SQL CheatSheet

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION TO SQL

1.1 What is SQL (Structured Query Language)?

SQL stands for Structured Query Language. It is a special language used to communicate with databases. SQL helps you create, manage, and manipulate data in a database. Think of it like a set of instructions that tells the computer how to interact with the information in a database. It allows you to:

- Create new databases and tables.
- Add, update, or delete data in tables.
- Retrieve data from tables using queries.
- Control access to the data and ensure data integrity.

1.2 Types of SQL Statements

- SQL statements are categorized based on their function. There are four main types of SQL statements:

1.2.1 DDL (Data Definition Language):

- DDL statements are used to define the structure of a database and its tables. They help you create and modify tables and other database objects. Examples of DDL statements include:
  - CREATE: Creates new databases, tables, or other objects.
  - ALTER: Modifies an existing database or table.
  - DROP: Deletes a database or table.

1.2.2 DML (Data Manipulation Language):

DML statements are used to manage the data inside the tables. They help you insert, update, delete, and retrieve data. Examples of DML statements include:

- SELECT: Retrieves data from a table.
- INSERT: Adds new rows of data to a table.
- UPDATE: Changes existing data in a table.
- DELETE: Removes data from a table.

1.2.3 DCL (Data Control Language):

DCL statements are used to control access to the database. They help you grant or deny permissions to users. Examples of DCL statements include:

- GRANT: Gives specific permissions to a user.
- REVOKE: Removes specific permissions from a user.

1. INTRODUCTION TO SQL

1.1 What is SQL (Structured Query Language)?

1.2.4 TCL (Transaction Control Language):

- TCL statements are used to manage changes to data in a database. They help you ensure that changes are saved or undone. Examples of TCL statements include:
- COMMIT: Saves all changes made during the current transaction.
- ROLLBACK: Undoes all changes made during the current transaction.
- SAVEPOINT: Sets a point to which you can later roll back.

1.3 SQL Syntax Overview

SQL uses specific rules (syntax) that you must follow when writing statements. These rules are like grammar rules in a language. Here are some basic things to keep in mind:

- SQL statements are usually written in uppercase letters, but the system doesn't mind if you use lowercase.
- SQL statements end with a semicolon (;), though it is often optional depending on the SQL system.
- Keywords in SQL like SELECT, INSERT, UPDATE, and DELETE are case-insensitive, which means you can write them in any case (e.g., select, Select, SELECT).

1.4 SQL Case Sensitivity and Conventions

In SQL, some things are case-sensitive, while others are not. Here are some general rules:

- SQL keywords (e.g., SELECT, INSERT) are not case-sensitive.
- Table names and column names may be case-sensitive, depending on the system you're using.
- It's a good practice to use consistent capitalization (e.g., writing keywords in uppercase) to make your SQL statements easy to read.

2. DATABASE CONCEPTS

2.1 Databases and Tables

- A database is like a big container that holds all your data. Inside a database, data is stored in tables. A table is like a spreadsheet where information is stored in rows and columns. Each row is a record, and each column represents a specific type of data (e.g., name, age, date).

2.2 Data Types in SQL

- Data types define what kind of data each column in a table can hold. Here are the main types of data used in SQL:

2.2.1 Numeric Types:

- Numeric types store numbers. Common numeric types include:
- INT: A whole number, like 1, 2, or 100.
- DECIMAL: A number with decimal points, like 3.14 or 100.50.

2.2.2 String Types:

- String types store text. Common string types include:
- CHAR: A fixed-length text, like 'John' (always 4 characters).
- VARCHAR: A variable-length text, like 'John' or 'Alice'. It can be shorter or longer, depending on the data.
- TEXT: A type used to store large amounts of text.

2.2.3 Date and Time Types:

- These types store date and time information. Common date and time types include:
- DATE: Stores only the date, like 2025-05-01.
- TIME: Stores only the time, like 14:30:00.
- DATETIME: Stores both date and time, like 2025-05-01 14:30:00.

2.2.4 Boolean and Binary Types:

Boolean types store true or false values. Binary types store binary data (like images or files). Common types include:

- BOOLEAN: Can store TRUE or FALSE.
- BLOB: Stands for Binary Large Object, used for storing images, audio, etc.

2. DATABASE CONCEPTS

2.3 Primary Key, Foreign Key, and Constraints

- In a database, you often need rules to make sure the data is correct and linked properly. Here are some key concepts:

2.3.1 Primary Key:

- A primary key is a column (or set of columns) that uniquely identifies each row in a table. No two rows can have the same primary key. It ensures that each record is unique.

2.3.2 Foreign Key:

- A foreign key is a column (or set of columns) that links one table to another. It points to the primary key in another table, creating a relationship between the two tables.

2.3.3 NOT NULL:

- The NOT NULL constraint ensures that a column cannot have a null (empty) value. Every record in this column must have a value.

2.3.4 UNIQUE:

- The UNIQUE constraint ensures that every value in a column is different. No two rows can have the same value for this column.

2.3.5 CHECK:

- The CHECK constraint allows you to set a condition for the data in a column. For example, you can make sure that the age column only contains values greater than or equal to 18.

2.3.6 DEFAULT:

- The DEFAULT constraint provides a default value for a column when no value is specified.

2.3.7 AUTO_INCREMENT:

- The AUTO_INCREMENT constraint automatically increases the value of a column (usually used for primary keys). For example, every time you add a new record, the ID number will automatically be 1 higher than the last one.

3. BASIC SQL COMMANDS

3.1 SELECT: Retrieving Data

- The SELECT statement is used to retrieve data from one or more tables in a database. It's one of the most commonly used SQL commands.

3.1.1 Simple SELECT Statement:

- The basic SELECT statement retrieves all columns and rows from a table:

```
SELECT * FROM table_name;
```

- "*" means all columns.
- table_name is the name of the table you want to get data from.

Example:

```
SELECT * FROM students;
```

- This will show all data from the students table.

3.1.2 Distinct Values:

- Sometimes, you may want to retrieve only unique (distinct) values from a column. To do this, you can use the DISTINCT keyword:

```
SELECT DISTINCT column_name FROM table_name;
```

Example:

```
SELECT DISTINCT country FROM students;
```

- This will give you a list of unique countries in the students table.

3.2 INSERT: Inserting Data

- The INSERT statement is used to add new rows of data into a table.

3.2.1 Inserting Single and Multiple Rows:

- To insert a single row:

```
INSERT INTO table_name (column1, column2, column3) VALUES (value1, value2,
value3);
```

- Example:

```
INSERT INTO students (name, age, country) VALUES ('John', 20, 'USA');
```

3. BASIC SQL COMMANDS

3.2.1 Inserting Single and Multiple Rows:

- Example:

```
INSERT INTO table_name (column1, column2, column3) VALUES (value1, value2,
value3), (value4, value5, value6);
```

- Example:

```
INSERT INTO students (name, age, country) VALUES ('Alice', 22, 'UK'), ('Bob',
21, 'Canada');
```

3.3 UPDATE: Updating Data

- The UPDATE statement is used to modify existing data in a table.

3.3.1 Updating Specific Columns:

- To update specific columns in a row:

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

- Example:

```
UPDATE students SET age = 21 WHERE name = 'John';
```

- This will change the age of John to 21 in the students table.

3.4 DELETE: Deleting Data

- The DELETE statement is used to remove data from a table.

3.4.1 DELETE vs. TRUNCATE vs. DROP:

- DELETE: Removes specific rows from a table based on a condition. It can be rolled back and is slower than TRUNCATE.

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

- Example:

```
DELETE FROM students WHERE age < 18;
```

3. BASIC SQL COMMANDS

3.4 DELETE: Deleting Data

- TRUNCATE: Removes all rows from a table. It's faster than DELETE but cannot be rolled back.

```
TRUNCATE TABLE table_name;
```

- DROP: Completely removes a table or database, including its structure and data. This action cannot be undone.

```
DROP TABLE table_name;
```

4. FILTERING AND SORTING DATA

4.1 WHERE Clause

- The WHERE clause is used to filter data based on certain conditions. It limits the rows returned by a SELECT, UPDATE, or DELETE statement.

4.1.1 Comparison Operators (=, <, >, <=, >=, <>):

- These operators are used to compare values:
- =: Equal to.
- <: Less than.
- >: Greater than.
- <=: Less than or equal to.
- >=: Greater than or equal to.
- <>: Not equal to.

- Example:

```
SELECT * FROM students WHERE age >= 18;
```

- This will return students who are 18 or older.

4.1.2 Logical Operators (AND, OR, NOT):

Logical operators help you combine multiple conditions:

- AND: Both conditions must be true.
- OR: At least one condition must be true.
- NOT: Reverses the result (True becomes False and vice versa).

Example:

```
SELECT * FROM students WHERE age >= 18 AND country = 'USA';
```

This will return students who are 18 or older and live in the USA.

4.1.3 Pattern Matching (LIKE, REGEXP):

- The LIKE operator is used for pattern matching. You can use % for multiple characters and _ for a single character.
  - LIKE: Matches a pattern.
  - REGEXP: Matches a regular expression pattern.

Example:

```
SELECT * FROM students WHERE name LIKE 'J%';
```

This will return all students whose name starts with "J".

4. FILTERING AND SORTING DATA

4.1.4 Null Values (IS NULL, IS NOT NULL):

- IS NULL checks if a value is null (empty), and IS NOT NULL checks if a value is not null.

```
SELECT * FROM students WHERE country IS NULL;
```

This will return students who don't have a country listed.

4.2 ORDER BY Clause

The ORDER BY clause is used to sort the result set.

- 4.2.1 Sorting Data in Ascending and Descending Order:
- You can sort data in ascending (ASC) or descending (DESC) order:

```
SELECT * FROM students ORDER BY age ASC;
```

- This will sort students by age in ascending order.
- To sort in descending order:

```
SELECT * FROM students ORDER BY age DESC;
```

4.2.2 Sorting by Multiple Columns:

- You can sort by multiple columns:

```
SELECT * FROM students ORDER BY country ASC, age DESC;
```

- This will first sort by country in ascending order, and if countries are the same, it will sort by age in descending order.

4.3 LIMIT and OFFSET

- The LIMIT and OFFSET clauses are used to control how many rows are returned by a query.

4.3.1 Fetching a Specific Number of Rows:

- The LIMIT clause specifies the maximum number of rows to return.

```
SELECT * FROM students LIMIT 5;
```

- This will return only the first 5 rows.

4. FILTERING AND SORTING DATA

4.3.2 Pagination Techniques:

- You can combine LIMIT with OFFSET to implement pagination. OFFSET skips a specific number of rows:

```
SELECT * FROM students LIMIT 5 OFFSET 10;
```

- This will skip the first 10 rows and return the next 5 rows. This is useful for displaying results in pages.

5. AGGREGATE FUNCTIONS

5.1 Using Aggregate Functions

- Aggregate functions are used to perform calculations on multiple rows of data and return a single value.

5.1.1 COUNT():

- The COUNT() function returns the number of rows in a table or the number of non-null values in a column.

```
SELECT COUNT(*) FROM students;
```

- This will return the total number of rows in the students table.

5.1.2 SUM():

- The SUM() function adds up the values in a numeric column.

```
SELECT SUM(age) FROM students;
```

- This will give you the total of all ages in the students table.

5.1.3 AVG():

- The AVG() function calculates the average value of a numeric column.

```
SELECT AVG(age) FROM students;
```

- This will give you the average age of all students.

5.1.4 MIN():

- The MIN() function returns the smallest value in a column.

```
SELECT MIN(age) FROM students;
```

- This will give you the smallest age in the students table.

5.1.5 MAX():

- The MAX() function returns the largest value in a column.

```
SELECT MAX(age) FROM students;
```

- This will give you the largest age in the students table.

5. AGGREGATE FUNCTIONS

5.2 GROUP BY Clause

- The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows, like calculating aggregates for each group.

5.2.1 Grouping Rows by Specific Columns:

- To group data by a specific column and apply an aggregate function:

```sql
SELECT country, COUNT(*) FROM students GROUP BY country;
```

- This will count the number of students in each country.
- You can group by multiple columns:

```sql
SELECT country, age, COUNT(*) FROM students GROUP BY country, age;
```

- This will count the number of students for each combination of country and age.

5.3 HAVING Clause

- The HAVING clause is used to filter the result set after the GROUP BY operation. It's similar to the WHERE clause, but WHERE filters rows before grouping, while HAVING filters after grouping.

5.3.1 Filtering Grouped Data:

- To filter groups based on an aggregate function:

```sql
SELECT country, COUNT(*) FROM students GROUP BY country HAVING COUNT(*) > 5;
```

- This will return only the countries that have more than 5 students.

6. JOINS AND RELATIONSHIPS

6.1 INNER JOIN: Fetching Matching Records

- The INNER JOIN fetches records that have matching values in both tables. If no match is found, the row is excluded.

6.1.1 Example:

```
SELECT students.name, courses.name FROM students
INNER JOIN enrollments ON students.id = enrollments.student_id
INNER JOIN courses ON enrollments.course_id = courses.id;
```

- This will return the names of students and the courses they are enrolled in, where there is a match between the students and courses tables.

6.2 LEFT JOIN (or LEFT OUTER JOIN): Fetching All Records from Left Table

- The LEFT JOIN (or LEFT OUTER JOIN) fetches all rows from the left table, and the matching rows from the right table. If no match is found, the result will contain NULL for columns from the right table.

6.2.1 Example:

```
SELECT students.name, courses.name FROM students
LEFT JOIN enrollments ON students.id = enrollments.student_id
LEFT JOIN courses ON enrollments.course_id = courses.id;
```

- This will return all students, along with the courses they are enrolled in (if any).

6.3 RIGHT JOIN (or RIGHT OUTER JOIN): Fetching All Records from Right Table

- The RIGHT JOIN (or RIGHT OUTER JOIN) fetches all rows from the right table, and the matching rows from the left table. If no match is found, the result will contain NULL for columns from the left table.

6.3.1 Example:

```
SELECT students.name, courses.name FROM students
RIGHT JOIN enrollments ON students.id = enrollments.student_id
RIGHT JOIN courses ON enrollments.course_id = courses.id;
```

6. JOINS AND RELATIONSHIPS

6.1 INNER JOIN: Fetching Matching Records

- The INNER JOIN fetches records that have matching values in both tables. If no match is found, the row is excluded.

6.1.1 Example:

```
SELECT students.name, courses.name FROM students
INNER JOIN enrollments ON students.id = enrollments.student_id
INNER JOIN courses ON enrollments.course_id = courses.id;
```

- This will return the names of students and the courses they are enrolled in, where there is a match between the students and courses tables.

6.2 LEFT JOIN (or LEFT OUTER JOIN): Fetching All Records from Left Table

- The LEFT JOIN (or LEFT OUTER JOIN) fetches all rows from the left table, and the matching rows from the right table. If no match is found, the result will contain NULL for columns from the right table.

6.2.1 Example:

```
SELECT students.name, courses.name FROM students
LEFT JOIN enrollments ON students.id = enrollments.student_id
LEFT JOIN courses ON enrollments.course_id = courses.id;
```

- This will return all students, along with the courses they are enrolled in (if any).

6.3 RIGHT JOIN (or RIGHT OUTER JOIN): Fetching All Records from Right Table

- The RIGHT JOIN (or RIGHT OUTER JOIN) fetches all rows from the right table, and the matching rows from the left table. If no match is found, the result will contain NULL for columns from the left table.

6.3.1 Example:

```
SELECT students.name, courses.name FROM students
RIGHT JOIN enrollments ON students.id = enrollments.student_id
RIGHT JOIN courses ON enrollments.course_id = courses.id;
```

- This will return all courses, along with the students enrolled in them (if any).

6. JOINS AND RELATIONSHIPS

6.4 FULL OUTER JOIN: Fetching All Records from Both Tables

- The FULL OUTER JOIN returns all rows when there is a match in either the left or right table. If no match is found, it will return NULL for the non-matching side.

6.4.1 Example:

```
SELECT students.name, courses.name FROM students
FULL OUTER JOIN enrollments ON students.id = enrollments.student_id
FULL OUTER JOIN courses ON enrollments.course_id = courses.id;
```

- This will return all students and all courses, showing NULL where there is no match.

6.5 CROSS JOIN: Cartesian Product of Two Tables

- **T**he CROSS JOIN returns the Cartesian product of two tables, meaning each row from the first table is combined with each row from the second table.

6.5.1 Example:

```
SELECT students.name, courses.name FROM students
CROSS JOIN courses;
```

- This will return a list of every student paired with every course.

6.6 Self Join: Joining a Table to Itself

- A self join is used when you need to join a table to itself. This can be useful for hierarchical data, such as employees and managers.

6.6.1 Example:

```
SELECT e.name AS Employee, m.name AS Manager FROM employees e
LEFT JOIN employees m ON e.manager_id = m.id;
```

- This will return a list of employees along with their managers.

6.7 Joins with Aliases

- Aliases are used to give a table or a column a temporary name, making the query easier to read, especially in joins.

6. JOINS AND RELATIONSHIPS

6.7.1 Example:

```
SELECT s.name AS StudentName, c.name AS CourseName FROM students s
INNER JOIN enrollments e ON s.id = e.student_id
INNER JOIN courses c ON e.course_id = c.id;
```

- In this example, s is an alias for the students table, and c is an alias for the courses table.

7. SUBQUERIES

7.1 Types of Subqueries

- A subquery is a query within another query. Subqueries can be used in SELECT, FROM, WHERE, HAVING, and INSERT clauses.

7.1.1 Single-row Subqueries:

- A single-row subquery returns only one row and is usually used with comparison operators.

```
SELECT name FROM students WHERE age = (SELECT MAX(age) FROM students);
```

- This will return the name of the student with the highest age.

7.1.2 Multi-row Subqueries:

- A multi-row subquery returns more than one row and is used with operators like IN, ANY, ALL.

```
SELECT name FROM students WHERE age IN (SELECT age FROM students WHERE
country = 'USA');
```

- This will return the names of students whose age is the same as any student from the USA.

7.1.3 Correlated Subqueries:

- A correlated subquery depends on the outer query. It's evaluated once for each row in the outer query.

```
SELECT name FROM students s WHERE age > (SELECT AVG(age) FROM students WHERE
country = s.country);
```

- This will return the names of students whose age is greater than the average age of students in the same country.

7.2 Using Subqueries in SELECT, FROM, WHERE, HAVING, and INSERT

7.2.1 In SELECT:

```
SELECT name, (SELECT MAX(age) FROM students) AS max_age FROM students;
```

7.2.2 In FROM:

```
SELECT avg_age FROM (SELECT AVG(age) AS avg_age FROM students) AS subquery;
```

7. SUBQUERIES

7.2.3 In WHERE:

```sql
SELECT name FROM students WHERE age = (SELECT MAX(age) FROM students);
```

7.2.4 In HAVING:

```sql
SELECT country, COUNT(*) FROM students GROUP BY country HAVING COUNT(*) >
(SELECT AVG(count) FROM (SELECT COUNT(*) FROM students GROUP BY country) AS
subquery);
```

7.2.5 In INSERT:

```sql
INSERT INTO students (name, age) SELECT 'Jane', 22 FROM DUAL WHERE NOT EXISTS
(SELECT 1 FROM students WHERE name = 'Jane');
```

8. SET OPERATIONS

8.1 UNION: Combining Result Sets (Distinct)

- The UNION operator is used to combine the results of two or more SELECT statements into a single result set. It returns only distinct values, meaning duplicates are removed.

8.1.1 Example:

```
SELECT name FROM students WHERE country = 'USA'
UNION
SELECT name FROM students WHERE country = 'Canada';
```

- This will return a list of student names who are either from the USA or Canada, without any duplicates.

8.2 UNION ALL: Combining Result Sets (Including Duplicates)

- The UNION ALL operator combines the results of two or more SELECT statements into a single result set, including all duplicates.

8.2.1 Example:

```
SELECT name FROM students WHERE country = 'USA'
UNION ALL
SELECT name FROM students WHERE country = 'Canada';
```

- This will return a list of student names from both the USA and Canada, including any duplicates.

8.3 INTERSECT: Returning Common Results

- The INTERSECT operator is used to return only the rows that are common in both result sets.

8.3.1 Example:

```
SELECT name FROM students WHERE country = 'USA'
INTERSECT
SELECT name FROM students WHERE country = 'Canada';
```

- This will return the names of students who are in both the USA and Canada.

8.4 EXCEPT: Returning Results in One Query but Not the Other

- The EXCEPT operator returns rows from the first query that do not appear in the second query.

8. SET OPERATIONS

8.4.1 Example:

```
SELECT name FROM students WHERE country = 'USA'
EXCEPT
SELECT name FROM students WHERE country = 'Canada';
```

- This will return the names of students from the USA who are not also in Canada.

9. SQL DATA MANIPULATION

9.1 INSERT INTO SELECT: Inserting Data from One Table to Another

- The INSERT INTO SELECT statement allows you to insert data from one table into another. This is useful when you want to copy data from one table into another.

9.1.1 Example:

```
INSERT INTO students_backup (name, age, country)
SELECT name, age, country FROM students;
```

- This will copy all data from the students table to the students_backup table.

9.2 UPDATE JOIN: Updating Records Based on Data from Another Table

- The UPDATE JOIN statement is used to update records in one table based on values from another table.

9.2.1 Example:

```
UPDATE students s
JOIN courses c ON s.id = c.student_id
SET s.age = 21
WHERE c.name = 'Math';
```

- This will update the age of students enrolled in the 'Math' course to 21.

9.3 DELETE JOIN: Deleting Records Based on Data from Another Table

- The DELETE JOIN statement is used to delete records in one table based on data from another table.

9.3.1 Example:

```
DELETE s FROM students s
JOIN enrollments e ON s.id = e.student_id
WHERE e.course_id = 101;
```

- This will delete students who are enrolled in the course with course_id 101.

10. INDEXES AND PERFORMANCE OPTIMIZATION

10.1 What is an Index in SQL?

- An index is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and time for updates and inserts. It works like an index in a book, helping the database quickly find the rows that match a query.

10.2 Creating and Dropping Indexes

10.2.1 CREATE INDEX:

- The CREATE INDEX statement is used to create an index on one or more columns of a table to improve the query performance.

```
CREATE INDEX idx_name ON students(name);
```

This will create an index on the name column of the students table.

10.2.2 DROP INDEX:

- The DROP INDEX statement is used to delete an index from a table.

```
DROP INDEX idx_name;
```

- This will remove the index called idx_name.

10.3 Using Indexes to Speed Up Queries

- Indexes are used to speed up the process of retrieving data from a table. Without an index, the database must scan the entire table to find the requested data. With an index, the database can quickly locate the data.

10.3.1 Example:

```
SELECT name FROM students WHERE country = 'USA';
```

- If there is an index on the country column, the query will run faster because the index will allow the database to quickly locate the rows where country = 'USA'.

10. INDEXES AND PERFORMANCE OPTIMIZATION

10.4 Unique Indexes vs Non-Unique Indexes

10.4.1 Unique Indexes:

- A unique index ensures that the values in the indexed column are unique, meaning no two rows can have the same value for that column.

```
CREATE UNIQUE INDEX idx_unique_name ON students(name);
```

10.4.2 Non-Unique Indexes:

- A non-unique index allows duplicate values in the indexed column.

```
CREATE INDEX idx_non_unique_country ON students(country);
```

10.5 Composite Indexes

- A composite index is an index that includes more than one column. It is used when queries filter on multiple columns.

10.5.1 Example:

```
CREATE INDEX idx_composite ON students(country, age);
```

- This will create an index on both the country and age columns.

10.6 Query Optimization Techniques

10.6.1 EXPLAIN Command for Query Analysis

- The EXPLAIN command is used to analyze how a query will be executed by the database and can help you identify performance bottlenecks.

10.6.1.1 Example:

```
EXPLAIN SELECT name FROM students WHERE country = 'USA';
```

- This will provide information about how the query is executed, such as which indexes are used.

10.6.2 Caching and Index Usage

- Using indexes efficiently can significantly reduce query execution time. The database may also cache the results of frequently used queries, improving performance.

10. INDEXES AND PERFORMANCE OPTIMIZATION

10.6.3 Optimizing JOIN Queries

- When joining large tables, ensure that indexes are used on the columns involved in the JOIN. This can speed up the query execution time.

10.6.3.1 Example:

```
SELECT s.name, c.name
FROM students s
JOIN enrollments e ON s.id = e.student_id
JOIN courses c ON e.course_id = c.id;
```

- Indexing the student_id and course_id columns can speed up the JOIN operation.

11. TRANSACTIONS

11.1 What is a Transaction?

- A transaction is a sequence of one or more SQL operations that are executed as a single unit. A transaction ensures that either all the operations are completed successfully, or none of them are applied, maintaining the integrity of the database.
- For example, transferring money from one account to another is a transaction. If one part of the operation (e.g., deducting from one account) fails, the entire transaction is rolled back, ensuring no money is lost.

11.2 Transaction Control Statements

11.2.1 BEGIN TRANSACTION:

- The BEGIN TRANSACTION statement marks the beginning of a transaction. It is used to start a set of operations that should be executed as a single unit.

```
BEGIN TRANSACTION;
```

11.2.2 COMMIT: Saving Changes

- The COMMIT statement is used to save all the changes made during a transaction. Once a transaction is committed, the changes are permanent.

```
COMMIT;
```

- This statement confirms all changes made to the database during the transaction.

11.2.3 ROLLBACK: Reverting Changes

- The ROLLBACK statement is used to undo all changes made during the current transaction. It is typically used when an error occurs and you want to revert to the previous state.

```
ROLLBACK;
```

- This will undo all changes made after the BEGIN TRANSACTION statement.

11.2.4 SAVEPOINT: Setting and Rolling Back to a Point

- A SAVEPOINT allows you to set a specific point in a transaction to which you can later roll back. This is useful for undoing part of a transaction, rather than the entire transaction.

```
SAVEPOINT savepoint_name;
```

11. TRANSACTIONS

11.2.4 SAVEPOINT: Setting and Rolling Back to a Point

- You can then roll back to that specific point:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

11.3 ACID Properties (Atomicity, Consistency, Isolation, Durability)

11.3.1 Atomicity:

- A transaction is atomic, meaning it is treated as a single unit of work. Either all the operations within the transaction are completed successfully, or none of them are applied.

11.3.2 Consistency:

- A transaction must transition the database from one valid state to another valid state, maintaining the integrity of the data.

11.3.3 Isolation:

- Each transaction is isolated from others. The changes made by one transaction are not visible to others until the transaction is committed.

11.3.4 Durability:

- Once a transaction is committed, its changes are permanent, even if the system crashes afterward.

12. NORMALIZATION AND DENORMALIZATION

12.1 What is Database Normalization?

- Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. The goal is to separate data into related tables and eliminate unnecessary duplication of data.

12.2 Types of Normal Forms

12.2.1 1st Normal Form (1NF):

- A table is in 1st Normal Form (1NF) if it contains only atomic (indivisible) values, and each column contains only one type of data.

Example:

- A table that contains a list of items and their prices must have each item in a separate row, not as a comma-separated list.

12.2.2 2nd Normal Form (2NF):

- A table is in 2nd Normal Form (2NF) if it is in 1st Normal Form (1NF) and all non-key attributes are fully functionally dependent on the primary key.

Example:

- If a table has a StudentID and CourseID as a composite primary key, and StudentName is only dependent on StudentID, then StudentName should be moved to another table.

12.2.3 3rd Normal Form (3NF):

- A table is in 3rd Normal Form (3NF) if it is in 2nd Normal Form (2NF) and there is no transitive dependency between non-key attributes. This means that non-key attributes should not depend on other non-key attributes.

Example:

- If a table has StudentID, CourseID, and InstructorName, and InstructorName depends on CourseID, then InstructorName should be moved to a separate table.

12.2.4 Boyce-Codd Normal Form (BCNF):

- A table is in Boyce-Codd Normal Form (BCNF) if it is in 3rd Normal Form (3NF) and, for every functional dependency, the determinant is a candidate key.

12.3 Denormalization: Trade-offs Between Performance and Data Integrity

- Denormalization is the process of combining tables to reduce the number of joins needed for querying data. While it can improve performance by reducing the complexity of queries, it may also introduce redundancy and violate the principles of normalization.

12. NORMALIZATION AND DENORMALIZATION

When to Denormalize:

- When performance is more critical than data integrity.
- When queries are slow due to the need for multiple joins.
- When the database design is too complex and needs to be simplified for efficiency.

However, denormalization may cause problems such as data redundancy, which can lead to inconsistencies when updating data.

13. VIEWS

13.1 What are Views in SQL?

- A view is a virtual table created by a query that combines data from one or more tables. Views simplify complex queries by presenting data in a more understandable or user-friendly way.

13.1.1 Example:

```sql
CREATE VIEW student_view AS
SELECT name, age FROM students WHERE country = 'USA';
```

- This creates a view that shows the names and ages of students from the USA.

13.2 Creating and Dropping Views

13.2.1 CREATE VIEW:

- To create a view, you use the CREATE VIEW statement followed by a SELECT query.

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

13.2.2 DROP VIEW:

- To remove a view, you use the DROP VIEW statement.

```sql
DROP VIEW view_name;
```

13.3 Updating Data Through Views

- Some views are updatable, meaning you can insert, update, or delete data through the view, and those changes will reflect in the underlying tables.
- However, not all views are updatable. A view that involves complex joins or aggregate functions may not be updatable.

Example of updating data through a view:

```sql
UPDATE student_view
SET age = 22
WHERE name = 'John';
```

13. VIEWS

13.4 View Performance Considerations

- While views are helpful for simplifying queries, they can sometimes have performance drawbacks. Each time you query a view, the database must run the underlying query, which can be slow for complex views.
- To improve performance, consider using indexed views (materialized views) or optimizing the underlying query.

14. STORED PROCEDURES AND FUNCTIONS

14.1 Introduction to Stored Procedures

- Stored procedures are pre-written SQL queries that are stored in the database. These procedures can be executed on demand and help in reusing SQL code, making the system more efficient.

14.1.1 Creating and Executing Stored Procedures:

- To create a stored procedure, you use the CREATE PROCEDURE statement. Once created, the stored procedure can be executed using the CALL statement.

```
CREATE PROCEDURE procedure_name (parameters)
BEGIN
    -- SQL statements
END;
```

Example:

```
CREATE PROCEDURE get_student_info (IN student_id INT)
BEGIN
    SELECT name, age FROM students WHERE id = student_id;
END;
```

- To execute the procedure:

```
CALL get_student_info(1);
```

14.1.2 Parameters in Stored Procedures:

- Stored procedures can accept input parameters and return output parameters. Input parameters are passed during execution, while output parameters return data.

```
CREATE PROCEDURE example_procedure (IN input_param INT, OUT output_param INT)
BEGIN
    SET output_param = input_param * 2;
END;
```

14.2 User-defined Functions (UDFs)

- User-defined functions (UDFs) allow you to create reusable functions in SQL. Unlike stored procedures, functions return a value and can be used directly in SQL queries.

14. STORED PROCEDURES AND FUNCTIONS

14.2 User-defined Functions (UDFs)

- User-defined functions (UDFs) allow you to create reusable functions in SQL. Unlike stored procedures, functions return a value and can be used directly in SQL queries.

```
CREATE FUNCTION function_name (parameters)
RETURNS return_type
BEGIN
    -- function body
    RETURN value;
END;
```

Example:

```
CREATE FUNCTION get_discount(price DECIMAL)
RETURNS DECIMAL
BEGIN
    RETURN price * 0.1;
END;
```

- You can then use the function in a SQL query:

```
SELECT price, get_discount(price) FROM products;
```

14.3 Advantages and Disadvantages of Stored Procedures

- **Advantages:**
  - Reusability: Stored procedures can be reused multiple times, improving code maintainability.
  - Performance: Stored procedures are precompiled, which can lead to faster execution.
  - Security: Stored procedures can limit access to certain operations, providing an additional layer of security.
- **Disadvantages:**
  - Complexity: Stored procedures can become complex, especially for large databases.
  - Portability: Stored procedures are often specific to a particular database management system (DBMS), making them less portable.

15. TRIGGERS

15.1 What are Triggers?

- A trigger is a special type of stored procedure that is automatically executed when a specified event occurs in the database. Triggers are typically used to enforce business rules or data integrity.

15.2 Types of Triggers

15.2.1 BEFORE Trigger:

- A BEFORE trigger is executed before an insert, update, or delete operation on a table.

```
CREATE TRIGGER before_insert_trigger
BEFORE INSERT ON table_name
FOR EACH ROW
BEGIN
    -- Trigger logic
END;
```

15.2.2 AFTER Trigger:

- An AFTER trigger is executed after an insert, update, or delete operation on a table.

```
CREATE TRIGGER after_insert_trigger
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
    -- Trigger logic
END;
```

15.2.3 INSTEAD OF Trigger:

- An INSTEAD OF trigger replaces the normal action (insert, update, or delete) with custom logic.

```
CREATE TRIGGER instead_of_trigger
INSTEAD OF UPDATE ON table_name
FOR EACH ROW
BEGIN
    -- Custom logic to replace the update action
END;
```

15. TRIGGERS

15.3 Creating and Dropping Triggers

15.3.1 CREATE TRIGGER:

- To create a trigger, you use the CREATE TRIGGER statement, followed by the trigger type (BEFORE, AFTER, INSTEAD OF) and the event (INSERT, UPDATE, DELETE).

```
CREATE TRIGGER trigger_name
BEFORE INSERT ON table_name
FOR EACH ROW
BEGIN
    -- Trigger logic
END;
```

15.3.2 DROP TRIGGER:

- To remove a trigger, you use the DROP TRIGGER statement.

```
DROP TRIGGER trigger_name;
```

15.4 Triggering Events (INSERT, UPDATE, DELETE):

Triggers can be set to fire on different events:

- INSERT: Executes when a new record is added.
- UPDATE: Executes when an existing record is modified.
- DELETE: Executes when a record is removed.

Example of an AFTER INSERT trigger:

```
CREATE TRIGGER after_insert_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO log_table (action, employee_id, timestamp)
    VALUES ('INSERT', NEW.id, NOW());
END;
```

16. USER MANAGEMENT AND SECURITY

16.1 Creating and Managing Users

- In SQL, you can create new users and assign them specific permissions.

16.1.1 CREATE USER:

- To create a new user, use the CREATE USER statement:

```
CREATE USER 'username'@'hostname' IDENTIFIED BY 'password';
```

16.1.2 GRANT:

- To grant privileges to a user, use the GRANT statement:

```
GRANT SELECT, INSERT ON database_name TO 'username'@'hostname';
```

16.1.3 REVOKE:

- To revoke privileges from a user, use the REVOKE statement:

```
REVOKE SELECT, INSERT ON database_name FROM 'username'@'hostname';
```

16.2 Managing Permissions

Permissions control what a user can and cannot do within the database.
Permissions include:

- SELECT: Allows reading data from tables.
- INSERT: Allows inserting data into tables.
- UPDATE: Allows modifying data in tables.
- DELETE: Allows removing data from tables.

16.3 Security Considerations in SQL

16.3.1 SQL Injection Prevention:

- SQL injection is a common security vulnerability where attackers manipulate SQL queries. To prevent SQL injection, use parameterized queries or prepared statements.

Example in Python:

```
cursor.execute("SELECT * FROM users WHERE username = %s AND password = %s",
(username, password))
```

16. USER MANAGEMENT AND SECURITY

16.3.2 Role-Based Access Control (RBAC):

- RBAC is a security model that restricts access based on user roles. Users are assigned roles, and each role has specific permissions.

Example:

```
CREATE ROLE admin;
GRANT SELECT, INSERT, DELETE ON database_name TO admin;
```

17. DATABASE DESIGN

17.1 Entity-Relationship (ER) Diagrams

- Entity-Relationship (ER) diagrams are visual representations of a database's structure. They help define the entities (tables), their attributes (columns), and the relationships between them. ER diagrams use different symbols to represent entities, attributes, and relationships.
- Entities: Represent tables in the database.
- Attributes: Represent columns in a table.
- Relationships: Define how two or more entities are related.

Example:

```
| Student |<------> | Course |
```

- This example shows a relationship between a "Student" entity and a "Course" entity.

17.2 Table Design and Relationships

- Database tables can have different types of relationships:

17.2.1 One-to-One Relationship:

- In this relationship, each record in one table is linked to exactly one record in another table.
- Example: A person has only one passport, and each passport is assigned to only one person.

```
CREATE TABLE person (
    person_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE passport (
    passport_id INT PRIMARY KEY,
    person_id INT,
    FOREIGN KEY (person_id) REFERENCES person(person_id)
);
```

17.2.2 One-to-Many Relationship:

- A record in one table can relate to multiple records in another table.
- Example: A customer can place multiple orders, but each order belongs to only one customer.

17. DATABASE DESIGN

17.2.2 One-to-Many Relationship:

```sql
CREATE TABLE customer (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

17.2.3 Many-to-Many Relationship:

- Multiple records in one table can be associated with multiple records in another table.
- Example: A student can enroll in multiple courses, and each course can have multiple students.

```sql
CREATE TABLE student (
    student_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE course (
    course_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE enrollment (
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES student(student_id),
    FOREIGN KEY (course_id) REFERENCES course(course_id)
);
```

17.3 Foreign Keys and Referential Integrity

- Foreign Key: A foreign key is a column or a set of columns in one table that refers to the primary key in another table. This establishes a link between the two tables.
- Referential Integrity: This ensures that relationships between tables remain consistent. For example, a foreign key value in the child table must match a primary key value in the parent table.

18. BACKUP AND RESTORATION

18.1 SQL Backup Strategies

18.1.1 BACKUP DATABASE:

- This command creates a backup of an entire database. It is typically used to safeguard all tables, stored procedures, and data within a database.

```
BACKUP DATABASE database_name TO DISK = 'backup_file.bak';
```

18.1.2 BACKUP TABLE:

- This command allows you to back up individual tables.

```
BACKUP TABLE table_name TO 'backup_file.sql';
```

18.2 Restoring Data

18.2.1 RESTORE DATABASE:

- This command restores a backup of a database to its previous state.

```
RESTORE DATABASE database_name FROM DISK = 'backup_file.bak';
```

18.2.2 LOAD DATA INFILE:

- This command is used to load data from a file into a table.

```
LOAD DATA INFILE 'file_path' INTO TABLE table_name;
```

18.3 Incremental and Full Backups

- Incremental Backup: This only backs up changes made since the last backup. It helps save storage space and reduces backup time.
- Full Backup: A full backup backs up the entire database or table, including all data, structures, and objects.

19. SQL BEST PRACTICES

19.1 Writing Efficient SQL Queries

- Use WHERE Clause: Always filter results using WHERE to avoid retrieving unnecessary data.
- Use Indexes: Indexes can help speed up query performance, especially on large tables.
- Avoid Subqueries When Possible: Try to use JOIN instead of subqueries, as they can be more efficient.

19.2 Avoiding Common Pitfalls in SQL

19.2.1 Using SELECT :

- Avoid using SELECT * in production queries. Always specify the columns you need to retrieve. It improves performance and prevents unnecessary data retrieval.

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

19.2.2 Using DISTINCT Improperly:

- Only use DISTINCT when necessary, as it can slow down queries by eliminating duplicate results.

```sql
SELECT DISTINCT column_name FROM table_name;
```

19.2.3 Using Subqueries Excessively:

- Subqueries can be inefficient. Instead, try to use joins or temporary tables when possible.

19.3 Naming Conventions for Tables and Columns

- Use clear, descriptive names for tables and columns (e.g., employees instead of emp).
- Use underscores to separate words (e.g., first_name instead of firstname).
- Avoid using reserved keywords (e.g., SELECT, TABLE).

19.4 Commenting SQL Code

- Always add comments to explain complex queries or logic. It makes it easier for others to understand and maintain your code.

```sql
-- This query retrieves all employees from the 'employees' table
SELECT * FROM employees;
```

19. SQL BEST PRACTICES

19.5 Documenting Queries and Procedures

- Document complex queries and stored procedures with brief explanations, parameter descriptions, and any special conditions. This is especially important when working in teams.

20. SQL RESOURCES

20.1 SQL Documentation and References

- Official SQL documentation is a valuable resource for understanding SQL syntax and functionality. Refer to your DBMS's documentation (e.g., MySQL, PostgreSQL, SQL Server).

20.2 Online Communities and Forums for SQL

- Stack Overflow: A popular forum for asking SQL-related questions.
- SQLServerCentral: A community for SQL Server users.
- Reddit (r/SQL): A place to discuss SQL concepts and share knowledge.

20.3 SQL Tools and IDEs

- MySQL Workbench: A powerful tool for designing and managing MySQL databases.
- SQL Server Management Studio (SSMS): An IDE for managing SQL Server databases.
- DBeaver: A cross-platform database tool that supports many databases.

20.4 SQL Tutorials and Courses

- SQLZoo: An interactive platform to learn SQL.
- Codecademy: Offers beginner to advanced courses on SQL.
- Udemy: Provides various SQL courses for all levels.

20.5 SQL Conferences and Meetups

- Attending conferences and meetups can help you stay updated with the latest trends in SQL and connect with professionals in the field.
- SQL Saturday: A free, global event focusing on SQL Server.
- SQLBits: A community-driven conference for SQL Server professionals

Was this post helpful ?

Follow Our 2nd Account

Follow For More

Tap Here