SQL Interview Questions for Experienced Candidates (3+ years)

Data Types & Miscellaneous Concepts

Normalization & Schema Design

Database Design Principles

Indexing Strategies & Keys

1. What is a SQL index and what are different types of indexes (clustered, non-clustered, unique, etc.)?

Answer:
An **index** in SQL is a database object that enhances the speed of data retrieval operations on a table by providing a fast lookup mechanism, similar to an index in a book. While indexes improve SELECT query performance, they require additional storage and can add overhead to data modification operations (INSERT, UPDATE, DELETE) because the index must be maintained as data changes.

- **Clustered Index:** Determines the physical order of data rows in the table. Each table can have only one clustered index because the data rows themselves can be sorted in only one order. The clustered index is typically created on the primary key by default.
- **Non-Clustered Index:** A separate structure from the table data that contains a sorted list of key values and pointers (row locators) to the actual data rows. A table can have multiple non-clustered indexes. Non-clustered indexes are useful for speeding up queries that filter or sort by columns other than the clustered index key.
- **Unique Index:** Ensures that all values in the indexed column(s) are unique across the table. Unique indexes are often used to enforce business rules, such as unique email addresses or usernames.
- **Composite Index:** An index that covers two or more columns. Composite indexes are useful for queries that filter or sort by multiple columns together.
- **Full-Text Index:** Specialized index for efficient searching of large text-based columns, supporting advanced search features like stemming and proximity.
- **Spatial Index:** Used for indexing spatial data types (such as geometry or geography), enabling efficient spatial queries (e.g., finding points within a region).

Syntax & Example:

- **Clustered Index:**

```sql
CREATE CLUSTERED INDEX idx_employee_id ON Employees(EmployeeID);
```

This creates a clustered index on the `EmployeeID` column of the `Employees` table. The data rows will be physically ordered by `EmployeeID`.

- **Non-Clustered Index:**

```sql
CREATE NONCLUSTERED INDEX idx_employee_lastname ON Employees(LastName);
```

This creates a non-clustered index on the `LastName` column. The index is stored separately from the table data and contains pointers to the actual rows.

- **Unique Index:**

```sql
CREATE UNIQUE INDEX idx_employee_email ON Employees(Email);
```

This ensures that all values in the `Email` column are unique.

- **Composite Index:**

```sql
CREATE INDEX idx_employee_dept_salary ON Employees(DepartmentID, Salary);
```

This creates an index on both `DepartmentID` and `Salary`. Useful for queries filtering or sorting by both columns.

- **Full-Text Index (SQL Server Example):**

```sql
CREATE FULLTEXT INDEX ON Employees(Resume);
```

This enables advanced text search capabilities on the `Resume` column.

- **Spatial Index (SQL Server Example):**

```sql
CREATE SPATIAL INDEX idx_location ON Locations(GeoPoint);
```

This allows efficient spatial queries on the `GeoPoint` column.

Summary: Indexes are essential for optimizing query performance. Choose the appropriate index type based on your query patterns and data structure. While indexes speed up data retrieval, they can slow down data modifications and require additional storage, so use them judiciously.

2. What is the difference between a heap (no clustered index) and a table with a clustered index, and how can you identify a heap table?

Answer:

A **heap** is a table that does **not** have a clustered index. In a heap, data rows are stored in no particular order, and their physical location is identified by a Row Identifier (RID). In contrast, a table with a **clustered index** stores its data rows in the order of the clustered index key, which means the physical order of the rows matches the logical order of the index.

- **Heap Table:**
  - No clustered index exists on the table.
  - Data is unordered; new rows are placed wherever space is available.
  - Row locations are tracked by RIDs (Row Identifiers).
  - Heaps are typically faster for bulk inserts and minimal indexing, but slower for searches and lookups.
- **Clustered Index Table:**
  - Has a clustered index, which determines the physical order of the data rows.
  - Data is stored and retrieved in the order of the clustered index key.
  - Only one clustered index is allowed per table, as data can be physically sorted in only one way.
  - Clustered indexes improve search, range queries, and ordered retrieval performance.

How to Identify a Heap Table:

- In SQL Server, you can query the `sys.indexes` system catalog view. If a table has an entry with `index_id = 0`, it is a heap. If it has `index_id = 1`, it has a clustered index.
- Other databases have similar metadata tables or commands to check for clustered indexes.

Syntax & Examples:

- **Create a heap (no clustered index):**

```
CREATE TABLE HeapTable (
    ID INT,
    Name VARCHAR(50)
);
-- No clustered index is created, so this table is a heap.
```

This table is a heap because it does not have a clustered index.

- **Add a clustered index to convert the heap:**

```
CREATE CLUSTERED INDEX idx_id ON HeapTable(ID);
```

After this, `HeapTable` is no longer a heap; it is now ordered by `ID`.

- **Identify a heap in SQL Server:**

```sql
SELECT name, index_id
FROM sys.indexes
WHERE object_id = OBJECT_ID('HeapTable');
```

If you see a row with `index_id = 0`, the table is a heap. If `index_id = 1` exists, it has a clustered index.

Heap vs Clustered Index Table Comparison

Aspect	Heap Table	Clustered Index Table
Physical Order	Unordered	Ordered by index key
Index ID (SQL Server)	0	1
Performance	Faster for bulk inserts, slower for searches	Faster for searches, range queries, and ordered retrieval
Use Case	Staging tables, ETL, minimal indexing	Transactional tables, reporting, frequent lookups

Summary: A heap is a table without a clustered index, storing data in no defined order and identified by `index_id = 0`. A clustered index table stores data in the order of the index key (`index_id = 1`), improving search and range query performance. Choose based on your workload and query patterns.

---

3. What is the difference between a PRIMARY KEY and a UNIQUE KEY (or unique index) in SQL?

Answer:
Both **PRIMARY KEY** and **UNIQUE KEY** (or unique index) are used to enforce uniqueness of values in one or more columns, but they have important differences in behavior, usage, and constraints.

PRIMARY KEY vs UNIQUE KEY

Aspect	PRIMARY KEY	UNIQUE KEY
Purpose	Uniquely identifies each row in a table	Ensures all values in the column(s) are unique
Number per Table	Only one PRIMARY KEY allowed	Multiple UNIQUE KEYs allowed
NULL Values	Cannot contain NULLs (all columns must be NOT NULL)	Can contain a single NULL (in most databases); some allow multiple NULLs
Index Type	Creates a unique clustered index by default (if none exists)	Creates a unique non-clustered index by default
Usage	Main identifier for table rows; used for relationships (foreign keys)	Enforces alternate unique constraints (e.g., email, username)

Syntax & Example:

- **Defining PRIMARY KEY and UNIQUE KEY in CREATE TABLE:**

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,         -- PRIMARY KEY: unique, not null, only one
per table
    Email VARCHAR(100) UNIQUE,          -- UNIQUE KEY: alternate unique
constraint, can be null
    PhoneNumber VARCHAR(20)
);
```

Here, `EmployeeID` is the primary key (unique, not null). `Email` is a unique key (can be null in some databases).

- **Adding a UNIQUE KEY after table creation:**

```
ALTER TABLE Employees ADD CONSTRAINT uq_emp_phone UNIQUE (PhoneNumber);
```

This adds a unique constraint to `PhoneNumber`, ensuring all phone numbers are unique (but can be null).

- **Composite Keys:** Both PRIMARY KEY and UNIQUE KEY can be defined on multiple columns (composite keys).

```
-- Composite PRIMARY KEY
CREATE TABLE ProjectAssignments (
    EmployeeID INT,
    ProjectID INT,
    AssignmentDate DATE,
    PRIMARY KEY (EmployeeID, ProjectID)
);

-- Composite UNIQUE KEY
ALTER TABLE Employees ADD CONSTRAINT uq_emp_email_phone UNIQUE (Email,
PhoneNumber);
```

Summary: Use **PRIMARY KEY** for the main unique identifier of a table (no NULLs, only one per table). Use **UNIQUE KEY** for enforcing alternate unique constraints (can be multiple per table, allows NULLs). Both help maintain data integrity, but PRIMARY KEY is essential for relational design and foreign key references.

---

4. What are index “forwarding pointers” in a heap table, and how do they affect query performance?

Answer:

In SQL Server, a **heap table** is a table without a clustered index. When a row in a heap table is updated and the new data no longer fits in its original location (data page), SQL Server may move the row to a new page with enough space. To maintain references from non-clustered indexes or other pointers, SQL Server leaves a

forwarding pointer at the original location. This pointer directs any access to the old location to the new location of the row.

- **How Forwarding Pointers Are Created:**
  - Occurs when an `UPDATE` operation increases the row size and there is not enough space on the original data page.
  - The original row is replaced with a small stub (the forwarding pointer), and the full row is moved to a new page.
- **Impact on Query Performance:**
  - When a query or non-clustered index seeks the row, SQL Server first reads the original page, finds the forwarding pointer, and then reads the new page to retrieve the actual row data.
  - This results in **additional I/O operations** (extra page reads), increasing latency and reducing query performance, especially if many forwarding pointers exist.
  - Forwarding pointers can accumulate over time in heavily updated heap tables, leading to significant performance degradation.
- **Resolution and Maintenance:**
  - Forwarding pointers are removed by rebuilding the heap (using `ALTER TABLE ... REBUILD`) or by creating a clustered index on the table (which reorganizes the data and eliminates forwarding pointers).
  - Regular maintenance is recommended for heaps with frequent updates to avoid performance issues.

Example:

```sql
-- Update a row in a heap table that causes it to move
UPDATE HeapTable SET Name = REPLICATE('A', 1000) WHERE ID = 1;
-- If the new value doesn't fit on the original page, a forwarding pointer is
created.
```

Subsequent queries accessing this row will first read the original page, follow the forwarding pointer, and then read the new page, resulting in extra I/O.

How to Remove Forwarding Pointers:

```sql
-- Rebuild the heap to remove forwarding pointers
ALTER TABLE HeapTable REBUILD;

-- Or create a clustered index (which also removes forwarding pointers)
CREATE CLUSTERED INDEX idx_id ON HeapTable(ID);
```

Summary: Forwarding pointers in heap tables are created when updated rows are moved to new pages, leaving a pointer behind. They cause extra I/O and degrade query performance, especially with frequent updates. Regularly rebuilding the heap or adding a clustered index can eliminate forwarding pointers and restore optimal performance.

---

5. What is a composite index, and how do you choose the order of columns in it for optimal performance?

Answer:

A **composite index** (also called a **multi-column index**) is an index that includes two or more columns from the same table. Composite indexes are useful for optimizing queries that filter or sort on multiple columns together. The **order of columns** in a composite index is critical for its effectiveness and determines which queries can benefit from the index.

- **How Composite Indexes Work:**
  - The index is built in the order of the specified columns (e.g., `(A, B, C)`).
  - Queries that filter on the **leading column(s)** (e.g., `A` or `A, B`) can use the index efficiently.
  - If a query filters only on a non-leading column (e.g., `B` or `C` without `A`), the index may not be used or may be less efficient.
- **Choosing Column Order:**
  - Place the **most selective** (most unique) column first, especially if it is commonly used in `WHERE` or `JOIN` conditions.
  - Consider the most common query patterns: columns used together in filters or sorts should appear in the same order as in the queries.
  - If sorting is important (e.g., `ORDER BY`), include those columns in the index in the same order.
- **Index Coverage:**
  - A composite index on `(A, B, C)` can be used for queries filtering on `A`, `A, B`, or `A, B, C`, but not efficiently for `B` or `C` alone.

Composite Index Usage Examples

Index Definition	Query Example	Index Used?
`(DepartmentID, Salary)`	`WHERE DepartmentID = 10`	Yes
`(DepartmentID, Salary)`	`WHERE DepartmentID = 10 AND Salary > 50000`	Yes
`(DepartmentID, Salary)`	`WHERE Salary > 50000`	No (or less efficient)

Syntax & Example:

```sql
-- Composite index on DepartmentID and Salary
CREATE INDEX idx_dept_salary ON Employees(DepartmentID, Salary);
```

This index is optimal for queries like `WHERE DepartmentID = ? AND Salary > ?` or `WHERE DepartmentID = ?`. It is not efficient for queries filtering only on `Salary`.

- **Tip:** Analyze your most frequent queries and their filter conditions to determine the best column order for composite indexes.
- **Best Practice:** Avoid creating overly wide composite indexes (many columns), as they increase storage and maintenance overhead.

Summary: A composite index covers multiple columns and is most effective when queries filter on the leading column(s). Choose the column order based on selectivity and common query patterns to maximize performance.

---

6. When should you use a covering index, and how does it improve the performance of a query?

Answer:

A **covering index** is an index that contains all the columns required to satisfy a query—either as part of the index key or as included (non-key) columns. When a query can be answered entirely from the index, the database engine does not need to access the underlying table (heap or clustered index), resulting in significant performance improvements.

- **When to Use:**
  - For queries that are run frequently and always retrieve the same set of columns.
  - When you want to minimize I/O and speed up SELECT operations, especially in read-heavy workloads.
  - For queries that filter on one or more columns and return additional columns (e.g., reporting, dashboards).
- **How It Improves Performance:**
  - Eliminates the need for "bookmark lookups" or "key lookups"—the extra step of fetching data from the base table after finding matching rows in the index.
  - Reduces disk I/O and CPU usage, as all required data is available in the index structure itself.
  - Improves query response time, especially for large tables or high-concurrency environments.

Covering Index Example and Benefits

Query	Covering Index	Benefit
`SELECT Salary, FirstName FROM Employees WHERE DepartmentID = ?`	`CREATE INDEX idx_covering ON Employees(DepartmentID) INCLUDE (Salary, FirstName);`	Query is satisfied entirely from the index—no table lookup needed.

Syntax & Example:

```
-- Create a covering index for a query filtering by DepartmentID and returning
Salary and FirstName
CREATE INDEX idx_covering ON Employees(DepartmentID) INCLUDE (Salary, FirstName);
```

This index allows the database to answer `SELECT Salary, FirstName FROM Employees WHERE DepartmentID = ?` using only the index, with no need to access the base table.

- **Tip:** Use covering indexes for high-traffic queries that always access the same columns. Avoid including too many columns, as this increases index size and maintenance overhead.
- **Best Practice:** Analyze your workload and use query execution plans to identify queries that would benefit most from covering indexes.

Summary: Covering indexes are highly effective for optimizing read-heavy workloads with predictable query patterns. They reduce I/O by allowing the database to serve queries directly from the index, improving speed and scalability.

7. How does the existence of an index on a column affect INSERT, UPDATE, and DELETE performance on a table?

Answer:

While indexes are essential for improving the speed of **SELECT** queries, they introduce additional overhead for data modification operations such as **INSERT**, **UPDATE**, and **DELETE**. This is because the database must maintain the index structures in sync with the underlying table data.

- **INSERT:**
  - When a new row is inserted, the database must add corresponding entries to all relevant indexes.
  - If there are multiple indexes, each one must be updated, which increases the time required for the insert operation.
  - Bulk inserts can be significantly slower if many indexes exist.
- **UPDATE:**
  - If an **indexed column** is updated, the database must remove the old index entry and add a new one reflecting the updated value.
  - Updates to non-indexed columns have minimal impact on indexes, but updates to indexed columns can be costly, especially if the index is large or complex (e.g., composite or unique indexes).
  - Frequent updates to indexed columns can lead to index fragmentation, further degrading performance over time.
- **DELETE:**
  - When a row is deleted, all associated index entries must also be removed.
  - Deleting rows from a table with many indexes takes longer than from a table with few or no indexes.
  - Large-scale deletes can cause index fragmentation, requiring periodic maintenance (e.g., index rebuilds).

Impact of Indexes on Data Modification Operations

Operation	Effect of Indexes	Best Practice
INSERT	Each index must be updated with new row data, increasing insert time.	Minimize indexes on tables with heavy insert workloads.
UPDATE	Updates to indexed columns require index maintenance (remove old, add new entry).	Avoid frequent updates to indexed columns; monitor fragmentation.
DELETE	Index entries must be removed for each deleted row, slowing down deletes.	Batch deletes and rebuild indexes periodically.

Example:

```sql
-- Insert into a table with indexes
INSERT INTO Employees (EmployeeID, FirstName, LastName) VALUES (1, 'Ashish',
'Zope');
-- The database updates all relevant indexes after the insert.

-- Update an indexed column
UPDATE Employees SET LastName = 'Patil' WHERE EmployeeID = 1;
-- The index on LastName (if exists) must be updated.

-- Delete a row
DELETE FROM Employees WHERE EmployeeID = 1;
-- All index entries for EmployeeID = 1 are removed.
```

The more indexes a table has, the more work the database must do for each insert, update, or delete operation.

- **Tip:** Only create indexes that are necessary for your most important queries. Excessive indexing can degrade write performance and increase storage requirements.
- **Best Practice:** Regularly review and optimize indexes, especially on tables with heavy write activity. Consider dropping unused indexes and scheduling index maintenance (rebuild/reorganize) as needed.

Summary: Indexes improve read (SELECT) performance but slow down write operations (INSERT, UPDATE, DELETE) due to the need to maintain index structures. Always balance the number and type of indexes based on your application's read/write workload.

---

8. What is index selectivity, and why is it important for query optimization?

Answer:

Index selectivity measures how well an index distinguishes between rows in a table. It is defined as the ratio of the number of distinct values in an indexed column (or set of columns) to the total number of rows in the table. High selectivity means the column has many unique values (e.g., a primary key), while low selectivity means the column has many repeated values (e.g., a boolean or gender column).

- **High Selectivity:** An index is highly selective if most values are unique or nearly unique. Such indexes are very effective for filtering queries, as they quickly narrow down the result set to a small number of rows.
- **Low Selectivity:** An index is poorly selective if most values are repeated (few unique values). These indexes are less useful for filtering, as they do not significantly reduce the number of rows scanned.
- **Formula:** `Selectivity = (Number of Distinct Values) / (Total Number of Rows)`

Index Selectivity Examples

Column	Distinct Values	Total Rows	Selectivity	Index Usefulness
EmployeeID	10,000	10,000	1.0 (high)	Very effective
Gender	2	10,000	0.0002 (low)	Not effective
DepartmentID	20	10,000	0.002 (low)	Limited usefulness

- **Why It Matters:** The database query optimizer uses selectivity to decide whether to use an index for a query. High-selectivity indexes allow the optimizer to quickly locate a small subset of rows, making queries much faster. Low-selectivity indexes may be ignored, as scanning the index provides little benefit over scanning the entire table.
- **Composite Indexes:** Selectivity can be improved by creating composite indexes on multiple columns, especially if the combination of columns is highly unique.

Syntax & Example:

```
-- High selectivity: EmployeeID (unique for each row)
CREATE INDEX idx_employee_id ON Employees(EmployeeID);

-- Low selectivity: Gender (few unique values)
CREATE INDEX idx_gender ON Employees(Gender);
```

The `idx_employee_id` index is highly selective and efficient for lookups. The `idx_gender` index is less useful because many rows share the same value.

- **Tip:** Use indexes on columns with high selectivity (e.g., primary keys, unique identifiers) for best query performance.
- **Best Practice:** Avoid indexing columns with very low selectivity unless they are frequently used in combination with other columns in composite indexes.

Summary: Index selectivity is a key factor in query optimization. High-selectivity indexes enable efficient data retrieval and are preferred by the query optimizer. Always consider selectivity when designing indexes for your tables.

---

9. How many clustered indexes can a table have, and why?

Answer:

A table can have **only one clustered index**. This is because a clustered index determines the **physical order** of the data rows in the table, and the data can be physically sorted in only one way at a time.

- **Reason:** The clustered index sorts and stores the data rows of the table based on the index key. Since the actual data rows are organized on disk according to this key, only one such ordering is possible.
- **Non-Clustered Indexes:** In contrast, a table can have multiple non-clustered indexes. These are separate structures that reference the data rows without affecting their physical order.
- **Primary Key and Clustered Index:** By default, when you define a primary key on a table (and no clustered index exists), most databases automatically create a clustered index on the primary key column(s). However, you can explicitly specify a different column for the clustered index if needed.
- **Attempting Multiple Clustered Indexes:** If you try to create a second clustered index on a table, the database will return an error, as only one is allowed.

Clustered vs Non-Clustered Indexes

Index Type	Physical Data Order	Number Allowed per Table	Typical Use

Index Type	Physical Data Order	Number Allowed per Table	Typical Use
Clustered Index	Yes (orders data rows)	1	Primary key, main lookup column
Non-Clustered Index	No (separate structure)	Many (subject to DB limits)	Alternate queries, searches, sorts

Syntax & Example:

```
-- Create a clustered index (only one allowed per table)
CREATE CLUSTERED INDEX idx_emp_id ON Employees(EmployeeID);

-- Create multiple non-clustered indexes (allowed)
CREATE NONCLUSTERED INDEX idx_emp_email ON Employees(Email);
CREATE NONCLUSTERED INDEX idx_emp_phone ON Employees(PhoneNumber);

-- Attempting to create a second clustered index will fail
CREATE CLUSTERED INDEX idx_emp_name ON Employees(LastName); -- Error!
```

Only one clustered index can exist per table. Non-clustered indexes can be created on other columns as needed.

- **Tip:** Choose the clustered index carefully, as it affects the physical storage and performance of range queries and lookups.
- **Best Practice:** Use the clustered index for columns that are frequently searched for ranges or used in sorting, such as primary keys or date columns.

Summary: Each table can have only **one clustered index** because it defines the physical order of the data rows. You can create multiple non-clustered indexes to support additional query patterns.

---

10. What is index fragmentation, and how can it be resolved or mitigated in a large database?

Answer:
Index fragmentation refers to the condition where the physical order of pages in an index becomes misaligned with the logical order of the index keys. This misalignment leads to inefficient disk I/O, increased page reads, and slower query performance, especially for range scans and large data sets.

- **Types of Fragmentation:**
  - **Internal Fragmentation:** Occurs when index pages have excessive free space (low page density), often due to frequent row deletions or updates that reduce row size.
  - **External Fragmentation:** Happens when the logical order of index pages does not match their physical order on disk, causing additional I/O as the database engine jumps between non-contiguous pages.
- **Causes:**
  - Frequent `INSERT`, `UPDATE`, and `DELETE` operations, especially on tables with random key values (e.g., GUIDs).

  - Page splits, where new rows are inserted into full pages, causing the page to split and data to be moved to a new page.
  - Bulk data modifications or large-scale deletes.
- **Impact:**
  - Slower query performance due to increased logical and physical reads.
  - Higher I/O and CPU usage, especially for range queries and index scans.
  - Reduced cache efficiency, as more pages are needed to store the same amount of data.
- **Detection:**
  - Use database-specific tools or system views to monitor fragmentation levels (e.g., `sys.dm_db_index_physical_stats` in SQL Server).
- **Resolution:**
  - **Rebuild Index:** Physically drops and recreates the index, removing all fragmentation and compacting pages. This is more resource-intensive but fully defragments the index.
  - **Reorganize Index:** Defragments the leaf level of the index by reordering pages and compacting rows. Less intensive than rebuild, suitable for moderate fragmentation.
- **Mitigation & Best Practices:**
  - Schedule regular index maintenance (rebuild or reorganize) based on fragmentation thresholds (e.g., rebuild if fragmentation > 30%, reorganize if 5–30%).
  - Monitor fragmentation using system views or maintenance plans.
  - Choose appropriate fill factors when creating indexes to reduce page splits.
  - Consider partitioning large tables to localize fragmentation and maintenance.

Index Fragmentation: Detection and Resolution

Action	SQL Server Example	Purpose
Detect Fragmentation	`SELECT index_id, avg_fragmentation_in_percent FROM sys.dm_db_index_physical_stats(DB_ID(), OBJECT_ID('Employees'), NULL, NULL, 'LIMITED');`	Check current fragmentation levels
Rebuild Index	`ALTER INDEX idx_emp_id ON Employees REBUILD;`	Removes all fragmentation, recreates index
Reorganize Index	`ALTER INDEX idx_emp_id ON Employees REORGANIZE;`	Defragments leaf level, less intensive

Syntax & Example:

```sql
-- Detect fragmentation (SQL Server)
SELECT index_id, avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats(DB_ID(), OBJECT_ID('Employees'), NULL, NULL,
'LIMITED');

-- Rebuild an index (removes all fragmentation)
ALTER INDEX idx_emp_id ON Employees REBUILD;
```

```
-- Reorganize an index (less intensive, for moderate fragmentation)
ALTER INDEX idx_emp_id ON Employees REORGANIZE;
```

Schedule these commands as part of regular maintenance, especially for large or frequently updated tables.

- **Tip:** Automate index maintenance using database jobs or maintenance plans.
- **Best Practice:** Monitor fragmentation regularly and adjust maintenance frequency based on workload and performance needs.

Summary: Index fragmentation degrades query performance by causing inefficient I/O. Regularly detect, rebuild, or reorganize indexes to maintain optimal performance, especially in large, high-transaction databases.

Mastering SQL Joins

1. What are the different types of SQL joins (`INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`, `CROSS JOIN`, etc.) and when would you use each?

Answer:
SQL joins are used to combine rows from two or more tables based on related columns. The main types of joins are:

- **INNER JOIN:** Returns only rows where there is a match in both tables. Use when you want records that exist in both tables.
- **LEFT JOIN (LEFT OUTER JOIN):** Returns all rows from the left table and matched rows from the right table. If there is no match, columns from the right table are NULL. Use when you want all records from the left table, regardless of matches in the right.
- **RIGHT JOIN (RIGHT OUTER JOIN):** Returns all rows from the right table and matched rows from the left table. If there is no match, columns from the left table are NULL. Use when you want all records from the right table, regardless of matches in the left.
- **FULL JOIN (FULL OUTER JOIN):** Returns all rows from both tables. Where there is a match, rows are combined; where there is no match, columns from the missing side are NULL. Use when you want all records from both tables, matched or unmatched.
- **CROSS JOIN:** Returns the Cartesian product of both tables (every row of the first table joined with every row of the second). Use rarely, typically for generating all possible combinations.

SQL Join Types Overview

Join Type	Description	When to Use
INNER JOIN	Rows with matching values in both tables	When you need only records present in both tables
LEFT JOIN	All rows from left table, matched rows from right; unmatched right rows are NULL	When you want all records from the left table

Join Type	Description	When to Use
RIGHT JOIN	All rows from right table, matched rows from left; unmatched left rows are NULL	When you want all records from the right table
FULL OUTER JOIN	All rows from both tables; unmatched columns are NULL	When you want all records from both tables
CROSS JOIN	Cartesian product (all possible combinations)	Rarely used; for generating combinations

Syntax & Examples:

- **INNER JOIN:**

```sql
SELECT a.*, b.*
FROM TableA a
INNER JOIN TableB b ON a.id = b.a_id;
```

Returns only rows where `a.id = b.a_id`.

- **LEFT JOIN:**

```sql
SELECT a.*, b.*
FROM TableA a
LEFT JOIN TableB b ON a.id = b.a_id;
```

Returns all rows from `TableA`; `TableB` columns are NULL where no match exists.

- **RIGHT JOIN:**

```sql
SELECT a.*, b.*
FROM TableA a
RIGHT JOIN TableB b ON a.id = b.a_id;
```

Returns all rows from `TableB`; `TableA` columns are NULL where no match exists.

- **FULL OUTER JOIN:**

```sql
SELECT a.*, b.*
FROM TableA a
FULL OUTER JOIN TableB b ON a.id = b.a_id;
```

Returns all rows from both tables; unmatched columns are NULL.

- **CROSS JOIN:**

```
SELECT a.*, b.*
FROM TableA a
CROSS JOIN TableB b;
```

Returns every possible combination of rows from `TableA` and `TableB`.

Summary:
- Use **INNER JOIN** for matching rows only.
- Use **LEFT JOIN** to include all left table rows.
- Use **RIGHT JOIN** to include all right table rows.
- Use **FULL OUTER JOIN** to include all rows from both tables.
- Use **CROSS JOIN** for all combinations (rarely needed in practice).

---

2. What is the difference between a `CROSS JOIN` and a `FULL OUTER JOIN`?

Answer:
`CROSS JOIN` and `FULL OUTER JOIN` are both used to combine rows from two tables, but they serve very different purposes and produce fundamentally different results:

Join Type Comparison

Feature	CROSS JOIN	FULL OUTER JOIN
Purpose	Returns the Cartesian product of both tables (all possible combinations of rows).	Returns all rows from both tables, matching rows where possible, and filling with NULLs where there is no match.
Join Condition	No join condition is used.	Join condition is required (typically ON clause).
Result Size	Number of rows = rows in TableA × rows in TableB.	Number of rows = all matched rows + unmatched rows from both tables.
Typical Use Case	Generating all possible combinations (e.g., scheduling, permutations).	Combining all data from both tables, showing matches and non-matches.

Syntax & Examples:

- **CROSS JOIN:**

```
SELECT a.*, b.*
FROM TableA a
CROSS JOIN TableB b;
```

This returns every possible combination of rows from `TableA` and `TableB`. If TableA has 3 rows and TableB has 2 rows, the result will have 6 rows.

- **FULL OUTER JOIN:**

```
SELECT a.*, b.*
FROM TableA a
FULL OUTER JOIN TableB b ON a.id = b.a_id;
```

This returns all rows from both tables. Where there is a match on `a.id = b.a_id`, the rows are combined. Where there is no match, columns from the missing side are filled with `NULL`.

Summary:
- **CROSS JOIN** produces the Cartesian product, combining every row from the first table with every row from the second table, regardless of any relationship.
- **FULL OUTER JOIN** returns all matched rows based on a join condition, plus all unmatched rows from both tables, using `NULL` for missing values.
- Use **CROSS JOIN** for generating all possible combinations; use **FULL OUTER JOIN** when you want to see all data from both tables, including non-matching rows.

---

3. Write a SQL query to retrieve the first and last names of employees along with the names of their managers (given `Employees` and `Managers` tables).

Answer:
To retrieve employee names along with their managers' names, you need to join the `Employees` table with the `Managers` table using a foreign key relationship (e.g., `ManagerID` in `Employees` referencing `Managers.ManagerID`).

- **INNER JOIN:** Returns only employees who have a matching manager.
- **LEFT JOIN:** Returns all employees, including those without a manager (manager fields will be `NULL`).

Example:

```
SELECT
    e.FirstName AS EmployeeFirstName,
    e.LastName AS EmployeeLastName,
    m.FirstName AS ManagerFirstName,
    m.LastName AS ManagerLastName
FROM Employees e
LEFT JOIN Managers m ON e.ManagerID = m.ManagerID;
```

Explanation:
- **LEFT JOIN** is used to include all employees, even those without a manager.
- **INNER JOIN** would exclude employees who do not have a manager.

Employees and Managers Join Result

Join Type	Result	Use Case
INNER JOIN	Only employees with a manager are shown.	Exclude employees without managers.
LEFT JOIN	All employees are shown; manager fields are NULL if no manager.	Include all employees, even those without managers.

---

4. Write a SQL query to find the average salary for each department, given tables `Employees` (with `DepartmentID`) and `Departments` (with `DepartmentName`).

Answer:
To calculate the average salary for each department, join the `Employees` table with the `Departments` table on `DepartmentID`, then use `GROUP BY` to aggregate by department.

- **INNER JOIN:** Returns only departments that have at least one employee.
- **LEFT JOIN:** Returns all departments, showing `NULL` for average salary if there are no employees in a department.

Example:

```
SELECT
    d.DepartmentName,
    AVG(e.Salary) AS AverageSalary
FROM Departments d
LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID
GROUP BY d.DepartmentName;
```

Explanation:
- **LEFT JOIN** ensures all departments are listed, even those without employees.
- **AVG(e.Salary)** computes the average salary per department.
- **GROUP BY d.DepartmentName** groups results by department.

---

5. Write a SQL query to list all products that have never been ordered (products in a `Product` table with no matching rows in the `Orders` table).

Answer:
To find products that have never been ordered, identify products in the `Product` table that do not have any corresponding entries in the `Orders` table. This can be done using a **LEFT JOIN** and checking for `NULL` in the joined table, or by using a `NOT EXISTS` or `NOT IN` subquery.

- **LEFT JOIN:** Returns all products, and for those with no matching order, the order fields will be `NULL`. Filter these using `WHERE o.OrderID IS NULL`.
- **NOT EXISTS:** Checks for products where no matching order exists.
- **NOT IN:** Selects products whose IDs are not present in the `Orders` table.

Example using LEFT JOIN:

```
SELECT p.ProductID, p.ProductName
FROM Product p
LEFT JOIN Orders o ON p.ProductID = o.ProductID
WHERE o.OrderID IS NULL;
```

Example using NOT EXISTS:

```
SELECT p.ProductID, p.ProductName
FROM Product p
WHERE NOT EXISTS (
    SELECT 1 FROM Orders o WHERE o.ProductID = p.ProductID
);
```

Example using NOT IN:

```
SELECT p.ProductID, p.ProductName
FROM Product p
WHERE p.ProductID NOT IN (
    SELECT o.ProductID FROM Orders o
);
```

Explanation:
- **LEFT JOIN** with `WHERE o.OrderID IS NULL` finds products with no orders.
- **NOT EXISTS** and **NOT IN** are alternative approaches, often preferred for readability or performance depending on the database.
- These queries help identify products that may need promotion or removal due to lack of sales.

Approaches to Find Unordered Products

Approach	When to Use
LEFT JOIN + IS NULL	Simple, readable, works well for moderate data sizes.
NOT EXISTS	Efficient for large datasets, especially with proper indexing.
NOT IN	Readable, but can have issues with NULLs in subquery results.

6. Write a SQL query to list all employees who are also managers (for example, employees who appear as managers in the same table).

Answer:

To find employees who are also managers, use a **self-join** on the `Employees` table. This means joining the table to itself, matching employees whose `EmployeeID` appears as a `ManagerID` for other employees.

- **Self-Join:** The `Employees` table is joined to itself, using aliases to distinguish between the "employee" and the "manager" roles.
- **INNER JOIN:** Returns only those employees who are referenced as managers by at least one other employee.
- **DISTINCT:** Used to avoid duplicate rows if an employee manages multiple people.

Example:

```
SELECT DISTINCT
    e.EmployeeID,
    e.FirstName,
    e.LastName
FROM Employees e
INNER JOIN Employees m ON e.EmployeeID = m.ManagerID;
```

Explanation:
- **e** represents the employee who is also a manager.
- **m** represents employees who report to a manager.
- The join condition `e.EmployeeID = m.ManagerID` finds all employees who are listed as a manager for someone else.
- **DISTINCT** ensures each manager appears only once, even if they manage multiple employees.

Employees Who Are Also Managers

EmployeeID	FirstName	LastName
1	Ashish	Zope
2	Sunil	Patil

Summary:
- Use a **self-join** to identify employees who are also managers.
- This pattern is common in organizational hierarchies where the manager and employee data are stored in the same table.
- The approach can be extended to retrieve additional information, such as the number of direct reports each manager has.

---

7. What is a `self-join`, and when might you use it? Provide an example scenario.

Answer:
A **self-join** is a regular join, but the table is joined with itself. This is useful when you want to compare rows within the same table or establish relationships between rows in the same table, such as hierarchical or recursive relationships (e.g., employees and their managers).

- **Self-Join:** The same table is referenced twice in the query, using different aliases to distinguish between the two roles (e.g., employee and manager).

- **Common Use Cases:** Organizational hierarchies, bill of materials, finding pairs of related records, comparing rows within a table.

Example Scenario:
Suppose you have an `Employees` table where each employee may have a manager, and both employees and managers are stored in the same table.

Example Query:
To list each employee along with their manager's name, you can use a self-join:

```sql
SELECT
    e.FirstName AS EmployeeFirstName,
    e.LastName AS EmployeeLastName,
    m.FirstName AS ManagerFirstName,
    m.LastName AS ManagerLastName
FROM Employees e
LEFT JOIN Employees m ON e.ManagerID = m.EmployeeID;
```

Explanation:
- **e** is the alias for the employee.
- **m** is the alias for the manager.
- The join condition `e.ManagerID = m.EmployeeID` links each employee to their manager.
- **LEFT JOIN** ensures employees without a manager are included, with manager fields as NULL.

Self-Join vs Regular Join

Join Type	Purpose	Example
Self-Join	Relate rows within the same table (e.g., employee-manager relationship)	List employees and their managers using `Employees` table
Regular Join	Relate rows between different tables	Join `Employees` and `Departments` to get department names

Summary:
- A **self-join** is a powerful tool for querying hierarchical or related data within the same table.
- It is commonly used for organizational charts, bill of materials, and other recursive relationships.
- Use table aliases to clearly distinguish the roles of each instance of the table in the query.

---

8. How would you join more than two tables in a single SQL query? What factors affect the performance when joining multiple tables?

Answer:
Joining more than two tables in a single SQL query is common in real-world scenarios, such as retrieving employee details along with their department and manager information. This is achieved by chaining multiple **JOIN** clauses together, each connecting two tables at a time.

- **Multiple Joins:** You can join as many tables as needed by specifying additional `JOIN` clauses, using appropriate join conditions for each pair.
- **Types of Joins:** Any combination of `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, etc., can be used depending on the data you want to retrieve.
- **Aliases:** Table aliases help keep queries readable, especially when joining several tables.

Example:

Suppose you have the following tables:

- **Employees** (`EmployeeID`, `FirstName`, `LastName`, `DepartmentID`, `ManagerID`, `Salary`)
- **Departments** (`DepartmentID`, `DepartmentName`)
- **Managers** (`ManagerID`, `FirstName`, `LastName`)

To retrieve each employee's name, department, manager's name, and salary:

```
SELECT
    e.FirstName AS EmployeeFirstName,
    e.LastName AS EmployeeLastName,
    d.DepartmentName,
    m.FirstName AS ManagerFirstName,
    m.LastName AS ManagerLastName,
    e.Salary
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID
LEFT JOIN Managers m ON e.ManagerID = m.ManagerID;
```

Explanation:

- **LEFT JOIN** is used to include all employees, even if they do not have a department or manager.
- Each `JOIN` connects two tables at a time, building up the result set.
- Aliases (`e`, `d`, `m`) make the query concise and readable.

Performance Factors When Joining Multiple Tables:

Performance Factors

Factor	Impact	Best Practice
Indexes	Lack of indexes on join columns can cause slow queries.	Create indexes on columns used in `ON` clauses (e.g., `DepartmentID`, `ManagerID`).
Join Order	Joining large tables first can increase intermediate result size.	Join smaller or filtered tables first when possible.
Join Type	OUTER joins can be slower than INNER joins due to more data being returned.	Use `INNER JOIN` when possible for better performance.
Data Volume	Large tables increase processing time and memory usage.	Filter data early using `WHERE` clauses.

Factor	Impact	Best Practice
Query Complexity	Complex queries with many joins can be harder to optimize.	Break down complex queries or use views for clarity.

Summary:
- You can join multiple tables by chaining `JOIN` clauses.
- Use table aliases for readability.
- Performance depends on indexes, join order, join type, data volume, and query complexity.
- Always test and optimize queries, especially as the number of joins increases.

---

9. Explain how an `OUTER JOIN` works when one side has no matching rows. How does this differ from an `INNER JOIN` in practice?

Answer:
An **OUTER JOIN** returns all rows from one (or both) tables, even if there are no matching rows in the joined table. When there is no match, the columns from the missing side are filled with `NULL` values. In contrast, an **INNER JOIN** only returns rows where there is a match in both tables.

- **LEFT OUTER JOIN (LEFT JOIN):** Returns all rows from the left table, and matched rows from the right table. If there is no match, right table columns are `NULL`.
- **RIGHT OUTER JOIN (RIGHT JOIN):** Returns all rows from the right table, and matched rows from the left table. If there is no match, left table columns are `NULL`.
- **FULL OUTER JOIN:** Returns all rows from both tables, with `NULL` in columns where there is no match.
- **INNER JOIN:** Returns only rows where there is a match in both tables.

Example Scenario:
Suppose you have the following tables:

Employees Table

EmployeeID	FirstName	LastName	DepartmentID
1	Ashish	Zope	10
2	Sunil	Patil	20
3	Ravi	Chaudhari	NULL

Departments Table

DepartmentID	DepartmentName
10	Engineering
20	Data Science
30	HR

LEFT OUTER JOIN Example:

```
SELECT
    e.FirstName,
    e.LastName,
    d.DepartmentName
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

Result:

LEFT OUTER JOIN Result

FirstName	LastName	DepartmentName
Ashish	Zope	Engineering
Sunil	Patil	Data Science
Ravi	Chaudhari	NULL

Notice that **Ravi Chaudhari** has no department, so `DepartmentName` is `NULL`. If you used an **INNER JOIN**, Ravi would not appear in the result.

INNER JOIN Example:

```
SELECT
    e.FirstName,
    e.LastName,
    d.DepartmentName
FROM Employees e
INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

Result:

INNER JOIN Result

FirstName	LastName	DepartmentName
Ashish	Zope	Engineering
Sunil	Patil	Data Science

Explanation:
- **OUTER JOIN** includes all rows from one or both tables, filling in `NULL` where there is no match.
- **INNER JOIN** only includes rows where there is a match in both tables.

- Use **OUTER JOIN** when you want to see all records from one side, even if there are no matches on the other side (e.g., all employees, even those without a department).

INNER JOIN vs OUTER JOIN

Join Type	Rows Returned	NULLs for Missing Data?	Example Use Case
INNER JOIN	Only matching rows	No	Employees with a department
LEFT OUTER JOIN	All left table rows	Yes, for right table columns	All employees, even those without a department
RIGHT OUTER JOIN	All right table rows	Yes, for left table columns	All departments, even those without employees
FULL OUTER JOIN	All rows from both tables	Yes, for missing matches on either side	All employees and all departments, showing all possible matches and non-matches

Summary:
- **OUTER JOIN** is useful for finding unmatched data (e.g., employees without departments, or departments without employees).
- **INNER JOIN** is used when you only care about records that exist in both tables.
- In practice, **OUTER JOIN** helps in reporting, auditing, and identifying missing relationships in your data.

Working with Views

1. What is a database view, and can you update data in the base tables through it?

Answer:
A **database view** is a virtual table defined by a SQL query. It does not store data itself, but presents data dynamically from one or more underlying tables. Views are used to simplify complex queries, provide abstraction, and restrict access to sensitive data.

- **Updatable Views:** If a view is based on a single table and does not use aggregates, GROUP BY, DISTINCT, or joins, you can usually perform `INSERT`, `UPDATE`, and `DELETE` operations through the view. The changes affect the underlying base table.
- **Read-Only Views:** Views that include joins, aggregate functions, GROUP BY, DISTINCT, or other complex logic are typically **read-only**. You cannot update data through these views.

Updatable vs Read-Only Views

View Type	Can Update Base Table?	Example
Simple (single table, no aggregates)	Yes	`SELECT EmployeeID, FirstName FROM Employees`
Complex (joins, GROUP BY, aggregates)	No	`SELECT DepartmentID, COUNT(*) FROM Employees GROUP BY DepartmentID`

Example:

```
-- Simple updatable view
CREATE VIEW vw_EmployeeNames AS
SELECT EmployeeID, FirstName, LastName FROM Employees;

-- Update through the view (affects base table)
UPDATE vw_EmployeeNames SET FirstName = 'John' WHERE EmployeeID = 1;
```

Summary: Views provide abstraction, security, and query simplification. You can update data through a view only if it is simple and directly maps to a single base table without complex logic.

---

2. What is the difference between a standard view and a materialized (or indexed) view?

Answer:
A **standard view** is a virtual table that does not store any data itself; it simply runs the underlying query each time you access it, always reflecting the current state of the base tables. In contrast, a **materialized view** (or **indexed view** in SQL Server) physically stores the result set of its defining query, providing faster access at the cost of storage and refresh overhead.

Standard View vs Materialized View

Feature	Standard View	Materialized/Indexed View
Storage	No data stored; always queries base tables	Physically stores query result set
Performance	Slower for complex or large queries	Much faster for repeated, heavy queries
Data Freshness	Always up-to-date with base tables	May be stale; requires manual or scheduled refresh
Use Case	Abstraction, security, simplified queries	Performance optimization for reporting, analytics
Maintenance	No maintenance needed	Requires refresh and storage management

Example:

```
-- Standard view (always current)
CREATE VIEW vw_EmployeeSummary AS
SELECT DepartmentID, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY DepartmentID;

-- Materialized view (Oracle/PostgreSQL) or indexed view (SQL Server)
CREATE MATERIALIZED VIEW mv_EmployeeSummary AS
SELECT DepartmentID, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY DepartmentID;
-- In SQL Server, use indexed view syntax with SCHEMABINDING and a unique
clustered index
```

Summary: Use **standard views** for abstraction, security, and up-to-date data. Use **materialized views** or **indexed views** to boost performance for complex, resource-intensive queries where real-time data is not critical.

---

3. What happens if a materialized view is being refreshed (complete refresh) and a user queries it at the same time?

Answer:
When a **materialized view** undergoes a **complete refresh**, the entire result set is rebuilt from scratch. If a user queries the view during this process, the behavior depends on the database system and refresh method:

- **Most databases (e.g., Oracle):** The materialized view remains available during the refresh. Users querying the view will see the **previous (old) data** until the refresh completes, ensuring data consistency and uninterrupted access.
- **Some systems or configurations:** If the refresh locks the view or replaces it atomically, queries may be **blocked** until the refresh finishes, or users may receive an error indicating the view is temporarily unavailable.

Materialized View Refresh Behavior

Database	User Query During Refresh
Oracle	Serves old data until refresh completes
SQL Server (Indexed View)	Always up-to-date; no explicit refresh needed
PostgreSQL	May block or error if `WITH NO DATA` is used

- **Tip:** Use `FAST REFRESH` (if supported) to minimize downtime and keep the view more current with incremental changes.
- **Best Practice:** Schedule complete refreshes during off-peak hours to reduce user impact.

Summary: During a complete refresh, most databases serve the old data to users until the refresh is finished, ensuring uninterrupted access. Always check your database documentation for specific behavior and consider refresh strategies that balance performance and data freshness.

---

4. When would you use a view in a database design? What benefits do views provide (e.g. security, abstraction)?

Answer:

A **view** is a virtual table based on the result of a SQL query. Views are a powerful tool in database design, offering several key benefits:

- **Security:** Limit user access to sensitive data by exposing only specific columns or rows. For example, you can create a view that omits salary information, allowing users to query employee names without seeing confidential details.
- **Abstraction:** Hide complex joins, aggregations, or calculations behind a simple interface. This allows end users and applications to interact with straightforward queries, without needing to understand the underlying schema or logic.
- **Simplification:** Provide a consistent and simplified way to access data for reporting, analytics, or application development. Views can encapsulate frequently used queries, reducing code duplication and errors.
- **Consistency:** Centralize business logic, calculations, or filters in one place. This ensures that all users and applications apply the same rules and definitions, improving data integrity and maintainability.

Benefits of Using Views

Benefit	Description	Example
Security	Restrict access to sensitive columns or rows	`vw_PublicEmployees` hides salary data
Abstraction	Hide complex joins/calculations	View for sales summary with aggregations
Simplification	Provide a simple interface for users	View for reporting on active customers
Consistency	Standardize business logic in one place	View for tax calculations

Example:

```sql
-- Hide salary details from most users
CREATE VIEW vw_PublicEmployees AS
SELECT EmployeeID, FirstName, LastName FROM Employees;
```

Summary: Views are essential for enforcing security, simplifying data access, and centralizing business logic. They enable you to present a tailored, consistent, and secure interface to your data, improving both usability and maintainability.

---

5. Can you create an index on a view? If so, what are the implications (e.g. indexed view in SQL Server)?

Answer:

Yes, in some databases—most notably **SQL Server**—you can create an **indexed view** (also known as a

materialized view). An indexed view physically stores the result set of the view and allows you to create indexes on it, just like a regular table.

- **Performance Benefits:** Indexed views can dramatically improve query performance for complex aggregations, joins, or calculations by persisting the computed results and allowing fast index-based lookups.
- **Storage & Maintenance Overhead:** Since the data is physically stored, indexed views consume additional disk space. They also require maintenance—any changes to the underlying tables (INSERT, UPDATE, DELETE) will trigger updates to the indexed view, potentially impacting write performance.
- **Restrictions:** Not all views are eligible for indexing. The view definition must be **deterministic** (no randomness or non-deterministic functions), use `WITH SCHEMABINDING`, and meet other database-specific requirements (e.g., no outer joins, no subqueries in SELECT list, etc.).
- **Use Cases:** Indexed views are ideal for reporting, dashboards, or analytics scenarios where the same complex query is run frequently and up-to-date data is not required in real time.

Indexed View Example (SQL Server)

Step	SQL Syntax

Create a view with SCHEMABINDING	`CREATE VIEW vw_SalesSummary WITH SCHEMABINDING`
	`AS`
	`SELECT StoreID, SUM(SalesAmount) AS TotalSales`
	`FROM dbo.Sales`
	`GROUP BY StoreID;`

Create a unique clustered index on the view	`CREATE UNIQUE CLUSTERED INDEX`
	`idx_SalesSummary_StoreID`
	`ON vw_SalesSummary(StoreID);`

- **Tip:** Indexed views are automatically maintained by SQL Server, so any changes to the base tables are reflected in the view. However, this can slow down data modifications.
- **Best Practice:** Use indexed views for scenarios with heavy read and aggregation workloads, but evaluate the impact on write performance and storage.

Summary: Indexed views (materialized views) can significantly speed up complex queries by storing precomputed results and allowing indexing, but they come with additional storage and maintenance costs. Use them judiciously for performance-critical reporting and analytics.

---

6. How do you modify or drop a view if the underlying table schema changes?

Answer:

When the schema of a base table changes (such as adding, renaming, or dropping columns), any dependent views may become invalid or outdated. To keep your views in sync with the underlying tables, you need to **modify** or **drop** the affected views accordingly.

- **Modify a View:**
  - Use `CREATE OR REPLACE VIEW` (Oracle, PostgreSQL, MySQL 5.7+), or `ALTER VIEW` (SQL Server) to update the view definition to match the new table schema.
  - Update the SELECT statement in the view to reflect any changes in column names, data types, or structure.
- **Drop a View:**
  - Use `DROP VIEW view_name;` to remove the view if it is no longer needed or cannot be updated to match the new schema.
- **Tip:** If a column referenced by a view is dropped or renamed in the base table, the view will become invalid and must be recreated or altered before it can be used again.

Modifying and Dropping Views

Action	SQL Syntax	Notes

Modify	`-- Oracle/PostgreSQL/MySQL`
	`CREATE OR REPLACE VIEW vw_EmployeeNames AS`
	`SELECT EmployeeID, FirstName FROM Employees;`	

-- SQL Server ALTER VIEW vw_EmployeeNames AS SELECT EmployeeID, FirstName FROM Employees; Update the view definition to match the new schema. Drop

```
DROP VIEW vw_EmployeeNames;
```

```
        </td>
        <td style="border:1px solid #D5D8DC; padding:8px;">Removes the view from the database.</td>
    </tr>
</tbody>
```

Summary: Always review and update dependent views after making changes to base table schemas. Modify the view to match the new structure, or drop and recreate it as needed to avoid errors and maintain data integrity.

---

7. What is the difference between a view and a temporary table?

Answer:

A **view** and a **temporary table** are both used to simplify complex queries and manage data, but they serve different purposes and have distinct behaviors in SQL databases.

View vs Temporary Table

Aspect	View	Temporary Table

Aspect	View	Temporary Table
Persistence	Definition persists in the database; always reflects current data in base tables	Exists only for the duration of a session or transaction; dropped automatically
Storage	No data stored (unless materialized); acts as a saved query	Physically stores data in tempdb or memory
Usage	Abstraction, security, reusable logic, simplified access	Storing intermediate results, breaking down complex queries, batch processing
Modification	Cannot be directly modified (unless updatable view)	Can be modified (INSERT, UPDATE, DELETE)
Scope	Global (available to all users with permission)	Session or transaction-specific (not visible to others)

Example:

```sql
-- Create a view (persists, always current)
CREATE VIEW vw_ActiveEmployees AS
SELECT EmployeeID, FirstName, LastName FROM Employees WHERE IsActive = 1;

-- Create a temporary table (session-specific, stores data)
CREATE TABLE #TempEmployees (
    EmployeeID INT,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50)
);

INSERT INTO #TempEmployees
SELECT EmployeeID, FirstName, LastName FROM Employees WHERE IsActive = 1;
```

- **View:** Use when you need a reusable, secure, and always up-to-date abstraction over your data.
- **Temporary Table:** Use when you need to store and manipulate intermediate results within a session or batch process.

Summary: Views provide abstraction, security, and simplified access to live data, while **temporary tables** are ideal for storing and processing intermediate results during complex operations. Choose based on your need for persistence, scope, and data manipulation.

Stored Procedures & Functions

1. What is the difference between a stored procedure and a user-defined function in SQL (aside from return value)?

Answer:

A **stored procedure** and a **user-defined function (UDF)** are both programmable objects in SQL, but they serve different purposes and have distinct behaviors beyond just their return values.

- **Stored Procedure:** A stored procedure is a precompiled set of SQL statements that can perform a wide range of operations, including modifying data (INSERT, UPDATE, DELETE), controlling transactions, and returning result sets or output parameters. Stored procedures are typically invoked using the `EXEC` or `CALL` statement and can contain complex logic, error handling, and flow control (IF, WHILE, etc.).
- **User-Defined Function (UDF):** A UDF is a routine that returns a single value (scalar function) or a table (table-valued function). UDFs are designed to be used within SQL expressions, such as in `SELECT`, `WHERE`, or `JOIN` clauses. They are generally side-effect free and cannot modify database state (with rare exceptions in some databases).

Stored Procedure vs User-Defined Function

Aspect	Stored Procedure	User-Defined Function
Can modify data	Yes (DML operations allowed)	No (read-only; cannot modify data)
Can be used in SELECT/WHERE/JOIN	No	Yes
Transaction control	Yes (can begin, commit, rollback)	No
Return type	None, scalar value, output parameters, or result set	Scalar value or table
Error handling	Can use TRY...CATCH or equivalent	Limited or not supported
Usage context	Called independently via EXEC/CALL	Used inline within SQL statements

Example:

```sql
-- Stored Procedure Example (SQL Server)
CREATE PROCEDURE UpdateEmployeeSalary
    @EmpID INT,
    @NewSalary DECIMAL(10,2)
AS
BEGIN
    UPDATE Employees SET Salary = @NewSalary WHERE EmployeeID = @EmpID;
END;

-- User-Defined Function Example (SQL Server)
CREATE FUNCTION dbo.GetEmployeeFullName(@EmpID INT)
RETURNS NVARCHAR(100)
AS
BEGIN
    DECLARE @FullName NVARCHAR(100);
    SELECT @FullName = FirstName + ' ' + LastName FROM Employees WHERE EmployeeID
= @EmpID;
    RETURN @FullName;
END;
```

Summary: Use stored procedures for complex operations, data modifications, and transaction control. Use user-defined functions for reusable computations or logic that can be embedded in queries. Functions are more restrictive but integrate seamlessly into SQL expressions, while procedures offer greater flexibility and control.

---

2. What are the advantages and disadvantages of using stored procedures?

Answer:
Stored procedures are precompiled collections of SQL statements and optional control-of-flow logic stored under a name and processed as a unit. They offer several benefits but also come with some trade-offs.

- **Advantages:**
    - **Encapsulation of Business Logic:** Centralizes complex business rules and data processing within the database, ensuring consistency and reusability across applications.
    - **Performance Improvement:** Stored procedures are precompiled and cached by the database engine, reducing parsing and execution time for repeated calls.
    - **Security:** You can grant users permission to execute procedures without giving direct access to underlying tables, reducing the risk of unauthorized data access or modification.
    - **Reduced Network Traffic:** Multiple SQL statements can be batched and executed in a single call, minimizing round-trips between application and database servers.
    - **Maintainability:** Changes to business logic can be made in one place (the procedure) without modifying application code.
- **Disadvantages:**
    - **Deployment and Versioning Complexity:** Managing changes to stored procedures across environments (development, staging, production) can be more challenging than deploying application code, especially in teams or CI/CD pipelines.
    - **Potential for Increased Database Load:** Moving significant logic to the database can shift processing load from application servers to the database server, which may become a bottleneck under heavy use.
    - **Split Logic:** Business logic may be divided between application code and stored procedures, making it harder to trace, debug, and maintain overall system behavior.
    - **Portability Issues:** Stored procedure syntax and features can vary between database systems, making migrations or multi-database support more difficult.
    - **Testing and Debugging:** Debugging stored procedures is often less straightforward than debugging application code, and automated testing tools may be less mature.

Summary: Stored procedures are powerful for encapsulating logic, improving performance, and enhancing security, but they require careful management to avoid maintainability and deployment challenges. Use them when centralized logic and performance gains outweigh the added complexity.

---

3. Can you perform INSERT/UPDATE/DELETE operations inside a SQL function? Why or why not?

Answer:

In most relational database systems, **user-defined functions (UDFs) are not allowed to perform data modification operations** such as `INSERT`, `UPDATE`, or `DELETE` on tables. This restriction is enforced to ensure that functions remain **deterministic** (always return the same result for the same input) and **side-effect free**. Functions are designed to be used within queries (e.g., in `SELECT`, `WHERE`, or `JOIN` clauses), and allowing data modifications could lead to unpredictable results, recursion, or performance issues.

- **SQL Server:** Scalar and table-valued functions cannot modify database state. Attempting to use DML statements inside a function will result in an error.
- **PostgreSQL:** Functions written in SQL cannot perform DML; only procedural languages (like PL/pgSQL) allow limited modifications, and even then, such functions are not safe for use in queries.
- **Oracle:** Standard SQL functions cannot modify data, but **autonomous transactions** in PL/SQL allow limited exceptions (not recommended for general use).

Reason:

- **Determinism:** Functions must return consistent results and not depend on or alter external state.
- **Query Safety:** Functions are often called many times during query execution; side effects could cause data corruption or unpredictable behavior.
- **Optimization:** The query optimizer assumes functions are read-only, enabling better execution plans.

Example (SQL Server):

```sql
-- This will fail with an error
CREATE FUNCTION dbo.TryInsertEmployee(@Name NVARCHAR(50))
RETURNS INT
AS
BEGIN
    INSERT INTO Employees (FirstName) VALUES (@Name); -- Not allowed
    RETURN 1;
END;
-- Error: Invalid use of a side-effecting operator 'INSERT' within a function.
```

Summary: Use **stored procedures** for operations that modify data. Use **functions** for computations, lookups, or logic that must be embedded in queries and remain side-effect free.

---

4. What is a table-valued function and when would you use one in a query?

Answer:

A **table-valued function (TVF)** is a user-defined function that returns a table data type, rather than a single scalar value. TVFs can be used in the `FROM` clause of a query, just like a regular table or view. They are powerful for encapsulating reusable logic that produces a set of rows, such as filtering, joining, or transforming data based on input parameters.

- **Inline TVF:** Returns the result of a single SELECT statement. Efficient and commonly used for simple logic.
- **Multi-statement TVF:** Allows multiple statements and more complex logic, but may have performance overhead.

- **Use Cases:** Encapsulate business rules, simplify complex joins, return filtered or computed sets, or modularize query logic for reuse.

Example (SQL Server):

```
-- Inline table-valued function: returns all employees in a given department
CREATE FUNCTION dbo.GetEmployeesByDept(@DeptID INT)
RETURNS TABLE
AS
RETURN (
    SELECT EmployeeID, FirstName, LastName
    FROM Employees
    WHERE DepartmentID = @DeptID
);

-- Usage in a query
SELECT * FROM dbo.GetEmployeesByDept(10);
```

Explanation:
- The function `GetEmployeesByDept` takes a department ID as input and returns a table of employees in that department.
- You can use the function in the `FROM` clause, join it with other tables, or filter its results further.
- TVFs help modularize and reuse query logic, making code easier to maintain and test.

Table-Valued Function Use Cases

Scenario	Example Usage
Reusable filtering logic	Get all active customers by region
Encapsulate business rules	Return employees eligible for a bonus
Simplify complex joins	Return orders with computed totals

Summary: Table-valued functions are ideal for encapsulating and reusing set-based query logic. Use them to simplify queries, enforce consistency, and modularize business rules that return multiple rows.

5. When would you use a stored procedure instead of inline SQL queries in an application?

Answer:
A **stored procedure** is preferred over inline SQL queries in an application when you need to encapsulate business logic, improve security, enhance maintainability, or optimize performance. Stored procedures are precompiled and stored in the database, allowing you to centralize logic, reduce code duplication, and control access more effectively.

- **Centralized Business Logic:** Complex calculations, validations, or data processing rules can be implemented once in a stored procedure and reused by multiple applications or users, ensuring consistency and reducing maintenance effort.

- **Security:** By granting users permission to execute stored procedures (rather than direct table access), you can restrict data access and prevent unauthorized modifications. This also helps prevent SQL injection attacks, as parameters are handled safely by the database engine.
- **Performance:** Stored procedures are compiled and cached by the database, reducing parsing and execution time for repeated operations. They can also batch multiple SQL statements, minimizing network round-trips between the application and the database.
- **Maintainability:** Changes to business logic or data access can be made in the stored procedure without modifying application code, simplifying updates and deployments.
- **Transaction Control:** Stored procedures can manage transactions (BEGIN, COMMIT, ROLLBACK) to ensure atomicity and consistency for multi-step operations.
- **Auditing and Logging:** Procedures can include logic for auditing, logging, or error handling, which is harder to enforce with scattered inline SQL.

Example Scenario:

Suppose you have a payroll application that needs to calculate bonuses, update salaries, and log changes. Instead of writing inline SQL in the application, you can create a stored procedure:

```
CREATE PROCEDURE ProcessPayroll
    @EmpID INT,
    @Bonus DECIMAL(10,2)
AS
BEGIN
    BEGIN TRANSACTION;
    UPDATE Employees SET Salary = Salary + @Bonus WHERE EmployeeID = @EmpID;
    INSERT INTO PayrollAudit (EmployeeID, Bonus, ProcessedAt) VALUES (@EmpID,
@Bonus, GETDATE());
    COMMIT TRANSACTION;
END;
```

The application simply calls `EXEC ProcessPayroll @EmpID, @Bonus`, ensuring all business rules, auditing, and transaction control are handled in one place.

Summary: Use stored procedures for reusable, secure, and efficient data operations, especially when logic is complex, shared, or requires transaction management. Inline SQL is suitable for simple, one-off queries, but stored procedures offer greater control and maintainability for enterprise applications.

---

6. How do you pass parameters to and receive results from stored procedures?

Answer:

Stored procedures accept parameters to make them flexible and reusable. Parameters can be of three types: **input** (provide values to the procedure), **output** (return values from the procedure), and **input/output** (both supply and return values). Results from stored procedures can be received in several ways:

- **Input Parameters:** Used to pass values into the procedure for processing (e.g., filtering by ID).
- **Output Parameters:** Used to return values back to the caller (e.g., computed results, status codes).
- **Return Values:** A single integer value can be returned using the `RETURN` statement (often used for status or error codes).

- **Result Sets:** Procedures can return one or more result sets using SELECT statements, which can be read by the calling application.

Example (SQL Server):

```
-- Define a stored procedure with input and output parameters
CREATE PROCEDURE GetEmployeeDetails
    @EmpID INT,                        -- Input parameter
    @FirstName NVARCHAR(50) OUTPUT,    -- Output parameter
    @LastName NVARCHAR(50) OUTPUT      -- Output parameter
AS
BEGIN
    SELECT
        @FirstName = FirstName,
        @LastName = LastName
    FROM Employees
    WHERE EmployeeID = @EmpID;
END;

-- Call the procedure and receive output parameters
DECLARE @FName NVARCHAR(50), @LName NVARCHAR(50);
EXEC GetEmployeeDetails 1, @FName OUTPUT, @LName OUTPUT;
SELECT @FName AS FirstName, @LName AS LastName;
```

Returning a Result Set:
A procedure can also return a result set directly using SELECT:

```
CREATE PROCEDURE GetEmployeesByDepartment
    @DeptID INT
AS
BEGIN
    SELECT EmployeeID, FirstName, LastName
    FROM Employees
    WHERE DepartmentID = @DeptID;
END;

-- Usage:
EXEC GetEmployeesByDepartment 10;
```

Returning a Value with RETURN:
You can use RETURN to send back a single integer (commonly used for status codes):

```
CREATE PROCEDURE CheckEmployeeExists
    @EmpID INT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Employees WHERE EmployeeID = @EmpID)
        RETURN 1; -- Exists
    ELSE
        RETURN 0; -- Does not exist
END;

-- Usage:
DECLARE @Result INT;
EXEC @Result = CheckEmployeeExists 1;
SELECT @Result AS ExistsFlag;
```

- **Input parameters** are specified in the procedure definition and passed when calling the procedure.
- **Output parameters** must be declared as `OUTPUT` in both the procedure and the call.
- **Result sets** are retrieved using `SELECT` inside the procedure and read by the caller (application or script).
- **Return values** are captured using the `RETURN` keyword and can be assigned to a variable when executing the procedure.

Summary: Stored procedures support flexible parameter passing and can return results via output parameters, result sets, or return values. Choose the method that best fits your use case and calling environment.

---

7. How would you debug or test a slow or failing stored procedure in production?

Answer:
Debugging or testing a slow or failing stored procedure in production requires a systematic approach to identify the root cause and minimize impact on users. Here are the recommended steps:

- **Capture the Actual Execution Plan:** Use tools like SQL Server Management Studio (SSMS), `SHOWPLAN`, or `EXPLAIN` to obtain the execution plan. Look for table scans, missing indexes, or expensive operations that may cause slowness.
- **Monitor Performance Metrics:** Check CPU, memory, and I/O usage on the database server. Use built-in monitoring tools (e.g., SQL Profiler, Extended Events, Performance Monitor) to track resource consumption during procedure execution.
- **Enable Logging or Add Debug Output:** Temporarily add `PRINT` statements, logging to a debug table, or use `RAISE NOTICE` (PostgreSQL) to trace variable values and control flow. This helps pinpoint where the procedure is failing or slowing down.
- **Check for Blocking and Deadlocks:** Use system views (e.g., `sys.dm_tran_locks`, `sp_who2`) or monitoring tools to detect if the procedure is waiting on locks or involved in a deadlock.
- **Test with Representative Data in a Non-Production Environment:** Reproduce the issue using a copy of production data in a development or staging environment. This allows safe experimentation and tuning without affecting live users.
- **Review Recent Changes:** Investigate any recent schema modifications, index changes, or data growth that could impact performance. Roll back or adjust changes if necessary.
- **Optimize Queries and Indexes:** Refactor inefficient queries, add or update indexes, and avoid unnecessary cursors or loops. Use query hints judiciously if needed.
- **Check for Parameter Sniffing Issues:** If performance varies with different inputs, consider using local variables or `OPTION (RECOMPILE)` to avoid suboptimal plans.
- **Analyze Error Logs and Exception Handling:** Review database and application logs for error messages or exceptions related to the procedure.

Summary: Debugging stored procedures involves analyzing execution plans, monitoring system resources, tracing logic, and testing in a safe environment. Always document findings and changes, and coordinate with your DBA or operations team when working in production.

---

8. How do you grant a user permission to execute a specific stored procedure?

Answer:
To allow a user to execute a specific stored procedure without granting broader access to the underlying tables or database, you use the `GRANT EXECUTE` statement. This provides the user with permission to run the procedure, but not to modify or view the procedure's code or access other database objects directly.

- **Principle of Least Privilege:** Granting execute permission on a stored procedure is a best practice for security, as it restricts the user's actions to only what the procedure allows.
- **Granular Control:** You can grant execute permission on individual procedures, groups of procedures, or all procedures within a schema, depending on your requirements.
- **Revoking Permission:** If needed, you can later revoke the permission using the `REVOKE EXECUTE` statement.

Syntax & Example (SQL Server):

```sql
-- Grant execute permission on a specific stored procedure to a user
GRANT EXECUTE ON OBJECT::GetEmployeeByID TO [username];
```

- `OBJECT::GetEmployeeByID` specifies the stored procedure.
- `[username]` is the database user or role you want to grant permission to.

Example (Oracle):

```sql
GRANT EXECUTE ON GetEmployeeByID TO username;
```

Explanation:
- The user can now execute the `GetEmployeeByID` procedure using `EXEC GetEmployeeByID` or equivalent.
- The user cannot alter, drop, or view the procedure's definition unless granted additional permissions.
- This approach is commonly used to expose business logic through stored procedures while protecting sensitive data and enforcing security policies.

Summary: Use `GRANT EXECUTE` to securely allow users to run specific stored procedures, following the principle of least privilege and maintaining tight control over database access.

---

1. What is a trigger in SQL, and when would you use one? Give an example use case.

Answer:
A **trigger** is a special type of stored procedure that is automatically executed (or "triggered") by the database engine in response to specific events on a table or view, such as `INSERT`, `UPDATE`, or `DELETE` operations. Triggers are used to enforce business rules, maintain data integrity, automate system tasks, or audit changes without requiring explicit calls from application code.

- **Use Cases:**
    - **Auditing:** Automatically log changes to sensitive data (e.g., salary updates, deletions).
    - **Enforcing Business Rules:** Prevent invalid data changes, such as disallowing deletion of a parent row if child rows exist.
    - **Maintaining Derived Data:** Keep summary or aggregate tables up to date when base data changes.
    - **Cascading Actions:** Implement custom cascading updates or deletes not natively supported by foreign keys.
    - **Synchronizing Tables:** Copy or replicate data between tables automatically.

Example:

```sql
-- Audit trigger: Log every employee salary change
CREATE TRIGGER trg_AuditSalaryChange
ON Employees
AFTER UPDATE
AS
BEGIN
    INSERT INTO SalaryAudit (EmployeeID, OldSalary, NewSalary, ChangedAt)
    SELECT i.EmployeeID, d.Salary, i.Salary, GETDATE()
    FROM inserted i
    JOIN deleted d ON i.EmployeeID = d.EmployeeID
    WHERE i.Salary <> d.Salary;
END;
```

This trigger fires automatically after any `UPDATE` to the `Employees` table. If an employee's salary changes, it inserts a record into the `SalaryAudit` table, capturing the employee ID, old salary, new salary, and the timestamp of the change. This is useful for auditing and compliance purposes, ensuring all salary modifications are tracked without requiring changes to application code.

Summary: Triggers are powerful tools for automating responses to data changes, enforcing complex business logic, and maintaining data integrity. Use them judiciously, as they can introduce hidden logic and impact performance if overused.

---

2. What is the difference between an AFTER trigger and an INSTEAD OF trigger (e.g. in SQL Server)?

Answer:
In SQL Server, **AFTER** and **INSTEAD OF** triggers are both used to respond to data modification events (`INSERT`, `UPDATE`, `DELETE`), but they differ in when and how they execute:

- **AFTER Trigger:** Executes **after** the triggering DML operation has completed and the data has been modified in the table. Used for auditing, enforcing business rules, or cascading changes. If the trigger fails, the entire transaction is rolled back.
- **INSTEAD OF Trigger:** Executes **instead of** the triggering DML operation. The original operation does not occur unless explicitly performed inside the trigger. Commonly used on views to enable custom logic for `INSERT`, `UPDATE`, or `DELETE` operations that would otherwise be invalid.

AFTER vs INSTEAD OF Trigger Comparison

Trigger Type	When It Fires	Typical Use Case
AFTER	After the DML operation completes	Auditing, enforcing rules, cascading actions
INSTEAD OF	In place of the DML operation	Custom logic for views, complex updates

Example:

```sql
-- AFTER trigger on a table (audit log)
CREATE TRIGGER trg_AfterInsert
ON Employees
AFTER INSERT
AS
BEGIN
    INSERT INTO AuditLog (EmployeeID, Action, ActionDate)
    SELECT EmployeeID, 'INSERT', GETDATE() FROM inserted;
END;

-- INSTEAD OF trigger on a view (custom update logic)
CREATE VIEW vw_EmployeeDept AS
SELECT e.EmployeeID, e.FirstName, d.DepartmentName
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID;

CREATE TRIGGER trg_UpdateEmpDept
ON vw_EmployeeDept
INSTEAD OF UPDATE
AS
BEGIN
    UPDATE Employees
    SET FirstName = i.FirstName
    FROM inserted i
    WHERE Employees.EmployeeID = i.EmployeeID;
END;
```

Summary: Use **AFTER triggers** to respond to completed data changes on tables. Use **INSTEAD OF triggers** to override or customize DML operations, especially on views or when special handling is required.

---

3. What are the “inserted” and “deleted” magic tables in SQL Server triggers?

Answer:
In SQL Server, **inserted** and **deleted** are special (magic) tables available inside triggers:

- **inserted:** Holds the new rows for `INSERT` and `UPDATE` operations.
- **deleted:** Holds the old rows for `DELETE` and `UPDATE` operations.

Example: In an `AFTER UPDATE` trigger, `inserted` has new values, `deleted` has old values.

---

4. How can triggers be used to enforce business rules or data integrity (e.g. auditing changes, simulating foreign keys)?

Answer:

Triggers can enforce business rules by validating data, preventing invalid changes, logging modifications, or simulating constraints not natively supported (e.g., cascading deletes, custom referential integrity).

- **Auditing:** Log changes to sensitive data (e.g., salary updates).
- **Enforcing Rules:** Prevent deletion of parent rows if child rows exist (simulate foreign key).
- **Data Integrity:** Automatically update related tables or maintain derived columns.

Example:

```sql
-- Prevent deleting a department if employees exist
CREATE TRIGGER trg_PreventDeptDelete
ON Departments
INSTEAD OF DELETE
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Employees e JOIN deleted d ON e.DepartmentID = d.DepartmentID)
        RAISERROR('Cannot delete department with employees.', 16, 1);
    ELSE
        DELETE FROM Departments WHERE DepartmentID IN (SELECT DepartmentID FROM deleted);
END;
```

This trigger blocks deletion of a department if employees are assigned to it.

---

5. What are the potential drawbacks of using triggers (such as performance impact or hidden logic)?

Answer:

Triggers can introduce hidden logic and performance overhead:

- **Performance:** Triggers add extra processing to DML operations, potentially slowing down inserts, updates, or deletes.
- **Hidden Logic:** Business rules in triggers may not be obvious to developers, making debugging and maintenance harder.
- **Complexity:** Nested or recursive triggers can cause unexpected behavior.
- **Portability:** Trigger syntax and behavior can vary between database systems.

Summary: Use triggers judiciously; document their behavior and monitor performance.

---

6. How do INSTEAD OF triggers on a view work?

Answer:

INSTEAD OF triggers on a view intercept `INSERT`, `UPDATE`, or `DELETE` operations and allow you to define custom logic for how those operations are handled. This is useful for updatable views that join multiple tables or require special handling.

Example:

```
-- Allow updates to a view that joins Employees and Departments
CREATE VIEW vw_EmployeeDept AS
SELECT e.EmployeeID, e.FirstName, d.DepartmentName
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID;

CREATE TRIGGER trg_UpdateEmpDept
ON vw_EmployeeDept
INSTEAD OF UPDATE
AS
BEGIN
    UPDATE Employees
    SET FirstName = i.FirstName
    FROM inserted i
    WHERE Employees.EmployeeID = i.EmployeeID;
END;
```

This trigger allows updates to the `FirstName` column via the view.

---

7. Can triggers call stored procedures, and are there any limitations to doing that?

Answer:

Yes, triggers can call stored procedures. However, there are limitations:

- **Side Effects:** Procedures called from triggers should not commit/rollback transactions independently.
- **Performance:** Long-running procedures can slow down DML operations.
- **Recursion:** Be careful to avoid recursive trigger/procedure calls.
- **Permissions:** The trigger must have permission to execute the procedure.

Summary: Triggers can call stored procedures, but keep logic efficient and avoid transactional conflicts.

Transactions & Concurrency Control

---

1. What are the ACID properties of a database transaction (atomicity, consistency, isolation, durability)?

Answer:

The **ACID** properties are a set of four key guarantees that ensure reliable processing of database transactions. They are fundamental to maintaining data integrity, especially in multi-user and concurrent environments. Each letter in ACID stands for a specific property:

ACID Properties Explained

Property	Description	Example

Property	Description	Example
Atomicity	A transaction is an indivisible unit: either all its operations succeed, or none do. If any part fails, the entire transaction is rolled back.	Transferring money between accounts: both debit and credit must succeed, or neither happens.
Consistency	A transaction brings the database from one valid state to another, maintaining all defined rules (constraints, triggers, cascades).	A transaction cannot violate foreign key or unique constraints.
Isolation	Concurrent transactions do not affect each other's intermediate states. Each transaction executes as if it were the only one running.	Two users updating the same data do not see each other's uncommitted changes.
Durability	Once a transaction is committed, its changes are permanent—even in the event of a system crash or power failure.	After a successful transfer, the updated balances remain even if the server restarts.

- **Atomicity:** Guarantees "all or nothing" execution for transactions.
- **Consistency:** Ensures data validity before and after the transaction.
- **Isolation:** Prevents concurrent transactions from interfering with each other.
- **Durability:** Makes committed changes permanent and crash-proof.

Summary: The ACID properties are essential for reliable, predictable, and safe database operations. They protect data integrity and enable robust multi-user applications.

---

2. What are the different SQL isolation levels (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE), and what phenomena do they prevent (dirty reads, non-repeatable reads, phantom reads)?

Answer:
SQL isolation levels define how transaction integrity is maintained when multiple transactions occur concurrently. Each level balances data consistency and system performance by controlling which changes made by one transaction are visible to others. The main phenomena controlled by isolation levels are:

- **Dirty Read:** Reading uncommitted changes from another transaction.
- **Non-Repeatable Read:** A row read twice in the same transaction returns different values (due to another transaction's update/commit).
- **Phantom Read:** New rows added or removed by another transaction appear/disappear in repeated queries within the same transaction.

SQL Isolation Levels and Prevented Phenomena

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Description

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Description
READ UNCOMMITTED	Possible	Possible	Possible	Lowest isolation; transactions can see uncommitted changes from others. Fastest, but least safe.
READ COMMITTED	Prevented	Possible	Possible	Default in most databases; only committed data is visible. Non-repeatable and phantom reads can occur.
REPEATABLE READ	Prevented	Prevented	Possible	Ensures rows read cannot change during the transaction. New rows (phantoms) may still appear.
SERIALIZABLE	Prevented	Prevented	Prevented	Highest isolation; transactions are fully isolated as if run sequentially. Safest, but slowest and most restrictive.

Summary: Higher isolation levels prevent more anomalies but can reduce concurrency and increase locking. Choose the level that best fits your application's consistency and performance needs.

- **READ UNCOMMITTED:** Use for reporting or analytics where dirty data is acceptable and performance is critical.
- **READ COMMITTED:** Good default for most OLTP workloads; balances safety and concurrency.
- **REPEATABLE READ:** Use when you need consistent reads of the same data within a transaction (e.g., financial calculations).
- **SERIALIZABLE:** Use for critical operations requiring strict consistency, such as end-of-day batch processing.

Isolation Level Use Cases

Isolation Level	Typical Use Case
READ UNCOMMITTED	Fast reporting, analytics, non-critical reads
READ COMMITTED	General OLTP, web applications
REPEATABLE READ	Financial calculations, batch processing
SERIALIZABLE	Critical transactions, strict data integrity

Tip: Always test your application's behavior under different isolation levels to ensure the right balance between consistency and performance.

3. What is a deadlock in database terms, and how can you prevent or resolve deadlocks?

Answer:

A **deadlock** is a situation in a database where two or more transactions are each waiting for the other to release a lock, causing all of them to remain blocked indefinitely. Deadlocks are a classic concurrency problem in multi-user database systems and can halt progress unless resolved by the database engine.

Deadlock Scenario Example

Transaction 1	Transaction 2
Locks **Table A** Tries to lock **Table B** (blocked)	Locks **Table B** Tries to lock **Table A** (blocked)

- **Symptoms:** Queries hang or time out; database engine logs deadlock errors.
- **Detection:** Most modern databases automatically detect deadlocks and resolve them by rolling back one of the involved transactions (the "deadlock victim").

How to Prevent Deadlocks:

- **Access Resources in a Consistent Order:** Always acquire locks on tables and rows in the same order in all transactions.
- **Keep Transactions Short:** Minimize the time locks are held by keeping transactions as brief as possible.
- **Use Lower Isolation Levels:** Where appropriate, use `READ COMMITTED` or `READ UNCOMMITTED` to reduce locking contention.
- **Avoid User Interaction in Transactions:** Do not prompt for user input or perform long-running operations inside transactions.
- **Index Appropriately:** Proper indexing can reduce the number of locked rows and the duration of locks.

How to Resolve Deadlocks:

- **Automatic Resolution:** The database engine detects the deadlock and rolls back one transaction, allowing others to proceed.
- **Manual Intervention:** If deadlocks are frequent, analyze deadlock graphs or logs to identify problematic queries and refactor them.
- **Retry Logic:** Implement retry mechanisms in application code to handle deadlock errors gracefully.

Deadlock Prevention & Resolution Strategies

Strategy	Description
Consistent Lock Order	Always lock tables/rows in the same order in all transactions.
Short Transactions	Reduce lock duration by keeping transactions brief.
Retry Logic	Automatically retry transactions that are rolled back due to deadlocks.
Analyze Deadlock Graphs	Use database tools to identify and fix deadlock-prone queries.

Summary: Deadlocks are a natural risk in concurrent systems. Prevent them by designing transactions carefully and resolving them by letting the database engine choose a victim and retrying as needed.

4. How do you control transactions in SQL (BEGIN, COMMIT, ROLLBACK)? Give an example of using a transaction in a stored procedure or batch.

Answer:

A **transaction** is a sequence of one or more SQL statements executed as a single unit of work. Transactions ensure that either all operations succeed (commit) or none take effect (rollback), preserving data integrity. You control transactions using the following commands:

- **BEGIN TRANSACTION** (or `START TRANSACTION`): Starts a new transaction block.
- **COMMIT:** Saves all changes made during the transaction to the database.
- **ROLLBACK:** Undoes all changes made during the transaction, reverting the database to its previous state.

Transactions are essential for operations that must be atomic, such as transferring money between accounts, batch updates, or multi-step data modifications.

Transaction Control Commands

Command	Purpose
`BEGIN TRANSACTION`	Start a new transaction
`COMMIT`	Make all changes permanent
`ROLLBACK`	Undo all changes since the transaction began

Example: Atomic Money Transfer

Suppose you want to transfer funds between two accounts. Both updates must succeed together, or neither should happen.

```sql
BEGIN TRANSACTION;
    UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
    UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
    IF @@ERROR <> 0
        ROLLBACK;
    ELSE
        COMMIT;
```

This ensures that if either update fails (e.g., insufficient funds, constraint violation), the entire transaction is rolled back and no money is lost or created.

Example: Transaction in a Stored Procedure (SQL Server)

```sql
CREATE PROCEDURE TransferFunds
    @FromAccount INT,
    @ToAccount INT,
    @Amount DECIMAL(10,2)
```

```
AS
BEGIN
    BEGIN TRANSACTION;
    BEGIN TRY
        UPDATE Accounts SET Balance = Balance - @Amount WHERE AccountID =
@FromAccount;
        UPDATE Accounts SET Balance = Balance + @Amount WHERE AccountID =
@ToAccount;
        COMMIT;
    END TRY
    BEGIN CATCH
        ROLLBACK;
        THROW;
    END CATCH
END;
```

This stored procedure safely transfers funds and ensures all-or-nothing execution, with error handling for robustness.

- **Tip:** Always use transactions for multi-step operations that must be atomic and consistent.
- **Note:** Syntax may vary slightly between SQL dialects (e.g., MySQL uses `START TRANSACTION`).

Summary: Use `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` to control atomicity and consistency in SQL. Transactions are critical for reliable, error-proof data operations.

---

5. If you run a long SELECT query on a table while another transaction is updating rows in that table, will your session see the old data or new data by default? (Consider default isolation level behavior.)

Answer:
By default, most databases use the **READ COMMITTED** isolation level. In this mode, a `SELECT` query only sees data that has been committed at the moment each row is read—not necessarily at the start of the query. This means:

- If another transaction updates and commits a row **before** your SELECT reads it, you will see the **new (updated)** data.
- If your SELECT reads a row **before** the other transaction commits its update, you will see the **old (pre-update)** data.
- You may see a mix of old and new data within the same query if the update commits during your SELECT's execution (this is called a **non-repeatable read**).

Isolation Level Impact on SELECT Visibility

Isolation Level	SELECT Sees	Phenomena
READ COMMITTED (default)	Latest committed data at time of row read	Non-repeatable reads possible
REPEATABLE READ	Data as of start of query/transaction	Prevents non-repeatable reads

Isolation Level	SELECT Sees	Phenomena
SERIALIZABLE	Data as of start of transaction; strictest	Prevents phantoms and non-repeatable reads

Example Scenario:

1. **Transaction A** starts a long SELECT on `Employees`.
2. **Transaction B** updates a row and commits while Transaction A's SELECT is still running.
3. If Transaction A's SELECT has not yet read that row, it will see the updated value. If it already read the row, it will see the old value.

Summary: With the default **READ COMMITTED** isolation, your SELECT may see a mix of old and new data depending on when each row is read and when updates are committed. For consistent results, use **REPEATABLE READ** or **SERIALIZABLE** isolation levels.

---

6. What is the difference between pessimistic and optimistic locking, and when would you use each?

Answer:
Locking strategies are essential for managing concurrent access to data in multi-user database environments. The two primary approaches are **pessimistic locking** and **optimistic locking**. Each has its own use cases, advantages, and trade-offs.

Pessimistic vs Optimistic Locking

Aspect	Pessimistic Locking	Optimistic Locking
How it works	Locks data as soon as it is read or before it is updated, preventing others from modifying it until the transaction completes.	Allows multiple transactions to read data without locking; checks for conflicts only when updating, typically using a version or timestamp column.
Concurrency	Lower concurrency; can cause blocking and deadlocks.	Higher concurrency; conflicts detected at commit time.
Best for	High-conflict scenarios (e.g., banking, inventory management).	Low-conflict scenarios (e.g., web apps, reporting, most OLTP workloads).
Implementation	Database locks (row/table locks, SELECT ... FOR UPDATE).	Version columns, timestamps, or checksums.
Drawbacks	Can reduce throughput and cause contention.	May require retry logic if conflicts are detected.

Pessimistic Locking Example:

```
-- Lock a row for update (blocks others)
SELECT * FROM accounts WHERE account_id = 1 FOR UPDATE;
```

```
-- Perform updates, then COMMIT or ROLLBACK
```

This prevents other transactions from reading or updating the locked row until the transaction completes.

Optimistic Locking Example:

Suppose you have a `version` column in your table:

```
-- Step 1: Read the row and note the version
SELECT account_id, balance, version FROM accounts WHERE account_id = 1;

-- Step 2: Attempt to update, checking the version
UPDATE accounts
SET balance = balance + 100, version = version + 1
WHERE account_id = 1 AND version = 5;
```

If the version has changed since you read it, the update affects 0 rows, indicating a conflict (someone else updated it first).

- **Pessimistic Locking:** Use when data contention is high and consistency is critical (e.g., financial transactions).
- **Optimistic Locking:** Use when conflicts are rare and you want maximum concurrency (e.g., most web applications).

Summary: Pessimistic locking prevents conflicts by locking data up front but can reduce performance. **Optimistic locking** allows more concurrency and only checks for conflicts at update time, making it ideal for most modern applications where write conflicts are infrequent.

---

7. What is a savepoint in a transaction, and how do you use it?

Answer:

A **savepoint** is a marker set within a transaction that allows you to partially roll back to a specific point, rather than rolling back the entire transaction. Savepoints provide finer control over error handling and complex transactional logic, especially in long or multi-step transactions. They are useful when you want to undo only part of the work done in a transaction without losing all previous successful operations.

- **Granular Rollback:** Roll back to a savepoint to undo only the changes made after that point.
- **Multiple Savepoints:** You can set multiple savepoints within a single transaction and roll back to any of them as needed.
- **Supported In:** Most major databases (SQL Server, PostgreSQL, Oracle, MySQL with InnoDB) support savepoints.

Savepoint Commands Overview

Command	Purpose
`SAVEPOINT savepoint_name;`	Creates a named savepoint within the current transaction.

Command	Purpose
`ROLLBACK TO SAVEPOINT savepoint_name;`	Undoes all changes after the savepoint, but keeps the transaction open.
`RELEASE SAVEPOINT savepoint_name;`	Removes the savepoint (optional; some DBs do this automatically).

Example: Using Savepoints in a Transaction

Suppose you are transferring funds between two accounts and want to ensure that if the second update fails, you can roll back only that part, not the entire transaction.

```
BEGIN TRANSACTION;
    UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
    SAVEPOINT after_debit;
    UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
    -- Suppose the above update fails, roll back to the savepoint:
    ROLLBACK TO SAVEPOINT after_debit;
    -- Optionally, try a different operation or log the error
COMMIT;
```

In this example, if the credit to AccountID 2 fails, only that operation is undone; the debit from AccountID 1 remains unless you roll back further or the whole transaction.

- **Tip:** Use savepoints for complex business logic, error recovery, or when partial success is acceptable within a transaction.
- **Note:** Savepoints are only valid within the current transaction scope. Rolling back the entire transaction discards all savepoints.

Summary: Savepoints provide flexible, granular control over transaction rollback, making them invaluable for robust error handling and complex transactional workflows.

---

8. How do two-phase commit protocols work in distributed transactions?

Answer:

The **two-phase commit (2PC)** protocol is a standard method for ensuring **atomicity** and **consistency** across multiple databases or systems involved in a single distributed transaction. It coordinates all participating nodes (databases, services, etc.) so that either **all** commit the transaction or **all** roll it back, even in the presence of failures. This prevents data inconsistencies in distributed environments.

Two-Phase Commit Protocol Steps

Phase	Coordinator Action	Participant Action
1. Prepare (Voting)	Sends **PREPARE** request to all participants, asking if they can commit.	Each participant tries to prepare (e.g., writes to a log, locks resources) and replies **YES** (ready to commit) or **NO** (cannot commit).

Phase	Coordinator Action	Participant Action
2. Commit (Decision)	If all reply **YES**, sends **COMMIT** to all; if any reply **NO**, sends **ROLLBACK** to all.	On **COMMIT**, each participant commits and releases resources. On **ROLLBACK**, each undoes changes.

Example Scenario:

- **Bank Transfer:** Moving money from an account in Database A to another in Database B. Both must succeed or fail together.

1. **Prepare Phase:** Coordinator asks both databases if they can commit the transfer. Each checks constraints and locks rows.
2. **Commit Phase:** If both reply YES, coordinator tells both to commit. If either says NO (e.g., insufficient funds), coordinator tells both to roll back.

Two-Phase Commit: Key Points

Advantage	Drawback
Ensures atomicity and consistency across distributed systems	Can block resources if a participant or coordinator crashes (blocking protocol)
Prevents partial commits in distributed transactions	Adds network and logging overhead; not ideal for high-latency or unreliable networks

Summary: The **two-phase commit protocol** is the foundation for reliable distributed transactions, ensuring all-or-nothing outcomes across multiple systems. While robust, it can introduce blocking and performance trade-offs, so it's best used when strong consistency is required.

---

9. How can you identify and terminate a blocking or long-running transaction in a SQL database?

Answer:
Identifying and terminating blocking or long-running transactions is crucial for maintaining database performance and availability. Blocking transactions can prevent other queries from executing, leading to slowdowns or application timeouts. Most relational databases provide tools and system views to help you detect and resolve these issues.

- **Identify Blocking or Long-Running Transactions:**
  - **SQL Server:** Use system views such as `sys.dm_exec_requests`, `sys.dm_tran_locks`, and `sys.dm_os_waiting_tasks` to find blocking sessions. The `sp_who2` stored procedure is also commonly used to list active sessions and identify blockers.
  - **Oracle:** Query `v$session`, `v$locked_object`, and `v$session_longops` to find sessions holding locks or running long operations.
  - **PostgreSQL:** Use `pg_stat_activity` and `pg_locks` to monitor active queries and lock holders.
  - **MySQL:** Use `SHOW PROCESSLIST` or query `information_schema.PROCESSLIST` to see running queries and their states.
- **Terminate the Problematic Session:**

  - **SQL Server:** Use `KILL session_id;` to terminate a specific session.
  - **Oracle:** Use `ALTER SYSTEM KILL SESSION 'sid,serial#';` to end a session.
  - **PostgreSQL:** Use `SELECT pg_terminate_backend(pid);` to terminate a backend process.
  - **MySQL:** Use `KILL thread_id;` to stop a running query or connection.

Identifying and Terminating Blocking Sessions by Database

Database	Identify Blocking Session	Terminate Session
SQL Server	`sp_who2`, `sys.dm_exec_requests`	`KILL session_id;`
Oracle	`v$session`, `v$locked_object`	`ALTER SYSTEM KILL SESSION 'sid,serial#';`
PostgreSQL	`pg_stat_activity`, `pg_locks`	`SELECT pg_terminate_backend(pid);`
MySQL	`SHOW PROCESSLIST`	`KILL thread_id;`

Example (SQL Server):

```
-- Find blocking sessions
EXEC sp_who2;

-- Terminate a session (replace 53 with the actual session_id)
KILL 53;
```

Example (PostgreSQL):

```
-- Find long-running queries
SELECT pid, usename, state, query_start, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY query_start;

-- Terminate a session (replace 12345 with the actual pid)
SELECT pg_terminate_backend(12345);
```

- **Tip:** Always investigate the cause of blocking before terminating sessions, as killing a session can roll back uncommitted work and affect users or applications.
- **Monitor regularly:** Set up alerts for long-running or blocking transactions to proactively manage performance.

Summary: Use system views and monitoring tools to identify blocking or long-running transactions, and terminate them using the appropriate command for your database system. Regular monitoring and quick intervention help maintain database health and application responsiveness.

---

10. What is deadlock detection, and how does the database engine choose a deadlock victim?

Answer:

Deadlock detection is a mechanism used by database engines to identify situations where two or more transactions are waiting indefinitely for each other to release locks, resulting in a cycle that cannot be resolved without intervention. Instead of letting transactions hang forever, the database actively monitors for deadlocks and takes action to resolve them automatically.

- **Detection Process:** The database engine periodically scans the lock manager's wait-for graph to look for cycles (deadlocks). If a cycle is found, it means a deadlock exists.
- **Choosing a Victim:** Once a deadlock is detected, the engine must break the cycle by rolling back one of the involved transactions—the "deadlock victim." The choice is typically based on minimizing the cost and impact:
    - **Least Cost:** The transaction that has done the least amount of work (fewest changes, shortest duration, lowest resource usage) is usually chosen as the victim.
    - **User Priority:** Some databases allow you to set a deadlock priority, so lower-priority transactions are more likely to be chosen as victims.
    - **Rollback:** The victim transaction is rolled back, releasing its locks and allowing the other transactions to proceed. The application receives an error and can retry if needed.

Deadlock Detection & Resolution Workflow

Step	Description
1. Monitor Locks	Database engine tracks which transactions are waiting for which locks.
2. Detect Cycle	Engine scans for cycles in the wait-for graph (deadlock detection algorithm).
3. Select Victim	Chooses the transaction with the lowest cost or priority to roll back.
4. Rollback & Notify	Victim transaction is rolled back, locks are released, and an error is sent to the application.
5. Continue	Other transactions can now proceed, breaking the deadlock.

Example (SQL Server):

SQL Server automatically detects deadlocks and chooses a victim based on the `DEADLOCK_PRIORITY` and the amount of work done. The victim receives error 1205 ("Transaction (Process ID) was deadlocked on resources...").

Summary: Deadlock detection ensures that your database does not hang indefinitely due to resource contention. By automatically identifying and resolving deadlocks, the engine maintains system availability and allows applications to handle errors gracefully—often by retrying the failed transaction.

Performance Tuning & Query Optimization

1. What is a query execution plan and how do you use it to improve performance?

Answer:

A **query execution plan** is a detailed roadmap generated by the database engine that outlines how a SQL query will be executed. It includes the sequence of operations (such as scans, joins, sorts), the access methods (e.g., index seek, table scan), and the estimated cost of each step. By analyzing the execution plan, you can identify inefficiencies like full table scans, missing indexes, or suboptimal join strategies, and then optimize your queries or schema accordingly.

- **How to View:** Use `EXPLAIN` (MySQL, PostgreSQL), `EXPLAIN PLAN` (Oracle), or graphical tools (SQL Server Management Studio, pgAdmin, etc.).
- **Optimization:** Look for expensive operations (e.g., table scans), add or adjust indexes, rewrite queries, or refactor schema based on plan insights.

Common Execution Plan Operators

Operator	Description
Table Scan	Reads all rows in a table; slow for large tables
Index Seek	Efficiently finds rows using an index
Nested Loop Join	Efficient for small result sets or indexed joins
Hash Join	Good for joining large, unsorted data sets

Syntax & Example:

```
EXPLAIN SELECT * FROM Employees WHERE DepartmentID = 10;
```

This command shows the execution plan for the query, including whether an index is used.

Summary: Always review execution plans for slow queries to identify and resolve performance bottlenecks.

---

2. How would you optimize a slow SQL query in production?

Answer:
Optimizing a slow query involves a systematic approach:

- **Analyze the Execution Plan:** Identify bottlenecks such as full table scans, missing indexes, or expensive joins.
- **Add Indexes:** Create indexes on columns used in WHERE, JOIN, and ORDER BY clauses.
- **Rewrite Queries:** Simplify complex queries, avoid unnecessary subqueries, and use set-based operations instead of row-by-row processing.
- **Select Only Needed Columns:** Avoid `SELECT *`; retrieve only the columns you need.
- **Partition Large Tables:** Use table partitioning to improve query performance on very large datasets.
- **Update Statistics & Rebuild Indexes:** Ensure the query optimizer has up-to-date statistics and that indexes are not fragmented.

Optimization Techniques Comparison

Technique	When to Use

Technique	When to Use
Indexing	Frequent filtering or joining on columns
Query Rewrite	Complex or inefficient queries
Partitioning	Very large tables

Summary: Use a combination of indexing, query rewriting, and regular maintenance for optimal performance.

---

3. What is the difference between UNION and UNION ALL in SQL, and when would you use each?

Answer:
UNION combines the results of two queries and removes duplicate rows. **UNION ALL** combines results and keeps all duplicates. **UNION ALL** is faster because it does not perform the extra step of removing duplicates.

UNION vs UNION ALL

Operator	Duplicates Removed?	Performance
UNION	Yes	Slower (deduplication required)
UNION ALL	No	Faster

Syntax & Example:

```
SELECT Name FROM Employees
UNION
SELECT Name FROM Managers;

SELECT Name FROM Employees
UNION ALL
SELECT Name FROM Managers;
```

Summary: Use **UNION ALL** for better performance if you do not need to remove duplicates.

---

4. How can you find duplicate rows in a table using SQL?

Answer:
To identify duplicate rows in a table, you typically use the `GROUP BY` clause on the columns that define a duplicate, combined with the `HAVING` clause to filter groups that occur more than once. This approach helps you find which values are repeated and how many times they appear.

- **Step 1:** Decide which columns define a duplicate (e.g., all columns, or a subset such as `email` or `first_name, last_name`).
- **Step 2:** Use `GROUP BY` on those columns and count the occurrences.
- **Step 3:** Use `HAVING COUNT(*) > 1` to filter only the duplicates.

Syntax & Example:

```
-- Example: Find duplicates based on column1 and column2
SELECT column1, column2, COUNT(*) AS duplicate_count
FROM table_name
GROUP BY column1, column2
HAVING COUNT(*) > 1;
```

This query lists each combination of `column1` and `column2` that appears more than once, along with the number of times it occurs.

Example Scenario:
Suppose you have a `users` table with columns `email` and `username`, and you want to find duplicate emails:

```
SELECT email, COUNT(*) AS duplicate_count
FROM users
GROUP BY email
HAVING COUNT(*) > 1;
```

This returns all email addresses that appear more than once in the `users` table.

Sample Output

email	duplicate_count
ashish@example.com	3
sunil@example.com	2

To retrieve the full duplicate rows (not just the duplicate values), you can join the results back to the original table:

```
SELECT t.*
FROM table_name t
JOIN (
    SELECT column1, column2
    FROM table_name
    GROUP BY column1, column2
    HAVING COUNT(*) > 1
) dup
ON t.column1 = dup.column1 AND t.column2 = dup.column2;
```

This returns all rows from `table_name` that are considered duplicates based on `column1` and `column2`.

Summary: Use `GROUP BY` and `HAVING COUNT(*) > 1` to find duplicates. Adjust the columns in `GROUP BY` to match your definition of a duplicate row.

5. Write a SQL query to find the 10th highest salary in an Employee table.

Answer:

To find the 10th highest salary, you need to rank the unique salary values in descending order and select the one at position 10. There are several approaches, depending on your SQL dialect and requirements (e.g., handling duplicate salaries).

- **Approach 1: Subquery with DISTINCT, ORDER BY, LIMIT/OFFSET (MySQL, PostgreSQL)**

This method selects the top 10 unique salaries in descending order, then picks the minimum (which is the 10th highest).

```
SELECT MIN(Salary) AS TenthHighestSalary
FROM (
    SELECT DISTINCT Salary
    FROM Employee
    ORDER BY Salary DESC
    LIMIT 10
) AS Top10;
```

This returns the 10th highest unique salary. If there are fewer than 10 unique salaries, it returns NULL.

- **Approach 2: Using OFFSET (MySQL, PostgreSQL)**

You can also use `OFFSET` to directly get the 10th highest salary:

```
SELECT DISTINCT Salary
FROM Employee
ORDER BY Salary DESC
LIMIT 1 OFFSET 9;
```

This returns the 10th highest unique salary (OFFSET is zero-based).

- **Approach 3: Using Window Functions (SQL Server, PostgreSQL, Oracle)**

Window functions like `DENSE_RANK()` can be used to assign a rank to each unique salary:

```
SELECT Salary AS TenthHighestSalary
FROM (
    SELECT Salary, DENSE_RANK() OVER (ORDER BY Salary DESC) AS rnk
    FROM Employee
) t
WHERE rnk = 10;
```

This works in databases that support window functions and handles ties (duplicate salaries) correctly.

- **Approach 4: Correlated Subquery (Standard SQL)**

This approach counts how many distinct salaries are greater than the current one:

```sql
SELECT Salary AS TenthHighestSalary
FROM Employee e1
WHERE (
    SELECT COUNT(DISTINCT Salary)
    FROM Employee e2
    WHERE e2.Salary > e1.Salary
) = 9
LIMIT 1;
```

This returns the 10th highest unique salary. If there are fewer than 10 unique salaries, it returns no row.

Summary: Use `DISTINCT` to ignore duplicate salaries. Choose the approach that fits your SQL dialect and performance needs. Window functions are preferred for clarity and efficiency in modern databases.

---

6. How would you retrieve the last 5 records (by date or ID) from a table?

Answer:
To retrieve the last 5 records from a table, you need to determine which column defines the "order" of your data—typically a date column (e.g., `created_at`, `order_date`) or a unique, incrementing ID (e.g., `id`, `record_id`). You then sort the table in descending order by that column and use the `LIMIT` clause (or its equivalent) to fetch only the top 5 rows. This approach works in most SQL databases, including MySQL, PostgreSQL, and SQLite.

- **Step 1:** Identify the column that determines the record order (e.g., `date_column` or `id`).
- **Step 2:** Use `ORDER BY` in descending order (`DESC`) to bring the latest or highest values to the top.
- **Step 3:** Use `LIMIT 5` to restrict the result to the last 5 records.

Syntax & Examples:

- **By Date:**

```sql
SELECT *
FROM table_name
ORDER BY date_column DESC
LIMIT 5;
```

This returns the 5 most recent records based on `date_column`.

- **By ID (assuming higher IDs are newer):**

```sql
SELECT *
FROM table_name
ORDER BY id DESC
LIMIT 5;
```

This returns the 5 records with the highest (latest) IDs.

- **To return the last 5 records in ascending order (oldest to newest among the last 5):**

```
SELECT *
FROM (
    SELECT *
    FROM table_name
    ORDER BY date_column DESC
    LIMIT 5
) AS last_five
ORDER BY date_column ASC;
```

This subquery first selects the last 5 records, then reorders them in ascending order.

Retrieval Methods by SQL Dialect

Database	Syntax
MySQL / PostgreSQL / SQLite	`ORDER BY column DESC LIMIT 5`
SQL Server	`SELECT TOP 5 * FROM table_name ORDER BY column DESC`
Oracle	`SELECT * FROM (SELECT * FROM table_name ORDER BY column DESC) WHERE ROWNUM <= 5`

Summary: Use `ORDER BY ... DESC LIMIT 5` to get the last 5 records by date or ID. Adjust the column and syntax for your database system.

---

7. Write a SQL query to exclude specific values (e.g., select all rows except those where ID is X or Y).

Answer:

To exclude specific values from your query results, use the `NOT IN` operator or multiple `!=` (or `<>`) conditions in the `WHERE` clause. This is useful when you want to filter out rows with certain values in a column, such as excluding students with specific IDs.

- **NOT IN:** Excludes all rows where the column matches any value in the list.
- **!= or <>:** Excludes rows matching a single value; combine with `AND` for multiple exclusions.
- **NOT EQUALS (for a single value):** Use `WHERE ID != X` or `WHERE ID <> X`.

Syntax & Examples:

- **Exclude multiple values using NOT IN:**

```
SELECT *
FROM Student
WHERE ID NOT IN (101, 102);
```

This query returns all students except those with ID 101 or 102.

- **Exclude a single value using != (or <>):**

```sql
SELECT *
FROM Student
WHERE ID != 101;
-- or
SELECT *
FROM Student
WHERE ID <> 101;
```

This returns all students except the one with ID 101.

- **Exclude multiple values using AND:**

```sql
SELECT *
FROM Student
WHERE ID != 101 AND ID != 102;
```

This is equivalent to using `NOT IN (101, 102)`.

- **Exclude values in a string column:**

```sql
SELECT *
FROM Student
WHERE Name NOT IN ('Ashish', 'Sunil');
```

This excludes students named Ashish or Sunil.

Exclusion Methods Comparison

Method	Use Case	Example
NOT IN	Exclude multiple values	`ID NOT IN (101, 102)`
!= or <>	Exclude a single value	`ID != 101`
AND with !=	Exclude several values (few)	`ID != 101 AND ID != 102`

Notes:

- If the column contains `NULL` values, `NOT IN` may return no rows if any value in the list is `NULL`. Use `IS NOT NULL` if needed.
- For large exclusion lists, `NOT IN` is more concise and readable.

Summary: Use `NOT IN` to exclude multiple values, or `!=`/`<>` for single values. Adjust the column and values as needed for your query.

---

8. How do you retrieve the Nth record (e.g., the 3rd record) from a table?

Answer:
Retrieving the Nth record from a table depends on the SQL dialect and whether you want the Nth row in a specific order. The most common approach is to use `ORDER BY` with `LIMIT` and `OFFSET` (supported in MySQL, PostgreSQL, SQLite), or `ROW_NUMBER()` window function (in SQL Server, PostgreSQL, Oracle).

- **Step 1:** Decide the column(s) that define the order (e.g., `created_at`, `id`).
- **Step 2:** Use `ORDER BY` to specify the order.
- **Step 3:** Use `OFFSET` to skip the first N-1 rows and `LIMIT 1` to get the Nth row.

Syntax & Examples:

- **MySQL / PostgreSQL / SQLite:**

```
SELECT *
FROM table_name
ORDER BY ordering_column
LIMIT 1 OFFSET N-1;
```

For example, to get the 3rd record (N=3):

```
SELECT *
FROM table_name
ORDER BY ordering_column
LIMIT 1 OFFSET 2;
```

This returns the 3rd row in the specified order (OFFSET is zero-based).

- **SQL Server:**

```
SELECT *
FROM (
    SELECT *, ROW_NUMBER() OVER (ORDER BY ordering_column) AS rn
    FROM table_name
) t
WHERE rn = N;
```

Replace `N` with the desired row number (e.g., 3 for the 3rd record).

- **Oracle:**

```
SELECT *
FROM (
    SELECT t.*, ROW_NUMBER() OVER (ORDER BY ordering_column) AS rn
    FROM table_name t
)
WHERE rn = N;
```

Nth Record Retrieval Methods by SQL Dialect

Database	Syntax
MySQL / PostgreSQL / SQLite	`ORDER BY ... LIMIT 1 OFFSET N-1`
SQL Server / Oracle	`ROW_NUMBER() OVER (ORDER BY ...) = N`

Notes:

- If the table has no explicit ordering, the result may be unpredictable. Always use `ORDER BY` for deterministic results.
- For large tables, ensure the ordering column is indexed for better performance.

Summary: Use `ORDER BY` with `LIMIT 1 OFFSET N-1` (or `ROW_NUMBER()` in SQL Server/Oracle) to retrieve the Nth record in a specified order.

---

9. How do you obtain the CREATE TABLE DDL for an existing table in SQL?

Answer:
To obtain the **CREATE TABLE** DDL (Data Definition Language) statement for an existing table, you use database-specific commands or tools that generate the SQL statement required to recreate the table structure, including columns, data types, constraints, indexes, and other properties. This is useful for documentation, migration, backup, or recreating tables in another environment.

DDL Retrieval Methods by Database

Database	How to Get CREATE TABLE DDL	Example
MySQL	Use `SHOW CREATE TABLE` command	`SHOW CREATE TABLE employees;`
PostgreSQL	Use `pg_dump` with `--schema-only` or query `pg_catalog`/`information_schema`. Many GUI tools (e.g., pgAdmin) also provide "Generate SQL" or "DDL" options.	`pg_dump -U username -d dbname -t employees --schema-only`
SQL Server	Use SQL Server Management Studio (SSMS): right-click table → Script Table as → CREATE To → New Query Editor Window. Or use `sp_help` for table details.	*SSMS GUI: Script Table as → CREATE To*

Database	How to Get CREATE TABLE DDL	Example
Oracle	Use `DBMS_METADATA.GET_DDL` function or tools like SQL Developer.	`SELECT DBMS_METADATA.GET_DDL('TABLE', 'EMPLOYEES') FROM DUAL;`
SQLite	Query `sqlite_master` table for the SQL statement.	`SELECT sql FROM sqlite_master WHERE type='table' AND name='employees';`

- **GUI Tools:** Most database management tools (e.g., MySQL Workbench, pgAdmin, SSMS, Oracle SQL Developer) provide right-click options to generate the CREATE TABLE script for any table.
- **Command-Line Utilities:** Utilities like `mysqldump`, `pg_dump`, or `expdp` (Oracle) can export DDL for tables or entire schemas.
- **Information Schema:** For advanced scripting, you can query metadata tables (e.g., `information_schema.columns`) to reconstruct DDL, but this is rarely needed for standard use cases.

Example Output (MySQL):

```
SHOW CREATE TABLE employees;
```

SHOW CREATE TABLE employees Result

Table	Create Table

employees	`` CREATE TABLE `employees` ( ``
	`` `id` int NOT NULL AUTO_INCREMENT, ``
	`` `first_name` varchar(50) NOT NULL, ``
	`` `last_name` varchar(50) NOT NULL, ``
	`` PRIMARY KEY (`id`) ``
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;`

Summary: Use the appropriate command or tool for your database system to generate the CREATE TABLE DDL for an existing table. This is essential for migrations, backups, and documentation.

---

10. Explain the difference between the RANK() and DENSE_RANK() window functions.

Answer:

RANK() and **DENSE_RANK()** are window functions used to assign a ranking number to each row within a result set, based on the ordering of one or more columns. They are commonly used for tasks like leaderboard generation, top-N queries, and reporting. The key difference between them is how they handle ties (rows with equal values in the ordering column).

- **RANK():** Assigns the same rank to tied rows, but leaves gaps in the ranking sequence after the tie. The next rank after a tie is incremented by the number of tied rows.
- **DENSE_RANK():** Also assigns the same rank to tied rows, but does not leave gaps. The next rank after a tie is incremented by one, regardless of the number of tied rows.

Syntax:

```sql
SELECT
    column1,
    RANK() OVER (ORDER BY column2 DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY column2 DESC) AS dense_rank
FROM table_name;
```

Example:

Suppose you have a `Scores` table:

Player	Score
A	100
B	90
C	90
D	80

Applying `RANK()` and `DENSE_RANK()`:

Player	Score	RANK()	DENSE_RANK()
A	100	1	1
B	90	2	2
C	90	2	2
D	80	4	3

- With **RANK()**, both B and C have rank 2 (tie), and the next rank is 4 (skips 3).
- With **DENSE_RANK()**, both B and C have rank 2, and the next rank is 3 (no gap).

RANK() vs DENSE_RANK() Comparison

Function	Ranking Behavior	Example Output
RANK()	Skips ranks after ties (gaps in ranking)	1, 2, 2, 4
DENSE_RANK()	No gaps after ties (consecutive ranking)	1, 2, 2, 3

When to use:

- Use **RANK()** when you want to reflect the number of tied rows in the ranking (e.g., for competition scoring where the next rank skips ahead).
- Use **DENSE_RANK()** when you want consecutive ranking numbers without gaps, even if there are ties (e.g., for reporting or grouping).

Summary: RANK() leaves gaps after ties; **DENSE_RANK()** does not. Choose based on your ranking requirements.

---

11. When would you use ROW_NUMBER(), RANK(), or DENSE_RANK() in a query? Give a use case.

Answer:

`ROW_NUMBER()`, `RANK()`, and `DENSE_RANK()` are window functions used to assign a sequential number or rank to rows within a result set, based on a specified ordering. They are commonly used for tasks such as pagination, ranking, deduplication, and leaderboard generation. The choice among them depends on how you want to handle ties (rows with equal values in the ordering column).

- **ROW_NUMBER():** Assigns a unique sequential number to each row within the partition, regardless of ties. No two rows get the same number, even if their values are identical.
  - **Use Case:** Pagination (fetching rows N to M), removing duplicates (keeping only the first occurrence), or selecting the "top N" per group.
- **RANK():** Assigns the same rank to tied rows, but leaves gaps in the ranking sequence after ties. The next rank after a tie is incremented by the number of tied rows.
  - **Use Case:** Competition ranking where ties should skip ranks (e.g., Olympic medals: 1, 2, 2, 4).
- **DENSE_RANK():** Assigns the same rank to tied rows, but does not leave gaps. The next rank after a tie is incremented by one.
  - **Use Case:** Leaderboards or reporting where consecutive ranking is desired without gaps (e.g., 1, 2, 2, 3).

Syntax & Example:

Suppose you have a `Sales` table:

Employee	SalesAmount
Alice	500
Bob	400
Carol	400
Dave	300

To assign rankings based on `SalesAmount` (highest first):

```
SELECT
    Employee,
    SalesAmount,
    ROW_NUMBER() OVER (ORDER BY SalesAmount DESC) AS row_num,
    RANK() OVER (ORDER BY SalesAmount DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY SalesAmount DESC) AS dense_rank
FROM Sales;
```

Result:

Employee	SalesAmount	row_num	rank	dense_rank
Alice	500	1	1	1
Bob	400	2	2	2
Carol	400	3	2	2
Dave	300	4	4	3

- **ROW_NUMBER():** Each row gets a unique number (no ties).
- **RANK():** Bob and Carol tie for 2nd place, so both get rank 2; the next rank is 4 (gap).
- **DENSE_RANK():** Bob and Carol tie for 2nd, both get 2; the next rank is 3 (no gap).

Window Function Use Cases

Function	Typical Use Case
ROW_NUMBER()	Pagination, deduplication, selecting Nth row
RANK()	Competition ranking with gaps after ties
DENSE_RANK()	Leaderboard or reporting with consecutive ranks

Summary: Use `ROW_NUMBER()` for unique row numbering, `RANK()` for rankings with gaps after ties, and `DENSE_RANK()` for consecutive rankings without gaps. Choose based on your business logic and how you want to handle ties.

---

12. Write a SQL query to compute the median number of searches made by users, given a summary table of search counts.

Answer:

The **median** is the middle value in a sorted list of numbers. If the count of values is odd, the median is the value at the center. If the count is even, the median is the average of the two central values. Calculating the median in SQL requires ordering the data and identifying the middle value(s). This is typically done using window functions such as `ROW_NUMBER()`, `RANK()`, or `PERCENTILE_CONT()` (if supported by your database).

- **Step 1:** Order the search counts for all users.
- **Step 2:** Assign a row number to each record and count the total number of records.
- **Step 3:** Select the middle row(s) based on the total count.
- **Step 4:** For even counts, average the two middle values; for odd counts, select the single middle value.

Syntax & Example (Standard SQL):

Suppose you have a table `user_search_summary` with columns `user_id` and `search_count`.

```
SELECT AVG(search_count) AS median_searches
FROM (
    SELECT
        search_count,
        ROW_NUMBER() OVER (ORDER BY search_count) AS rn,
        COUNT(*) OVER () AS total
    FROM user_search_summary
) t
WHERE
    rn = (total + 1) / 2
    OR (total % 2 = 0 AND rn = (total / 2) + 1);
```

- This query works for both odd and even numbers of rows.
- For odd counts, it selects the middle row. For even counts, it averages the two central rows.
- `ROW_NUMBER()` assigns a unique sequential number to each row in order of `search_count`.
- `COUNT(*) OVER ()` gives the total number of rows.

Alternative (Using PERCENTILE_CONT, if supported):

Some databases (e.g., PostgreSQL, Oracle, SQL Server 2012+) support the `PERCENTILE_CONT` window function, which directly computes the median:

```
SELECT
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY search_count) AS median_searches
FROM user_search_summary;
```

- This is the most concise and efficient way to compute the median if your database supports it.

Sample Data:

user_id	search_count
1	5
2	2
3	7
4	3
5	4

- Ordered `search_count`: 2, 3, 4, 5, 7
- Median is 4 (the third value in the sorted list)

Summary: To compute the median in SQL, use window functions to rank and select the middle value(s), or use `PERCENTILE_CONT(0.5)` if available. The median provides a robust measure of central tendency, especially when data contains outliers.

13. Write a SQL query to calculate the sum of odd-numbered and even-numbered measurements separately for each day.

Answer:

To calculate the sum of odd-numbered and even-numbered measurements for each day, you can use a **CASE** expression inside the **SUM()** aggregate function. This allows you to conditionally sum values based on whether the `measurement_number` is odd or even. The `MOD()` (or `%` in some databases) function is used to determine if a number is odd (`MOD(measurement_number, 2) = 1`) or even (`MOD(measurement_number, 2) = 0`).

- **Step 1:** Use `CASE` to check if `measurement_number` is odd or even.
- **Step 2:** Use `SUM()` to aggregate the values for odd and even numbers separately.
- **Step 3:** `GROUP BY` the `day` column to get results per day.

Syntax & Example (Standard SQL):

```
SELECT
    day,
    SUM(CASE WHEN MOD(measurement_number, 2) = 1 THEN value ELSE 0 END) AS
odd_sum,
    SUM(CASE WHEN MOD(measurement_number, 2) = 0 THEN value ELSE 0 END) AS
even_sum
FROM measurements
GROUP BY day;
```

- `MOD(measurement_number, 2) = 1` checks for odd numbers.
- `MOD(measurement_number, 2) = 0` checks for even numbers.
- Replace `MOD` with `%` if your database uses that syntax (e.g., `measurement_number % 2`).

Example Data:

day	measurement_number	value
2024-06-01	1	10
2024-06-01	2	20
2024-06-01	3	30
2024-06-02	1	15
2024-06-02	2	25

Result:

day	odd_sum	even_sum
2024-06-01	40	20
2024-06-02	15	25

- For 2024-06-01: odd measurements (1 and 3) sum to 10 + 30 = 40; even measurement (2) is 20.
- For 2024-06-02: odd measurement (1) is 15; even measurement (2) is 25.

Summary: Use `CASE` with `SUM()` and `MOD()` (or `%`) to separate and aggregate odd and even measurement values for each day.

---

14. Write a SQL query to get the average review rating for each product for each month.

Answer:
To calculate the average review rating for each product for each month, you need to:

- Group reviews by **product** and by **month** (derived from the review date).
- Use an aggregate function (`AVG()`) to compute the average rating per group.
- Use a date truncation or formatting function to extract the month from the review date. The exact function depends on your SQL dialect.

Date Truncation Functions by SQL Dialect

Database	Function	Example
PostgreSQL	`DATE_TRUNC('month', review_date)`	`DATE_TRUNC('month', review_date)`
MySQL	`DATE_FORMAT(review_date, '%Y-%m')`	`DATE_FORMAT(review_date, '%Y-%m')`
SQL Server	`FORMAT(review_date, 'yyyy-MM')` or `YEAR(review_date), MONTH(review_date)`	`FORMAT(review_date, 'yyyy-MM')`
Oracle	`TO_CHAR(review_date, 'YYYY-MM')`	`TO_CHAR(review_date, 'YYYY-MM')`

Syntax & Examples:

- **PostgreSQL:**

```
SELECT
    product_id,
    DATE_TRUNC('month', review_date) AS review_month,
    AVG(rating) AS avg_rating
FROM reviews
GROUP BY product_id, DATE_TRUNC('month', review_date)
ORDER BY product_id, review_month;
```

- **MySQL:**

```
SELECT
    product_id,
    DATE_FORMAT(review_date, '%Y-%m') AS review_month,
    AVG(rating) AS avg_rating
FROM reviews
GROUP BY product_id, review_month
ORDER BY product_id, review_month;
```

- **SQL Server:**

```
SELECT
    product_id,
    FORMAT(review_date, 'yyyy-MM') AS review_month,
    AVG(rating) AS avg_rating
FROM reviews
GROUP BY product_id, FORMAT(review_date, 'yyyy-MM')
ORDER BY product_id, review_month;
```

- **Oracle:**

```
SELECT
    product_id,
    TO_CHAR(review_date, 'YYYY-MM') AS review_month,
    AVG(rating) AS avg_rating
FROM reviews
GROUP BY product_id, TO_CHAR(review_date, 'YYYY-MM')
ORDER BY product_id, review_month;
```

Sample Data:

product_id	review_date	rating
101	2024-05-10	4
101	2024-05-15	5
101	2024-06-01	3
102	2024-05-20	2
102	2024-06-05	4

Sample Output:

product_id	review_month	avg_rating
101	2024-05-01	4.5
101	2024-06-01	3.0
102	2024-05-01	2.0

product_id	review_month	avg_rating
102	2024-06-01	4.0

- **Note:** The `review_month` column format may vary by SQL dialect (e.g., `2024-05-01` or `2024-05`).

Summary: Use `GROUP BY` with a date truncation or formatting function to aggregate average ratings per product per month. Adjust the date function for your database system.