Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2019, 6(12):311-314**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Key Solutions to Optimize Database SQL Queries

Vishnupriya S Devarajulu

Vishnupriyasupriya@gmail.com

Abstract: Optimizing SQL Queries is crucial for enhancing the performance and efficiency of database-driven applications. This article explores key solutions to the performance issues in SQL queries, with code samples and detailed explanations. Best practices such as using indexes, avoiding unnecessary columns in SELECT statements, using schema names with object names, and optimizing joins and subqueries and other solutions are discussed. By following these optimization techniques, developers can provide more efficient database with improved application performance.

Keywords: Database Optimization, SQL Queries, Performance Tuning, Indexing, Query Optimization

Introduction

SQL queries are fundamental pillars to the database operations, such as data retrieval, manipulation, and storage. However, weak, inefficient SQL queries can lead to significant performance bottlenecks, there by affecting the overall performance of applications. This article provides best practical solutions for optimizing SQL queries, addressing performance issues with explanations and code samples.

Key Solutions

1. Use Proper Indexes

Issue: Lack of necessary indexes can lead to slow query performance due to full table scans.
Solution: Create indexes on columns that are used in WHERE clauses, JOIN conditions, and ORDER BY clauses.

```
CREATE INDEX idx_customer_lastname
ON Customers LastName;
```

Use Joins Instead of Subqueries

Issue: Subqueries are less efficient and harder for the query optimizer to optimize.
Solution: Use JOIN operations to improve query performance.

```
-- Subquery -- Inefficient way
SELECT FirstName, LastName
FROM Customers
WHERE CustomerID IN (SELECT CustomerID FROM Orders WHERE OrderDate > '2019-
01-01');

-- Join -- Efficient way
SELECT c.FirstName, c.LastName
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
```

```
WHERE o.OrderDate > '2019-01-01';
```

Avoid Using SELECT*
Issue: Selecting all columns can lead to inefficient use of resources and slower performance.
Solution: Select only the columns that are needed.

```
SELECT FirstName, LastName
FROM Customers
WHERE OrderName = 'Apple';
```

Filter Early
Issue: Filtering data late in the query can lead to processing large data sets unnecessarily.
Solution: Apply filters as early as possible in the query.

SELECT FirstName, LastName
FROM Customers
WHERE CustomerDetail = 'Apple'
ORDER BY LastName;

Avoid Functions on Indexed Columns
Issue: Using functions on indexed columns can prevent the use of indexes.
Solution: Avoid functions on indexed columns in WHERE clauses.

```
-- Inefficient way
SELECT * FROM Customers
WHERE YEAR(BirthDate) = 1990;

-- Efficient way
SELECT * FROM Customers
WHERE BirthDate BETWEEN '1990-01-01' AND '1990-12-31';
```

Use EXISTS Instead of IN for Subqueries
Issue: The IN clause is inefficient when used with subqueries.
Solution: Use EXISTS for better performance.

```
-- Using IN - Inefficient way
SELECT FirstName, LastName
FROM Customers
WHERE CustomerID IN (SELECT CustomerID FROM Orders WHERE OrderDate > '2019-
01-01');
-- Using EXISTS - Efficient way
SELECT FirstName, LastName
FROM Customers c
WHERE EXISTS (SELECT 1 FROM Orders o WHERE o.CustomerID = c.CustomerID AND
o.OrderDate > '2019-01-01');
```

Avoid Wildcards at the Beginning of LIKE
Issue: Using wildcards at the beginning of LIKE patterns prevents the use of indexes.
Solution: Avoid leading wildcards in LIKE patterns.

```
-- Inefficient way
SELECT * FROM Customers
```

```
WHERE LastName LIKE '%son';

-- Efficient way
SELECT * FROM Customers
WHERE LastName LIKE 'John%';
```

Optimize ORDER BY With Indexes
Issue: Sorting large data sets can lead to heavy resource utilization
Solution: Use indexes on columns used in ORDER BY clauses.

```
CREATE INDEX idx_customer_lastname
ON Customers (LastName);

SELECT FirstName, LastName
FROM Customers
ORDER BY LastName;
```

Use UNION ALL Instead Of UNION
Issue: UNION removes duplicates, which adds overhead.
Solution: Use UNION ALL if duplicates are not a concern.

```
-- Using UNION
SELECT FirstName FROM Customers
UNION
SELECT FirstName FROM Employees;

-- Using UNION ALL
SELECT FirstName FROM Customers
UNION ALL
SELECT FirstName FROM Employees;
```

Limit The Result Set
Issue: Retrieving more data than necessary leads to resource utilization
Solution: Use the LIMIT clause to restrict the number of rows returned.

```
SELECT FirstName, LastName
FROM Customers
ORDER BY LastName
LIMIT 10;
```

Conclusion

Optimizing SQL queries is crucial for providing and maintaining efficient and responsive Database. By implementing the proposed best practices such as using proper indexes, avoiding SELECT *, using joins instead of subqueries, avoiding functions on indexed columns, and limiting the result set, developers can significantly improve the performance and reliability of their SQL queries. Thus, it contributes to better resource utilization of Database and provides enhanced application performance.

References

[1]. Microsoft Docs. "Optimizing SQL Queries." Last modified June 6, 2017. https://docs.microsoft.com/en-us/sql/relational-databases/performance/optimizing-sql-queries
[2]. SQL Server Performance. "Best Practices for Writing SQL Queries." Last accessed November 15, 2018. https://www.sql-server-performance.com/best-practices-writing-sql-queries/

[3]. Redgate. "SQL Query Optimization Techniques." Last accessed October 10, 2019. https://www.red-gate.com/simple-talk/sql/performance/sql-query-optimization-techniques/

[4]. Stack Overflow. "Optimizing SQL Queries." Last accessed March 25, 2018. https://stackoverflow.com/questions/25278901/optimizing-sql-queries