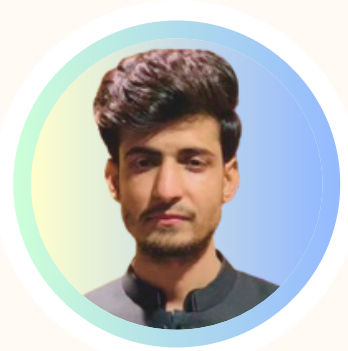
deepdecide.com

SQL DATA OPTIMIZATION

Optimizing data in SQL involves various techniques, such as indexing, query optimization, proper data modeling, and regular maintenance. Here are 20 detailed examples to illustrate these techniques.

Shah Jahan

linkedin.com/in/shah907



1. Creating an Index

Indexes improve query performance by allowing the database to quickly locate rows.

Example:

```
CREATE INDEX idx_job_title ON employees(job_title);
```

Creates an index named idx_job_title on the job_title column of the employees table.

This improves query performance when filtering by job_title.

2. Using Composite Indexes

Composite indexes cover multiple columns and are useful for queries involving multiple columns.

Example:

```
CREATE INDEX idx_name_job ON employees(name, job_title);
```

Creates an index on both name and job_title.

Optimizes queries that filter or sort by both columns

3. Optimizing Query with SELECT

Retrieve only necessary columns to reduce data load.

Inefficient Query:

```
SELECT * FROM employees WHERE job_title = 'Developer';
```

Optimized Query:

```
SELECT name, salary FROM employees WHERE job_title = 'Developer';
```

Only retrieves the name and salary columns, reducing the amount of data processed.

4. Using EXISTS Instead of IN

EXISTS can be more efficient than IN for subqueries.

Inefficient Query:

```
SELECT name FROM employees WHERE employee_id IN (SELECT
employee_id FROM projects WHERE status = 'active');
```

Optimized Query:

```
SELECT name FROM employees WHERE EXISTS (SELECT 1 FROM
projects WHERE projects.employee_id = employees.employee_id AND
status = 'active');
```

EXISTS stops searching after finding the first match, whereas IN processes all matches

5. Using JOINs Instead of Subqueries

JOINs can be more efficient than subqueries.

Inefficient Query:

```
SELECT name FROM employees WHERE employee_id IN (SELECT employee_id
FROM projects WHERE status = 'active');
```

Optimized Query:

```
SELECT employees.name FROM employees JOIN projects ON
employees.employee_id = projects.employee_id WHERE projects.status = 'active';
```

JOIN retrieves data in a single pass, whereas subqueries may require multiple passes.

6. Using Proper Data Types

Choose appropriate data types to optimize storage and performance.

Inefficient Query:

```
CREATE TABLE sales (
    sale_id INT,
    amount VARCHAR(50)
);
```

Optimized Query:

```
CREATE TABLE sales (
    sale_id INT,
    amount DECIMAL(10, 2)
);
```

Using DECIMAL for amount ensures proper storage and arithmetic operations.

7. Using Partitioning

Partition large tables to improve query performance.

Example:

```
CREATE TABLE orders (
    order_id INT,
    order_date DATE,
    customer_id INT,
    amount DECIMAL(10, 2)
) PARTITION BY RANGE (order_date) (
    PARTITION p0 VALUES LESS THAN ('2021-01-01'),
    PARTITION p1 VALUES LESS THAN ('2022-01-01'),
    PARTITION p2 VALUES LESS THAN (MAXVALUE)
);
```

Partitions the orders table by order_date.

Queries on specific date ranges will only scan the relevant partitions.

8. Using Indexes on Foreign Keys

Indexes on foreign keys can improve join performance.

Example:

```
CREATE INDEX idx_customer_id ON orders(customer_id);
```

Creates an index on the customer_id column of the orders table.

Improves performance of joins between orders and customers.

9. Using Covering Indexes

Covering indexes include all columns required by a query.

Example:

```
CREATE INDEX idx_employee_covering ON employees(employee_id, name, salary);
```

Index includes all columns (employee_id, name, salary) needed by a query.

The query can be satisfied entirely by the index without accessing the table.

10. Avoiding Functions on Indexed Columns

Avoid using functions on indexed columns as it prevents index usage.

Inefficient Query:

```
SELECT * FROM employees WHERE UPPER(name) = 'JOHN DOE';
```

Optimized Query:

```
SELECT * FROM employees WHERE name = 'John Doe';
```

Using UPPER on name prevents the use of an index on name.

Directly comparing the value allows the index to be used.

11. Using UNION ALL Instead of UNION

UNION ALL is faster than UNION because it doesn't remove duplicates.

Inefficient Query:

```
SELECT name FROM employees WHERE department = 'HR'
UNION
SELECT name FROM employees WHERE department = 'Finance';
```

Optimized Query:

```
SELECT name FROM employees WHERE department = 'HR'
UNION ALL
SELECT name FROM employees WHERE department = 'Finance';
```

UNION ALL combines results without checking for duplicates, improving performance.

12. Avoiding SELECT DISTINCT

Use SELECT DISTINCT only when necessary as it adds overhead.

Inefficient Query:

```
SELECT DISTINCT name FROM employees;
```

Optimized Query:

```
SELECT name FROM employees GROUP BY name;
```

GROUP BY can sometimes be more efficient than DISTINCT, especially with indexes.

13. Using Batch Updates

Batch updates reduce the number of transaction commits.

Inefficient Query:

```
UPDATE employees SET salary = salary * 1.1;
```

Optimized Query:

```
-- Batch size of 1000
DECLARE @BatchSize INT = 1000;
DECLARE @MinID INT = (SELECT MIN(employee_id) FROM employees);
DECLARE @MaxID INT = (SELECT MAX(employee_id) FROM employees);

WHILE @MinID <= @MaxID
BEGIN
    UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id BETWEEN @MinID AND @MinID + @BatchSize - 1;

    SET @MinID = @MinID + @BatchSize;
END;
```

Processes updates in batches of 1000 rows, reducing transaction overhead.

14. Using Stored Procedures

Stored procedures can precompile and optimize query execution plans.

Example:

```
CREATE PROCEDURE GetEmployeeByID
    @EmployeeID INT
AS
BEGIN
    SELECT * FROM employees WHERE employee_id = @EmployeeID;
END;
```

Encapsulates the query in a stored procedure, which can be optimized by the database.

15. Updating Statistics

Keeping statistics up-to-date helps the query optimizer.

Example:

```
UPDATE STATISTICS employees;
```

Updates statistics for the employees table, helping the optimizer choose the best execution plan.

16. Using Temporary Tables

Temporary tables can improve performance for complex queries.

Example:

```
CREATE TABLE #TempEmployees (employee_id INT, name NVARCHAR(100));

INSERT INTO #TempEmployees
SELECT employee_id, name FROM employees WHERE job_title = 'Developer';

SELECT * FROM #TempEmployees;
```

Stores intermediate results in a temporary table, reducing the complexity of the final query.

17. Using Query Hints

Query hints can force the optimizer to use a specific index.

Example:

```
SELECT * FROM employees WITH (INDEX(idx_job_title)) WHERE job_title =
'Developer';
```

Forces the query to use the idx_job_title index, which can improve performance.

18. Avoiding Cursors

Cursors can be slow; use set-based operations instead.

Inefficient Cursor:

```
DECLARE employee_cursor CURSOR FOR
SELECT employee_id, salary FROM employees;

OPEN employee_cursor;

FETCH NEXT FROM employee_cursor INTO @EmployeeID, @Salary;

WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE employees SET salary = salary * 1.1 WHERE employee_id =
@EmployeeID;
    FETCH NEXT FROM employee_cursor INTO @EmployeeID, @Salary;
END;

CLOSE employee_cursor;
DEALLOCATE employee_cursor;
```

Optimized Query:

```
UPDATE employees SET salary = salary * 1.1;
```

The set-based update is more efficient than using a cursor.

19. Avoiding OR in WHERE Clauses

Using OR in WHERE clauses can prevent index usage.

Inefficient Cursor:

```
SELECT * FROM employees WHERE job_title = 'Developer' OR job_title =
'Manager';
```

Optimized Query:

```
SELECT * FROM employees WHERE job_title IN ('Developer', 'Manager');
```

IN allows the use of an index, whereas OR may not.

20. Using Execution Plans

Analyze and optimize queries using execution plans.

Example:

```
EXPLAIN PLAN FOR
SELECT * FROM employees WHERE job_title = 'Developer';
```

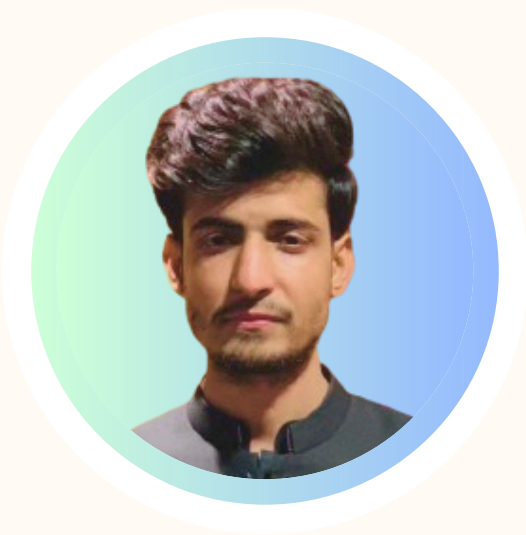
Provides the execution plan for the query, helping identify performance bottlenecks.

JOIN NOW

linkedin.com/in/shah907

deepdecide.com

Free Notes: https://deepdecide.com/resources



Shah Jahan