Case Study on SQL Server: Patient Management System

Problem Statement: Patient Management System with Performance Tuning

Background

CityCare Hospital requires a **Patient Management System** implemented in SQL Server to manage patient records, doctor information, and appointment scheduling. The system must handle daily operations efficiently, support reporting, and ensure scalability for a growing number of patients and appointments. To address potential performance bottlenecks as the database scales, the solution must incorporate **performance tuning** techniques, such as indexing, query optimization, and best practices for stored procedures and UDFs.

Objective

Design and implement a SQL Server database that:

- Manages patient, doctor, and appointment data with robust data manipulation and querying capabilities.
- Supports automation through stored procedures and UDFs.
- Incorporates **performance tuning** to ensure fast query execution, minimal resource usage, and scalability for large datasets.

SQL Server concepts

This case study includes the following concepts from SQL Server:

- **DML Commands**: Batch inserts, precise updates, and deletes to minimize overhead.
- **Joins**: Optimized with indexed columns and minimal column selection.
- **Filtering with WHERE**: Uses sargable conditions and indexes for faster execution.
- **GROUP BY Clause**: Leverages indexes for aggregation and early filtering.
- **Stored Procedure**: Includes SET NOCOUNT ON, error handling, and WITH RECOMPILE for dynamic query plans.
- **UDF**: Marked as deterministic with SCHEMABINDING for better performance.
- **Performance Tuning**:
    - Non-clustered indexes on foreign keys and filter columns (e.g., AppointmentDate, Status).
    - Covering indexes with INCLUDE for frequently selected columns.
    - Index maintenance with ALTER INDEX REBUILD.
    - Monitoring index usage with sys.dm_db_index_usage_stats.

Requirements

The database must include the following components and functionalities, with performance tuning considerations:

1. **Database Schema**:
    - **Patients Table**: Store patient details (PatientID, FirstName, LastName, DateOfBirth, Gender, ContactNumber).
    - **Doctors Table**: Store doctor details (DoctorID, DoctorName, Specialty, Department).
    - **Appointments Table**: Store appointment details (AppointmentID, PatientID, DoctorID, AppointmentDate, Reason, Status).
    - **Performance Tuning**:
        - Create **indexes** on frequently queried columns (e.g., PatientID, DoctorID, AppointmentDate, Status) to speed up searches and joins.
        - Use **clustered indexes** on primary keys and **non-clustered indexes** on foreign keys and filter columns.
        - Avoid over-indexing to minimize insert/update overhead.
2. **DML Operations**:
    - **Insert**, **Update**, and **Delete** records for patients, doctors, and appointments.
    - **Performance Tuning**:
        - Use batch inserts for sample data to reduce transaction overhead.
        - Ensure updates and deletes are targeted with precise WHERE clauses to avoid table scans.
        - Monitor index fragmentation and rebuild indexes if needed.
3. **Joins**:
    - Use **INNER JOIN** and **LEFT JOIN** to combine data for reports.
    - **Performance Tuning**:
        - Ensure joins use indexed columns (e.g., PatientID, DoctorID).
        - Avoid unnecessary columns in SELECT statements to reduce I/O.
        - Use query execution plans to identify and optimize costly join operations.
4. **Filtering with WHERE**:
    - Filter appointments by date, status, or department, and patients by age.
    - **Performance Tuning**:
        - Create indexes on filter columns (e.g., AppointmentDate, Status).

- Use sargable conditions (e.g., AppointmentDate >= '2025-08-01' instead of YEAR(AppointmentDate) = 2025) to leverage indexes.
- Avoid functions on indexed columns in WHERE clauses to prevent index scans.

5. **GROUP BY Clause**:
   - Aggregate data (e.g., appointments per department, completed appointments).
   - **Performance Tuning**:
     - Ensure GROUP BY columns are indexed to speed up aggregation.
     - Use computed columns or indexed views for frequently aggregated data, if applicable.
     - Optimize HAVING clauses to filter early in the query execution.
6. **Stored Procedure**:
   - Create a stored procedure to schedule appointments.
   - **Performance Tuning**:
     - Use parameterized queries to prevent SQL injection and enable query plan reuse.
     - Minimize locking by using appropriate transaction isolation levels (e.g., READ COMMITTED).
     - Include error handling to avoid performance degradation from unhandled exceptions.
7. **User-Defined Function (UDF)**:
   - Create a UDF to calculate patient age.
   - **Performance Tuning**:
     - Use deterministic scalar UDFs for better performance.
     - Consider inlining calculations in queries instead of UDFs for frequently executed operations to avoid overhead.
     - Test UDF performance with large datasets and optimize if necessary.
8. **Comprehensive Queries**:
   - Generate reports combining joins, filtering, and UDFs.
   - **Performance Tuning**:
     - Use covering indexes to include all columns needed for a query in the index itself.
     - Break complex queries into smaller, indexed temporary tables if needed.

- Analyze query execution plans to identify and resolve performance bottlenecks.

Performance Tuning Instructions

- **Indexing Strategy**:
  - Create a clustered index on primary keys (e.g., PatientID, DoctorID, AppointmentID).
  - Create non-clustered indexes on foreign keys (e.g., PatientID, DoctorID in Appointments) and frequently filtered columns (e.g., AppointmentDate, Status).
  - Use INCLUDE in non-clustered indexes to cover frequently selected columns (e.g., Reason, Status).
  - Monitor index usage with sys.dm_db_index_usage_stats and remove unused indexes.
  - Rebuild fragmented indexes using ALTER INDEX REBUILD if fragmentation exceeds 30%.
- **Query Optimization**:
  - Use query execution plans in SSMS to identify table scans or costly operations.
  - Replace correlated subqueries with joins where possible to improve performance.
  - Use TOP the query execution plan for each query to identify bottlenecks and optimize performance.