PRACTICAL 1

Aim:

To write DDL Queries to demonstrate the use of objects such as Table, View, Index, Sequence, Synonym, different constraints(Pk, FK, Not NULL Constraints) etc. Write at least 10 queries on the suitable database application using DML statements. Ref: Employees (columns: EmployeeID, FirstName, LastName, DepartmentID, Salary, HireDate) Departments (columns: DepartmentID, DepartmentName)

Objective:

To explore the creation and management of database objects such as tables, views, indexes, sequences, and synonyms, and to enforce constraints like Primary Key, Foreign Key, and NOT NULL.

Theory:

Data Definition Language (DDL):
DDL includes commands that define the structure of database schema objects like tables, views, and indexes. It also handles constraints to enforce data integrity.

- **CREATE TABLE:** Used to create a new table by specifying column definitions and optional constraints.
  Syntax:

```
CREATE TABLE table_name (
  column1 datatype [constraint],
  column2 datatype [constraint],
  );
```

- **CREATE VIEW:** Creates a virtual table based on a SELECT query. Views simplify complex queries and provide security by restricting access to specific data.
  Syntax:

```
CREATE VIEW view_name AS
SELECT columns
FROM table_name
WHERE condition;
```

- **CREATE INDEX:** Creates an index on a table column to optimize query performance.
  Syntax:

```
CREATE INDEX index_name ON table_name (column_name);
```

- **CREATE SEQUENCE:** Automatically generates unique sequential numbers. Commonly used for auto-incrementing primary keys.

Syntax:

```
CREATE SEQUENCE sequence_name
START WITH start_value
INCREMENT BY increment_value;
```

- **CREATE SYNONYM:** Provides an alias for another database object.
  Syntax:

```
CREATE SYNONYM synonym_name FOR object_name;
```

Conclusion:

DDL commands help define and manage the structure of database objects while ensuring data consistency through constraints.

PRACTICAL – 2

Aim:

To write DML Queries : Write at least 10 queries for suitable database application using DML statements –CRUD,sort,Aggregation and indexing queries. Ref: Products (columns: ProductID, ProductName, Category, Price, StockQuantity) Orders (columns: OrderID, OrderDate) OrderItems (columns: OrderItemID, OrderID, ProductID, Quantity)

Objective:

To perform operations like data insertion, retrieval, updating, and deletion using DML commands, along with sorting, aggregation, and indexing.

Theory:

Data Manipulation Language (DML):
DML statements manage the data within database tables. CRUD operations (Create, Read, Update, Delete) and aggregations are essential for manipulating and analyzing data.

- **INSERT INTO:** Adds new rows to a table.
  Syntax:

  ```
  INSERT INTO table_name (column1, column2, ...)
  VALUES (value1, value2, ...);
  ```

- **SELECT:** Retrieves data from a table. Sorting and filtering can be applied using clauses like WHERE and ORDER BY.
  Syntax:

  ```
  SELECT columns
  FROM table_name
  WHERE condition
  ORDER BY column_name;
  ```

- **UPDATE:** Modifies existing rows in a table based on specified conditions.
  Syntax:

  ```
  UPDATE table_name
  SET column_name = value
  WHERE condition;
  ```

- **DELETE:** Removes rows from a table based on a condition.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

- **Aggregate Functions:** Performs calculations on groups of rows. Examples include COUNT, SUM, AVG, MAX, and MIN.
  Syntax:

```
SELECT aggregate_function(column_name)
FROM table_name
GROUP BY column_name;
```

- **INDEX:** Speeds up data retrieval by creating an index on one or more columns.
  Syntax:

```
CREATE INDEX index_name ON table_name (column_name);
```

Conclusion:

DML commands enable the manipulation and retrieval of data, making it possible to maintain and query databases efficiently.

PRACTICAL – 3

Aim:

To write Joins queries : all types of Join, Sub-Query and View Employees (columns: EmployeeID, FirstName, LastName, DepartmentID, Salary) Departments (columns: DepartmentID, DepartmentName) Projects (columns: ProjectID, ProjectName) EmployeeProjects (columns: EmployeeID, ProjectID)

Objective:

To understand how to combine data from multiple tables using JOINs, and how to use subqueries and views for data organization and security.

Theory:

Joins:
Joins are used to combine rows from two or more tables based on a related column.

- **INNER JOIN:** Returns rows with matching values in both tables.
  Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2 ON table1.column = table2.column;
```

- **LEFT JOIN:** Returns all rows from the left table and matching rows from the right table.
  Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2 ON table1.column = table2.column;
```

- **RIGHT JOIN:** Returns all rows from the right table and matching rows from the left table.
  Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2 ON table1.column = table2.column;
```

- **FULL OUTER JOIN:** Combines the results of LEFT and RIGHT JOIN.

  Syntax:

```
SELECT columns
FROM table1
FULL OUTER JOIN table2 ON table1.column = table2.column;
```

Subqueries:
A query nested inside another query, often used to filter or calculate intermediate results.
Syntax:

```
SELECT columns

FROM table_name

WHERE column_name = (SELECT column FROM table_name WHERE condition);
```

Views:
As explained earlier in DDL, views provide a simplified and secure way to represent data.

Conclusion:

Joins, subqueries, and views simplify complex queries, enhance data representation, and optimize database management.

PRACTICAL – 4

Aim:

Stored Procedures: Named PL/Block: PL/Stored Procedure. Products (columns: ProductID, ProductName, Category, Price, StockQuantity) Orders (columns: OrderID, OrderDate) OrderItems (columns: OrderItemID, OrderID, ProductID, Quantity) Create a PL/stored procedure that takes a product ID and a quantity as input and does the following:Checks if the requested quantity is available in stock.If available, updates the stock quantity in the "Products" table and records the order in the "Orders" and "OrderItems" tables.If not available, raises an error message indicating insufficient stock.

Objective:

To create a PL/stored procedure for automating operations like checking stock, updating inventory, and recording orders, which helps in encapsulating logic for reusability and improved performance.

Theory:

Theory: Stored Procedure (Elaborated)

A **stored procedure** is a precompiled set of commands that resides in the database. It is used to perform specific tasks like inserting, updating, deleting, or retrieving data. Stored procedures are often designed to include complex logic and business rules that need to be consistently executed.

Key Features of Stored Procedures

1. **Precompilation**: Once created, the statements within the procedure are compiled and stored. This saves time during execution as the does not need to be parsed or optimized repeatedly.
2. **Modularity**: Logic can be encapsulated in a procedure and reused across multiple applications or modules.
3. **Security**: Stored procedures can limit direct access to the underlying data by exposing only the required logic to the user.
4. **Transaction Management**: Stored procedures allow for handling transactions (e.g., COMMIT, ROLLBACK) within a secure database environment.

Advantages of Stored Procedures (Detailed)

1. **Encapsulation of Logic**: Procedures allow the centralization of business rules, making them easy to maintain and modify without altering multiple application codes.
2. **Improved Performance**: Procedures are optimized and cached in the database after the first execution, reducing the overhead for subsequent executions.
3. **Security**: By providing controlled access to data via procedures, direct queries to tables can be restricted. Only authorized users can execute the procedure.

4. **Reusability**: A single stored procedure can be reused in multiple applications, reducing code duplication and ensuring consistency.

Components of a Stored Procedure Syntax

1. **Procedure Name**:
    - A unique identifier for the stored procedure within the database.
2. **Parameters**:
    - **IN**: Input-only parameter used to pass data to the procedure.
    - **OUT**: Output-only parameter used to return data from the procedure.
    - **IN OUT**: Parameter that passes data to and from the procedure.
3. **Variable Declarations**:
    - Local variables within the procedure to store intermediate results.
4. **Statements**:
    - The core logic of the procedure, including data manipulation (INSERT, UPDATE, DELETE, SELECT) and control statements (IF-ELSE, LOOP).

Syntax for Stored Procedure:

```
CREATE OR REPLACE PROCEDURE procedure_name (parameter1 datatype, parameter2 datatype, ...)

IS

 -- Variable declarations

BEGIN

 -- statements for the operation

 -- Update or retrieve data

END procedure_name;
```

Conclusion:

Stored procedures enhance database management by automating repetitive tasks, promoting consistency, improving performance, and increasing security.

PRACTICAL – 5

Aim:

To create a PL/function that takes a student's ID as input and does the following:Calculates the Grade Point Average (GPA) for the student based on their grades and credit hours.Returns the GPA as the function's result. Student GPA Calculation Consider university database with the following tables: Students (columns: StudentID, FirstName, LastName) Courses (columns: CourseID, CourseName) Grades (columns: GradeID, StudentID, CourseID, Grade, CreditHours)

Objective:

To create a PL/function that performs a specific calculation (such as GPA) and returns the result, encapsulating the logic for easier reuse in queries.

Theory:

A **stored function** is a programmable database object that performs a specific operation and returns a single value as a result. Unlike stored procedures, which are typically invoked to execute operations without returning a value (or returning multiple outputs), a stored function is designed primarily for computation or data transformation and must return exactly one value.

Stored functions can be called in various contexts, such as within a SELECT statement or as part of a conditional clause (WHERE, HAVING). This makes them a powerful tool for modular programming, reusable logic, and efficient data processing.

Key Features of Functions

1. **Return Values**: Functions are **required** to return a value. The RETURN statement is used within the function body to specify what value is returned.
2. **Use in Statements**: Functions can be called directly in queries. For example:

   SELECT function_name(column_name) FROM table_name;

   - They can be utilized in clauses like WHERE, ORDER BY, and HAVING.
3. **Modularity**: Functions encapsulate logic, breaking down complex operations into smaller, reusable components. This modularity makes the code more readable, maintainable, and consistent.
4. **Deterministic vs Non-Deterministic**:
   - **Deterministic Functions**: Always return the same result when given the same inputs. For example, a function that calculates the square of a number.
   - **Non-Deterministic Functions**: May return different results for the same inputs due to varying factors like system state or random number generation.

Advantages of Stored Functions

1. **Reusability**: Functions can be reused across different queries and applications, reducing code duplication.
2. **Efficiency**: By moving complex calculations into the database, functions reduce the overhead on the client-side application.
3. **Encapsulation**: Functions help encapsulate business logic, keeping the database layer consistent and manageable.
4. **Integration with SQL** :Functions can be seamlessly integrated into queries, enabling dynamic calculations or transformations during query execution.

Components of a Stored Function Syntax

A stored function includes several key components, structured as follows:

1. **Function Name**: A unique identifier for the function within the database.
2. **Parameters**: Functions can accept input parameters (optional). These parameters provide input data for the function's logic.
3. **Return Type**: Defines the data type of the value returned by the function (e.g., NUMBER, VARCHAR2).
4. **Local Declarations**: Variables and constants that are used within the function body.
5. **Function Body**: The main logic of the function, including statements, calculations, and the RETURN statement.
6. **Exception Handling**: Optional logic for handling runtime errors within the function.

Syntax for Function:

```
CREATE OR REPLACE FUNCTION function_name (parameter datatype)

RETURN return_datatype

IS

 -- Variable declarations

BEGIN

 -- Perform operations

 RETURN value;

END function_name;
```

Conclusion:

Functions provide an efficient mechanism for performing calculations or data transformations, making them an essential tool for modular database programming.

PRACTICAL – 6

Aim:

To create a PL/trigger that automatically executes when a new order is added to the Orders table. The trigger should do the following:Calculate the loyalty points earned for the order based on the order amount (e.g., 1 point for every $10 spent).Update the LoyaltyPoints field in the Customers table with the calculated points for the customer. Consider a customer database with the following tables: Customers (columns: CustomerID, FirstName, LastName, TotalPurchases, LoyaltyPoints) Orders (columns: OrderID, CustomerID, OrderDate, OrderAmount)

Objective:

To create database triggers that automatically respond to changes in database tables, ensuring that certain tasks (like updating loyalty points) are handled without manual intervention.

Theory:

A **database trigger** is a procedural database object that is automatically executed (or "triggered") when a specific event occurs within a database. These events can include **INSERT**, **UPDATE**, or **DELETE** operations on a table. Triggers allow developers to define automatic responses to such events, which are particularly useful for maintaining data integrity, enforcing complex business rules, and automating routine actions.

Triggers are not called directly; instead, they are activated implicitly by the database system when the specified event occurs.

Types of Triggers

Row-Level Trigger:

- A **row-level trigger** is executed for each row that is affected by the triggering event.
- If an operation modifies multiple rows, the trigger is executed once for each row.
- Common Use Case: Maintaining audit logs for each row change.

Statement-Level Trigger:

- A **statement-level trigger** executes **once per triggering statement**, regardless of the number of rows affected.
- Common Use Case: Performing summary updates or enforcing business rules at a higher level.

Before Trigger:

- A **before trigger** executes **before** the triggering event occurs.

- Common Use Case: Validating or modifying data before it is committed to the database.

After Trigger:

- An **after trigger** executes **after** the triggering event has successfully occurred.
- Common Use Case: Logging changes or performing dependent actions after the event.

Advantages of Triggers

1. **Automatic Execution**:
    - Triggers execute automatically in response to specific database events, reducing manual intervention.
2. **Data Integrity**:
    - Helps maintain consistency and integrity by enforcing constraints or rules.
3. **Auditing and Logging**:
    - Automatically records changes to the database for audit purposes.
4. **Centralized Logic**:
    - Business logic implemented in triggers is centralized, reducing duplication across applications.

Syntax for Trigger:

```
CREATE OR REPLACE TRIGGER trigger_name

BEFORE|AFTER event ON table_name

[FOR EACH ROW]

DECLARE

 -- Variable declarations

BEGIN

 -- Actions to be performed

END trigger_name;
```

Conclusion:

Triggers automate data management tasks and ensure data integrity by responding to changes in tables, reducing the need for manual intervention.

PRACTICAL – 7

Aim:

To create a PL/block that includes the following components: Cursor Declaration: Declare an explicit cursor named employee_cursor to retrieve e employee records from the Employees table. Cursor FOR LOOP: Use a FOR LOOP to iterate through the employees in the cursor. Salary Update: For each employee in the cursor, check if there is a corresponding record in the Salary Updates table with a New Salary value. If such a record exists, update the employee's salary in the Employees table with the new salary value. If there is no record in the Salary Updates table for the employee, leave the salary unchanged. Employees (columns: EmployeeID, FirstName, LastName, Salary) Salary Updates (columns: UpdateID, EmployeeID, NewSalary, UpdateDate)

Objective:

To use PL/cursors to retrieve and process data row by row, providing precise control over large datasets and enabling operations like salary updates for employees.

Theory:

Cursors in PL/provide a mechanism for handling the result set of a query, particularly when dealing with multiple rows of data. A cursor allows for row-by-row processing of query results. This is particularly useful when operations need to be performed on each individual row of the result set, such as updating or inserting data, or performing calculations.

In PL/, a **cursor** acts as a pointer to the context area, where the database stores the result set of the query. Cursors can be either **implicit** (created automatically by Oracle for single-row queries) or **explicit** (defined and controlled by the programmer for multi-row queries).

Types of Cursors in PL/SQL

Implicit Cursor:

- **Implicit cursors** are automatically created by Oracle when a query is executed that returns only one row. These cursors are used primarily for INSERT, UPDATE, DELETE, and single-row SELECT operations.
- Oracle implicitly manages the cursor behind the scenes, so the programmer does not need to declare it.
- Implicit cursors are not suitable for queries that return multiple rows, as they do not provide the control needed to process each row individually.

Explicit Cursor:

- **Explicit cursors** are defined and managed by the programmer, providing more control over how multiple rows are processed.

- An explicit cursor is typically used for queries that return multiple rows.
- The programmer defines the cursor, opens it, fetches rows from it, and finally closes it.
- The cursor allows you to fetch one row at a time, which is useful when you need to process data row by row (such as performing calculations or updates on each row).

Cursor FOR LOOP:

- **Cursor FOR LOOP** is a simplified version of using explicit cursors. It automatically declares, opens, fetches, and closes the cursor, and the loop processes each row returned by the cursor.
- This form eliminates the need for the programmer to explicitly handle cursor operations (like OPEN, FETCH, and CLOSE).
- It is ideal when you need to process all rows of the result set without having to manage the cursor manually.

Parameterized Cursor:

- A **parameterized cursor** allows you to pass parameters to the cursor at the time of its declaration or execution, making it flexible for different queries based on input values.
- It enables you to reuse the cursor for various queries by modifying the parameters dynamically, which adds flexibility and reduces redundancy.

Cursor Operations

1. **Open**:
    - The OPEN operation is used to initialize the cursor and execute the query.
    - Once the cursor is opened, Oracle begins fetching rows from the result set.
    - The OPEN statement binds the query result to the cursor, making it ready for row-by-row processing.

      Syntax: OPEN cursor_name;
2. **Fetch**:
    - The FETCH operation retrieves one row at a time from the cursor's result set.
    - The fetched row is stored in variables or a record that corresponds to the columns selected by the cursor.
    - After fetching a row, you can perform operations on it, such as updating or calculating values, before moving on to the next row.

- If no more rows are available, the FETCH operation will indicate that all rows have been processed.

  Syntax: FETCH cursor_name INTO variable1, variable2;

3. **Close**:
   - The CLOSE operation is used to release the cursor and its resources. After the cursor is closed, it cannot be used again unless it is reopened.
   - Closing the cursor is important for performance and resource management, as cursors consume memory and other database resources.

     Syntax: CLOSE cursor_name;

Syntax for Explicit Cursor:

```
DECLARE
  CURSOR cursor_name IS
    SELECT column1, column2 FROM table_name;
BEGIN
  OPEN cursor_name;
  LOOP
    FETCH cursor_name INTO variable1, variable2;
    EXIT WHEN cursor_name%NOTFOUND;
    -- Operations
  END LOOP;
  CLOSE cursor_name;
END;
```

Conclusion:

Cursors enable detailed, row-wise processing of data, which is essential for tasks requiring fine-grained control over each row, such as updates and complex calculations.

GROUP B

PRACTICAL – 1

Aim:

To design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators etc.).

Objective:

To perform CRUD operations in MongoDB, enabling the insertion, retrieval, updating, and deletion of documents within collections.

Theory:

MongoDB is a **No**(Not Only) database that is known for its flexibility, scalability, and performance. Unlike traditional relational databases, MongoDB stores data in a format known as **BSON** (Binary JSON), which allows it to store structured, semi-structured, and unstructured data as **documents** within **collections**. These documents are JSON-like objects that can store complex data structures, including nested arrays and embedded documents.

The four core operations for managing data in MongoDB are **CRUD operations**, which stand for **Create**, **Read**, **Update**, and **Delete**. These operations form the foundation of all database interactions and allow users to manipulate documents within collections. Below is a detailed explanation of each CRUD operation:

. Create (Insert Documents)

The **Create** operation is used to add new documents to a collection in MongoDB. When inserting a new document, MongoDB automatically generates a unique identifier (_id) for the document if one is not provided. The insertOne() and insertMany() methods are used for inserting a single document or multiple documents, respectively.

- **insertOne()**: Inserts a single document into the collection.
- **insertMany()**: Inserts multiple documents into the collection at once.

Syntax:

```
// Insert a single document
db.collection.insertOne({
  name: "Alice",
  age: 25,
  city: "New York"
});
```

```
// Insert multiple documents
db.collection.insertMany([
  { name: "Bob", age: 30, city: "Los Angeles" },
  { name: "Charlie", age: 22, city: "Chicago" }
]);
```

2. Read (Retrieve Documents)

The **Read** operation is used to retrieve documents from a collection. MongoDB provides a powerful query language to retrieve documents based on various criteria. The find() method is the primary way to retrieve documents from a collection. It can be used with or without filters to specify the query conditions. If no filter is provided, it retrieves all documents in the collection.

- **find()**: Retrieves documents from a collection based on the provided query criteria.
- **findOne()**: Retrieves a single document that matches the query criteria.

Syntax:

```
// Retrieve all documents in the collection
db.collection.find();

db.collection.find({ age: { $gt: 25 } });  // Find documents where age > 25

// Retrieve a single document
db.collection.findOne({ name: "Alice" });
```

3. Update (Modify Existing Documents)

The **Update** operation is used to modify existing documents within a collection. MongoDB provides the updateOne(), updateMany(), and replaceOne() methods to perform update operations.

- **updateOne()**: Updates a single document that matches the query criteria.
- **updateMany()**: Updates multiple documents that match the query criteria.
- **replaceOne()**: Replaces an entire document with a new one, based on the query condition.

MongoDB also provides update operators such as $set, $inc, $push, etc., to specify how the document should be updated.

Syntax:

```
// Update a single document
db.collection.updateOne(
  { name: "Alice" },
// Query condition
  { $set: { age: 26 } }
// Update operation
);

db.collection.updateMany(
  { age: { $gt: 25 } },
// Query condition
]
```

```
  { $set: { status: "active" }
}   // Update operation

);

db.collection.replaceOne(
  { name: "Alice" },
// Query condition

  { name: "Alice", age: 27,
city: "New York" }  // New
document to replace the
old one

);
```

4. Delete (Remove Documents)

The **Delete** operation is used to remove documents from a collection. MongoDB provides the deleteOne() and deleteMany() methods to delete documents that match specified conditions.

- **deleteOne()**: Deletes a single document that matches the query criteria.
- **deleteMany()**: Deletes multiple documents that match the query criteria.

Syntax:

```
// Delete a single document

db.collection.deleteOne({ name: "Alice" });

db.collection.deleteMany({ age: { $lt: 30 }
}); // Delete multiple documents
```

Conclusion:

CRUD operations are essential for managing data in MongoDB, providing an intuitive interface for creating, querying, updating, and deleting documents.

PRACTICAL – 2

Aim:

To design and Develop MongoDB Queries using aggregation and indexing with suitable example using MongoDB.

Objective:

To demonstrate how MongoDB's aggregation framework and indexing can be used to efficiently process and query large datasets.

Theory:

Aggregation Process in MongoDB:

Aggregation in MongoDB typically involves transforming data into a summarized or computed format. The **aggregation framework** provides several stages, each of which performs a specific operation on the data, and the stages are linked together in a **pipeline**.

1. Aggregation Pipeline

The **Aggregation Pipeline** is a series of stages that process data in sequence. Each stage in the pipeline performs an operation on the data and passes the result to the next stage. The pipeline allows developers to build complex queries by combining multiple operations, such as filtering, sorting, grouping, and calculating. The **pipeline** is an efficient and flexible way to query and aggregate data.

Each stage in the aggregation pipeline processes the documents and can filter, group, or reshape the documents, depending on the type of operation defined for that stage. Some of the common stages in an aggregation pipeline are:

- **$match**: Filters the documents based on specified criteria. It is equivalent to a WHERE clause in .
- **$group**: Groups the documents based on specific field(s), and aggregates data using functions like sum, avg, max, min, etc.
- **$sort**: Sorts the documents in a specific order (e.g., ascending or descending).
- **$project**: Specifies which fields to include or exclude from the documents, or creates new computed fields.
- **$limit**: Limits the number of documents returned after the aggregation.

- **$skip**: Skips a specified number of documents before returning the results.
- **$unwind**: Deconstructs an array field from the documents to output one document for each element in the array.
- **$lookup**: Performs a left outer join to another collection in the database.

The stages are executed in the order in which they appear in the pipeline, and the output of each stage becomes the input for the next stage.

Syntax for Aggregation:

```
db.collection.aggregate([
  { $match: {criteria} },    // Filter documents
  { $group: {_id: "$field", total: {$sum: "$value"}} },    // Group and aggregate
  { $sort: {field: 1} }   // Sort results
]);
```

Syntax for Indexing:

```
db.collection.createIndex({field1: 1, field2: -1});
```

Conclusion:

Aggregation and indexing are crucial tools for efficiently processing and querying large datasets in MongoDB. Aggregation transforms and processes data, while indexing optimizes query performance.

PRACTICAL – 3

Aim:

To implement Map reduces operation with suitable example using MongoDB.

Objective:

To implement a MapReduce operation in MongoDB for complex data transformations and aggregations.

Theory:

MapReduce is a programming model used for processing large datasets in parallel and distributed systems. In MongoDB, MapReduce allows users to apply custom functions to process and aggregate data, which is especially useful when performing complex operations that cannot be done with traditional aggregation pipelines.

MapReduce in MongoDB involves two main functions: **Map** and **Reduce**. Here's a more detailed breakdown of each:

1. Map Function:

The **Map** function is responsible for transforming input data into key-value pairs. Each input document is processed by the Map function, which outputs key-value pairs that are grouped together based on the key. These key-value pairs serve as the foundation for the next stage of aggregation.

- **Input**: A collection of documents.
- **Output**: Key-value pairs, where the key is typically the field or value you want to aggregate, and the value is typically some computed data or transformation of the document.

Example: Let's say you have a collection of documents representing purchases:

```
{
  "item": "apple",
  "price": 1.5,
  "quantity": 3
}
```

You could write a Map function to emit key-value pairs where the key is the item name (e.g., "apple"), and the value is the total cost of the item (price * quantity):

```
function mapFunction() {
  emit(this.item, this.price * this.quantity);
}
```

In this case:

- **Key**: The item (e.g., "apple")
- **Value**: The total price for that item (e.g., 1.5 * 3 = 4.5)

2. Reduce Function:

The **Reduce** function takes the output from the Map function (i.e., the key-value pairs) and combines the values that share the same key into a single result. The reduce function aggregates the values based on the key, usually performing operations like summing, averaging, or concatenating the values.

- **Input**: The output of the Map function, a set of key-value pairs.
- **Output**: A single value that aggregates the data based on the key.

Example: For the Map function above, you might want to sum up the total value for each item. The Reduce function would aggregate the values for each item (key) and calculate the total:

```
function reduceFunction(key, values) {
  return Array.sum(values);  // Sum all the values for each key
}
```

In this case:

- **Input**: The values for a given item (e.g., [4.5, 5.0, 6.0]).
- **Output**: The sum of those values (e.g., 4.5 + 5.0 + 6.0 = 15.5).

3. Example in MongoDB:

Now, let's see how this works in MongoDB. Suppose you want to use MapReduce to calculate the total sales for each item in a collection of purchases:

```
db.purchases.mapReduce(
  function() { emit(this.item, this.price * this.quantity); },  // Map function
  function(key, values) { return Array.sum(values); },        // Reduce function
  { out: "total_sales" }                                      // Output collection
);
```

Conclusion:

MapReduce allows for advanced data processing and aggregation tasks, providing flexibility in MongoDB when built-in aggregation operators are insufficient.