Project 1 Instructions

Part 1 – Data Model (40 points)

* Using the supplied SRS, design a crows-foot ERD for the airline DBMS that satisfies first-normal form.
  + There shall be no data duplication across any of the relational tables
* Submit your ERD as an image in PDF format.
  + You can do this using MS Word and saving/printing as PDF.

Part 2 – Implementation (40 points)

* Write the DDL (CREATE) statements to create the database implementing the PK and FK constraints.
* Write the SQL queries/views for the highlighted requirements in section 2.3

Part 3 – Discussion (20 points)

* What types of performance constraints do you see Customers potentially facing if you do not create Views for the queries described in the CUSTOMER FUNCTIONS section?
  + Do you think this creates more overhead for the DB?
  + Do you think query performance will scale as the database scales up?
  + How would you alleviate any potential scaling problems with the DB?
  + Do you think a first-normal form RDBMS is best suited for a highly transactional web-application?
* Your answer should be a minimum of 1000 words or 4 paragraphs.
* Use and cite at least three sources to support your answers.

When considering the performance constraints that customers may face if views are not created for the queries described in the customer functions section, several factors come into play. Without views, each query would need to be executed in full each time it is requested, requiring the database to process complex joins, aggregations, and filters repeatedly. This redundancy leads to increased query execution time and higher resource utilization, particularly as the database grows and the number of queries increases. The absence of precomputed views means that users may experience longer wait times when retrieving flight schedules, reservation information, or sales data.

The increased processing demand without views does create more overhead for the database. Each individual query will require full table scans and expensive join operations, placing additional load on the database server. As a result, overall system performance may degrade, particularly during peak usage periods when multiple customers are requesting information simultaneously. With views, especially materialized views, precomputed results can be stored and refreshed periodically, reducing the burden on the system and improving response times.

Query performance is unlikely to scale efficiently as the database expands if views are not used. As the number of flights, customers, and reservations grows, queries that were previously manageable may become significantly slower. The complexity of joins across multiple tables, particularly with increasing volumes of data, leads to higher latency. Without optimization techniques such as indexing and caching, query execution times will continue to increase, negatively impacting the user experience.

To alleviate potential scaling problems, several optimizations can be implemented. Indexing key columns such as primary and foreign keys can greatly improve query performance by reducing the amount of data scanned during searches. Partitioning large tables can also distribute the workload more efficiently, preventing bottlenecks in database performance. Caching frequently accessed results and employing materialized views to store precomputed results can help reduce processing time. Additionally, optimizing queries through indexing, query rewriting, and denormalization of highly accessed data can improve response times significantly.

A first-normal form relational database management system (RDBMS) is structured to ensure data integrity and consistency, which is essential for transactional applications. However, for a highly transactional web application with a high volume of concurrent reads and writes, a fully normalized database can introduce performance trade-offs. The necessity of performing multiple joins to retrieve data can slow down queries, particularly in real-time applications. While normalization reduces redundancy and ensures data consistency, some denormalization might be beneficial in cases where performance is a priority. Hybrid approaches, such as using NoSQL databases for certain workloads or implementing caching mechanisms, can help balance the need for data consistency and high-speed transactions.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., & Gosain, D. (2012). A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, *1*(6), 1-5.

 This paper compares **relational databases (RDBMS)** and **NoSQL databases**, emphasizing that relational models provide **reliability, consistency, and robustness**, but may struggle with **scalability** under high loads.

 **NoSQL databases** (such as key-value stores, document stores, and column-oriented databases) **scale horizontally** and offer flexibility, but often sacrifice strict ACID compliance.

Kolonko, K. (2018). Performance comparison of the most popular relational and non-relational database management systems.

Gadiraju, K. K., Verma, M., Davis, K. C., & Talaga, P. G. (2016). Benchmarking performance for migrating a relational application to a parallel implementation. *Future Generation Computer Systems*, *63*, 148-156.

 This research investigates **scaling relational database applications** to **big data environments**, comparing MySQL with Hive in a **parallel computing** setup.

 The study finds that **scaling up data size** can significantly impact performance, and **alternative architectures like Hadoop** may provide better efficiency for analytical workloads.

 **Relevance to your database**: This supports the idea that **query optimization (e.g., indexing, partitioning, and materialized views)** is crucial to handling growth in a relational database.

Rolik, O., Ulianytska, K., Khmeliuk, M., Khmeliuk, V., & Kolomiiets, U. (2021, December). Increase efficiency of relational databases using instruments of second normal form. In *2021 IEEE 3rd International Conference on Advanced Trends in Information Theory (ATIT)* (pp. 221-225). IEEE.

 This paper explores how **normalization affects query performance**, particularly in **OLTP (Online Transaction Processing) vs. OLAP (Online Analytical Processing) systems**.

 The study discusses **query optimization techniques** such as **using compound primary keys, query indexing, and optimizing SQL query structures** to improve execution times.

 **Relevance to your database**: Since **flight reservations and payments** require **real-time transactions**, **indexing and query optimization strategies** are crucial for maintaining efficient performance. The study suggests that **denormalizing highly accessed tables** can reduce costly joins.

**Kharade, K. G., Kharade, S. K., Kumbhar, V. S., & Kamat, R. K. (2020). A comparative analysis of using indexed view to improve the performance of SQL. *Analysis of Using Indexed View to Improve the Performance of SQL*, 6-13.**

**Cioloca, C., & Georgescu, M. (2011). Increasing database performance using indexes. *Database Systems Journal*, *2*(2), 13-22.**

**Optimizing Database Performance for Customer Functions**

**Performance Constraints Without Views**

Without utilizing **views**, customers and applications querying the database could encounter significant **performance constraints**, including:

1. **Increased Query Complexity**
   * Customers or applications must manually execute complex **joins and aggregations**.
   * Writing and debugging queries repeatedly increases **development time** and the risk of errors.
2. **Slower Query Execution**
   * Without views, queries must be **compiled and executed** each time they are run.
   * Queries that involve **multiple joins across large tables** result in slower responses.
3. **Higher Load on the Database Server**
   * The DBMS has to compute results in real-time instead of using **precomputed** data from views.
   * This leads to increased **CPU and memory usage**, which may cause performance bottlenecks.

**Does This Create More Overhead for the Database?**

Yes, avoiding **views** can result in **higher overhead**, such as:

* **Repeated Execution Cost:** Every query execution requires parsing, optimization, and execution.
* **Higher Resource Consumption:** CPU, RAM, and disk I/O are used inefficiently, especially with frequent queries.
* **More Network Latency:** If an application fetches and processes data separately, multiple round trips to the database increase latency.

Using **views** reduces the need for redundant query executions and optimizes performance by caching query plans.

**Will Query Performance Scale as the Database Grows?**

Without **views** and proper **indexing**, query performance will not scale efficiently due to:

1. **Growing Data Volume** – Larger datasets increase **join processing** and **filtering times**.
2. **Higher Concurrency** – Multiple customers querying large tables simultaneously can cause **resource contention**.
3. **Frequent Transactions** – Inserts, updates, and deletes will degrade performance if queries are not optimized.

**How to Alleviate Scaling Issues?**

To ensure **database scalability**, the following optimizations should be applied:

1. **Use Materialized Views**
   * Unlike regular views, **materialized views** store precomputed results for faster retrieval.
   * This significantly improves **query performance** but requires periodic refreshing.
2. **Partitioning Large Tables**
   * **Horizontal partitioning** (sharding) splits large tables into smaller, manageable sections.
   * Example: Partition flight\_schedules by **date** or passengers by **region**.
3. **Indexing Critical Columns**
   * Implement **B-tree indexes** for frequently queried columns (flight\_number, passenger\_id).
   * Consider **covering indexes** for queries filtering and returning data from the same columns.
4. **Read-Replica Databases**
   * Offload **read-heavy operations** to replica databases.
   * Helps scale applications where customers **frequently read but don’t modify data**.
5. **Implement Caching Strategies**
   * Use **Redis** or **Memcached** to store **frequent query results**.
   * Reduces the need for **re-executing complex queries**.

**Is First-Normal Form (1NF) Best for a Highly Transactional Web Application?**

No, strictly enforcing **1NF** is not sufficient for high-performance transactional applications.

✅ **Pros of 1NF:**

* Eliminates **duplicate data** and **ensures atomic values**.
* Simplifies **data consistency** across transactions.

❌ **Cons of 1NF for a Web Application:**

* **Expensive Joins** – Strict **normalization (3NF or higher)** requires **multiple joins**, slowing down reads.
* **High Transactional Load** – A **denormalized approach** (precomputed columns or JSONB fields) can **improve response times**.

**Better Approach?**

For **highly transactional web applications**, a hybrid approach is recommended:

1. **3NF for critical data consistency** (e.g., payments, transactions).
2. **Denormalized tables for read-heavy operations** (e.g., precomputed sales reports).
3. **Hybrid models with NoSQL stores** (e.g., storing customer preferences in Redis).

**Final Conclusion**

* **Views optimize query performance** and reduce application complexity.
* **Without views, performance degrades** as the database scales.
* **Indexes, partitioning, and caching** are key to **handling database growth**.
* **1NF alone is not ideal** for high-traffic web applications—**a hybrid approach is best**.