“skill”:”Java”}, $or:[{”salary”:”80000”}, {”salary”:”95000”}])
```

MongoDB `find` Method Syntax:

javascript — Copy code

```
db.collection_name.find(selection_criteria, fields_required)
```

- **`db.collection_name`**: Specifies the MongoDB collection you want to query.
- **`find`**: The method used to query the collection.
- **`selection_criteria`**: Optional criteria to filter the documents you want to retrieve. An empty document **`{}`** means no specific criteria (retrieve all documents).
- **`fields_required`**: Specifies which fields to include or exclude in the output. The fields are specified in a document where the keys represent the field names, and the values are either **`1`** to include the field or **`0`** to exclude the field.

1. **Display `firstname` and `object_id` of all employees:**

```javascript
db.employee.find({}, {"firstname": 1})
```

   - `{}: all documents` - No selection criteria, meaning retrieve all documents.
   - `{"firstname": 1}` - Display the `firstname` field (`1` indicates inclusion).

   This query fetches all documents from the `employee` collection and displays only the `firstname` field.

2. **Display only `firstname` of all employees, excluding `_id`:**

```javascript
db.employee.find({}, {"firstname": 1, "_id": 0})
```

   - `{}: all documents` - No selection criteria, meaning retrieve all documents.
   - `{"firstname": 1, "_id": 0}` - Display the `firstname` field (`1` indicates inclusion) and exclude the `_id` field (`0` indicates exclusion).

   This query fetches all documents from the `employee` collection, displaying only the `firstname` field and excluding the default `_id` field.

Update Operation ($set)

MongoDB `update` Operation Syntax:

```javascript
db.collection_name.update(
  { query_criteria },
  { $set: { update_fields } },
  { options }
)
```

- `db.collection_name`: Specifies the MongoDB collection you want to update.
- `update`: The method used to update documents in the collection.
- `{ query_criteria }`: Specifies the criteria to identify the documents that need to be updated.
- `{ $set: { update_fields } }`: Specifies the fields and their new values using the `$set` operator.
- `{ options }`: Optional parameters, such as `{multi: true}`, to control the update operation's behavior.

`db.collection_name.update(selection_criteria, update_value)`

Examples

Query: update salaries of all employees with skill “mongodb”

`db.employee.update({“skill”:”mongodb”},{$set:{“salary”:”1000000”}})`

To update all documents:

`db.employee.update({“skill”:”mongodb”},{$set:{“salary”:”1000000”}}) {multi:true}`

Since MongoDB is a distributed system, the two-phase commit method is used to ensure atomicity and consistency of multi-document transactions.

The `update` method is now deprecated in MongoDB, and it is recommended to use `updateOne`, `updateMany`, or `replaceOne` for more granular control over the update operations.

The following example uses the Collection.updateOne() method on the inventory
collection to update the first document where item equals "paper":

```
db.inventory.updateOne({ item: 'paper' },{$set: { 'size': 'cm
```

The update operation uses the $set operator to update the value of the size field to "cm" and the value of the status field to "P",
Update Multiple Documents

```
db.inventory.updateMany({ qty: { $lt: 50 } },{$set: { 'size':
```

The update operation uses the $set operator to update the value of the size field to "in" and the value of the status field to "P"
Replacing a document

```
db.inventory.replaceOne({ item: 'paper' },{item: 'paper',inst
```

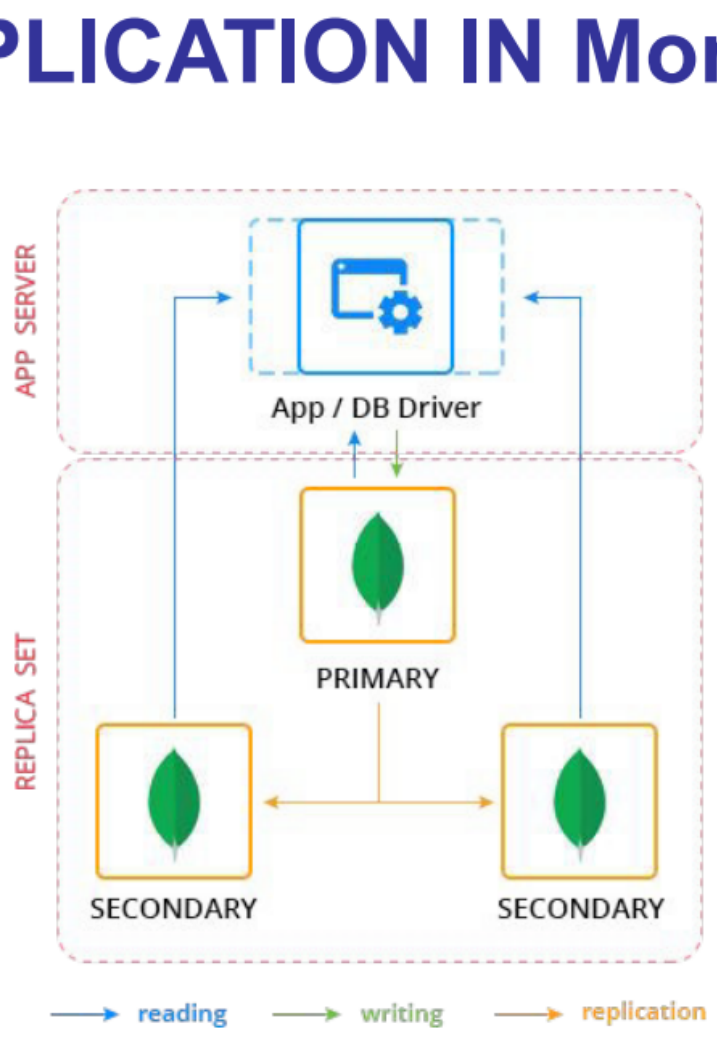
Delete : remove()

```
db.<collection_name>.remove(<condition>)
```

Query: Remove all employees skilled in “mongodb”

```
db.employee.remove({“skill”:”mongodb”}})
```

Replication

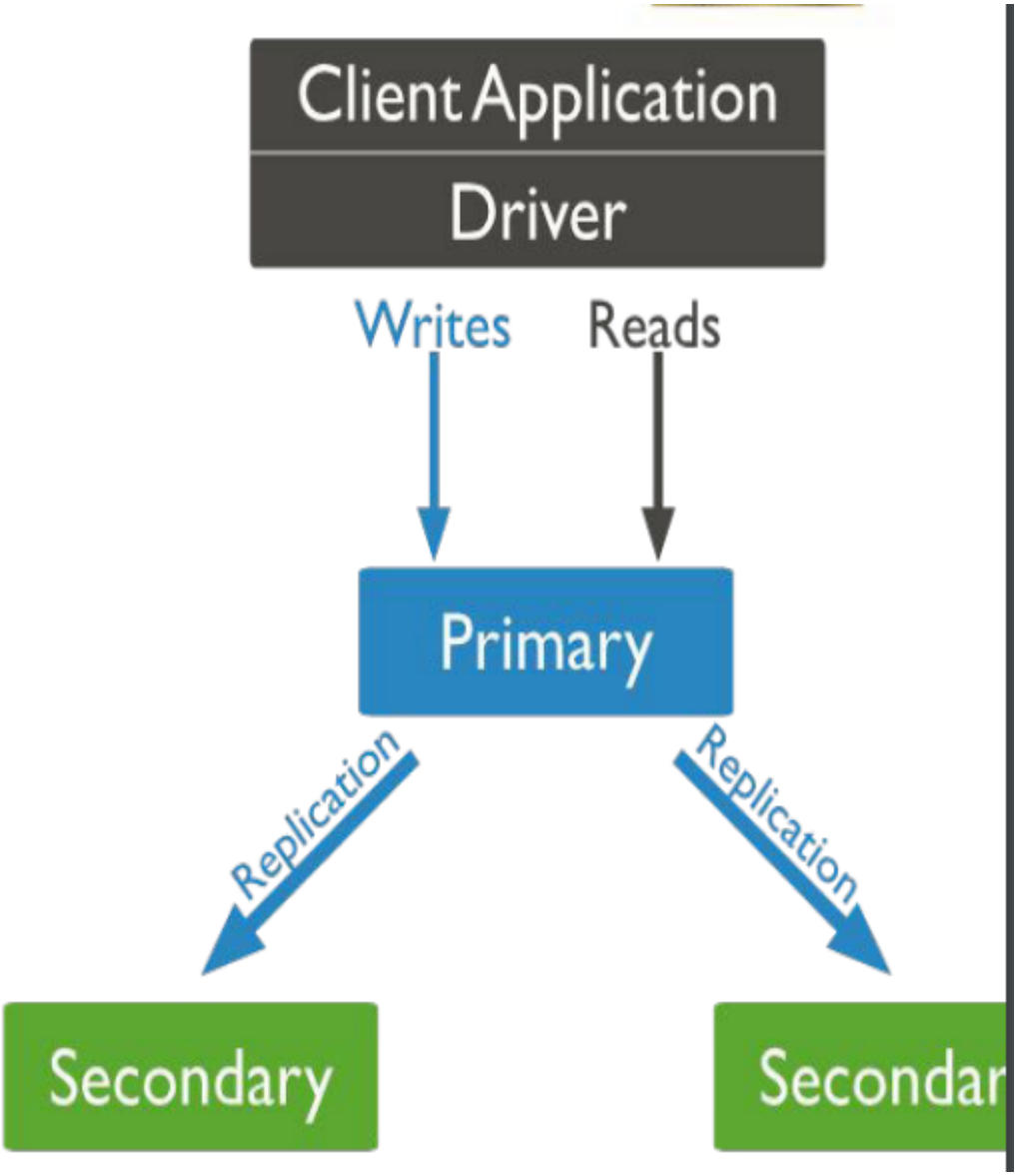


The concept of replica set is used in MongoDB to create multiple copies of the same data set on different nodes in the distributed system

if we want to replicate a particular document collection C. A replica set will have one primary copy of the collection C stored in one node N1, and at least one secondary copy (replica) of C stored at another node N2.

Replication in MongoDB:

1. **Minimum Participants:**
   - The total number of participants in a MongoDB replica set must be at least three.
   - If only one secondary copy is needed, an arbiter must run on the third node (N3) as a participant in the replica set.
2. **Arbiter Role:**
   - The arbiter does not hold a replica of the collection but participates in elections to choose a new primary in case the node storing the current primary copy fails.
3. **Odd Number of Members:**
   - If the total number of members in a replica set is `n` (one primary plus `i` secondaries), then `n` **must be an odd number.**
   - If `n` is not odd, an arbiter is added to ensure the election process functions correctly if the primary fails.
4. **Write Operations:**
   - In MongoDB replication, all write operations must be applied to the primary copy first and then propagated to the secondaries.
5. **Read Operations:**
   - Users can choose the read preference for their applications during read operations.
   - The default read preference processes all reads at the primary copy, meaning all read and write operations are performed at the primary node.
6. **Role of Secondary Copies:**
   - Secondary copies in MongoDB replication primarily serve to ensure system continuity if the primary fails.
   - MongoDB ensures that every read request gets the latest document value by utilizing secondary copies.



Sharding

Overview:

- Sharding is a method used in MongoDB for distributing or partitioning data across multiple machines.
- It is particularly useful when a single machine cannot handle large workloads, allowing for horizontal scaling.
- Horizontal scaling, also known as scale-out, involves adding machines to share the data set and load.

Need for Sharding:

- When a collection contains a large number of documents or requires substantial storage space, storing all documents on one node can lead to performance issues, especially with concurrent user operations.
- Sharding, or horizontal partitioning, divides the collection's documents into disjoint partitions called shards.

The partitioning field (shard key) must have two characteristics:

1. It must exist in every document in the collection
2. It must have an index.

Sharding Benefits:

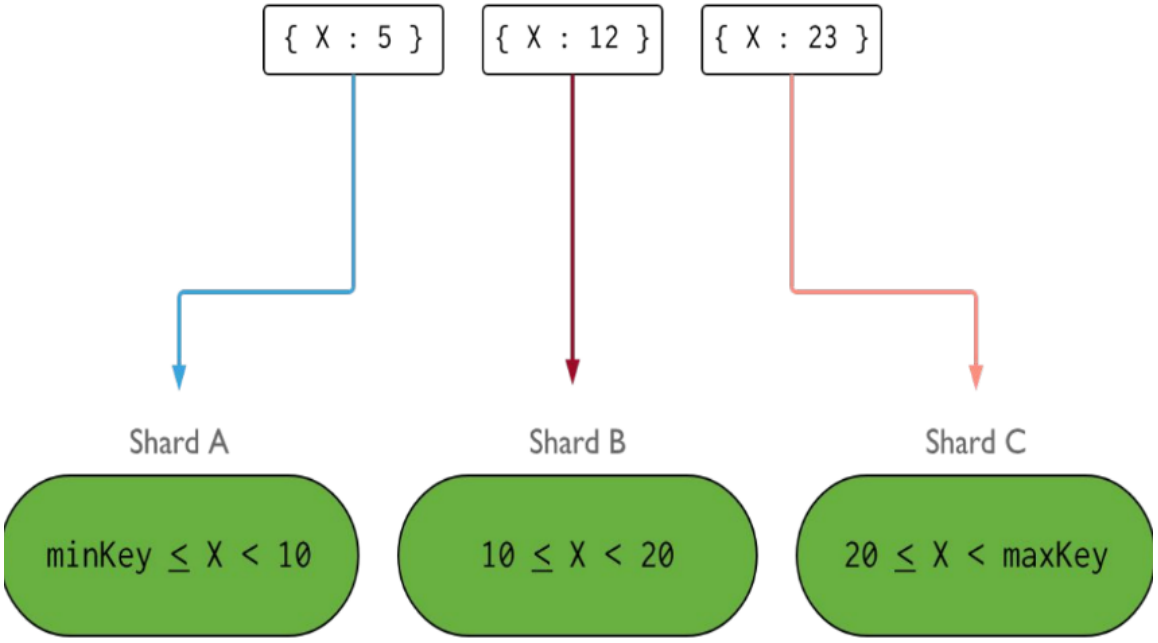
- Allows the addition of more nodes through horizontal scaling to handle big data and intense workloads.
- Achieves load balancing by storing shards of the collection on different nodes.

Sharding Methods:

- Two ways to partition a collection into shards in MongoDB: **Range partitioning and Hash partitioning.**
- Both methods require specifying a document field (shard key) for partitioning, which must exist in every document and have an index.

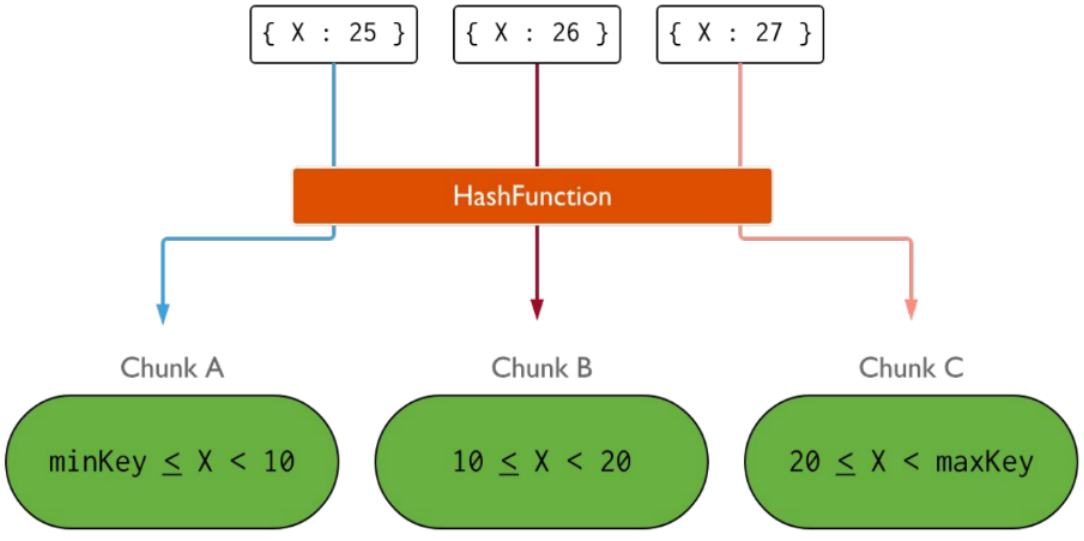
Range Partitioning:

- Creates chunks by specifying a range of key values.
- Example: If shard key values range from one to ten million, it can create ten ranges with each chunk containing key values in one range.



Hash Partitioning:

- Applies a hash function to each shard key, and partitioning is based on the hash values.



Choosing between Range and Hash Partitioning:

- **Range partitioning is preferred for common range queries**, ensuring they are submitted to a single node containing the required documents.

- **Hash partitioning may be preferable for individual document retrieval**, randomizing the distribution of shard key values into chunks.

Query Routing in Sharding:

- MongoDB queries are submitted to a query router when sharding is used.
- The **query router keeps track of which nodes contain which shards** based on the chosen partitioning method and shard keys.
- The query is routed to nodes that contain the shards holding the requested documents.

Integration with Replication:

- Sharding and replication are used together.
- **Sharding focuses on load balancing and horizontal scalability, while replication ensures system availability in the event of node failures in the distributed system.**

Consider a database db with the collection “posts”. Documents are inserted into this collection with this command:

```
db.posts.insertMany([
{
  title: "Intermittent fasting",
  body: "It promotes health benefits such as weight loss and
improved metabolic function.",
  category: "Health",
  likes: 17
 },
{
 title: "Benefits of Blockchain",
 body:"Blockchain provides decentralized and secure data storage, ,
enhancing transparency,traceability,trust in various industries.",
 category: "Technology",
 likes: 4
 },
{
  title: "2024 Elections",
  body: "Political parties are gearing up for the Lok Sabha
elections in 2024.",
  category: "News",
  likes: 65
  },

{
  title: "Audio Steganography",
  body:"An approach of hiding information within an audio signal.",
  category: "Technology",
  likes: 8
  },
```

Write commands in MongoDB for the following operations:

1. Insert a document with the following fields and values into this collection:

```
title: "G20 Summit",
body: "The summit addressed pressing geopolitical challenges.",
category: "News",
likes: 15
```

2. Retrieve the posts where category is "News"
3. Retrieve information of only title and body of all posts
4. Set the number of likes on the post "Benefits of Blockchain" to 8
5. Set the number of likes of a post to 10 if the number of likes is less than 5.
6. Delete the post with the title "2024 Elections".
7. Delete all the posts whose category is "News".
8. Retrieve posts whose category is either "Entertainment" or "Health"
9. Retrieve "Technology" category posts which have greater than 5 likes.
10. Retrieve all posts whose likes are less than or equal to 15.

1. **Insert a document:**

```
db.posts.insertOne({
   title: "G20 Summit",
   body: "The summit addressed pressing geopolitical
challenges.",
   category: "News",
   likes: 15
});
```

2. **Retrieve posts where category is "News":**

```
db.posts.find({ category: "News" });
```

3. **Retrieve information of title and body of all posts:**

```
db.posts.find({}, { title: 1, body: 1, _id: 0 });
```

4. **Set the number of likes on the post "Benefits of Blockchain" to 8:**

```
db.posts.updateOne({ title: "Benefits of Blockchain" }, { $set: { likes: 8 } });
```

5. **Set the number of likes of a post to 10 if less than 5:**

```
db.posts.updateMany({ likes: { $lt: 5 } }, { $set: { likes: 10 } });
```

6. **Delete the post with the title "2024 Elections":**

```
db.posts.deleteOne({ title: "2024 Elections" });
```

7. **Delete all posts whose category is "News":**

```
db.posts.deleteMany({ category: "News" });
```

8. **Retrieve posts whose category is either "Entertainment" or "Health":**

```
db.posts.find({ category: { $in: ["Entertainment", "Health"] } });
```

9. **Retrieve "Technology" category posts with more than 5 likes:**

```
db.posts.find({ category: "Technology", likes: { $gt: 5 } });
```

10. **Retrieve all posts whose likes are less than or equal to 15:**

```
db.posts.find({ likes: { $lte: 15 } });
```

These commands should help you perform the specified operations on your MongoDB collection. Adjust the collection and field names as needed based on your actual data structure.

DynamoDB

- Key: A unique identifier associated with a data item and is used to locate this data item rapidly.
- Value: The data item itself, and it can have very different formats for different key-value storage systems.
- **DynamoDB uses the concepts of tables, items, and attributes**
- Table: A table in DynamoDB does not have a schema; it holds a collection of self-describing items.
- Item: Each item will consist of a number of (attribute, value) pairs, and attribute values can be single-valued or multivalued.
- **The table name and a primary key is required when creating a table.**

Types of Primary Key

Single Attribute: This attribute is used to build a hash index on the items in the table.Called as a hash type primary key.

Pair Attribute : hash and range type primary key.

The primary key is a pair of attributes (A, B) where A is used for hashing and B is used for ordering the records which have the same A value

Voldermort

Voldemort is an open source system available through Apache 2.0 open source

High performance, high scalability, high availability (i.e replication, sharding, horizontal scalability) are realized through a technique to distribute the key-value pairs among the nodes of a distributed cluster known as **consistent hashing.**

get, put, delete

Voldemort Key-Value Distributed Data Source

Features

1. **Simple basic operations:**
   - Collection of key-value pairs is kept in a **store(s)**.
   - 3 operations: **get, put, delete**
     - s.get(k) : retrieves the value v associated with key k.
     - s.put(k, v) : inserts an item as a key-value pair with key k and value v.
     - s.delete(k) : deletes the item whose key is k from the store.
   - At the basic storage level, both keys and values are arrays of bytes (strings).

At the basic storage level, both keys and values are arrays of bytes (strings).

Features

2. **High-level formatted data values:**
   - The values (v) can be specified in **JSON** format.
   - Other formats can be specified if the application provides **conversion(serialization)** between user format and storage format as a *serializer class*.
   - The Serializer class must be provided by the user and will include operations to convert the user format into a string of bytes for storage as a value, and to convert back a string (array of bytes) retrieved via s.get(k) into the user format.
   - Voldemort has some built-in serializers for formats other than JSON.

Consistent Hashing

Overview:

- **Consistent hashing** is a variation of the data distribution algorithm used in Voldemort for distributing (key, value) pairs among nodes in a distributed cluster.
- It enables **horizontal scalability by allowing the addition of new nodes and replication of data.**
- A **hash function** is applied to the key of each (key, value) pair, determining the location where the item will be stored.

Key Concepts:

1. **Hash Ring:**

- Consider the range of all possible integer hash values (0 to Hmax) distributed evenly on a circle (hash ring).
- Nodes in the distributed system are also located on the same ring, with each node having several positions on the ring.

2. **Storage Decision:**
   - An item (k, v) is stored on the node whose position on the ring follows the position of h(k) on the ring in a clockwise direction.
   - Nodes have multiple locations on the ring, providing flexibility in storage assignments.
3. **Horizontal Scalability:**
   - When a new node is added, it can be placed in one or more locations on the ring based on its capacity.
   - A limited percentage of (key, value) items are reassigned to the new node using the consistent hashing placement algorithm.
4. **Item Replication:**
   - **Replication is achieved by placing specified replicas of an item on successive nodes on the ring in a clockwise direction.**
   - Sharding is inherently built into the method, horizontally partitioning items among nodes in the distributed system.
5. **Handling Node Failures:**
   - **When a node fails, the load of data items from that node can be distributed to other existing nodes whose labels follow the labels of the failed node on the ring.**
   - Nodes with higher capacity can have more locations on the ring, allowing them to store more items.

Example:

- In a scenario with three nodes (A, B, C) on the hash ring, each node has multiple placements on the ring.
- Items with hash values falling into specific ranges on the circle are stored on corresponding nodes (e.g., range 1 in node A, range 2 in node B, and range 3 in node C).

Benefits:

- Enables horizontal scalability, supporting the addition of new nodes.
- Facilitates data replication for fault tolerance.
- Supports automatic load balancing as nodes are added or removed from the system.

Consistent Versioning

1. C**oncurrent Writes:**
    - Concurrent writes are allowed, and each write is associated with a vector clock value.
2. **Reads with Different Versions:**
    - For a read where different versions of the same value (associated with the same key) are read from different nodes:
        - If the system can reconcile to a single final value, it passes that value to the read.
        - If the system can't reconcile to a single value, more than one version can be passed back to the application.
3. **Reconciliation Process:**
    - The application is responsible for reconciliation.
    - The application passes the reconciled value back to the nodes.

Graph Databases

- Data is represented as a graph - **collection of nodes(vertices) and edges.**
- Neo4j is an open source system, and it is implemented in Java.
- **Labels are used to group nodes**
    - Labels are optional and each node can have multiple labels
- Neo4j organizes data using the concepts of **nodes and relationships.**
- **Properties** store the data items associated with nodes and relationships.
- Relationships are **directed**; **each relationship has a start node and end node**

- ●Nodes in Neo4j correspond to entities
- Node labels correspond to entity types and subclasses
- Relationships correspond to relationship instances
- Properties correspond to attributes

Neo4j	ER Model
Relationship is directed.	Relationship is not directed.
Node may have no label in Neo4j.	Every entity must belong to an entity type, so a node must have a label.
Graph model of Neo4j is used as a basis for an actual high-performance distributed database system	The ER/EER(Enhanced ER) model is mainly used for database design.

Neo4j system creates an internal unique system-defined identifier for each node.

It is possible that some nodes have multiple labels.

Figure 24.4
Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships.

(a) creating some nodes for the COMPANY data (from Figure 5.6):

```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})
...
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})
...
CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})
...
CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})
...
```

A path specifies a traversal of part of the graph.

A schema is optional in Neo4j.

When a node is created, the Neo4j system creates an internal unique system-defined identifier for each node.

Neo4j has a high-level query language, Cypher.

(c) **Basic simplified syntax of some common Cypher clauses:**
Finding nodes and relationships that match a pattern: MATCH <pattern>
Specifying aggregates and other query variables: WITH <specifications>
Specifying conditions on the data to be retrieved: WHERE <condition>
Specifying the data to be returned: RETURN <data>
Ordering the data to be returned: ORDER BY <data>
Limiting the number of returned data items: LIMIT <max number>
Creating nodes: CREATE <node, optional labels and properties>
Creating relationships: CREATE <relationship, relationship type and optional properties>
Deletion: DELETE <nodes or relationships>
Specifying property values and labels: SET <property values and labels>
Removing property values and labels: REMOVE <property values and labels>

(d) **Examples of simple Cypher queries:**

```
1. MATCH (d : DEPARTMENT {Dno: '5'}) – [ : LocatedIn ] → (loc)
   RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) – [ w: WorksOn ] → (p)
   RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e ) – [ w: WorksOn ] → (p: PROJECT {Pno: 2})
   RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [ w: WorksOn ] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
5. MATCH (e) – [ w: WorksOn ] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
   LIMIT 10
6. MATCH (e) – [ w: WorksOn ] → (p)
   WITH e, COUNT(p) AS numOfprojs
   WHERE numOfprojs > 2
   RETURN e.Ename , numOfprojs
   ORDER BY numOfprojs
7. MATCH (e) – [ w: WorksOn ] → (p)
   RETURN e , w, p
   ORDER BY e.Ename
   LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: '2'})
   SET e.Job = 'Engineer'
```

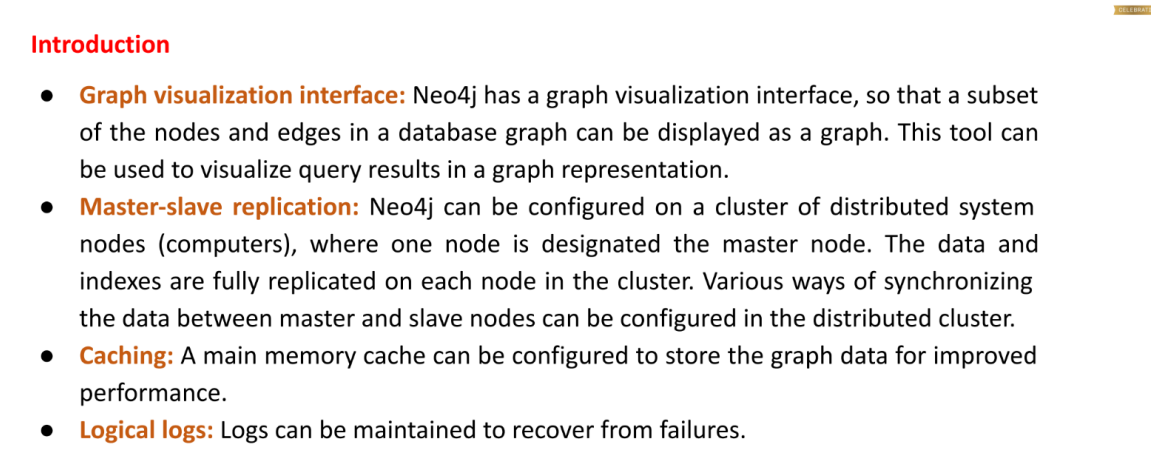
Query 1 in Figure 24.4(d) shows how to use the MATCH and RETURN clauses in a query, and the query retrieves the locations for department number 5. Match specifies the pattern and the query variables (d and loc) and RETURN specifies the query result to be retrieved by referring to the query variables.

Query 2 has three variables (e, w, and p), and returns the projects and hours per week that the employee with Empid = 2 works on.

Query 3, on the other hand, returns the employees and hours per week who work on the project with Pno = 2.

Query 4 illustrates the ORDER BY clause and returns all employees and the projects they work on, sorted by Ename. It is also possible to limit the number of returned results by using the LIMIT clause as in query 5, which

only returns the first 10 answers.
Query 6 illustrates the use of WITH and aggregation, although the WITH clause can be used to separate,clauses in a query even if there is no aggregation. Query 6 also illustrates the WHERE clause to specify additional conditions, and the query returns the employees who work on more than two projects, as well as the number of projects each employee works on. It is also common to return the nodes and relationships themselves in the query result, rather than the property values of the nodes as in the previous queries.
Query 7 is similar to query 5 but returns the nodes and relationships only, and
so the query result can be displayed as a graph using Neo4j's visualization tool. It is also possible to add or remove labels and properties from nodes. Query 8 shows how to add more properties to a node by adding a Job property to an employee node.




```
CREATE (:User { name: "@jeff" })-[:FOLLOWS]->(:User { name: "@neo" })
```

user is label()

start node is name{}

relationship is follow[]

end node is name[]

`(label{node})-[relationship]→(label{node})`

Neo4j can be a csv file or a JSON

```
It looks like you've provided a series of Cypher queries f

1. **MATCH (n) RETURN n:**
   - Matches all nodes in the graph and returns them.

2. **MATCH (n:Movie) RETURN n:**
   - Matches all nodes labeled as `Movie` and returns them

3. **MATCH (n:Movie{title:'Top Gun'}) RETURN n:**
   - Matches nodes labeled as `Movie` with the title 'Top

4. **MATCH (n:Movie) WHERE n.title = 'Top Gun' RETURN n:**
   - Matches nodes labeled as `Movie` where the title is '

5. **MATCH (m:Movie) WHERE m.released >= 2000 RETURN m.tit
   - Matches movies released in or after 2000 and returns

6. **MATCH (person:Person) WHERE person.name STARTS WITH '
   - Matches people whose names start with 'An' or end wit

7. **MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie) WHERE
   - Matches actors who acted in the movie 'The Green Mile

8. **MATCH (reviewer:Person)-[r:REVIEWED]->(movie:Movie) W
   - Matches reviewers who reviewed movies with a rating g

9. **MATCH (writer:Person{name:'Cameron Crowe'})-[:WROTE]-
   - Matches movies written by Cameron Crowe and actors wh

10. **MATCH (director:Person)-[:DIRECTED]->(m:Movie) RETUR
    - Matches directors who directed movies and returns th

11. **MATCH (reviewer:Person)-[r:REVIEWED]->(m:Movie) RETU
    - Matches reviewers who reviewed movies and returns th
```

12. **MATCH (reviewer:Person)-[r:REVIEWED]->(m:Movie) RETU
    - Same as the previous query, but results are ordered

13. **CREATE (n:MOVIE{title:"Movie1"}) RETURN n:**
    - Creates a new movie node with the title 'Movie1' and

14. **CREATE (n:MOVIE{title:"Movie2", released:2001}) RETU
    - Creates a new movie node with the title 'Movie2' and

15. **CREATE (n:MOVIE{title:"Movie3"}) RETURN n:**
    - Creates a new movie node with the title 'Movie3' and

16. **CREATE (n:PERSON{name:"Person1"}) RETURN n:**
    - Creates a new person node with the name 'Person1' an

17. **CREATE (n:PERSON{name:"Person2"}) RETURN n:**
    - Creates a new person node with the name 'Person2' an

18. **CREATE (n:PERSON{name:"Person3"}) RETURN n:**
    - Creates a new person node with the name 'Person3' an

19. **CREATE (n:PERSON{name:"Person4"}) RETURN n:**
    - Creates a new person node with the name 'Person4' an

20. **MATCH (a:PERSON), (b:MOVIE) WHERE a.name = "Person1"
    - Matches the person node 'Person1' and movie node 'Mo

21. **MATCH (a:PERSON), (b:MOVIE) WHERE a.name = "Person2"
    - Similar to the previous query but for 'Person2' and

22. **MATCH (a:PERSON), (b:MOVIE) WHERE a.name = "Person1"
    - Similar to the previous queries but for 'Person1' an

23. **MATCH (a:PERSON), (b:MOVIE) WHERE a.name = "Person3"
    - Similar to the previous queries but for 'Person3' an

24. **MATCH (a:PERSON), (b:MOVIE) WHERE a.name = "Person4"

```
    - Similar to the previous queries but for 'Person4' an

25. **MATCH (p:PERSON{name:"Person1"}) SET p.age = 54 RETU
    - Matches the person node 'Person1' and sets the age p

26. **MATCH (p:PERSON{name:"Person1"}) SET p:ACTOR RETURN
    - Matches the person node 'Person1' and adds the label

27. **MATCH (m:MOVIE{title:"Movie3"}) DELETE m:**
    - Matches the movie node 'Movie3' and deletes it.

28. **MATCH (m:MOVIE{title:"Movie2"}) DELETE m:**
    - Matches the movie node 'Movie2' and deletes it.

29. **MATCH (m:MOVIE{title:"Movie2"}) DETACH DELETE m:**
    - Matches the movie node 'Movie2' and deletes it along

30. **MATCH (n) RETURN n:**
    - Matches all nodes in the graph and returns them.

31. **MATCH (n) DETACH DELETE n:**
    - Matches all nodes in the graph and deletes them alon

32. **MATCH (n) RETURN n:**
    - Matches all nodes in the graph and returns them.
```

Read Clause

Sr.No	Read Clauses	Usage
1	MATCH	This clause is used to search the data with a specified pattern.
2	OPTIONAL MATCH	This is the same as match, the only difference being it can use nulls in case of missing parts of the pattern.
3	WHERE	This clause id is used to add contents to the CQL queries.
4	START	This clause is used to find the starting points through the legacy indexes.
5	LOAD CSV	This clause is used to import data from CSV files.

Write Clause

Sr.No	Write Clause	Usage
1	CREATE	This clause is used to create nodes, relationships, and properties.
2	MERGE	This clause verifies whether the specified pattern exists in the graph. If not, it creates the pattern.
3	SET	This clause is used to update labels on nodes, properties on nodes and relationships.
4	DELETE	This clause is used to delete nodes and relationships or paths etc. from the graph.
5	REMOVE	This clause is used to remove properties and elements from nodes and relationships.
6	FOREACH	This class is used to update the data within a list.
7	CREATE UNIQUE	Using the clauses CREATE and MATCH, you can get a unique pattern by matching the existing pattern and creating the missing one.
8	Importing CSV files with Cypher	Using Load CSV you can import data from .csv files.

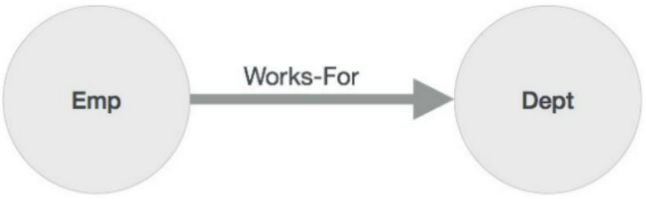
General Clause

Sr.No	General Clauses	Usage
1	RETURN	This clause is used to define what to include in the query result set.
2	ORDER BY	This clause is used to arrange the output of a query in order. It is used along with the clauses **RETURN** or **WITH**.
3	LIMIT	This clause is used to limit the rows in the result to a specific value.
4	SKIP	This clause is used to define from which row to start including the rows in the output.
5	WITH	This clause is used to chain the query parts together.
6	UNWIND	This clause is used to expand a list into a sequence of rows.
7	UNION	This clause is used to combine the result of multiple queries.
8	CALL	This clause is used to invoke a procedure deployed in the database.

CQL DataTypes

Sr.No	CQL Data Type	Usage
1	Boolean	It is used to represent Boolean literals: true, false.
2	byte	It is used to represent 8-bit integers.
3	short	It is used to represent 16-bit integers.
4	int	It is used to represent 32-bit integers.
5	long	It is used to represent 64-bit integers.
6	float	It is used to represent 32-bit floating-point numbers.
7	double	It is used to represent 64-bit floating-point numbers.
8	char	It is used to represent 16-bit characters.
9	String	It is used to represent Strings.

For example-



- Left side node has a Label: "Emp" and the right side node has a Label: "Dept".
- Relationship between those two nodes also has a Label: "WORKS_FOR".

Neo4js Commands

Create a Node:

```
CREATE (n)
```

- **Description:** This command creates a single node in the graph database.
- **Notes:**
  - `(n)` is a pattern that represents a node. `n` is an identifier for the node.
  - The identifier `n` is not the name of the node; it's a variable that can be used in subsequent queries.

Output: The output indicates that one node has been created.

View a Node:

```
MATCH (n) RETURN (n);
```

- **Description:** This command retrieves and returns all nodes labeled as `n` in the graph database.

- **Notes:**
  - `MATCH (n)` is a pattern that matches all nodes labeled as `n`.
  - `RETURN (n)` specifies that the matched nodes should be returned in the result.

Output: The output will display the nodes that match the pattern. Each node will be printed with its internal property `id`. The `id` is an internal identifier for the node, starting from zero, and it is unique within the database.

Delete all nodes:

```
Match(n) delete n
```

//delete all the nodes in the database if there are no relationships in the graph
Output: Deleted "x" nodes

Creating a node with a label:

```
Create (n:Person)
```

Output: Added 1 label, created 1 node

Search node with a labels:

```
Match (n) where n:Person return n
```

Output: Added 1 label, created 1 node
// returns a node with label Person
// displays both the id and label of the node

Create a node with multiple labels:

```
create (n : Person : Indian)
```

Output: Added 2 labels, created 1 node
// a node is created with two labels (person and Indian)
// a node can have multiple labels and a label can have multiple nodes

Search node by multiple labels:

```
Match (n) where n:Person:Indian return n
```

Output: Added 1 label, created 1 node,

// returns all the nodes with label Person and Indian
// displays both the id and label of the node

Add label to an existing node using set clause:

```
Match (n) set n:Employee return n
```

// add's a label Employee to all the nodes in the database

Add label to an existing node using set clause:

```
Match (n) set n:Employee return n
```

// add's a label Employee to all the nodes in the database

Add label to multiple selected nodes:

```
Match (n) where id (n) in [2,3] set n: TeamLeader return n
```

//add's a label "TeamLeader" to the nodes with internal id 2 and 3.

Remove multiple labels from all nodes in database:

```
Match (n) remove n:Food:Vacation return n
```

//remove's label Food and vacation from all the nodes which has both or either of these labels

Update a label of a node:

```
Match (n) where id(n) = 0 remove n:Manager set n:Director return n
```

//update's label of the node with internal id 0 from Manager to Director and return the node

Listing all labels for a node:

```
Match (n) where id(n)=0 return distinct labels(n)
```

//list all the labels associated with node with id=0

Delete is used to delete nodes and remove is used to remove labels

Delete a node with a given label:

```
Match (n) where n:TeamLeader delete n
```

// deletes all the nodes with the label TeamLeader

Create a node with properties (Properties are attributes of a node)

```
create ( x: Book  { title : “Database System”, author: “Navathe”, publisher : “Prearson” })
return x;
```

// creates a node with
label Book with 3 properties.
//property names are case sensitive and can contain underscore and alpha numeric characters //must start with a letter

💡 operation(node:label{property})

```
Match (n:Book) where n.price<1000 AND (n.author="Neel" or n.author = "Navathe")
return n;
```

// return all the nodes with label book with price <1000 and author with value Neel or Navathe

```
Create (x:Book { title:”the lives of others”, author : “Neel”, publisher: [“Chato & Windos”,”w norton”], price: 285.00, pages : 528, instock : false }) return x;
```

To change the title of the Book

```
Match(n) where n.title=“the lives of others” set n.title=“The lives” return n
```

//updates the title from “the lives of others” to “the lives” and returns the node

To update multiple properties:

```
Match(n:Book {title=“existing title}) set n.author=“Gosh”,n.price=324.00,n.stock=true
return n
```

//Searches for the nodes with label Book and title =“existing title” and update its

author, price and stock properties accordingly.

Remove property by setting the value to NULL

`Match(n:Book {author:“Neel”}) set n.publisher=NULL return n`

Remove property “pages” from all nodes

`Match (n) remove n.pages return n`
//to remove property pages from all the nodes

For Relationship

`create (node1) – [r:RelationshipType]->(node2)`

```
Create (chetan:author{name:”Chetan”,dob:1980,age:41})
Create (nagas:Book{title:”Nagas”,price:200,publication:”westland publication”}
```

//above to queries creates 2 nodes

`Create (chetan)-[r:author_of]->(nagas)`
//above query to create relationship between them

Examples

Create Nodes with Properties:

```
CREATE (person:Person {name: 'Alice', age: 30})
CREATE (company:Company {name: 'XYZ Corporation', indust
ry: 'Tech'})
```

- **Description:** Creates nodes with labels (`Person` and `Company`) and associated properties.

Create Relationships between Nodes:

```
MATCH (alice:Person), (company:Company)
WHERE alice.name = 'Alice' AND company.name = 'XYZ Corpo
ration'
CREATE (alice)-[:WORKS_AT]->(company)
```

- **Description:** Creates a relationship between the nodes representing Alice and the XYZ Corporation.

Query Nodes and Relationships:

```
MATCH (person:Person)-[r:WORKS_AT]->(company:Company)
WHERE person.name = 'Alice'
RETURN person, r, company
```

- **Description:** Queries for nodes and relationships where Alice works at a company.

Create Multiple Nodes and Relationships:

```
CREATE (bob:Person {name: 'Bob', age: 25}),
       (jane:Person {name: 'Jane', age: 28}),
       (startup:Company {name: 'ABC Startups', industry:
'Tech'})

MATCH (bob:Person), (jane:Person), (startup:Company)
CREATE (bob)-[:WORKS_AT]->(startup),
       (jane)-[:WORKS_AT]->(startup)
```

- **Description:** Creates multiple nodes and relationships in a single query.

Update Node Properties:

```
MATCH (person:Person {name: 'Alice'})
SET person.age = 31
RETURN person
```

- **Description:** Updates the age property of the node representing Alice.

Delete a Node and Relationships:

```
MATCH (person:Person {name: 'Bob'})-[r:WORKS_AT]->()
DELETE person, r
```

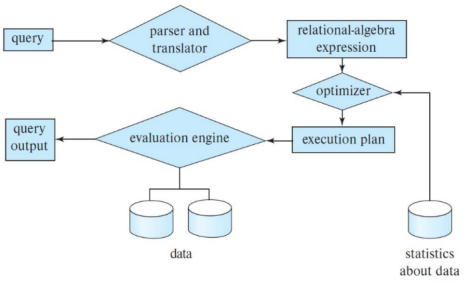
- **Description:** Deletes the node representing Bob and any relationships where Bob works.

Query Processing Language

Query processing refers to the range of activities involved in extracting data from a database. The activities include **translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries**.

Steps in Query Processing

1. Parsing and translation
   - Translate the query into its internal form. This is then translated into relational algebra.
   - Parser checks syntax and verifies relations.
2. Optimization
   - Construct a query-evaluation plan that minimizes the cost of query evaluation.
3. Evaluation
   - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Query Evaluation Plan:

A query evaluation plan, also known as a query-execution plan, is a sequence of primitive operations that specifies how to evaluate a query. These primitive operations, or evaluation primitives, are annotated with instructions that detail how to perform each operation**. The goal of a query evaluation plan is to guide the query-execution engine in executing the query and returning the desired results.**

Evaluation Primitives:

- Evaluation primitives are relational-algebra operations annotated with instructions on how to execute them.
- Instructions may specify the algorithm to be used for a specific operation or indicate the particular index or indices to use.

Example:

```
sqlCopy code
SELECT * FROM Employees WHERE Department = 'IT';
```

In the context of this query, an evaluation plan might include a selection operation annotated with instructions specifying the use of a particular index on the 'Department' column.

Measures of Query Cost:

To compare alternative evaluation plans for a query, it's essential to estimate their cost. The cost of query evaluation can be measured in terms of various resources, including:

1. **Disk Accesses: The number of times data needs to be read from or written to disk.**
2. **CPU Time: The time taken by the CPU to execute the query.**
3. **Communication Cost: In parallel and distributed database systems, the cost of communication between nodes.**

Focus on I/O Cost:

- For large databases residing on magnetic disks, the I/O (Input/Output) cost, such as the number of block transfers from disk and the number of seeks, usually dominates other costs.
- Early cost models often focused on estimating the I/O cost when evaluating query operations.

Considerations:

- The goal is to choose an evaluation plan with the lowest estimated cost.
- Different evaluation plans may exist for a query, and the query-execution engine uses these plans to efficiently retrieve the query results.

Example:

Consider a simple query that involves a selection operation on a relation `Employees` based on the department:

```
sqlCopy code
SELECT * FROM Employees WHERE Department = 'IT';
```

Assume that:

- The relation `Employees` is stored on a high-end magnetic disk with 4 KB blocks.
- The disk subsystem has an average block-transfer time (`tT`) of 0.1 msec and an average block-access time (`tS`) of 4 msec.

Estimation of Cost:(resource consumption–based model)

The cost estimate can be calculated using the formula:

(number of block) * (block transfer time) + (number of seeks) * (block access time)

$$Cost = b \cdot tT + S \cdot tS$$

Where:

- *b* is the number of blocks transferred.
- *S* is the number of seeks (random I/O accesses).

Let's say the query involves transferring 10 blocks (b=10) and requires 2 seeks (S=2).

Cost=10·0.1 msec+2·4 msec

Cost=1 msec+8 msec

Cost=9 msec

Interpretation:

The estimated cost of the query execution plan, in this case, is 9 milliseconds. This cost includes the time for transferring data blocks ($b \cdot tT$) and the time for seeks ($S \cdot tS$). It does not consider the cost of writing the final result back to disk.

Note:

- The values of *tT* and *tS* depend on the characteristics of the storage system. For a high-end magnetic disk with 4 KB blocks, tS=4 msec and

tT=0.1 msec.

- The actual response time might vary based on factors like the current state of the buffer, the size of the buffer, and the distribution of accesses among multiple disks.

Query Tree

Query Tree in Relational Algebra:

A query tree is a tree data structure that corresponds to a relational algebra expression. It visually represents the logical flow of operations in a query. In the context of a query tree:

- **Leaf Nodes:** Correspond to the input relations of the query.
- **Internal Nodes:** Represent relational algebra operations.

Execution of the Query Tree:

1. **Start at the Leaf Nodes:**
    - The order of execution begins at the leaf nodes, which represent the input database relations for the query.
    - Each leaf node typically corresponds to a base table or an intermediate result from a previous operation.
2. **Execute Internal Node Operations:**
    - Move upward in the tree, executing internal node operations whenever their operands (child nodes) are available.
    - The execution of an internal node involves applying the corresponding relational algebra operation to its operands.
3. **Replace Internal Node with Result Relation:**
    - After executing an internal node operation, replace the internal node with the relation that results from the execution.
    - This process continues until the entire tree is traversed.
4. **End at the Root Node:**
    - The execution order continues until reaching the root node, which represents the final operation of the query.

- The root node operation is executed, producing the result relation for the query.

Termination of Execution:

The execution of the query tree terminates when the root node operation is executed, and the result relation for the query is obtained. The entire process follows the logical flow of operations specified in the relational algebra expression.

Example:

Consider a simple relational algebra expression:

The corresponding query tree might look like this:

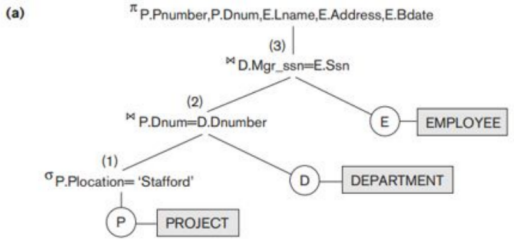
```
    π
    |
    σ
    |
Employee
```

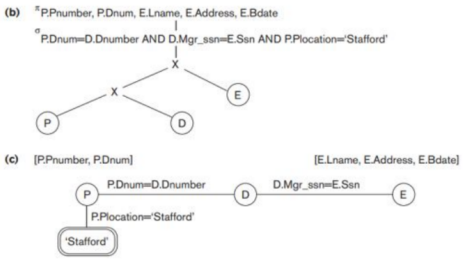
- Leaf Node: Represents the input relation `Employee`.
- Internal Node (σ): Represents the selection operation based on the condition `salary > 50000`.
- Internal Node (π): Represents the projection operation on the attribute `emp_name`.

The execution would start at the leaf node (`Employee`), execute the selection operation, then the projection operation, and finally obtain the result relation.

$\pi_{Pnumber, Dnum, Lname, Address, Bdate}$ (($\sigma_{Plocation='Stafford'}$(PROJECT)) $\bowtie_{Dnum=Dnumber}$(DEPARTMENT)) $\bowtie_{Mgr\_ssn=Ssn}$(EMPLOYEE))

```
SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM    PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
        P.Plocation= 'Stafford';
```





Selection Algorithm

Purpose of Selection Algorithms:

● Identify and retrieve rows from a database that satisfy specified conditions.

● Minimize the amount of data processed during query execution.

Types of Selection Algorithms:

Selection algorithms are methods used to retrieve specific records from a database based on certain conditions. Here are different types of selection algorithms:

A1: Linear Search, Equality on Key

- **Description:** Scans the entire dataset until the required record is found.
- **Cost:** At most *b* block transfers in the worst case, where *b* is the number of blocks in the dataset.

A2: Clustering B+-tree Index, Equality on Key

- **Description:** Utilizes a clustering B+-tree index for efficient lookup.
- **Cost:** Traverses the height of the tree plus one I/O to fetch the record.

A3: Clustering B+-tree Index, Equality on Non-key

- **Description:** Performs one seek for each level of the tree and one seek for the first block.

- **Cost:** *b* blocks containing records with the specified search key are read.

A4: Secondary B+-tree Index, Equality on Key

- **Description:** Similar to clustering index, but may involve secondary index.
- **Cost:** Comparable to A3.

A4: Secondary B+-tree Index, Equality on Non-key

- **Description:** Cost of index traversal is the same as A3, but each record may be on a different block, requiring a seek per record.
- **Cost:** One seek per record.

A5: Clustering B+-tree Index, Comparison

- **Description:** Identical to A3 but involves comparison instead of equality on non-key.
- **Cost:** Similar to A3.

A6: Secondary B+-tree Index, Comparison

- **Description:** Similar to A5, with potential high cost if *n* is large.
- **Cost:** Comparable to A5.

Sorting

Purpose of Sorting:

- **Ordering Data:**
  - Sorting arranges data in a specified order, making it easier to search and retrieve information efficiently.
- **Optimizing Join Operations:**
  - Sorting is often employed to enhance the performance of join operations, such as the subsequent sort-merge phase.

Sort-Merge Algorithm:

- **Definition:**
  - Sorts the records based on the join key and then merges them.
- **Process:**
  - Two main phases: Sorting and Merging.
    - Sorting: Arrange records based on the join key.
    - Merging: Combine sorted records to perform the join.
- **Advantages of Sort-Merge:**
  - Optimizes join operations, especially for large datasets.
  - Provides predictable performance irrespective of data distribution.
  - Easily adaptable to parallel processing for further speedup.
- **Use Cases:**
  - Effective for large datasets where other join algorithms may face performance challenges.
  - Particularly suited for equi-joins where records are matched based on equality conditions.

Hash Join

- **Process:**
  - **Build Phase:**
    - Hashing: Hash function distributes the build input into hash buckets.
    - Hash Table: Build a hash table storing the hashed values and corresponding data.
  - **Probe Phase:**
    - Hashing Again: Apply the same hash function to the probe input.
    - Matching: Probe the hash table for matching entries.
- **Advantages:**
  - Scales well with large datasets due to parallelization.

- Often faster than nested loop and merge join for suitable scenarios.

- **Limitations:**
  - Hash tables can be memory-intensive; consider available resources.
  - Strategies for handling hash collisions should be considered.

Merge Join

- **Working:**
  - Sort both input tables based on the join key.
  - Merge the sorted lists by comparing values in a step-by-step fashion.
  - Exploits the sorted nature of input, reducing the need for extensive random access.
  - commonly used for equi-joins.
- **Advantages:**
  - Efficiency: Especially efficient for large datasets where sorting overhead is outweighed by sequential access during merging.
  - Well-suited for equi-joins on indexed columns.
  - Deterministic Performance: Relatively predictable performance with a time complexity of O(n log n) for sorting and O(n) for merging.

Nested Join

- **Types of Nested Joins:**
  - **Nested Inner Join:**
    - Matching rows from the inner and outer tables based on specified conditions.
    - The result of the inner query is used as input for the outer query's join condition.
  - **Nested Outer Join:**

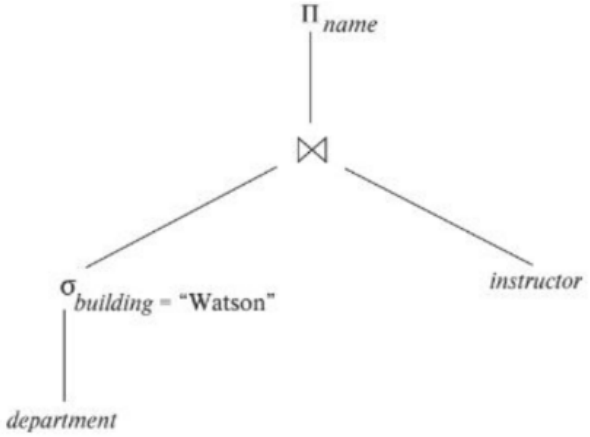
        - Retaining unmatched rows from the outer table while performing a join with the inner table.
    - **Considerations:**
        - Nested joins may impact performance; consider indexing and optimization.
        - Evaluate the trade-off between query readability and complexity.
        - Nested joins provide a powerful tool for handling complex relationships.
        - Understand the data structure and optimize nested joins for better performance.

Evaluation of Expression

1. Materialized evaluation:Bottom Up Approach

Evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.

$$\Pi_{name}(\sigma_{building = \text{“Watson”}}(department) \bowtie instructor)$$



In the operator tree, begin from the lowest-level operations (at the bottom of the tree) in the expression. The inputs to the lowest level operations are stored in the form of relations in the database.

Use temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database.

The join can now be evaluated, creating another temporary relation.

Continue it until you reach root

Evaluation as just described is called materialized evaluation, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

Cost Estimation of Materialized Evaluation in Database Management Systems

The cost estimation of materialized evaluation involves considering the costs of individual operations and the cost of writing intermediate results to disk. Here are the key steps and formulas for estimating the cost:

1. **Overall Cost Formula:**
    - **Overall Cost = Sum of costs of individual operations + Cost of writing intermediate results to disk**
2. **Number of Blocks Written Out (br):**
    - Estimate the number of tuples in the result relation (nr) and the blocking factor of the result relation (fr).
    - Use the formula: br = nr / fr
3. **Transfer Time Calculation:**
    - Calculate the transfer time by estimating the number of required disks.
    - The disk head may have moved between successive writes of the block.
    - Number of seeks = Γ (br / bb)
        - Here, bb is the size of the output buffer measured in blocks.

Double Buffering:

- Optimize the cost estimation using double buffering, where one buffer executes the algorithm continuously, and the other is being written out.
- Double buffering helps in parallelizing CPU activities with I/O activities, making the algorithm execute more efficiently.
- **It reduces the number of seeks by allocating extra blocks to the output buffer, allowing multiple blocks to be written out together.**

Drawback of Materialization:

- Produces a high number of temporary files, making query evaluation less efficient..

Note:

- The cost estimation in materialized evaluation is different from analyzing the cost of an algorithm, as it includes the cost of writing results to disk.

EXAMPLE

1. **Overall Cost Calculation:**
   - The overall cost is the sum of the costs of individual operations plus the cost of writing intermediate results to disk.
2. **Buffer Usage:**
   - Results are stored in a buffer, and when the buffer is full, the results are written to disk.
3. **Estimating Blocks Written (br):**
   - The number of blocks written out (\(b_r \)) can be estimated using the formula:

     $$[b_r = \frac{n_r}{f_r}]$$

     where:
     - \(n_r \) is the estimated number of tuples in the result relation \(r \).
     - \(f_r \) is the blocking factor of the result relation, i.e., the number of records of \(r \) that will fit in a block.

4. **Calculating Transfer Time:**
    - Transfer time is calculated by estimating the number of required disks.
5. **Disk Head Movement:**
    - The disk head may have moved between successive writes of a block.
6. **Number of Seeks Formula:**
    - The formula provided for the number of seeks is: $(\frac{\Gamma b_r}{bb})$ where:
        - (Γ) is the number of blocks to be read or written.
        - (b_r) is the estimated number of blocks written out.
        - (bb) is the size of the output buffer, measured in blocks.

Suppose you have a query that joins two tables, A and B, with A having an estimated 1000 tuples and B having an estimated 800 tuples. The blocking factor f_r is determined by the number of records of r that will fit in a block, let's say $f_r = 50$ records per block.

Using the formula:

$$b_r = \frac{n_r}{f_r}$$

$$b_r = \frac{1000}{50}$$

$$b_r = 20 \text{ blocks}$$

Now, let's consider that the output buffer size (bb) is 10 blocks, and the total number of blocks to be read or written (Γ) is 200 blocks.

Using the formula:

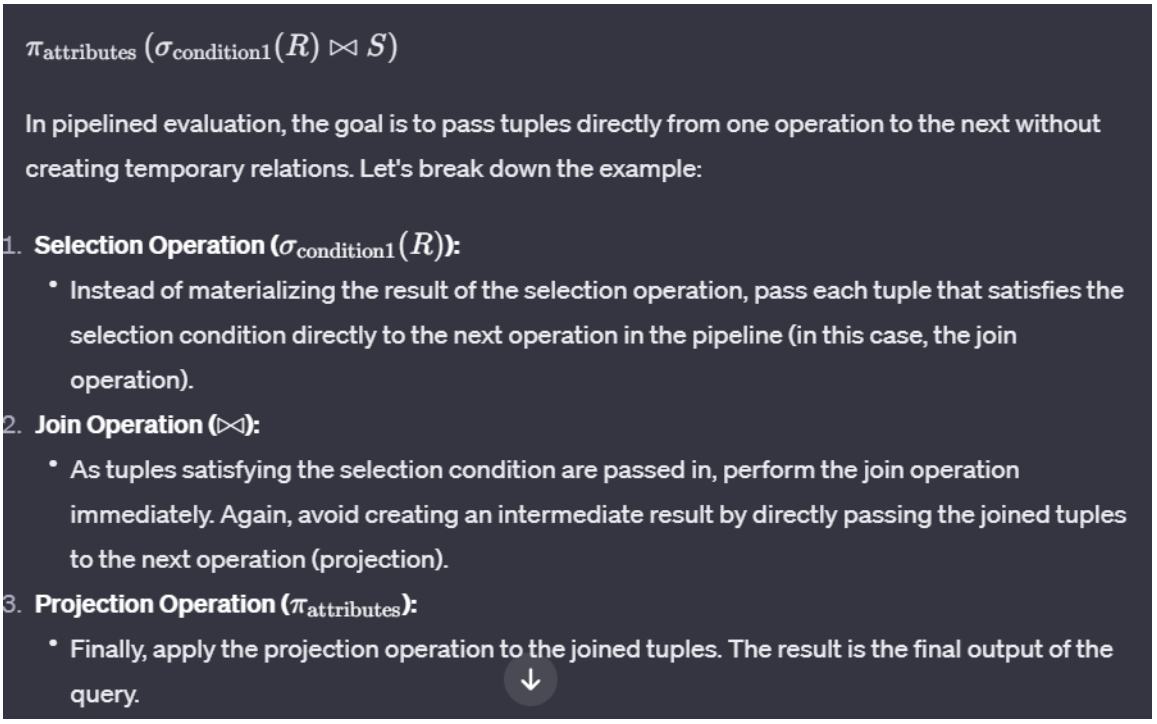
$$\text{Number of seeks} = \frac{200 \times 20}{10}$$

$$\text{Number of seeks} = 400 \text{ seeks}$$

In this example, the estimation suggests that 400 disk seeks would be required for the materialized evaluation, considering the given block sizes and buffer size.

2. Pipeline

Improve query-evaluation efficiency by reducing the number of temporary files that are produced.



$\pi_{\text{attributes}}\ (\sigma_{\text{condition1}}(R) \bowtie S)$

In pipelined evaluation, the goal is to pass tuples directly from one operation to the next without creating temporary relations. Let's break down the example:

1. **Selection Operation ($\sigma_{\text{condition1}}(R)$):**
   - Instead of materializing the result of the selection operation, pass each tuple that satisfies the selection condition directly to the next operation in the pipeline (in this case, the join operation).
2. **Join Operation ($\bowtie$):**
   - As tuples satisfying the selection condition are passed in, perform the join operation immediately. Again, avoid creating an intermediate result by directly passing the joined tuples to the next operation (projection).
3. **Projection Operation ($\pi_{\text{attributes}}$):**
   - Finally, apply the projection operation to the joined tuples. The result is the final output of the query.

Advantages of Pipelining:

- Reduces the cost of query evaluation by eliminating the need for temporary relations.
- Enables quick generation of query results as tuples are processed in a continuous flow.

Implementation:

- A pipeline can be implemented by constructing a single, complex operation that combines the operations in the pipeline.
- Reusing the code for individual operations is desirable for flexibility and maintainability.
- Pipelines can be executed in demand-driven (pull) or producer-driven (push) models.

Pipelining may not always be possible – e.g., sort, hash-join

Demand Driven (pull) Pipeline

In a demand-driven (or pull) pipeline model, the processing of operations in the pipeline is **initiated by the consumer** of the result. This means that each operation in the pipeline is executed only when the **consumer requests the next tuple or set of tuples**. The demand-driven pipeline is driven by the consumer's need for data, and operations are performed on an as-needed basis.

Let's consider how a demand-driven pipeline might work in the context of a query expression. Assume we have the following relational algebra expression:

$[\pi_{\text{attributes}} (\sigma_{\text{condition1}}(R) \bowtie S)]$

1. **Selection Operation** $(\sigma_{\text{condition1}}(R))$**:**
    - The consumer requests a tuple.
    - The selection operation retrieves tuples from relation (R), applies the selection condition, and returns the first tuple that satisfies the condition.
2. **Join Operation** $((\bowtie))$**:**
    - The consumer requests the next tuple.
    - The join operation receives the tuple from the selection operation, performs the join with relation (S), and returns the joined tuple to the consumer.
3. **Projection Operation** $(\pi_{\text{attributes}})$**:**
    - The consumer requests the next tuple.
    - The projection operation receives the tuple from the join operation, applies the projection, and returns the final projected tuple to the consumer.

In this model, each operation is executed in response to a demand from the consumer. **The operations are only performed when the next result is requested, allowing for more efficient resource utilization and avoiding unnecessary computation.**

Advantages of Demand-Driven Pipelines:

- Reduced resource consumption: Operations are performed only when needed, minimizing unnecessary computation.
- Efficient handling of large datasets: The pipeline processes data in a streaming fashion, which is particularly beneficial for large datasets.

Considerations:

- The demand-driven approach may introduce some latency, as operations are performed in response to requests.
- It is well-suited for scenarios where the consumer is actively interacting with the data and needs results on-demand.

Implementation

1. **Iterator Functions:**
    - **open():** Initiates the iterator and prepares it for processing. For a linear search, this might involve starting a file scan and recording the initial scan point.
    - **next():** Retrieves the next tuple as the output of the operation. In the case of a linear search, it continues the file scan from the recorded point, returning the next tuple that satisfies the selection condition.
    - **close():** Signals that no more tuples are required, concluding the iterator**.** This function is called when the consumer indicates that it has received all the needed tuples.
2. **Invocation Process:**
    - After invoking open(), each subsequent call to next() returns the next tuple in the result.
    - The operation may need to invoke open() and next() on its inputs to ensure that the necessary tuples are available when needed.
    - close() is invoked when the consumer no longer needs additional tuples.
3. **State Maintenance:**
    - The iterator maintains its state between calls to next(). For a linear search, this could involve keeping track of the current position in the file or database.

- Successive calls to next() receive tuples based on the current state.

Producer Driven (Push) Pipeline

In a producer-driven pipeline, the **operations produce tuples and push them downstream as soon as they are generated**. This is in contrast to a demand-driven pipeline, where the downstream operations explicitly request tuples as needed. Let's break down the key aspects of a producer-driven pipeline:

1. **Producer Operation:**
    - Each operation in the pipeline is implemented as a producer that generates tuples.
    - The producer operation has three primary functions: start(), produce(), and end()..
    - **Once a tuple is used from a pipelined input, it's removed from the input buffer.**
2. **Downstream Consumption:**
    - The downstream operations consume tuples as soon as they are produced.
    - There is no explicit request for tuples; instead, the tuples are pushed to the downstream operations by the producers.
3. **Flow of Data:**
    - Tuples flow from one operation to the next in a continuous manner.
    - **As soon as a producer generates a tuple, it is immediately sent to the next** operation in the pipeline without waiting for explicit requests.
4. **Efficiency Considerations:**
    - Producer-driven pipelines can be efficient for scenarios where tuples can be generated and pushed downstream without significant overhead.
    - They may be well-suited for streaming data or situations where the cost of generating tuples is relatively low.

5. **Synchronization:**
    - Coordination between producers and consumers is essential to ensure the smooth flow of data.
    - Synchronization mechanisms may be required to manage the data flow and prevent issues like data overflow or underflow.

Demand-driven Pipeline	Producer-driven Pipeline
It is similar to pulling data up from the top of an operation tree.	It is similar to pushing data up from the below of an operation tree.
Tuples are generated in a lazy manner.	Tuples are eagerly generated.
It is easy to implement.	It is not so easy to implement a producer-driven pipeline.
It is most commonly used for evaluating an expression.	It is typical so rarely used in the systems. But, it is good for systems such as parallel processing systems.

Pipelining	Materialization
It is a modern approach to evaluate multiple operations.	It is a traditional approach to evaluate multiple operations.
It does not use any temporary relations for storing the results of the evaluated operations.	It uses temporary relations for storing the results of the evaluated operations. So, it needs more temporary files and I/O.
It is a more efficient way of query evaluation as it quickly generates the results.	It is less efficient as it takes time to generate the query results.
It requires memory buffers at a high rate for generating outputs. Insufficient memory buffers will cause thrashing.	It does not have any higher requirements for memory buffers for query evaluation.
Poor performance if trashing occurs.	No trashing occurs in materialization. Thus, in such cases, materialization is having better performance.
It optimizes the cost of query evaluation. As it does not include the cost of reading and writing the temporary storages.	The overall cost includes the cost of operations plus the cost of reading and writing results on the temporary storage.

Query Optimization

1. **Purpose:**

- **Efficient Query Execution:** **Balance between maintaining query semantics and achieving optimal execution efficiency**
- **Transparent to Users:** Users are not expected to write queries with efficiency in mind; instead, the **database system is responsible for constructing optimized query evaluation plans.**

2. **Relational-Algebra Level:**
   - **Optimization Scope:** The optimization process occurs at the relational-algebra level, where the system seeks an equivalent but more efficient expression for query evaluation.
   - **Semantic Preservation:** The goal is to find an optimized expression that preserves the semantics of the original query while improving execution efficiency.
3. **Detailed Strategy Selection:**
   - **Algorithm and Index Selection:** Query optimization involves selecting detailed strategies for processing, such as choosing algorithms for operations and selecting specific indices.
   - **Strategic Decisions:** Various strategic decisions, including index usage, join strategies, and other optimization techniques, contribute to the overall efficiency of query execution.
4. **Significance of Optimization:**
   - **Cost Difference:** The difference in cost, measured in terms of evaluation time, between a well-optimized strategy and a suboptimal one can be substantial.
   - **Order of Magnitude:** Even small improvements in the optimization process can result in significant reductions in query execution time, sometimes by several orders of magnitude.
5. **Impact on Performance:**

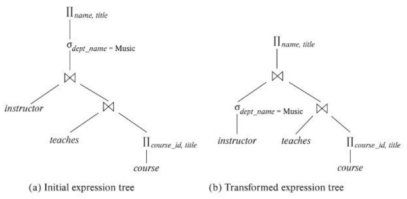
- **Investment in Time:** While optimization involves investing time in selecting the best strategy, the payoff in terms of improved query execution efficiency is often substantial.
- **Profound Impact:** Despite queries being executed only once, a well-optimized strategy can have a profound impact on performance, making the investment worthwhile.

6. **Continuous Improvement:**
    - **Dynamic Nature:** Query optimization is often a dynamic process, as the system adapts to changes in data, statistics, and system conditions.
    - **Adaptive Strategies:** Database systems may employ adaptive strategies that continuously monitor and adjust optimization decisions based on changing conditions.

Example:

Consider the following relational-algebra expression, for the query "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."

$$\Pi_{name,title}(\sigma_{dept\_name = \text{"Music"}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))))$$



(a) Initial expression tree (b) Transformed expression tree

Since we are concerned with only those tuples in the instructor relation that pertain to the Music department, we do not need to consider those tuples that do not have dept name ="Music".
By reducing the number of tuples of the instructor relation that we need to access, we reduce the size of the intermediate result.
Our query is now represented by the relational-algebra expression:

$$\Pi_{name,title}((\sigma_{dept\_name = \text{"Music"}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,title}(course)))$$

Query Evaluation Plan

Query-evaluation plan generation involves three key steps:

1. **Generate Logically Equivalent Expressions:**
    - The process begins by generating expressions that are logically equivalent to the given query.
    - Equivalent expressions have the same results on every legal database instance but may differ in their structure and operation order.
2. **Annotate Expressions for Alternative Plans:**

- Once logically equivalent expressions are generated, annotations are added to these expressions to **represent alternative plans for evaluation.**
- Annotations may include details about the choice of algorithms, indices, and other optimization techniques.
- These annotated expressions represent different strategies for executing the query.

3. **Estimate and Choose the Least-Cost Plan:**
    - The annotated expressions are then evaluated in terms of their estimated costs.
    - Cost estimation considers factors such as disk I/O, CPU time, and other resources.
    - The system selects the **plan with the least estimated cost as the optimal query-evaluation plan.**

Viewing Query Evaluation Plans:

- In some database systems, users can view the query evaluation plan before execution to understand how the system intends to process the query.
- SQL Server, for example, requires the execution of the command `SET SHOWPLAN_TEXT ON` before submitting the query. The execution plan is then displayed instead of executing the actual query.
- Other database systems, such as MySQL and PostgreSQL, use the `EXPLAIN` command to display the evaluation plan.

Transformation of Relational Expressions:

- The goal is to explore different ways of expressing the same query, each with varying evaluation costs.
- **Equivalence in relational algebra is based on the principle that two expressions are equivalent if they generate the same set of tuples on every legal database instance.**
- In SQL, where inputs and outputs are multisets of tuples, equivalence is determined by generating the same multiset of tuples on every legal database instance.

Equivalence in SQL:

- SQL uses the multiset version of relational algebra, where order is not relevant, and duplicates are allowed.

> 💡 **Equivalence in SQL is established by ensuring that two expressions produce the same multiset of tuples on all legal database instances.**

Equivalence Rules in Relational Algebra

Definition: Equivalence rules state that expressions of two forms are equivalent, meaning they generate the same result on any valid database instance.

1. Conjunctive Selection Operations

- **Rule:** Conjunctive selection operations can be deconstructed into a sequence of individual selections.
- **Transformation:** This is referred to as a cascade of σ (sigma).

2. Commutativity of Selection Operations

- **Rule:** Selection operations are commutative.

3. Cascade of Projection Operations

- **Rule:** Only the final operations in a sequence of projection operations are needed; the others can be omitted.
- **Transformation:** This is also known as a cascade of Π (pi) where \(L1 \subseteq L2 \subseteq \ldots \subseteq Ln\).

4. Selections with Cartesian Products and Theta Joins

- **Rule:** Selections can be combined with Cartesian products and theta joins.

5. Commutativity of Theta-Join Operations

- **Rule:** Theta-join operations are commutative. Natural joins are a special case of theta joins, and hence, natural joins are also commutative.

6. Associativity of Natural Joins and Theta Joins

- **Rule (a):** Natural-join operations are associative.
- **Rule (b):** Theta joins are associative in a specific manner.

7. Selection Distribution over Theta-Join Operations

- **Rule (a):** Selection distributes over the theta-join operation when attributes in the selection condition involve only one expression being joined.
- **Rule (b):** Selection distributes over the theta-join operation when selection conditions involve attributes of both expressions being joined.

8. Projection Distribution over Theta-Join Operations

- **Rule (a):** Projection distributes over the theta-join operation under certain conditions involving attribute sets.
- **Rule (b):** Similar equivalences hold for outer join operations $⟕$, $⟖$, and $⟗$.

9. Commutativity and Associativity of Set Operations

- **Rule (a):** Union and intersection operations are commutative.
- **Rule (b):** Union and intersection operations are associative.

10. Selection Distribution over Set Operations

- **Rule (a):** Selection distributes over union, intersection, and set-difference operations.
- **Rule (b):** This distribution does not hold for set difference.

11. Projection Distribution over Union Operation

- **Rule:** Projection distributes over the union operation when expressions have the same schema.

12. Selection Distribution over Aggregation

- **Rule:** Selection distributes over aggregation under specific conditions involving group by attributes and aggregate expressions.

13. Full Outer Join Commutativity and Exchange of Left and Right Outer Join

- **Rule (a):** Full outer join is commutative.
- **Rule (b):** Left outer join and right outer join can be exchanged.

14. Selection Distribution over Left and Right Outer Join

- **Rule:** Selection distributes over left and right outer join under certain conditions involving the selection condition.

15. Replacement of Outer Joins by Inner Joins

- **Rule:** Outer joins can be replaced by inner joins under specific conditions.

16. Selection Distribution over Outer Joins

- **Rule:** Selection distributes over outer joins under certain conditions.