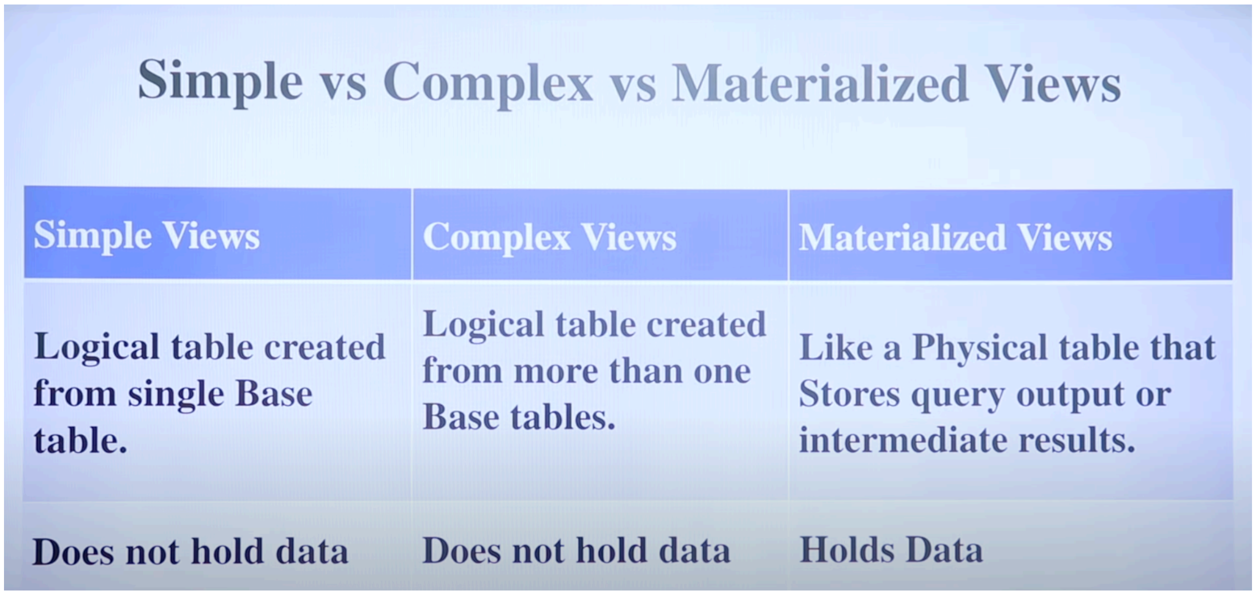

Simple vs Complex vs Materialized Views

Simple Views	Complex Views	Materialized Views
Logical table created from single Base table.	Logical table created from more than one Base tables.	Like a Physical table that Stores query output or intermediate results.
Does not hold data	Does not hold data	Holds Data

A materialized view is a database object that stores the pre-computed result of a query. Reading query execution plans using commands like `EXPLAIN ANALYZE` helps identify bottlenecks such as large table scans or suboptimal joins. Key performance metrics are latency (query response time) and throughput (queries processed per unit of time). Indexes and materialized views (MVs) can significantly improve query performance by reducing disk I/O and computation.

Materialized Views 📋

A materialized view (MV) stores the pre-computed result of a query. Unlike a regular view, which runs its defining query every time it's accessed, an MV stores the data physically, like a table. This makes querying the MV very fast.

Benefits:

- Speed: Significantly faster for complex, frequently run queries, especially those with aggregations or joins.
- Reduced Load: Decreases the load on base tables as complex computations are done less frequently.

Trade-offs:

- Storage: Consumes disk space to store the pre-computed data.
- Stale Data: The data in the MV is only as current as its last refresh. MVs need a

strategy to keep their data up-to-date.
- Refresh Overhead: Refreshing the MV can consume resources.

Query Example: Creating and Using a Materialized View

Let's assume we have a `sales` table:

```sql
CREATE TABLE sales (
  sale_id INT PRIMARY KEY,
  product_id INT,
  sale_date DATE,
  amount DECIMAL(10, 2)
);


-- Insert some sample data
INSERT INTO sales VALUES (1, 101, '2025-01-15', 50.00);
INSERT INTO sales VALUES (2, 102, '2025-01-16', 120.00);
INSERT INTO sales VALUES (3, 101, '2025-01-16', 55.00);
INSERT INTO sales VALUES (4, 103, '2025-01-17', 75.00);
INSERT INTO sales VALUES (5, 102, '2025-01-18', 130.00);
```

Query:

```sql
SELECT product_id, SUM(amount) AS total_sales

FROM sales
GROUP BY product_id;
```

Creating the Materialized View (Syntax varies by database; example for PostgreSQL):

```
CREATE MATERIALIZED VIEW mv_product_sales_summary AS

SELECT product_id, SUM(amount) AS total_sales

FROM sales

GROUP BY product_id;
```

Querying the Materialized View:

```
SELECT * FROM mv_product_sales_summary WHERE product_id = 101;
```

Refreshing the Materialized View:

```
REFRESH MATERIALIZED VIEW mv_product_sales_summary;
```

When are MVs most beneficial?

MVs provide the most significant benefit for queries that aggregate large volumes of data, involve complex joins, or are executed frequently for reporting and analytics, where slightly stale data is acceptable.

Query Execution Plans: Reading and Interpreting 📈

A query execution plan is the database's roadmap for executing a SQL query. It shows the steps the database will take, such as how tables are accessed (e.g., full scan vs. index scan) and joined.

How to obtain it:

Use the EXPLAIN command. For more detailed information including actual execution times, use EXPLAIN ANALYZE (available in databases like PostgreSQL).

Key parts to interpret:

- Scan Types:
    - Sequential Scan (Seq Scan) / Table Scan: Reads the entire table. Often bad for large tables if only a few rows are needed.
    - Index Scan: Uses an index to find rows. Generally much faster for selective queries.
    - Index Only Scan: All required data is retrieved directly from the index, without accessing the table itself. Very fast.
- Join Types:
    - Nested Loop Join: Iterates through rows of one table and, for each row, probes the other table. Good for small tables or when one table has a highly selective index on the join key.
    - Hash Join: Builds a hash table on one (smaller) table and then probes it with rows from the other (larger) table. Efficient for large tables with no suitable indexes for a nested loop.
    - Merge Join: Requires both inputs to be sorted on the join key. Efficient if data is already sorted or can be sorted cheaply.
- Costs/Rows: Estimates of how expensive an operation is (in arbitrary units) and the number of rows expected. `EXPLAIN ANALYZE` provides actual rows and timings.
- Filters: Conditions applied to reduce the dataset.

Practical Example: Improvements with Indexes and `EXPLAIN ANALYZE`

Let's use a `products` table:

```sql
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10, 2)
);


-- Insert a substantial amount of data (e.g., 100,000 rows)
-- For brevity, we'll imagine it's populated.
INSERT INTO products (product_id, product_name, category, price)
SELECT i, 'Product ' || i, CASE WHEN i % 3 = 0 THEN 'Electronics' WHEN i % 3 = 1 THEN 'Books'
ELSE 'Home Goods' END, (RANDOM() * 500 + 10)::DECIMAL(10,2)
FROM generate_series(1, 100000) AS i;
```

Scenario 1: Query without an index

```sql
EXPLAIN ANALYZE SELECT * FROM products WHERE category = 'Electronics';
```

Likely Output Snippet (Conceptual):

```
QUERY PLAN
-----------------------------------------------------------------------------------------
Seq Scan on products  (cost=0.00..1833.00 rows=33333 width=24) (actual time=0.015..25.123
rows=33333 loops=1)
  Filter: (category = 'Electronics'::text)
  Rows Removed by Filter: 66667
Planning Time: 0.075 ms
Execution Time: 26.543 ms
```

- `Seq Scan on products`: The database reads the entire `products` table.
- `cost=0.00..1833.00`: Estimated cost.
- `actual time=0.015..25.123`: The sequential scan itself took roughly 25 milliseconds.
- `Execution Time: 26.543 ms`: Total time to run the query.

Scenario 2: Creating an index and querying again

```sql
CREATE INDEX idx_products_category ON products(category);


EXPLAIN ANALYZE SELECT * FROM products WHERE category = 'Electronics';
```

Likely Output Snippet (Conceptual):

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on products  (cost=379.00..1280.75 rows=33333 width=24) (actual
time=2.012..5.532 rows=33333 loops=1)
  Recheck Cond: (category = 'Electronics'::text)
  Heap Blocks: exact=591
  ->  Bitmap Index Scan on idx_products_category  (cost=0.00..370.67 rows=33333 width=0)
(actual time=1.500..1.501 rows=33333 loops=1)
```

```
    Index Cond: (category = 'Electronics'::text)

Planning Time: 0.150 ms

Execution Time: 6.230 ms
```

- `Bitmap Index Scan on idx_products_category`: The database uses our new index.
- `Bitmap Heap Scan on products`: After finding relevant rows via the index, it fetches these rows from the table.
- `actual time=2.012..5.532`: The scan using the index is much faster.
- `Execution Time: 6.230 ms`: Total execution time is drastically reduced from ~26ms to ~6ms.

How to read it:

- Look for high costs or high actual times to identify slow steps.
- Compare estimated rows with actual rows. Large discrepancies can indicate outdated statistics or a poor plan.
- Prefer Index Scans over Seq Scans for selective queries on large tables.
- Note operations like "Sort" or "Hash," as these can be memory/CPU intensive.

Common Bottlenecks 🐢

1. Large Table Scans (Sequential Scans):
   - Problem: The database reads every row in a table to find the data it needs. This is very inefficient for large tables if the query is selective (i.e., only a few rows match the criteria).
   - Identification: Seen as `Seq Scan` (PostgreSQL) or `TABLE ACCESS FULL` (Oracle) in execution plans.
   - Solution: Create appropriate indexes on columns used in `WHERE` clauses or join conditions.
2. Suboptimal Joins:
   - Problem: The database chooses an inefficient method to join tables. This can happen due to missing indexes on join keys, outdated table statistics, or overly complex join conditions.
   - Identification: Look at the join types (`Nested Loop`, `Hash Join`, `Merge Join`)

  and their associated costs/times in the execution plan. A nested loop on two large tables without proper indexes is a classic example.
  - Solution:
    - Ensure indexes exist on foreign key columns and any other columns used in join conditions.
    - Keep database statistics up-to-date (`ANALYZE` command in PostgreSQL).
    - Sometimes, rewriting the query or breaking it into smaller parts can help the optimizer.

Basic Performance Metrics ⏱️

1. Latency:
   - Definition: The time it takes for a single query to complete from the moment it's sent to the database until the result is returned. Also known as response time.
   - Goal: Lower latency is better, leading to a snappier user experience.
   - Measurement: `EXPLAIN ANALYZE` shows execution time for a single query. Application performance monitoring (APM) tools can track average query latencies.
2. Throughput:
   - Definition: The number of queries the database can process per unit of time (e.g., queries per second, QPS, or transactions per second, TPS).
   - Goal: Higher throughput is better, indicating the system can handle more concurrent users/operations.
   - Measurement: Database-specific monitoring tools or load testing can measure throughput. Optimizing individual query latency often contributes to better overall throughput by freeing up resources faster.
   -

Okay, let's delve deeper into these join algorithms with SQL examples assuming we have two tables: `orders` and `customers`.

Table: `orders`

order_id	customer_id	order_date	total_amount
1	101	2025-05-20	100.00
2	102	2025-05-21	50.00
3	101	2025-05-22	75.00
4	103	2025-05-23	120.00
...	...	...	...

Table: `customers`

customer_id	name	city
101	Alice	New York
102	Bob	London

103	Charlie	Paris
...	...	...

We want to join these tables on `orders.customer_id = customers.customer_id`.

---

1. Nested Loop Join

As you mentioned, this algorithm iterates through each row of the outer table and, for each row, scans the inner table to find matching rows based on the join condition.

Scenario where it might be chosen:

- One of the tables is small.
- The inner table has a highly selective index on the join key.

Example (Conceptual - MySQL will decide the algorithm):

Let's imagine `customers` is the outer table and `orders` is the inner table, and there's an index on `orders.customer_id`.

The conceptual process would be:

1. Take the first row from `customers` (e.g., Alice, customer_id = 101).
2. Use the index on `orders.customer_id` to quickly find all rows in `orders` where `customer_id` is 101.
3. Combine the matching rows.
4. Repeat this for the next customer (Bob, customer_id = 102), and so on.

SQL Query:

```sql
SELECT o.order_id, c.name, o.order_date

FROM orders o

JOIN customers c ON o.customer_id = c.customer_id;
```

Why it's good in the described scenarios:

- Small outer table: If `customers` is small, iterating through it once is inexpensive. The cost then depends on how efficiently we can find matches in the `orders` table (hence the benefit of an index).
- Selective index on inner table: If there's a good index on `orders.customer_id`, the inner loop (the probe into `orders`) becomes very fast for each customer.

---

2. Hash Join

This algorithm builds an in-memory hash table on one of the tables (typically the smaller one) using the join key. Then, it iterates through the other table, and for each row, it calculates the hash of the join key and looks for matches in the hash table.

Scenario where it might be chosen:

- Joining large tables when there isn't a suitable index for a nested loop join.

Example (Conceptual):

1. MySQL might choose `customers` as the build table (assuming it's smaller).
2. It creates a hash table where the key is `customer_id`. The values in the hash table would point to the rows of the `customers` table.
3. Then, it iterates through each row in the `orders` table.
4. For each order, it calculates the hash of the `customer_id` from the `orders` table.
5. It uses this hash to quickly look up matching `customer_id` values in the hash table created from the `customers` table.
6. Matching rows are then combined.

SQL Query (same as before):

```SQL
SELECT o.order_id, c.name, o.order_date

FROM orders o

JOIN customers c ON o.customer_id = c.customer_id;
```

MySQL would likely choose a Hash Join in this scenario if both `orders` and `customers` are large and there isn't an efficient index that favors a Nested Loop Join.

Why it's good:

- Avoids repeated scanning of the inner table for each row of the outer table (as in a naive nested loop without an index).
- Generally more efficient for large datasets when indexes aren't optimally used.

---

3. Merge Join

This algorithm requires both input tables to be sorted on the join key. Once sorted, it iterates through both tables simultaneously, merging rows where the join keys match.

Scenario where it might be chosen:

- Both tables are already sorted on the join key.
- Sorting both tables is cheaper than the cost of other join algorithms.

Example (Conceptual):

1. First, both the `orders` table and the `customers` table would need to be sorted by `customer_id`.
2. Then, pointers would move through both sorted tables:
   - If the `customer_id` values at the current pointers match, the rows are joined, and both pointers advance.
   - If the `customer_id` in `orders` is smaller, the `orders` pointer advances.
   - If the `customer_id` in `customers` is smaller, the `customers` pointer advances.

SQL Query (same as before, but we might hint at sorting for demonstration, though MySQL usually decides):

```SQL
SELECT o.order_id, c.name, o.order_date

FROM orders o

JOIN customers c ON o.customer_id = c.customer_id

ORDER BY o.customer_id, c.customer_id; -- This just ensures the final output is sorted
```

Why it's good in the described scenarios:

- Already sorted data: If the data is already sorted (e.g., due to a previous operation or a clustered index), the merge join is very efficient as it avoids the overhead of building a hash table or repeatedly scanning.
- Cheap sorting: If the cost of sorting both tables is less than the cost of a hash join or a nested loop join (without a good index), a merge join might be preferred.

Important Note:

MySQL's query optimizer automatically chooses the join algorithm it deems most efficient based on various factors like table sizes, available indexes, and data distribution. You don't typically explicitly tell MySQL which join algorithm to use (though hints exist in some systems). Understanding these algorithms helps in predicting query performance and designing your database (e.g., choosing appropriate indexes).

CAP Theorem

The CAP Theorem, also known as Brewer's Theorem, states that any distributed data store can provide at most two out of the following three guarantees simultaneously:

- Consistency (C): All clients see the same data at the same time, regardless of which node they connect to. This is often referred to as linearizability. Once a write is complete, all future reads will see that write (or a more recent one).
- Availability (A): Every request to a non-failing node receives a response in a reasonable amount of time, without guarantee that the response contains the most recent write. In other words, the system is always operational.
- Partition Tolerance (P): The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between[1] nodes. Network partitions are inevitable in distributed systems.

The Trade-off:

The theorem implies that you have to make a trade-off between Consistency and Availability in the presence of network partitions. Since partition tolerance is generally considered a necessity for a distributed system, you typically have to choose between being:

- CP (Consistent and Partition Tolerant): Sacrifices availability. If a partition occurs, the system might refuse writes or reads on some nodes to ensure consistency. Examples include many traditional RDBMS and some NoSQL databases like MongoDB (with strong consistency settings) and CockroachDB.
- AP (Available and Partition Tolerant): Sacrifices consistency. If a partition occurs, the system will continue to process reads and writes, but the data on different nodes might become inconsistent temporarily. This is often referred to as eventual consistency. Examples include Cassandra and Couchbase.
- CA (Consistent and Available): This is only achievable in a non-distributed system or in a distributed system where network partitions never occur, which is not a realistic assumption for most distributed systems.

In simpler terms:

Imagine you have a distributed database with two nodes, and a network partition occurs, preventing them from communicating.

- If you prioritize Consistency (CP): When a write happens on one node, the other node must also be updated before the write is considered complete. If the nodes can't communicate due to a partition, the system might have to prevent the write from succeeding or make the data on the accessible node unavailable until the partition heals.
- If you prioritize Availability (AP): When a write happens on one node, it's accepted immediately. If the other node is partitioned, it won't receive the update right away. If a read request comes to the partitioned node, it will still serve the (potentially stale) data it has.