Documentation for basic.sql

Introduction

This document provides a detailed explanation of the SQL commands found in the `basic.sql` file. Each section covers a specific SQL concept, providing syntax from the file and a real-life example to illustrate its usage. This script demonstrates database creation, table management, data manipulation, and various querying techniques.

---

1. Database Operations

`CREATE DATABASE`
- **Purpose:** Creates a new database.
- **Syntax:**
```sql
CREATE DATABASE Collection;
CREATE DATABASE IF NOT EXISTS temp1;
```
- **Real-life Example:** Imagine you are building a new e-commerce website. You would start by creating a database to hold all your product, customer, and order information.
```sql
CREATE DATABASE eCommerceDB;
```

`DROP DATABASE`
- **Purpose:** Deletes an existing database. **Caution:** This action is irreversible and will delete all tables and data within the database.
- **Syntax:**
```sql
DROP DATABASE IF EXISTS temp1;
```
- **Real-life Example:** If you have a temporary or outdated database for a project that is no longer in development, you can delete it to free up server resources.
```sql
DROP DATABASE old_project_db;
```

`SHOW DATABASES`
- **Purpose:** Lists all available databases on the server.
- **Syntax:**
```sql
SHOW DATABASES;
```
- **Real-life Example:** A database administrator can use this command to get a quick overview of all databases currently managed by the server.

`USE`
- **Purpose:** Selects a database to work with. All subsequent SQL statements will be executed in the context of this database.
- **Syntax:**
```sql
USE Collection;
```
- **Real-life Example:** Before you can create tables for your e-commerce site, you must select the `eCommerceDB` to ensure the tables are created in the correct place.
```sql
USE eCommerceDB;
```

```

---

## 2. Table Operations

### `CREATE TABLE`
- **Purpose:** Creates a new table to store data.
- **Syntax:**
```sql
CREATE TABLE employee(
id INT PRIMARY KEY,
name VARCHAR(50),
dept VARCHAR(50),
university VARCHAR(50),
age INT NOT NULL,
salary INT
);
```
- **Real-life Example:** In an e-commerce database, you would create a `products` table to store information about each item you sell.
```sql
CREATE TABLE products (
product_id INT PRIMARY KEY,
product_name VARCHAR(100) NOT NULL,
price DECIMAL(10, 2),
stock_quantity INT
);
```

`ALTER TABLE`
- **Purpose:** Modifies an existing table's structure.
- **Sub-clauses:**
- `ADD COLUMN`: Adds a new column.
- `DROP COLUMN`: Removes a column.
- `RENAME TO`: Renames the table.
- `CHANGE COLUMN`: Renames and/or changes the data type of a column.
- **Syntax:**
```sql
-- Add new columns
ALTER TABLE employee
ADD COLUMN email VARCHAR(50) DEFAULT 'demo@gmail.com',
ADD COLUMN phone VARCHAR(50) DEFAULT '01512345678';

-- Drop a column
ALTER TABLE employee
DROP COLUMN phone;

-- Rename the table
ALTER TABLE employee
RENAME TO emp;

-- Change a column's name and type
ALTER TABLE emp
CHANGE COLUMN university uni VARCHAR(50);
```
- **Real-life Example:** If you decide to start tracking the manufacturer of your products, you can add a `manufacturer` column to your `products` table.
```sql

```
ALTER TABLE products
ADD COLUMN manufacturer VARCHAR(100);
```

`TRUNCATE TABLE`
- **Purpose:** Quickly deletes all rows from a table. It is faster than `DELETE` and cannot be rolled back.
- **Syntax:**
```sql
TRUNCATE TABLE emp;
```
- **Real-life Example:** Before running a large data import, you might want to clear a staging table of all its previous data.
```sql
TRUNCATE TABLE staging_import_data;
```

---

3. Data Manipulation Language (DML)

`INSERT INTO`
- **Purpose:** Adds new rows of data into a table.
- **Syntax:**
```sql
-- Inserting values for all columns
INSERT INTO employee VALUES(1, 'shaker', 'Java Baceknd', 'BUBT', 24, 35000);

-- Inserting values for specific columns
INSERT INTO employee (id, name, dept, university, age, salary)
VALUES (5, 'Dayan', 'UI', 'BUBT', 27, 30000);
```
- **Real-life Example:** When a new user signs up on your website, you insert their details into the `users` table.
```sql
INSERT INTO users (username, email, password_hash, registration_date)
VALUES ('john_doe', 'john.doe@example.com', 'hashed_password_string', NOW());
```

`UPDATE`
- **Purpose:** Modifies existing records in a table.
- **Syntax:**
```sql
UPDATE employee
SET university = 'DU'
WHERE id > 4;
```
- **Real-life Example:** If a user changes their shipping address, you update their record in the `customers` table.
```sql
UPDATE customers
SET shipping_address = '123 New Street, Anytown'
WHERE customer_id = 101;
```

`DELETE`
- **Purpose:** Removes existing records from a table.
- **Syntax:**
```sql
```

```sql
DELETE FROM employee
WHERE salary > 90000;
```
- **Real-life Example:** When a customer cancels their account, you might delete their data from the `users` table (subject to data retention policies).
```sql
DELETE FROM users
WHERE user_id = 57;
```

---

4. Data Querying and Clauses

`SELECT`
- **Purpose:** Retrieves data from one or more tables.
- **Syntax:**
```sql
-- Select all columns
SELECT * FROM employee;

-- Select specific columns with a condition
SELECT name FROM employee
WHERE name = 'Rifat';
```
- **Real-life Example:** To display all available products on your e-commerce homepage, you would select them from the `products` table.
```sql
SELECT product_name, price, image_url FROM products
WHERE stock_quantity > 0;
```

`ORDER BY` and `LIMIT`
- **Purpose:**
- `ORDER BY`: Sorts the result set in ascending (`ASC`) or descending (`DESC`) order.
- `LIMIT`: Restricts the number of rows returned.
- **Syntax:**
```sql
SELECT * FROM employee
ORDER BY salary DESC
LIMIT 3;
```
- **Real-life Example:** To find the top 5 most expensive products in your store:
```sql
SELECT product_name, price FROM products
ORDER BY price DESC
LIMIT 5;
```

Aggregate Functions
- **Purpose:** Perform a calculation on a set of values and return a single value.
- **Common Functions:** `COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`.
- **Syntax:**
```sql
SELECT COUNT(name) FROM employee;
SELECT MAX(salary) FROM employee;
SELECT AVG(salary) FROM employee;
```
- **Real-life Example:** A business manager might want a summary report of sales data.

```sql
SELECT
COUNT(order_id) AS total_orders,
SUM(order_total) AS total_revenue,
AVG(order_total) AS average_order_value
FROM sales;
```

`GROUP BY`
- **Purpose:** Groups rows that have the same values in specified columns into summary rows. Often used with aggregate functions.
- **Syntax:**
```sql
SELECT university, COUNT(salary) FROM employee
GROUP BY university;
```
- **Real-life Example:** To count the number of products in each category:
```sql
SELECT category, COUNT(product_id) AS number_of_products
FROM products
GROUP BY category;
```

`HAVING`
- **Purpose:** Filters results after aggregation. While `WHERE` filters rows before grouping, `HAVING` filters groups.
- **Syntax:**
```sql
SELECT COUNT(id), university FROM employee
GROUP BY university
HAVING MAX(salary) > 30000;
```
- **Real-life Example:** To find product categories where the average product price is over $100.
```sql
SELECT category, AVG(price) FROM products
GROUP BY category
HAVING AVG(price) > 100;
```

---

5. Joins

`INNER JOIN`
- **Purpose:** Returns records that have matching values in both tables.
- **Syntax:**
```sql
SELECT * FROM student
INNER JOIN course ON student.id = course.id;
```
- **Real-life Example:** To get a list of customers and the orders they have placed.
```sql
SELECT customers.customer_name, orders.order_date
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id;
```

`LEFT JOIN` (Outer Join)

- **Purpose:** Returns all records from the left table, and the matched records from the right table. The result is NULL from the right side if there is no match.
- **Syntax:**
```sql
SELECT * FROM student AS a
LEFT JOIN course AS b ON a.id = b.id;
```
- **Real-life Example:** To list all customers and any orders they may have placed. This will include customers who have never placed an order.
```sql
SELECT customers.customer_name, orders.order_id
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

`RIGHT JOIN` (Outer Join)
- **Purpose:** Returns all records from the right table, and the matched records from the left table. The result is NULL from the left side when there is no match.
- **Syntax:**
```sql
SELECT * FROM student AS a
RIGHT JOIN course AS b ON a.id = b.id;
```
- **Real-life Example:** To show all orders and the customer who placed them. This would include any (unlikely) orders that are not associated with a customer.
```sql
SELECT customers.customer_name, orders.order_id
FROM customers
RIGHT JOIN orders ON customers.customer_id = orders.customer_id;
```

`FULL JOIN` (Simulated with `UNION`)
- **Purpose:** Returns all records when there is a match in either the left or right table.
- **Syntax:**
```sql
SELECT * FROM student AS a
LEFT JOIN course AS b ON a.id = b.id
UNION
SELECT * FROM student AS a
RIGHT JOIN course AS b ON a.id = b.id;
```
- **Real-life Example:** You want a combined list of all employees and all departments, matching them where possible, but including all employees without a department and all departments without employees.

`SELF JOIN`
- **Purpose:** A regular join, but the table is joined with itself.
- **Syntax:**
```sql
SELECT a.name AS manager_name, b.name AS employee_name
FROM employee AS a
JOIN employee AS b ON a.id = b.manager_id;
```
- **Real-life Example:** In an `employees` table where one column is `employee_id` and another is `manager_id`, you can use a self-join to display each employee's name next to their manager's name.

---

6. Advanced Topics

`UNION`
- **Purpose:** Combines the result-set of two or more `SELECT` statements. It removes duplicate rows. `UNION ALL` includes duplicates.
- **Syntax:**
```sql
SELECT name FROM employee
UNION
SELECT name FROM employee;
```
- **Real-life Example:** To create a single mailing list from your `customers`, `suppliers`, and `partners` tables.
```sql
SELECT email FROM customers
UNION
SELECT email FROM suppliers;
```

Subqueries
- **Purpose:** A query nested inside another query.
- **Syntax:**
```sql
-- Subquery in the WHERE clause
SELECT full_name FROM emp
WHERE salary > (SELECT AVG(salary) FROM emp);

-- Subquery in the FROM clause
SELECT MAX(salary)
FROM (SELECT * FROM emp WHERE uni = 'DU') AS temp;
```
- **Real-life Example:** Find all products whose price is higher than the average price of all products.
```sql
SELECT product_name, price FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

`VIEW`
- **Purpose:** A virtual table based on the result-set of an SQL statement.
- **Syntax:**
```sql
CREATE VIEW view1 AS
SELECT id, full_name, uni FROM emp;
```
- **Real-life Example:** You can create a view to simplify a complex query that joins multiple tables. For instance, a `customer_order_summary` view could provide a denormalized look at customer and order data without having to write the join every time.
```sql
CREATE VIEW customer_order_summary AS
SELECT c.customer_name, p.product_name, o.order_date
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN products p ON o.product_id = p.product_id;

-- Now you can query the view easily
SELECT * FROM customer_order_summary WHERE customer_name = 'John Doe';
```
