upGrad

Lecture Notes:
Database Design and Advanced SQL Querying

Database Design

Key Definitions

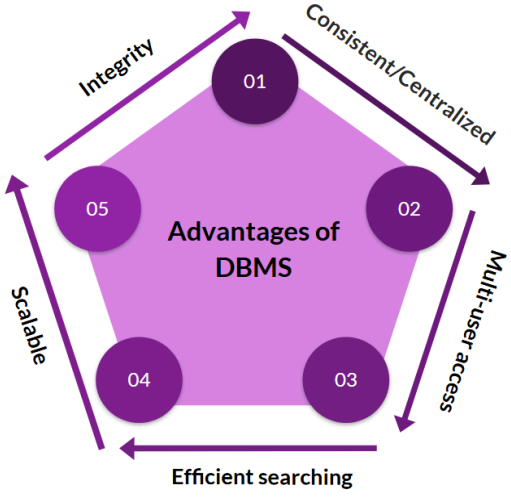
- **Database**: An organised collection of related data
- **DBMS (Database Management System)**: Software used to define, manipulate, retrieve, and manage data in a database
- **Relational Model**: A model of data as relations (tables)

Why Use a Database?

Spreadsheets become inefficient as data scales. Traditional file storage and management systems struggle with:

- Multi-user access
- Data integrity
- Security and scalability
- Scattered data and inconsistency

A DBMS allows applications and users to access and manage the data stored in databases. Some examples of widely used database management systems (DBMSs) include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and MongoDB.



upGrad

Relational Model

- Data is stored in **relations** (tables)
- An entity is any real-world object that is represented in the form of tables in a database
- Each relation is made up of **tuples** (rows) and **attributes** (columns)
- Tables can be related through **keys**

Features of the Relational Model

Feature	Description
Relational Structure	A relation has 2 parts: **Relation Heading** (identifier) and **Relational Body** Changes in storage do not affect the table structure
Data Integrity	Ensured via constraints (e.g., type, nullability)
Logical Data Independence	Schema changes do not impact existing queries
Data Manipulation	SQL allows us to retrieve, insert, and update data

Types of Constraints

- **Domain Constraint**: Values must belong to a specified data type/domain
- **Entity Integrity**: Primary key must be *unique* and *not null*
- **Referential Integrity**: Foreign keys must reference valid tuples

Keys

Key Type	Purpose
Primary Key	Uniquely identifies a tuple
Candidate Key	Any unique attribute(s)
Foreign Key	Links to primary key of another table
Composite Key	A key made of more than one column

- The primary key uniquely identifies records and **cannot be null**
- The foreign key links to the primary key in another table and **can be null**; however, **non-null values must match the referenced primary key**

Database Schema & ER Diagrams

Database Schema

- The logical structure of a database, defined as a collection of tables
- Includes table names, column names/types, keys, relationships
- Two commonly used schema structures are **star** and **snowflake**

ERD (Entity-Relationship Diagram)

- Visual tool to design databases
- Elements:
  - **Entity**: Object (table)
  - **Attribute**: Property (column)
  - **Relationship**: Association between entities

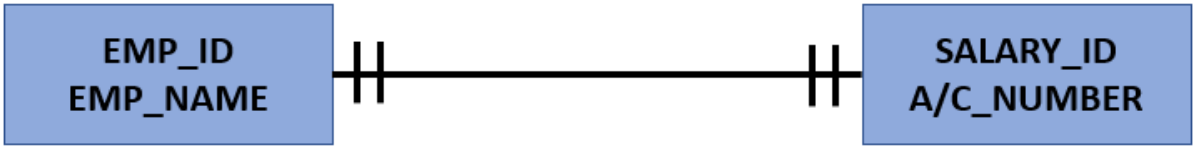
Example: A Student entity related to a Course entity via Enrolment

Think of schema as the structure of a building, and ERD as the blueprint for that structure.

Cardinality refers to the numerical relationship between entities. It defines how many instances of one entity relate to instances of another.

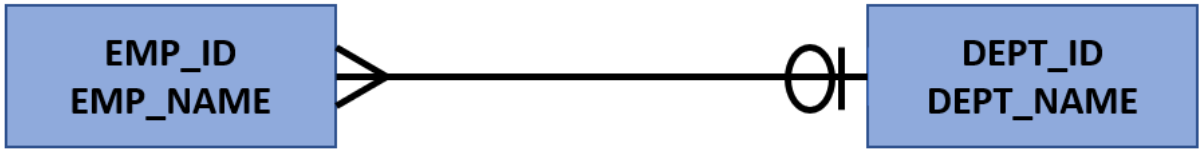
Cardinality can be expressed in the following three ways (notice the connectors):

One-to-One Relationship



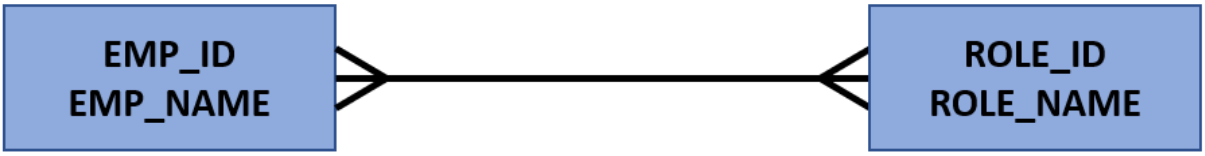
- **Only one employee corresponds to one salary account**
- **Only one salary account corresponds to one employee**

Many-to-Zero/One Relationship



- **Many employees *might* correspond to a *minimum* of zero or a *maximum* of one department**
- **A *minimum* of zero or one department corresponds to *multiple* employees**

Many-to-Many Relationship



- **Any employee in the 'Employee' table can have multiple roles**
- **Any role in the 'Roles' table can be assigned to multiple employees**

Normalisation

A process to organise data to:

- Eliminate redundancy
- Improve integrity and consistency

Normal Forms

Normal Form	Rule
1NF	Atomic (indivisible) values only
2NF	No partial dependency on part of a composite key
3NF	No transitive dependency on non-key attribute

A table in one normal form should satisfy the conditions of the preceding normal forms as well.
Always start with 1NF and progress as needed.

Databases are structured systems for managing data. A **DBMS** provides access to this data using SQL and ensures **data integrity and scalability**. The **relational model** organises data in tables connected by keys. Good database design uses **normalisation** and **ERDs** to minimise redundancy and optimise storage.

Further Reading

- Codd's 12 Rules of RDBMS
- ERD Tutorial (Lucidchart)

Ask an LLM

1. What are the advantages of a relational database over a spreadsheet?
2. Differentiate between primary key and candidate key with examples.
3. Given an unnormalised table, apply 1NF, 2NF, and 3NF.
4. Draw an ERD for a system with Students, Courses, and Instructors.
5. How does referential integrity affect data deletion?

Data Modelling

Key Definitions

- **Data Modelling**: The process of designing and structuring data to reflect real-world entities and relationships for effective storage, retrieval, and analysis.
- **Schema**: The structure of a database, described in terms of tables, attributes, and relationships.

A **data model** is like a **floor plan**. It doesn't build the system, but it's essential for building the right one.

upGrad

The Data Modelling Lifecycle

This is a continuous, iterative cycle, especially in data-driven applications.

Stage	Description
Design	Create an ERD from business requirements
Development	Implement the structure using a DBMS - tables and relationships
Manipulation	Populate tables with initial data
Revision	Iterate based on feedback or changing requirements
Production	Deploy the database and automate data flow (ETL pipelines, APIs)
Maintenance	Evolve the schema to accommodate structural or business changes

Relational Schemas

A **relational schema** defines how data is organised into tables and the relationships among them.

Advantages

- Clear, well-structured representation of data
- Optimised for **querying** (e.g., joins, indexing)
- Enforces **data integrity** via constraints

Disadvantages

- **Rigid structure,** not ideal for fast-evolving data
- Limited support for **semi-structured formats** (JSON, XML)
- Difficult to scale horizontally across multiple servers

upGrad

Relational vs Non-Relational Schemas

Feature	Relational	Non-Relational
Model Type	Tables, rows, and foreign keys	Documents, key-value, wide-column, graphs
Design Principle	**ACID** – atomicity, consistency, isolation, durability	**BASE** – basically available, soft state, eventual consistency
Structure	Fixed schema	Flexible schema
Scalability	Vertical	Horizontal
Query Language	SQL	Varies (MongoDB, Cassandra, etc.)
Use Case	Transactional systems	Analytics, distributed apps, IoT, ML pipelines

Use relational models for **structured, transactional data** and non-relational for **agility and scale**.

To summarise,

```
None
[Business Requirement]
        ↓
     [Design] ——————————> [ERD]
        ↓                    ↓
[Relational Schema]    [Tables + Relationships]

        ↓

[Database Development → Data Manipulation →
 → Revision → Production → Maintenance]
```

Ask an LLM

1. Draw a simple ERD for a flight booking system.
2. In what stage of the data modelling cycle would you validate foreign key relationships?
3. Compare relational and non-relational databases using the ACID/BASE framework. Suggest a real-world scenario where a NoSQL model is better than a relational one.

upGrad

Basic SQL Querying

What is SQL?

SQL (Structured Query Language) is a declarative programming language used to:

- Define the structure of databases
- Insert, retrieve, update, and delete data
- Enforce constraints and maintain data integrity

SQL is *declarative*: you specify *what* you want, not *how* to compute it.

SQL Commands

Data Definition Language (DDL)

Used to define or modify the database schema.

CREATE

Create databases and tables.

```
SQL
CREATE DATABASE <database_name>;
USE <database_name>;
```

Create a table named `movie_table` using the `CREATE TABLE` command.

```
SQL
CREATE TABLE movie_table ( mov_id int PRIMARY KEY,
movie_title varchar (30) NOT NULL);
```

The `CREATE` command also allows using a CHECK constraint to prevent the entry of invalid data.

ALTER

Modify schema objects

```sql
-- Modify table name
ALTER TABLE movie_table RENAME movie_main;

-- Adding columns to table
ALTER TABLE movie_main ADD country varchar (30);
```

DROP

Remove schema objects

```sql

-- Drop the table permanently

DROP TABLE movie_main;
```

TRUNCATE

Remove all records from a table

```sql

-- Remove all data from the table but keep the structure

TRUNCATE TABLE movie_main;
```

upGrad

Data Manipulation Language (DML)

Used to manipulate data:

INSERT

Add new rows to a table

```sql
INSERT INTO movie_main
(mov_id, movie_title, country)
VALUES
(210, 'The Amityville Horror', 'USA'),
(211, 'The Dark Knight', 'USA'),
(212, 'The Lion King', 'UK');
```

UPDATE

Modify existing rows; uses the SET command

```sql
-- Syntax
UPDATE <tablename>
SET col1= val1, col2=val2, …
WHERE condition;

-- example
UPDATE movie_main SET country = 'USA' WHERE mov_id =
212;
```

DELETE

Remove rows (slower than `TRUNCATE`)

```sql
DELETE FROM movie_main
WHERE mov_id = 212;
```

Additional notes:

- Use this link to learn more about different data types that can be used in MySQL
- Refer to this link to explore more about SQL commands other than DDL and DML
- **DROP** versus **TRUNCATE** versus **DELETE:**

DROP	TRUNCATE	DELETE
DDL command. Drops the table/ from the schema irreversibly.	DDL command. Drops the table rows only; keeps the schema intact.	DML command. Deletes particular rows specified by the user.

SELECT

`SELECT` statements are used to retrieve or select data from a table, and the returned data is stored in a result table, called a **result set**. We can use several other commands and queries along with `SELECT` statements to get the desired result.

Syntax:

```sql
SELECT {* | column1, column2..}
-- '*' selects all columns from a table
FROM {table1 AS [alias]}, {table_name2 AS [alias]}...
[WHERE  condition]
[ORDER BY  {column_name | column_#  [ ASC | DESC ]};
```

Clauses:

- `WHERE`:
    - Used as a conditional statement to filter out data
    - Operators used with `WHERE`: `OR`, `AND`, `IN`, and `BETWEEN`
- `ORDER BY`:
    - Takes the comma-separated column name(s) on which the data will be sorted either numerically or alphabetically
    - Each column name must be followed by ASC or DESC to specify ascending order or descending order for that column
    - If nothing is mentioned, the default sort is ascending

INSERT... SELECT Statements

The `INSERT ... SELECT` statement is used to insert data into a table by selecting data from another table.

```SQL
INSERT INTO target_table (column1, column2, ...)
SELECT column1, column2, ...
FROM source_table
WHERE condition;
```

Example: copying names and emails of active users from the `users` table into an `email_list` table

```SQL
INSERT INTO target_table (column1, column2, ...)
SELECT column1, column2, ...
FROM source_table
WHERE condition;
```

Note: Ensure that the columns in both the `SELECT` and `INSERT` match in number and data type.

upGrad

Common Keywords

Keyword(s)	Explanation
`LIMIT`	Returns only the specified number of rows
`DISTINCT`	Returns only unique values in the result
`IS NULL`	Filters rows where the column value is NULL
`IS NOT NULL`	Filters rows where the column has a value (not NULL)

Syntax	Required / Optional
SELECT <col1, col2, ..>	Required
FROM <table_name>	Required
WHERE condition	Optional
ORDER BY <order_col1, order_col2>	Optional
LIMIT <n_rows>	Optional

upGrad

Operators

Arithmetic Operators

Used to perform basic mathematical operations. The operations follow the BODMAS rule.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

Comparison Operators

Used to compare two values. Returns **TRUE** or **FALSE** as the output.

Operator	Description
=	Equal to
! = or <>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

upGrad

Logical Operators

Used to combine multiple conditions. Also returns **TRUE** or **FALSE**.

Operator	Description
AND	Returns TRUE if both conditions are TRUE
OR	Returns TRUE if at least one condition is TRUE
NOT	Negates a condition

Special Operators

These operators help filter results based on sets, ranges, or conditions involving subqueries.

Operator	Description	Example
ALL	TRUE if the condition is true for **all** values in the subquery	`price > ALL (SELECT price FROM Products WHERE category = 'A')`
ANY	TRUE if the condition is true for **any one** of the values	`salary > ANY (SELECT salary FROM Employees WHERE department = 'Sales')`
SOME	Synonym for ANY	`age < SOME (SELECT age FROM Students)`
BETWEEN	Checks whether a value is **within a range**, inclusive of boundary values	`SELECT * FROM Orders WHERE total BETWEEN 100 AND 500;`
IN	Checks if a value is **present in a given list or subquery result**	`SELECT * FROM Employees WHERE department IN ('HR', 'Finance', 'IT');`
EXISTS	Returns TRUE if a **subquery returns any rows** Often used to test **record existence** in related tables	`SELECT name FROM Customers WHERE EXISTS (SELECT * FROM Orders WHERE Orders.customer_id = Customers.id);`

upGrad

Single-Row Functions

Single-row functions operate on one row at a time and return one result per row. They are commonly used in `SELECT`, `WHERE`, or `ORDER BY` clauses.

Category	Function(s)	Description	Example
String	• `UPPER()` • `LOWER()` • `LENGTH()` • `CONCAT()` • `SUBSTRING()`	Manipulate or extract text	`SELECT UPPER(name) FROM Students;`
Numeric	• `ABS()` • `ROUND()` • `FLOOR()` • `CEIL()`	Perform numerical operations	`SELECT ROUND(salary, 0) FROM Employees;`
Date/Time	• `NOW()` • `CURDATE()` • `YEAR()` • `MONTH()` • `DATEDIFF()`	Extract or calculate date values	`SELECT YEAR(birthdate) FROM Users;`
Conversion	• `CAST()` • `CONVERT()`	Convert data from one type to another	`SELECT CAST('123' AS UNSIGNED);`

General Functions

Apart from the above categories, there are some more general single-row functions:

Function	Description
`COALESCE()`	Returns the first non-null value
`NULLIF()`	Returns `NULL` if both arguments are equal
`IFNULL()`	Replaces `NULL` with another value
`ISNULL()`	Checks if values are `NULL`

upGrad

Aggregate Functions in MySQL

Aggregate functions operate on **multiple rows** and return a **single result** (summary) for a group or entire table. Commonly used with `GROUP BY`.

Common Aggregate Functions

Function	Description
`COUNT()`	Counts rows (can exclude NULLs)
`SUM()`	Adds up values in a column
`AVG()`	Calculates average value
`MAX()`	Returns the highest value
`MIN()`	Returns the lowest value

With `GROUP BY`

Used to get aggregate values per group:

```sql
SELECT department, AVG(salary)
FROM Employees
GROUP BY department;
WHERE condition;
```

Note: Aggregate functions often ignore `NULL` values unless explicitly handled.

`HAVING` Clause

The `HAVING` clause is used to **filter groups** after aggregation, similar to `WHERE`, but it works **with aggregate functions**.

Key difference:

- `WHERE` filters **rows before** grouping.
- `HAVING` filters **groups after** `GROUP BY`.

Example:

```sql
SELECT department, COUNT(*) AS emp_count
FROM Employees
WHERE status = 'active'
GROUP BY department
HAVING COUNT(*) > 5;
```

Tip: Use `HAVING` only when you need to filter based on aggregate results.

upGrad

Nested Queries (Subqueries) in MySQL

Subqueries are queries nested inside another SQL statement. They help filter, compare, or create temporary/intermediate results.

FROM Subquery (Derived Table)

Used to treat the result of a subquery as a temporary table.

```sql
SELECT AVG(salary)
FROM (SELECT * FROM employees WHERE dept = 'Sales') AS temp;
```

- The subquery in the FROM clause is evaluated first, and then the results of the evaluation are stored in a new temporary relation.
- Next, the outer query is evaluated, selecting only those columns from the temporary relation that satisfy the requirements of the outer query.

WHERE / HAVING Subquery

Used to filter rows based on the result of another query.

```sql
SELECT name FROM employees
WHERE dept_id IN (SELECT id FROM departments WHERE region = 'Asia');
```

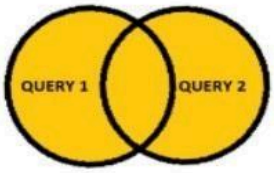
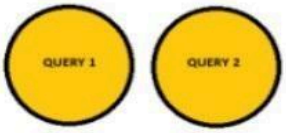
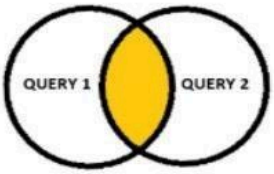
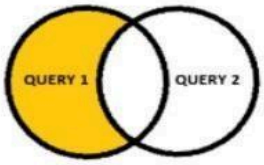
- First, the subquery executes, and we get a set of values in the WHERE/HAVING clause from the inner query complying with its requirements.
- Then, the outer query executes it and returns the required attributes from the table that satisfy the resultant WHERE/HAVING condition.

Note:

- Subqueries can return a single value, a list, or even a full table
- Assign an alias (`AS temp`) to the subquery to avoid errors
- Can be used in `SELECT`, `FROM`, `WHERE`, or `HAVING` clauses
- Must be enclosed in parentheses `( )`

upGrad

Set Operations

Operation	Condition	Diagrammatic Representation	Description	Syntax
UNION	1. Must be applied on the result of a SELECT Query. 2. The columns for both tables must be the same.		Returns the (unique) union of the select statements.	SELECT * FROM table1 UNION SELECT * FROM table2
UNION ALL			Returns all the values from both the select statements.	SELECT * FROM table1 UNION ALL SELECT * FROM table2
INTERSECT			Returns the intersection of the two select statements.	SELECT * FROM table1 INTERSECT SELECT * FROM table2;
MINUS			Returns the values from the first select statement after removing the common elements in both the select statements.	SELECT * FROM table1 MINUS SELECT * FROM table2;

MySQL does not support INTERSECT and MINUS. Hence, inner joins or nested queries can be used as alternatives.

Nested queries result in a lot of information loss, as they trim the tables to a scope limited to their specified conditions. This is not the case with JOINs.

When we join two or more tables, we have access to all the information around the shared entity that joins these tables.

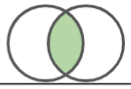
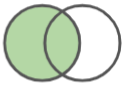
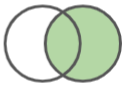
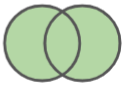
upGrad

Joins

Syntax:

```sql
SELECT
<Columns_from_Table1>,
<Columns_from_Table2>
FROM <Table1> AS <table_alias1>
JOIN <Table2> AS <table_alias2>
ON <joining_condition>
```

Types of joins

Join Type	Diagrammatic Representation	Left Table	Right Table	Output Table
INNER JOIN (Default)		id | val1 1 | 10 2 | 20 3 | 30	id | val2 2 | 20 3 | 30 4 | 40	id | val1 | val2 2 | 20 | 20 3 | 30 | 30
LEFT OUTER JOIN		id | val1 1 | 10 2 | 20 3 | 30	id | val2 2 | 20 3 | 30 4 | 40	id | val1 | val2 1 | 10 | - 2 | 20 | 20 3 | 30 | 30
RIGHT OUTER JOIN		id | val1 1 | 10 2 | 20 3 | 30	id | val2 2 | 20 3 | 30 4 | 40	id | val1 | val2 2 | 20 | 20 3 | 30 | 30 4 | - | 40
FULL OUTER JOIN		id | val1 1 | 10 2 | 20 3 | 30	id | val2 2 | 20 3 | 30 4 | 40	id | val1 | val2 1 | 10 | - 2 | 20 | 20 3 | 30 | 30 4 | - | 40
CROSS JOIN		id | val1 1 | 10 2 | 20	id | val2 2 | 20 3 | 30 4 | 40	id | val2 1 | 20 1 | 30 1 | 40 2 | 20 2 | 30 2 | 40

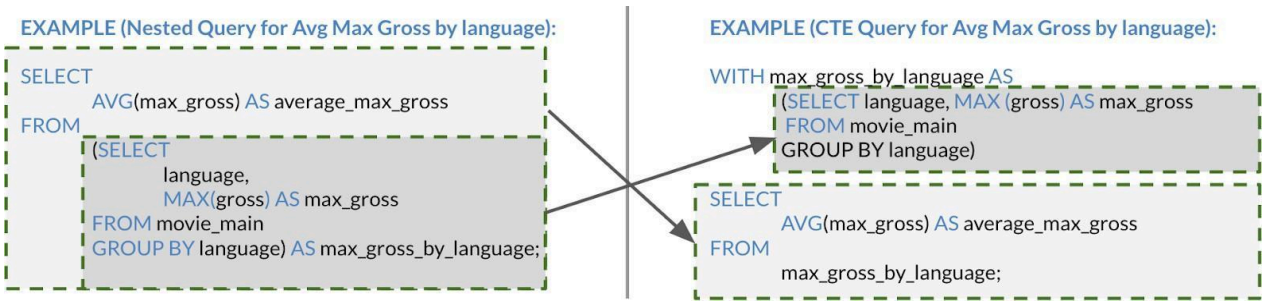
upGrad

Common Table Expressions

CTE query exists in memory only while the query is running. A CTE can be referenced multiple times in the same query.

Syntax:

```
SQL
WITH <CTE_name> AS (CTE definition/Subquery)
Outer Query
```



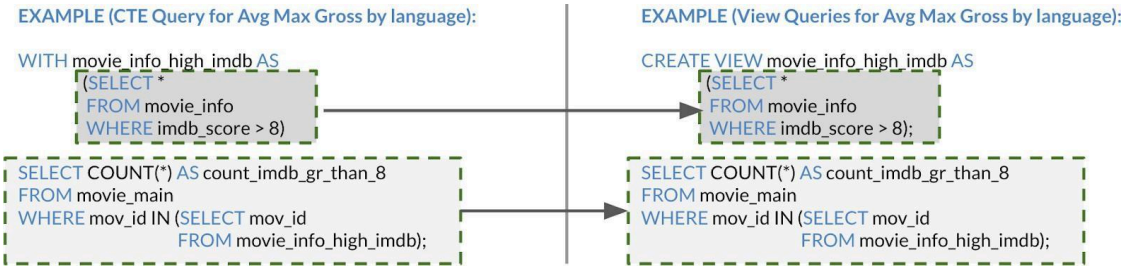
Views

A **view** is a virtual table created by operations on a single table or on multiple tables. A view can be treated like any other table and can be updated or deleted further. However, there is no separate storage layer for storing a view.

One major advantage of creating a view is data security. We can choose to display only select parts of the table.

Syntax:

```
SQL
CREATE VIEW <view_name> AS (SELECT <ColumnName(S)>
FROM <TableName>
WHERE condition ... ); //Subquery
```



CTE (`WITH`)	View (`CREATE VIEW`)
Temporary subquery (readable, modular)	Virtual table stored in the DB
Persists for one query	Persistent, can be reused across sessions
Not directly	Views can sometimes be updated (with restrictions)

Ask an LLM

1. Where can you use the `ALTER` command?
2. What is meant by shallow and deep copying a table?
3. When should you use `HAVING` and `WHERE`?

upGrad

Advanced SQL Querying

CASE Statements

Used for conditional logic in SQL, similar to if-else logic. It can be used in SELECT, WHERE, ORDER BY, etc.

Example:

```SQL
SELECT name, salary,
  CASE
    WHEN salary > 100000 THEN 'High'
    WHEN salary BETWEEN 50000 AND 100000 THEN 'Medium'
    ELSE 'Low'
  END AS salary_band
FROM employees;
```

- Once we classify rows into different categories using CASE statements, we can perform actions and operations on them just like a usual column to enrich our analyses.
- CASE statements lacking an ELSE command simply return a NULL in the rows that fail to satisfy any of the listed conditions.
- CASE statements can be used anywhere you have column definitions.
- Can be used with simple queries, aggregated queries, and grouped queries as well.

upGrad

Window Functions

- Operate over a window (set of rows) defined by PARTITION BY and ORDER BY
- Unlike GROUP BY, do not reduce the result set; output one row per input row

Syntax:

```sql
function_name(column) OVER (
  PARTITION BY partition_col
  ORDER BY order_col
  ROWS BETWEEN frame_specifications)
```

Partitioning and Ordering

- PARTITION BY: Resets the function for each group
- ORDER BY: Defines the sequence within each partition

Example:

```sql
SELECT
  department,
  employee,
  salary,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
dept_rank
FROM employees;
```

Ranking Functions

Function	Description
RANK()	Ranks with gaps for ties
DENSE_RANK()	Ranks without gaps
ROW_NUMBER()	Unique row number, no ties

Aggregate Window Functions

Apply aggregate functions over a moving or fixed window of rows.

Examples:

```sql
-- Running total
SUM(sales) OVER (PARTITION BY region ORDER BY date) AS running_total

-- Average salary per department
AVG(salary) OVER (PARTITION BY department) AS dept_avg
```

Note: These do not collapse rows like GROUP BY.

LAG & LEAD

Access values from previous or next rows within the window.

Function	Purpose
LAG()	Value from previous row
LEAD()	Value from next row

Example:

```sql
SELECT
  name,
  month,
  sales,
    LAG(sales, 1) OVER (PARTITION BY name ORDER BY month) AS
prev_month_sales,
    LEAD(sales, 1) OVER (PARTITION BY name ORDER BY month) AS
next_month_sales
FROM sales_data;
```

Frame Specifications

Control how many rows are visible to the window function.

Syntax:

```sql
ROWS BETWEEN preceding_rows AND following_rows
```

Options:

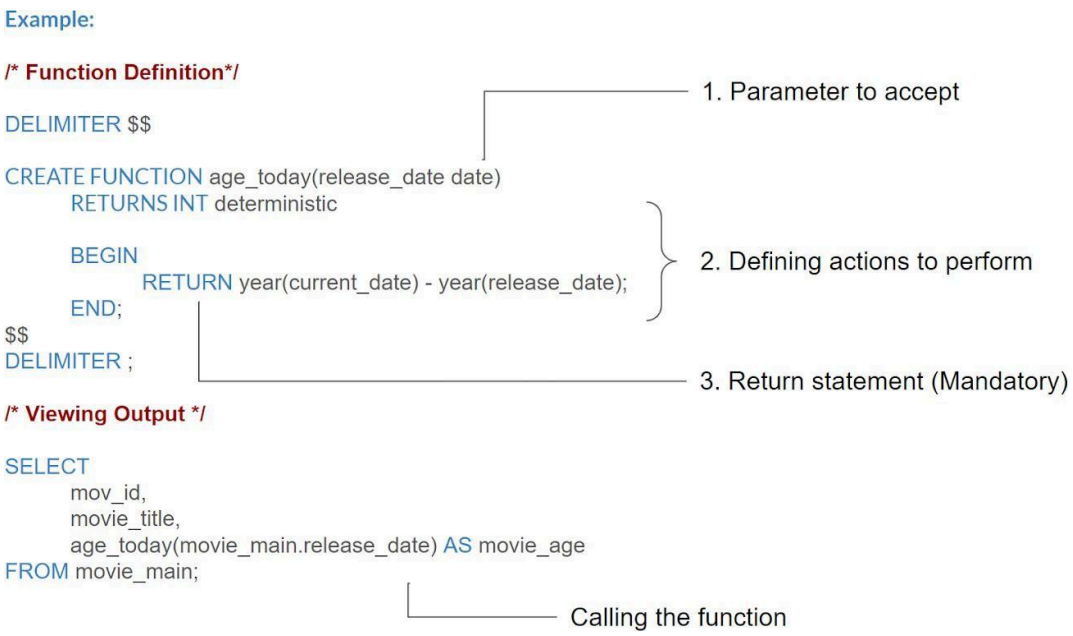
Clause	Description
UNBOUNDED PRECEDING	From start of the partition
CURRENT ROW	Up to the current row
N PRECEDING / FOLLOWING	Relative row offsets
UNBOUNDED FOLLOWING	Up to end of the partition

Example:

```sql
SUM(sales) OVER (
  PARTITION BY region
  ORDER BY date
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
) AS three_day_total
```

UDF (User-defined Functions)

A UDF or user-defined function has a lot of benefits. It makes the code more readable, and it is easier to use once defined.



Note: You need to use the DROP FUNCTION to recompile an already defined function.

```sql
DROP FUNCTION IF EXISTS function_name;
```

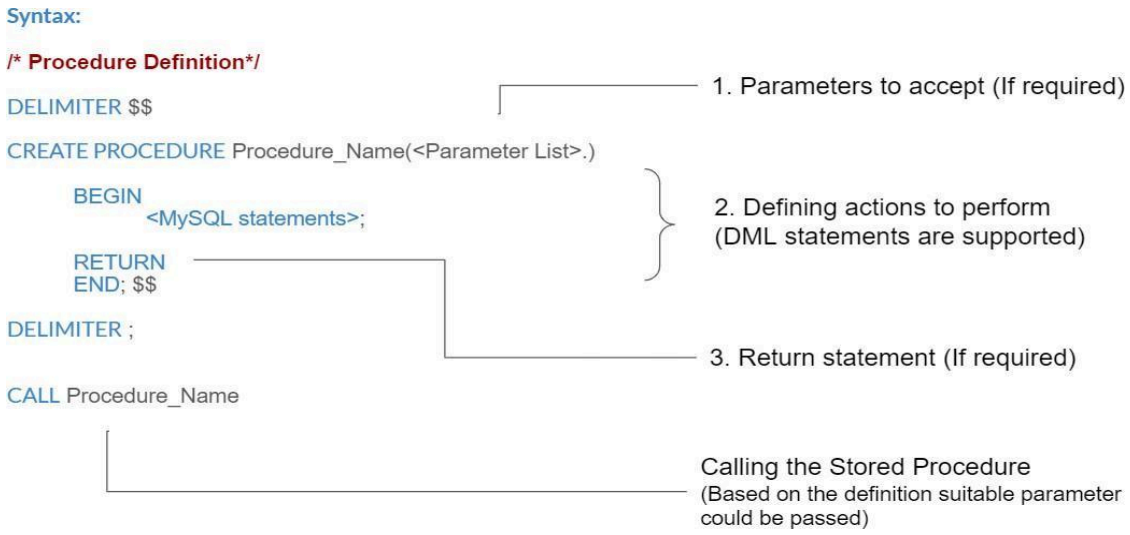
upGrad

Stored procedures

Stored Procedures offer a number of advantages over UDFs. Some of the advantages include the following:

- Stored Procedures do not necessarily need to have a RETURN statement.
- Support both input and output parameters.
- Also support DML commands such as UPDATE, DELETE, and INSERT.

The syntax structure for a stored procedure is similar to that for a UDF:



Aspect	Stored Procedure	User Defined Function
Returning a Value	May or may not return a value.	Mandatory to return a value.
Compilation Order	Compiled once, executed whenever called.	Compiled every time it is called / executed.
Calling Procedure / Function	Can call a Function.	Cannot call a Procedure.
DML / DDL Statements Available	Can use SELECT, INSERT, UPDATE and DELETE.	Can only use SELECT.
Compatibility with Try-Catch Blocks	Can use TRY-CATCH block to handle exceptions.	Cannot use TRY-CATCH blocks.
Compatibility with CTEs	Can use CTEs.	Cannot use CTEs.
Type of Parameters Accepted	Can have Input or Output parameters.	Can only have Input parameters.

upGrad

Indexing

Indexing is an effective way to optimise query execution, as it selects the required data values instead of processing the entire table. The syntaxes for CREATING, ADDING, and DROPPING an index are as follows:

Creating an Index

```SQL
CREATE INDEX index_name ON table_name (column1, column2,..);
```

Adding an Index

```SQL
ALTER TABLE table_name ADD INDEX index_name(column1, column2,..);
```

Dropping an Index

```SQL
ALTER TABLE table_name DROP INDEX index_name;
```

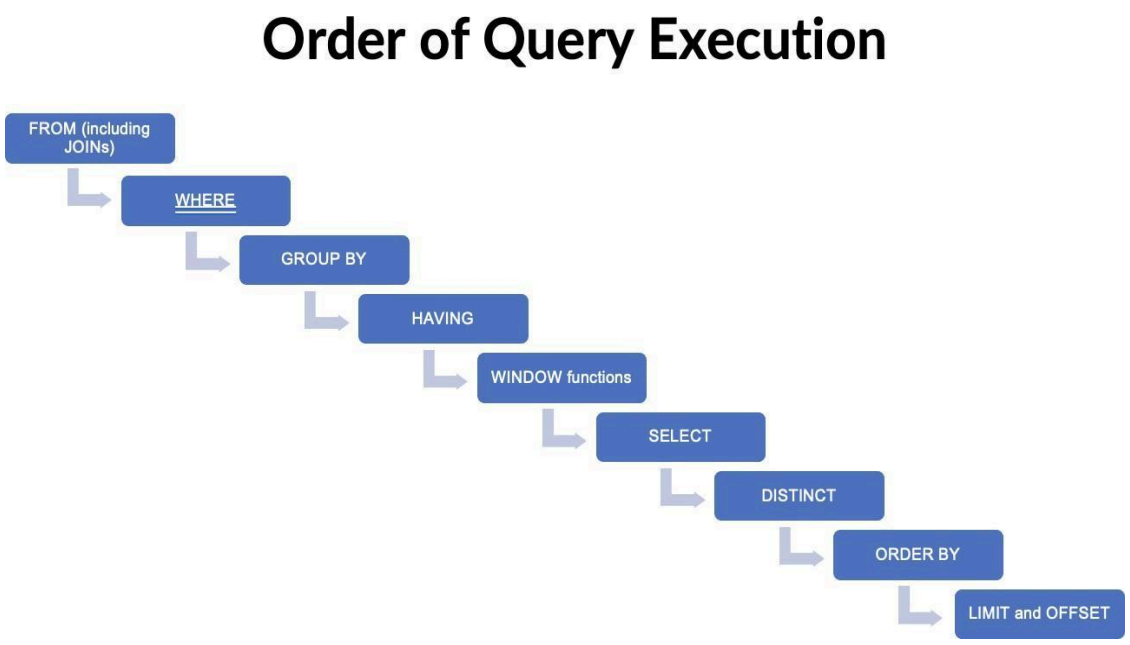
There are two types of indexed data structures: Clustered and Non-clustered indexes.

Additionally, there are multiple types of indexes within our tool set.

Type	Example(s)	Use
Unique	mov_id	Search unique value columns, e.x., user_id.
Primary	mov_id	Primary key of table.
Regular	language /language_country	Searched regularly, not necessarily unique.
Fulltext	movie_title	To search text columns.
Descending	release_date DESC	To look up recent data.

Order of query execution

The order in which the different SQL statements are executed is different from the order of their appearance in a query.



Query optimisation techniques: The points you should remember while writing a query are as follows:

- Use inner joins wherever possible to avoid any unnecessary rows in the resultant table.
- Apply all the required filters to get only the required data values from multiple tables.
- Index the columns that are frequently used in the WHERE clause.
- Avoid using DISTINCT while using the GROUP BY clause, as it slows down query processing.
- Avoid using SELECT * as much as possible. Select only the required columns.
- Use the ORDER BY clause only if it is absolutely necessary, as it is processed late in a query.
- Avoid using LIMIT and OFFSET as much as possible. Instead, apply appropriate filters using the WHERE clause.

Joins versus nested queries

Some notable differences between join operations and nested queries are below:

- Join operations are better in optimisation.
- Join operations take more time, as they fetch the whole data with attributes, unlike nested queries, which fetch only the relevant data from the tables.
- Nested queries are easier to read, understand, and evaluate.
- Nested queries can be used to return a scalar value as well as a row set, whereas Joins can only be used to return rows.
- Joins are powerful relational operators, whereas nested queries are not.

Ask an LLM

- How is ROW_NUMBER() different from RANK() when used in a window?
- What does the frame clause ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW mean?
- What is the difference between a stored procedure and a user-defined function (UDF)?
- When would you use a UDF inside a SELECT statement?
- Why is indexing important in SQL?
- What are the trade-offs of having too many indexes on a table?
- Can a stored procedure return a result set? How?

upGrad

Topics Commonly Asked in SQL Interviews

1. SQL Joins, especially INNER JOIN vs LEFT JOIN
2. GROUP BY with aggregate functions and HAVING
3. Subqueries in SELECT, WHERE, and FROM clauses
4. Window functions, ROW_NUMBER(), RANK(), PARTITION BY
5. Difference between WHERE and HAVING
6. Use of DISTINCT, LIMIT, IS NULL, COALESCE
7. Indexing: what it is, and trade-offs involved
8. CTE vs View and when to use each
9. Writing queries to fetch nth highest value (e.g., second highest salary)
10. Stored Procedures and UDFs - what they are, when to use them