SQL Basics to Advanced: Cheat Sheet and Notes by Durvesh Sunil Baharwal

Welcome to my SQL notes! These notes are a compilation of key concepts, best practices, and insights that I've gathered throughout my learning journey in SQL. Whether you're just starting out or looking to deepen your understanding, I hope you'll find these notes helpful.

I'm passionate about data analysis, artificial intelligence, and leveraging technology to solve complex problems. These notes are part of my continuous effort to share knowledge and contribute to the tech community.

1. Introduction to SQL

SQL (Structured Query Language) is a standard programming language used to interact with and manage relational databases. It allows users to create, read, update, and delete data within a database, making it an essential tool for data management. SQL is highly structured and provides a straightforward syntax for executing queries, making it accessible for both beginners and experienced database professionals.

2. Types of SQL Commands

SQL commands are categorized into several types based on their functionality:

DDL (Data Definition Language): Defines the structure of the database schema.

1. **CREATE**: Creates a new table, database, index, or view.
2. **ALTER**: Modifies an existing database object like a table.
3. **DROP**: Deletes objects like tables or databases.

DML (Data Manipulation Language): Manipulates the data stored in the database.

4. **INSERT**: Adds new records to a table.
5. **UPDATE**: Modifies existing data in a table.
6. **DELETE**: Removes records from a table.

DQL (Data Query Language): Queries the database to retrieve data.

7. **SELECT**: Retrieves data from one or more tables.

DCL (Data Control Language): Controls access to the data in the database.

8. **GRANT**: Provides users with access privileges.
9. **REVOKE**: Removes access privileges from users.

TCL (Transaction Control Language): Manages transactions within the database.

10. **COMMIT**: Saves all changes made during the current transaction.
11. **ROLLBACK**: Reverts changes made during the current transaction.
12. **SAVEPOINT**: Sets a savepoint within a transaction to roll back to later.

3. Data Types

SQL supports various data types to store different kinds of data:

- **Numeric Types**: `INT`, `FLOAT`, `DECIMAL`, `NUMERIC`
- **Character/String Types**: `CHAR`, `VARCHAR`, `TEXT`
- **Date/Time Types**: `DATE`, `TIME`, `DATETIME`, `TIMESTAMP`
- **Binary Types**: `BINARY`, `VARBINARY`
- **Boolean Types**: `BOOLEAN`

4. Basic SQL Syntax and Uses

CREATE

Used to create new database objects like tables, indexes, views, or databases

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    ...
    columnN datatype constraint
);
```

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    HireDate DATE,
    Salary DECIMAL(10, 2));
```

ALTER

Used to modify an existing database object like a table by adding, deleting, or modifying columns

```
ALTER TABLE table_name
ADD column_name datatype;

ALTER TABLE table_name
DROP COLUMN column_name;

ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

Example:

```
ALTER TABLE Employees
ADD Email VARCHAR(100);

ALTER TABLE Employees
DROP COLUMN Salary;

ALTER TABLE Employees
MODIFY COLUMN LastName VARCHAR(100);
```

DROP

Used to delete entire database objects like tables, views, or databases. This operation is irreversible.

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE Employees;
```

INSERT INTO

Insert new records into a table.

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

Example:

```
INSERT INTO employees (first_name,
last_name, department)
VALUES ('John', 'Doe', 'Marketing');
```

UPDATE

Update existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2,
...
WHERE condition;
```

Example:

```
UPDATE employees
SET salary = 60000
WHERE employee_id = 101;
```

DELETE

Delete records from a table.

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

Example:

```
DELETE FROM employees
WHERE department = 'HR';
```

SELECT

Retrieve data from a database.

```
SELECT column1, column2, ...
FROM table_name;
```

Example:

```
SELECT first_name, last_name
FROM employees;
```

DESCRIBE

The DESCRIBE or DESC command is used to display the structure of a table, including its columns, data types, and constraints. This command provides a quick overview of the table's schema.

```
DESCRIBE table_name;
```

Example:

```
DESCRIBE employees;
```

ORDER BY

Sort the result set in ascending or descending order.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 ASC|DESC;
```

Example:

```
SELECT first_name, last_name
FROM employees
ORDER BY last_name ASC;
```

AND, OR, NOT

Combine multiple conditions.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2;

SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2;

SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

Example:

```
SELECT first_name, last_name
FROM employees
WHERE department = 'HR' AND salary >
50000;

SELECT first_name, last_name
FROM employees
WHERE department = 'HR' OR salary >
50000;

SELECT first_name, last_name
FROM employees
WHERE department = 'HR' NOT salary >
50000;
```

LIMIT

Limit the number of records returned.

```
SELECT column1, column2, ...
FROM table_name
LIMIT number;
```

Example:

```
SELECT first_name, last_name
FROM employees
LIMIT 10;
```

LIKE

Search for a specified pattern in a column.

```
SELECT column1, column2, ...
FROM table_name
WHERE column1 LIKE pattern;
```

Example:

```
SELECT first_name, last_name
FROM employees
WHERE first_name LIKE 'J%';
```

IN

Specify multiple values in a WHERE clause.

```
SELECT column1, column2, ...
FROM table_name
WHERE column1 IN (value1, value2, ...);
```

Example:

```
SELECT first_name, last_name
FROM employees
WHERE department IN ('HR', 'Marketing');
```

BETWEEN

Select values within a range.

```
SELECT column1, column2, ...
FROM table_name
WHERE column1 BETWEEN value1 AND value2;
```

Example:

```
SELECT first_name, last_name
FROM employees
WHERE salary BETWEEN 50000 AND 70000;
```

DISTINCT

Return only distinct (unique) values.

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

Example:

```
SELECT DISTINCT department
FROM employees;
```

ALIAS

Give a table or a column a temporary name.

```
SELECT column_name AS alias_name
FROM table_name AS alias_name;
```

Example:

```
SELECT first_name AS fname, last_name AS
lname
FROM employees AS emp;
```

5. Aggregate Functions

COUNT

Return the number of rows that match a condition.

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT COUNT(employee_id)
FROM employees
WHERE department = 'HR';
```

SUM

Return the total sum of a numeric column.

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT SUM(salary)
FROM employees
WHERE department = 'HR';
```

AVG

Return the average value of a numeric column.

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT AVG(salary)
FROM employees
WHERE department = 'HR';
```

MIN

Return the smallest value in a column.

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT MIN(salary)
FROM employees;
```

MAX

Return the largest value in a column.

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT MAX(salary)
FROM employees;
```

6. String Functions

CONCAT

Concatenates two or more strings into one string.

```
CONCAT(string1, string2, ...);
```

Example:

```
SELECT CONCAT('Hello', ' ', 'World') AS
Greeting;
```

SUBSTRING

Extracts a substring from a string, starting from a specified position.

```
SUBSTRING(string, start_position,
length);
```

Example:

```
SELECT SUBSTRING('Hello World', 7, 5) AS
ExtractedString;
```

LENGTH

Returns the length of a string.

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT AVG(salary)
FROM employees
WHERE department = 'HR';
```

UPPER

Converts all characters in a string to uppercase.

```
UPPER(string);
```

Example:

```
SELECT UPPER('Hello World') AS Upper-
caseString;
```

LOWER

Converts all characters in a string to lowercase.

```
LOWER(string);
```

Example:

```
SELECT LOWER('Hello World') AS Lower-
caseString;
```

7. Date Functions

NOW

Returns the current date and time.

```
NOW();
```

Example:

```
SELECT NOW() AS CurrentDateTime;
```

CURDATE

Returns the current date.

```
CURDATE();
```

Example:

```
SELECT CURDATE() AS CurrentDate;
```

DATEDIFF

Returns the length of a string.

```
DATEDIFF(date1, date2);
```

Example:

```
SELECT DATEDIFF('2024-12-31', '2024-01-
01') AS DaysDifference;
```

DATEADD

Adds a specified time interval to a date.

```
DATEADD(interval, number, date);
```

Example:

```
SELECT DATEADD(DAY, 10, '2024-08-24') AS
NewDate;
```

8. Numeric Functions

ROUND

Rounds a number to a specified number of decimal places.

```
ROUND(number, decimal_places);
```

Example:

```
SELECT ROUND(123.4567, 2) AS Rounded-
Number;
```

ABS

Returns the absolute value of a number.

```
ABS(number);
```

Example:

```
SELECT ABS(-123) AS AbsoluteValue;
```

MOD

Returns the remainder of a division operation.

```
MOD(dividend, divisor);
```

Example:

```
SELECT MOD(10, 3) AS Modulus;
```

CEILING

Returns the smallest integer greater than or equal to a number.

```
CEILING(number);
```

Example:

```
SELECT CEILING(123.456) AS CeilingValue;
```

FLOOR

Returns the largest integer less than or equal to a number.

```
FLOOR(number);
```

Example:

```
SELECT FLOOR(123.456) AS FloorValue;
```

9. Conversion Functions

CAST

Converts an expression from one data type to another.

```
CAST(expression AS target_data_type);
```

Example:

```
SELECT CAST('123' AS INT) AS Convert-
edValue;
```

CONVERT

Converts an expression from one data type to another, with an optional style argument.

```
CONVERT(target_data_type, expression, style);
```

Example:

```
SELECT CONVERT(DATE, '2024-08-24', 23) AS ConvertedDate;
```

10. Window Functions

ROW_NUMBER()

Assigns a unique sequential integer to rows within a partition of a result set.

```
SELECT column1, ROW_NUMBER() OVER (PARTITION BY column2 ORDER BY column3) AS row_num
FROM table_name;
```

RANK()

Assigns a rank to each row within a partition, with gaps between ranks where there are ties.

```
SELECT column1, RANK() OVER (PARTITION BY column2 ORDER BY column3) AS rank
FROM table_name;
```

DENSE_RANK()

Similar to RANK(), but without gaps in the ranking sequence.

```
SELECT column1, DENSE_RANK() OVER (PARTITION BY column2 ORDER BY column3) AS dense_rank
FROM table_name;
```

NTILE()

Distributes rows into a specified number of groups, and assigns a group number to each row.

```
SELECT column1, NTILE(4) OVER (ORDER BY column2) AS quartile
FROM table_name;
```

Example:

```
SELECT employee_id, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
       RANK() OVER (ORDER BY salary DESC) AS rank,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank,
       NTILE(4) OVER (ORDER BY salary DESC) AS quartile
FROM employees;
```

11. Joins

INNER JOIN

Select records that have matching values in both tables.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Example:

```
SELECT employees.first_name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

LEFT JOIN (or LEFT OUTER JOIN)

Return all records from the left table and matched records from the right table.

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.col-
umn_name;
```

Example:

```
SELECT employees.first_name, depart-
ments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = depart-
ments.department_id;
```

RIGHT JOIN (or RIGHT OUTER JOIN)

Return all records from the right table and matched records from the left table.

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.col-
umn_name;
```

Example:

```
SELECT employees.first_name, depart-
ments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = depart-
ments.department_id;
```

FULL JOIN (or FULL OUTER JOIN)

Return all records when there is a match in either left or right table.

```
SELECT column_name(s)
FROM table1
FULL JOIN table2
ON table1.column_name = table2.col-
umn_name;
```

Example:

```
SELECT employees.first_name, depart-
ments.de
partment_name
FROM employees
FULL JOIN departments
ON employees.department_id = depart-
ments.department_id;
```

CROSS JOIN

Return the Cartesian product of the two tables.

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```

Example:

```
SELECT employees.first_name, depart-
ments.department_name
FROM employees
CROSS JOIN departments;
```

SELF JOIN

A self join is a regular join but the table is joined with itself. Useful for querying hierarchical data or comparing rows within the same table.

```
SELECT A.column1, B.column2, ...
FROM table_name A
JOIN table_name B
ON A.common_column = B.common_column;
```

Example:

```
SELECT A.employee_id, A.first_name,
B.manager_id, B.first_name AS man-
ager_name
FROM employees A
JOIN employees B ON A.manager_id = B.em-
ployee_id;
```

ANTI JOIN

Returns rows from the first table where there are no matching rows in the second table. Often implemented using a LEFT JOIN with a WHERE clause checking for NULL values.

```
SELECT A.column1, B.column2, ...
FROM table_name A
LEFT JOIN table_name B
ON A.common_column = B.common_column;
WHERE B.table_name IS NULL;
```

Example:

```
 SELECT A.employee_id, A.first_name
FROM employees A
LEFT JOIN projects B ON A.employee_id =
B.employee_id
WHERE B.employee_id IS NULL;
```

SEMI JOIN

Returns rows from the first table where there is a matching row in the second table, but does not return rows from the second table..

```
SELECT A.column1, B.column2, ...
FROM table_name1 A
WHERE EXISTS (
    SELECT 1
    FROM table_name2 B
    WHERE A.column1 = B.column1
);
```

Example:

```
SELECT A.employee_id, A.first_name
FROM employees A
WHERE EXISTS (
    SELECT 1
    FROM projects B
    WHERE A.employee_id = B.employee_id
);
```

12. Subqueries

A query nested inside another query.

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name = (SELECT column_name
FROM table_name WHERE condition);
```

Example:

```
SELECT first_name, last_name
FROM employees
WHERE salary > (SELECT AVG(salary) FROM
employees);
```

Temporary Tables

Global Temporary Tables:

Accessible by any user and session. The data is visible to all sessions until the table is dropped.

```
CREATE GLOBAL TEMPORARY TABLE temp_table_name (...);
```

Local Temporary Tables:

Accessible only within the session that created it. Automatically dropped when the session ends.

```
CREATE TEMPORARY TABLE temp_table_name (...);
```

Example:

```
CREATE TEMPORARY TABLE temp_employees (
    employee_id INT,
    first_name VARCHAR(50)
);

INSERT INTO temp_employees (employee_id, first_name)
VALUES (1, 'John'), (2, 'Jane');

SELECT * FROM temp_employees;
```

WITH Clause and Common Table Expressions (CTEs)

Temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. Simplifies complex queries by breaking them down into simpler parts.

```
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT *
FROM cte_name
WHERE condition;
```

Example:

```
WITH EmployeeCTE AS (
    SELECT employee_id, first_name, department_id
    FROM employees
    WHERE salary > 50000
)
SELECT *
FROM EmployeeCTE
WHERE department_id = 1;
```

13. Grouping Data

GROUP BY

Group rows that have the same values in specified columns.

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

Example:

```
SELECT department, COUNT(employee_id)
FROM employees
GROUP BY department;
```

WHERE

Filter records before grouping.

```
SELECT column_name(s), aggregate_func-
tion(column_name)
FROM table_name
WHERE condition
GROUP BY column_name(s);
```

Example:

```
SELECT department, COUNT(employee_id)
FROM employees
WHERE salary > 50000
GROUP BY department;
```

HAVING

Filter records after grouping.

```
SELECT column_name(s), aggregate_func-
tion(column_name)
FROM table_name
GROUP BY column_name(s)
HAVING condition;
```

Example:

```
SELECT department, COUNT(employee_id)
FROM employees
GROUP BY department
HAVING COUNT(employee_id) > 10;
```

EXCEPT

The EXCEPT operator is used to return the rows from the first SELECT statement that are not present in the second SELECT statement. It is essentially a subtraction operation between two result sets.

```
SELECT column1, column2, ...
FROM table1
EXCEPT
SELECT column1, column2, ...
FROM table2;
```

Example:

```
SELECT employee_id, first_name, last_name
FROM employees
EXCEPT
SELECT employee_id, first_name, last_name
FROM retired_employees;
```

Note: EXCEPT is supported in some SQL databases like SQL Server. In other systems like MySQL, you might use NOT IN or a LEFT JOIN with WHERE clause as an alternative.

UNION

The UNION operator is used to combine the results of two or more SELECT statements into a single result set. The UNION operator removes duplicate rows unless you use UNION ALL, which includes duplicates. The number and order of columns must be the same in both queries, and the data types must be compatible.

Syntax:

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

Example:

```
SELECT first_name, last_name, department
FROM employees
WHERE department = 'HR'
UNION
SELECT first_name, last_name, department
FROM contractors
WHERE department = 'HR';
```

UNION ALL:

```
SELECT first_name, last_name, department
FROM employees
WHERE department = 'HR'
UNION ALL
SELECT first_name, last_name, department
FROM contractors
WHERE department = 'HR';
```

This query returns the same combined list, but without removing duplicates

INTERSECT

The INTERSECT operator is used to return only the rows that are common to both SELECT statements. It is essentially an intersection operation between two result sets.

Like UNION, the number and order of columns must be the same in both queries, and the data types must be compatible.

Syntax:

```
SELECT column1, column2, ...
FROM table1
INTERSECT
SELECT column1, column2, ...
FROM table2;
```

Example:

```
SELECT employee_id, first_name, last_name
FROM employees
INTERSECT
SELECT employee_id, first_name, last_name
FROM retired_employees;
```

14. Advanced Topics

a. Views

A virtual table based on the result-set of an SQL statement.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
CREATE VIEW high_salary_employees AS
SELECT first_name, last_name, salary
FROM employees
WHERE salary > 70000;
```

b. Indexes

Indexes in SQL are used to improve the speed of data retrieval operations by creating pointers to the data stored in tables. They allow the database to find rows more quickly, without having to scan the entire table.

Types of Indexes:

i. **Primary Index**: Automatically created when a primary key is defined. It ensures that the column(s) defined as the primary key is unique and indexed.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50));
```

ii. **Unique Index**: Ensures that all the values in the indexed column are unique, similar to the primary key but can be applied to any column.

Example:

```
CREATE UNIQUE INDEX idx_unique_email ON Employees (Email);
```

iii. **Composite Index**: An index on multiple columns of a table. It is useful for queries that filter or sort on multiple columns.

Example:

```
CREATE INDEX idx_name ON Employees (LastName, FirstName);
```

iv. **Full-text Index**: Used to speed up searches on large text fields. It allows for more complex queries like searching for words within a text.

Example:

```
CREATE FULLTEXT INDEX idx_fulltext_bio ON Employees (Bio);
```

Advanced Indexing:

i. **Clustered Index**: Sorts and stores the data rows in the table based on the index key. Only one clustered index can be created per table, as the data rows themselves can only be sorted in one order.

Example:

```
CREATE CLUSTERED INDEX idx_employee_id ON employees(employee_id);
```

ii. **Non-Clustered Index**: Does not alter the physical order of the table; instead, creates a separate structure within the table. Multiple non-clustered indexes can be created on a single table.

Example:

```
CREATE NONCLUSTERED INDEX idx_last_name ON employees(last_name);
```

iii. Bitmap Indexes: Used primarily in data warehousing environments where AND, OR, and NOT operations are frequently used. Uses bitmap vectors to represent data, which are efficient for low-cardinality columns.

Example:

```
CREATE BITMAP INDEX idx_gender ON employees(gender);
```

c. Partitioning

Table Partitioning: Divides a large table into smaller, more manageable pieces while retaining the overall structure of the table. Helps in improving query performance and ease of maintenance.

```
CREATE TABLE sales (
    sale_id INT,
    sale_date DATE,
    amount DECIMAL(10, 2)
)
PARTITION BY RANGE (sale_date) (
    PARTITION p0 VALUES LESS THAN ('2023-
01-01'),
    PARTITION p1 VALUES LESS THAN ('2024-
01-01'),
    PARTITION p2 VALUES LESS THAN
(MAXVALUE)
);
```

Example:

```
CREATE TABLE orders (
    order_id INT,
    order_date DATE,
    customer_id INT
)
PARTITION BY RANGE (order_date) (
    PARTITION p1 VALUES LESS THAN ('2022-
01-01'),
    PARTITION p2 VALUES LESS THAN ('2023-
01-01'),
    PARTITION p3 VALUES LESS THAN ('2024-
01-01')
);
```

d. Constraints

Constraints in SQL enforce rules at the database level to ensure data integrity and accuracy. They prevent invalid data from being entered into the database.

Types of Constraints:

i. **Primary Key:** Uniquely identifies each record in a table. Each table can have only one primary key, and it cannot contain NULL values.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);
```

ii. **Foreign Key**: Links one table to another by referencing the primary key in another table. It ensures referential integrity.

Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    EmployeeID INT,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
);
```

iii. **UNIQUE**: Ensures all values in a column or a set of columns are unique across the table.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE
);
```

iv. **NOT NULL**: Ensures that a column cannot have a NULL value.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL
);
```

v. **CHECK**: Ensures that all values in a column satisfy a specific condition..

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Age INT CHECK (Age >= 18)
);
```

vi. **DEFAULT**: Provides a default value for a column when no value is specified.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    HireDate DATE DEFAULT GETDATE()
);
```

e. GRANT and REVOKE

The GRANT and REVOKE commands in SQL are used to manage permissions for database users, allowing or restricting their access to specific database objects like tables, views, and procedures.

GRANT:

The GRANT command is used to give specific privileges to users or roles. These privileges can include rights to SELECT, INSERT, UPDATE, DELETE data, or even administrative privileges like creating tables or altering schemas.

```
GRANT privilege_name ON object_name TO {user_name | PUBLIC | role_name};
```

Example:

```
GRANT SELECT, INSERT ON Employees TO JohnDoe;
```

WITH GRANT OPTION: Allows the user to grant the same privileges to others.

```
GRANT SELECT ON Employees TO JohnDoe WITH GRANT OPTION;
```

REVOKE:

The REVOKE command is used to remove previously granted privileges from users or roles. This ensures that a user no longer has access to perform certain actions on the database objects.

```
REVOKE privilege_name ON object_name FROM {user_name | PUBLIC | role_name};
```

Example:

```
REVOKE SELECT, INSERT ON Employees FROM JohnDoe;
```

f. Stored Procedures

A prepared SQL code that you can save and reuse.

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    SQL_statement;
END;
```

Example:

```
CREATE PROCEDURE getEmployees
AS
BEGIN
    SELECT first_name, last_name FROM em-
ployees;
END;
```

Stored Procedures VS Functions

Stored Procedure: A set of SQL statements that perform a specific task, such as inserting data or updating data.

Can perform actions like INSERT, UPDATE, DELETE. Can return zero or more results, but does not return a value.

Example:

```
CREATE PROCEDURE getEmployeeData()
AS
BEGIN
    SELECT * FROM employees;
END;
```

Function: A routine that can perform calculations and return a single value.

Typically used for operations like mathematical calculations or returning a single result. Cannot perform actions like INSERT, UPDATE, DELETE (except in some SQL dialects).

Example:

```
CREATE FUNCTION getEmployeeCount()
RETURNS INT
AS
BEGIN
    RETURN (SELECT COUNT(*) FROM employ-
ees);
END;
```

Recursive Stored Procedure: A stored procedure that calls itself until a specified condition is met.

Example:

```
CREATE PROCEDURE CalculateFactorial(@Number INT, @Result INT OUTPUT)
AS
BEGIN
    IF @Number = 1
        SET @Result = 1;
    ELSE
    BEGIN
        DECLARE @Temp INT;
        SET @Temp = @Number - 1;
        EXEC CalculateFactorial @Temp, @Result OUTPUT;
        SET @Result = @Result * @Number;
    END;
END;
```

15. Triggers

A Trigger in SQL is a set of commands that automatically executes (or "fires") in response to certain events on a table or view. Triggers are used to enforce business rules, maintain data integrity, and synchronize tables.

Basic Syntax:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- SQL statements
END;
```

Types of Triggers:

a. **BEFORE Triggers:** These triggers are executed before the triggering event (INSERT, UPDATE, DELETE) is performed on the table.

   Use Case: Often used to validate or modify data before it gets inserted or updated in the database.

```
CREATE TRIGGER trg_before_insert
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
    IF NEW.Salary < 0 THEN
        SIGNAL SQLSTATE '45000' SET MES-
SAGE_TEXT = 'Salary cannot be negative';
    END IF;
END;
```

b. **AFTER Triggers:** These triggers are executed after the triggering event has been performed on the table.

   Use Case: Often used for auditing, logging changes, or updating related tables.

```
CREATE TRIGGER trg_after_update
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO AuditLog(EmployeeID, Old-
Salary, NewSalary, ChangeDate)
    VALUES  (OLD.EmployeeID,  OLD.Salary,
NEW.Salary, NOW());
END;
```

c. **INSTEAD OF Triggers:** These triggers are executed instead of the triggering event. They are commonly used on views where the underlying data is being modified in a way that is different from the standard DML operation.

   Use Case: Useful when you need to perform complex processing or when modifying data in a view.

```
CREATE TRIGGER trg_instead_of_update
INSTEAD OF UPDATE ON EmployeeView
FOR EACH ROW
BEGIN
    UPDATE Employees
    SET FirstName = NEW.FirstName, Last-
Name = NEW.LastName
    WHERE EmployeeID = OLD.EmployeeID;
END;
```

Trigger Components:

a. Trigger Event: The event that causes the trigger to execute (e.g., INSERT, UPDATE, DELETE).
b. Trigger Condition: A condition that must be met for the trigger to execute.
c. Trigger Action: The set of SQL statements that execute when the trigger event occurs and the condition is met.
d. NEW and OLD Keywords:
   NEW: Refers to the new data that is going to be inserted or updated.
   OLD: Refers to the existing data that is being updated or deleted.

16. Exception Handling in SQL

TRY...CATCH in SQL Server

SQL Server uses `TRY...CATCH` blocks to handle exceptions. When an error occurs inside the `TRY` block, control is transferred to the `CATCH` block.

Syntax

```
BEGIN TRY
    -- SQL statements that may raise an exception
    SQL_statement;
END TRY
BEGIN CATCH
    -- SQL statements to handle the exception
    SQL_statement;
END CATCH;
```

Error Functions in CATCH Block

Inside the `CATCH` block, you can use several functions to retrieve information about the error:

- **`ERROR_NUMBER()`**: Returns the error number.
- **`ERROR_SEVERITY()`**: Returns the severity of the error.
- **`ERROR_STATE()`**: Returns the state number of the error.
- **`ERROR_PROCEDURE()`**: Returns the name of the stored procedure or trigger where the error occurred.
- **`ERROR_LINE()`**: Returns the line number where the error occurred.
- **`ERROR_MESSAGE()`**: Returns the full text of the error message.

Example

```
BEGIN TRY
    -- Attempt to divide by zero
    SELECT 1 / 0;
END TRY
BEGIN CATCH
    -- Handle the error
    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_SEVERITY() AS ErrorSeverity,
        ERROR_STATE() AS ErrorState,
        ERROR_PROCEDURE() AS ErrorProcedure,
        ERROR_LINE() AS ErrorLine,
        ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

Explanation:

- **TRY Block**: Contains the code that may cause an exception (e.g., a division by zero).
- **CATCH Block**: Executes if an error occurs in the TRY block. It retrieves and displays details about the error using the functions mentioned above.

Handling Transactions

In scenarios involving transactions, exception handling ensures data integrity.

Example with Transactions

```
BEGIN TRY
    BEGIN TRANSACTION;

    -- Execute some SQL statements
    UPDATE accounts
    SET balance = balance - 100
    WHERE account_id = 1;

    UPDATE accounts
    SET balance = balance + 100
    WHERE account_id = 2;

    -- Commit transaction if everything is fine
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    -- Rollback transaction in case of an error
```

```
    ROLLBACK TRANSACTION;

    -- Handle the error
    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

Explanation:

- **BEGIN TRANSACTION**: Starts a new transaction.
- **COMMIT TRANSACTION**: Commits the transaction if no errors occur.
- **ROLLBACK TRANSACTION**: Rolls back the transaction if an error is encountered, ensuring that no partial updates are made to the database.

Exception Handling in PL/SQL (Oracle)

For those using Oracle databases, exception handling is done differently, typically using the `EXCEPTION` block.

Syntax

```
BEGIN
    -- SQL statements
EXCEPTION
    WHEN exception_name THEN
        -- Handle the exception
        SQL_statement;
    WHEN OTHERS THEN
        -- Handle all other exceptions
        SQL_statement;
END;
```

Example

```
BEGIN
    -- Attempt to divide by zero
    SELECT 1 / 0;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Division by zero error!');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
END;
```

Explanation:

- **ZERO_DIVIDE**: A predefined exception that catches division by zero errors.
- **OTHERS**: A catch-all for any other exceptions not explicitly handled.

Appendix: SQL Basics to Advanced

A. SQL Basics

1. Introduction to SQL

- Overview of SQL and its importance in database management.

2. Types of SQL Commands

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DQL (Data Query Language)
- DCL (Data Control Language)
- TCL (Transaction Control Language)

3. Data Types

- Numeric Types
- Character/String Types
- Date/Time Types
- Binary Types
- Boolean Types

4. Basic SQL Syntax and Uses

- CREATE
- ALTER
- DROP
- INSERT
- UPDATE
- DELETE
- SELECT
- DESCRIBE

B. SQL Clauses

5. WHERE, HAVING, and GROUP BY Clauses

- WHERE: Filtering rows before grouping.
- HAVING: Filtering groups after aggregation.
- GROUP BY: Grouping rows that have the same values in specified columns.

6. ORDER BY

- Sorting result sets in ascending or descending order.

7. AND, OR, NOT

- Combining multiple conditions in queries.

8. LIMIT

- Limiting the number of records returned by a query.

9. LIKE

- Searching for patterns in string data.

10. IN and BETWEEN

- IN: Specifying multiple values in a WHERE clause.
- BETWEEN: Selecting values within a specified range.

11. DISTINCT

- Returning only distinct (unique) values.

12. ALIAS

- Giving a table or a column a temporary name.

C. SQL Functions

13. Aggregate Functions

- COUNT
- SUM
- AVG
- MIN
- MAX

14. String Functions

- CONCAT
- SUBSTRING
- LENGTH
- UPPER
- LOWER

15. Date Functions

- NOW
- CURDATE
- DATEDIFF
- DATEADD

16. Numeric Functions

- ROUND
- ABS
- MOD
- CEILING
- FLOOR

17. Conversion Functions

- CAST
- CONVERT

D. Advanced SQL

18. Joins

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- CROSS JOIN
- Self Join
- Anti Join
- Semi Join

19. Window Functions

- ROW_NUMBER()
- RANK()
- DENSE_RANK()
- NTILE()

20. Set Operations

- UNION
- UNION ALL
- INTERSECT
- EXCEPT

21. Subqueries

- Queries nested inside another query.

22. Grouping Data

- GROUP BY
- HAVING

23. Views

- Creating virtual tables based on result sets.

24. Indexes

- Primary Index
- Unique Index
- Composite Index
- Full-text Index
- Clustered vs Non-Clustered Indexes
- Bitmap Indexes

25. Constraints

- PRIMARY KEY
- FOREIGN KEY
- UNIQUE
- NOT NULL
- CHECK
- DEFAULT

26. Triggers

- BEFORE Triggers
- AFTER Triggers
- INSTEAD OF Triggers

27. Stored Procedures and Functions

- Stored Procedures
- Functions
- Recursive Stored Procedures

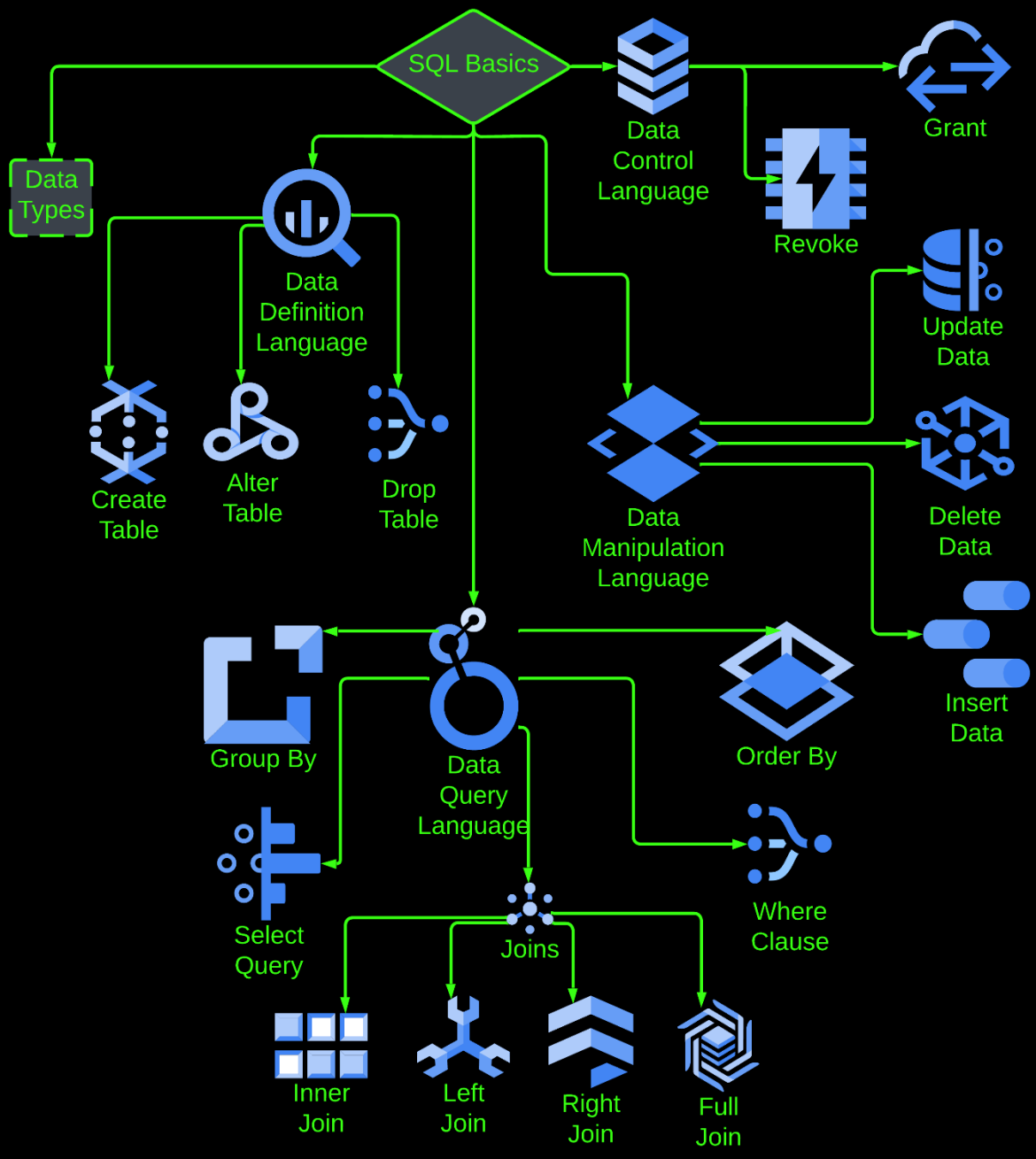
28. Common Table Expressions (CTEs)

- Simplifying complex queries with CTEs.

29. Partitioning

- Table Partitioning: Improving query performance and maintenance.

Flow Chart for better understanding



Thank you for taking the time to go through these SQL notes. I hope they've provided you with valuable insights and a deeper understanding of SQL. From foundational queries to advanced techniques, these notes are designed to serve as a reliable resource in your journey as a data professional.

If you're interested in collaborating on projects, participating in hackathons, or simply exchanging ideas, I'd love to connect! I'm always on the lookout for exciting opportunities to work with like-minded individuals who share a passion for technology and innovation.

You can reach out to me on LinkedIn or explore more of my work on GitHub. Let's build something amazing together.