Day 18- Facilitation Guide

Index

For (1.5 hrs) ILT

During the previous session, we explored PreparedStatement,Callable Statement and using JDBC with other databases.

We covered a variety of important concepts:

Prepared Statement:

A Prepared Statement is a feature in database programming that allows for the precompilation and optimization of SQL queries before execution. It is mainly used to improve the performance and security of database operations. Prepared Statements have placeholders for parameters, and these placeholders can be dynamically filled with values at runtime.

Callable Statement:

A Callable Statement is a JDBC feature used to call stored procedures or functions in a database. It is specifically designed for interacting with database procedures and functions. Callable Statements can handle input and output parameters, making them suitable for invoking complex database operations. They enable the execution of database transactions, the retrieval of result sets, and the management of IN, OUT, and INOUT parameters.

In this session, we will delve into performance Tuning and the Security concept.

I. Performance Tuning

SQL tuning is the process of enhancing SQL queries to speed up the performance of your server. Its main goal is to shorten the time it takes for a user to receive a response after sending a query and to utilize fewer resources in the process. The idea is that users can occasionally produce the same intended result set with a faster-running query.

There is not just one tool or method for optimizing SQL speed. Instead, it's a set of procedures that utilize a variety of methods, and procedures. Let's talk about some of the major factors that will influence how many computations you must perform and how long it takes for your query to run:

- Table size: Performance may be impacted if your query hits one or more tables with millions of rows or more.
- Joins: Your query is likely to be slow if it joins two tables in a way that significantly raises the number of rows in the return set.
- Aggregations: Adding several rows together to create a single result needs more processing than just retrieving those values individually.
- Other users executing queries: The more queries a database has open at once, the more it must process at once, and the slower it will all be. It can be particularly problematic if other users are using a lot of resources to perform queries that meet some of the aforementioned requirements.

a) **The theory behind query run time**

A database is a piece of software that runs on a computer, and is subject to the same limitations as all software—it can only process as much information as its hardware is capable of handling. The way to make a query run faster is to reduce the number of calculations that the software (and therefore hardware) must perform. To do this, you'll need some understanding of how SQL actually makes calculations. First, let's address some of the high-level things that will affect the number of calculations you need to make, and therefore your queries runtime:

- Table size: If your query hits one or more tables with millions of rows or more, it could affect performance.
- Joins: If your query joins two tables in a way that substantially increases the row count of the result set, your query is likely to be slow. There's an example of this in the subqueries lesson.
- Aggregations: Combining multiple rows to produce a result requires more computation than simply retrieving those rows.

Query runtime is also dependent on some things that you can't really control related to the database itself:

- Other users running queries: The more queries running concurrently on a database, the more the database must process at a given time and the slower everything will run. It can be especially bad if others are running particularly resource-intensive queries that fulfill some of the above criteria.
- Database software and optimization: This is something you probably can't control, but if you know the system you're using, you can work within its bounds to make your queries more efficient.

Ways to Find Slow SQL Queries in SQL Server to Measure SQL Server Performance:

1. **Create an Execution Plan:** It is essential to be able to create execution plans, which you can accomplish with SQL Server Management Studio, in order to diagnose delayed queries. After the queries run, actual execution plans are generated. To create an execution plan:

- Start by selecting “Database Engine Query” from the toolbar of SQL Server Management Studio.
- Enter the query after that, and then select “Include Actual Execution Plan” from the Query option.
- It’s time to run your query at this point. You can do that by pressing F5 or the “Execute” toolbar button.
- The execution plan will then be shown in the results pane, under the “Execution Pane” tab, in SQL Server Management Studio.

2. Monitor Resource Usage: The performance of a SQL database is greatly influenced by resource use. Monitoring resource use is important since you can’t improve what you don’t measure. Use the System Monitor tool on Windows to evaluate SQL Server’s performance. You may view SQL Server objects, performance counters, and other object activity with it. Simultaneously watch Windows and SQL Server counters with System Monitor to see if there is any correlation between the two services’ performance.

3. Use SQL DMVs to Find Slow Queries: The abundance of dynamic management views (DMVs) that SQL Server includes is one of its best features. There are many of them, and they may offer a lot of knowledge on a variety of subjects.

Various DMVs are available that offer information on query stats, execution plans, recent queries, and much more. These can be combined to offer some incredible insights.

Optimizing a Query for SQL:

The server/database is crucial. If a query is inefficient or contains errors, it will consume up the production database's resources and slow down or disconnect other users. You must optimize your queries to have the least possible negative influence on database performance. The following techniques can be used to optimize SQL queries:

1. SELECT fields instead of using SELECT *:

Many SQL developers use the SELECT * shortcut to query all of the data in a table while executing exploratory queries. However, if a table contains a lot of fields and rows, the database would be taxed by superfluous data queries.

By using the SELECT statement, one may direct the database to only query the data you actually need to suit your business needs. For example:

Inefficient:
```
Select * from Student;
```

Efficient:
```
SELECT FirstName, LastName,
DateOfBirth, Gender, Email, Phone FROM Student;
```

2. Avoid SELECT DISTINCT:

It is practical to get rid of duplicates from a query by using SELECT DISTINCT. To get separate results, SELECT DISTINCT GROUPs for every field in the query. However, a lot of computing power is needed to achieve this goal. Furthermore, data may be inaccurately classified to a certain extent. Choose more fields to produce distinct results instead of using SELECT DISTINCT.

Inefficient and inaccurate:

SELECT DISTINCT FirstName, LastName,
DateOfBirth, Gender, Email, Phone FROM Student;

Efficient and accurate:

SELECT FirstName, LastName,
DateOfBirth, Gender, Email, Phone FROM Student;

3. Create queries with INNER JOIN (not WHERE or cross join):

A Cartesian Connection, also known as a Cartesian Product or a CROSS JOIN, is produced by this kind of join. A Cartesian Join creates every conceivable combination of the variables. If we had 1,000 customers and 1,000 in total sales in this example, the query would first produce 1,000,000 results before filtering for the 1,000 entries where CustomerID is correctly connected. The database has performed 100 times more work than was necessary, therefore this is a wasteful use of its resources. Due to the possibility of producing billions or trillions of results, Cartesian Joins pose a particular challenge for large-scale databases.

To prevent creating a Cartesian Join, use INNER JOIN instead:

select Student.FirstName,Course.CourseTitle,Score.CreditObtained from Score INNER JOIN Course on Course.CourseId=Score.CourseId INNER JOIN Student ON Student.StudentId=Score.StudentId;

The database would only generate the limited desired records where StudentId is equal.

4. Use wildcards at the end of a phrase only:

Wildcards enable the broadest search when searching unencrypted material, such as names or cities. However, the most extensive search is also the least effective. The database is required to search all records for a match anywhere inside the chosen field when a leading wildcard is used, particularly when combined with an ending wildcard. Think about using this search and find cities that start with “No”:

select Firstname from student where FirstName like '%j%';

The expected results are John, and Jane and Jim, which will be returned by this query.

5. Use LIMIT to sample query results:

Use a LIMIT statement to check if the results will be pleasing and useful before executing a query for the first time. (In some DBMS systems, the terms TOP and LIMIT are used synonymously.) Only the given number of records are returned by the LIMIT statement. By using a LIMIT statement, you can avoid stressing the production database with a big query only to discover that it needs to be edited or improved.

Select * from student limit 3;

Question: What are the best practices for writing efficient SQL queries? What common mistakes should be avoided?

b) Index Tuning:

Index tuning in SQL involves designing and maintaining indexes on database tables to improve query performance

Advantages of Index Tuning:
The performance of queries and databases can be improved by using the Index tuning wizard. It accomplishes this using the following methods:

1. It makes recommendations for the best index usage based on the analysis of queries performed by the query optimizer in relation to workload.
2.The changes in query distribution, index utilization, and performance are examined to determine the impact. Additionally, it suggests strategies to fine-tune the database for a select group of problematic queries.
3.SQL profiler is used to record activity traces and to improve performance. To capture a broad range of data, the trace can be prolonged for a while.

Here's an example of how to perform index tuning:

Suppose you have a table named "Employees" with the following structure:

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Department VARCHAR(50),
   Salary INT
);
```

You frequently run queries to retrieve employees based on their department. Without any indexes, the query might look like this:

SELECT * FROM Employees WHERE Department = 'Sales';

To optimize this query, you can create an index on the "Department" column:
CREATE INDEX idx_Department ON Employees (Department);

This index speeds up queries that filter employees by department. When you run the same query with the index in place, it can quickly locate the rows that match the department criteria, resulting in faster query performance.

Execution: When you execute this SELECT query, the database management system will automatically use the index "idx_Department" to find the rows that match the criteria.Which results in improved query performance. This is especially beneficial when dealing with large datasets, as it significantly reduces the time needed to find the relevant records.

Here are some key points to keep in mind when tuning indexes:

However, it's important to note that over-indexing can be detrimental to performance, so it's essential to choose which columns to index carefully. Consider the types of queries you frequently run and the columns involved in those queries.

- **Primary Key:** A primary key creates a unique index automatically. Ensure that you have a primary key on your table for data integrity.

- **Unique Indexes:** Use unique indexes when you need to enforce uniqueness on one or more columns.

- **Composite Indexes:** If queries often involve multiple columns, consider creating composite indexes that cover all the columns needed for those queries.

- **Clustered Indexes:** In some database systems (e.g., SQL Server), you can define a clustered index on a table, which determines the physical order of data rows. Use clustered indexes for columns that are frequently used in range queries, such as date ranges.

- **Regular Monitoring:** Regularly monitor index performance and query execution plans to identify areas where indexes can be beneficial.

- **Index Maintenance:** Ensure that indexes are regularly maintained, which might involve rebuilding or reorganizing them, especially in high-insert, update, or delete environments.

- **Drop Unnecessary Indexes:** If you find that certain indexes are no longer beneficial or are rarely used, consider dropping them to reduce the overhead of maintaining unnecessary indexes.

Index tuning should be an ongoing process that adapts to the changing needs of your application. By carefully selecting and maintaining indexes, you can significantly improve SQL query performance.

Let’s us see Few more common methods for SQL performance tuning include:

- **Database Schema Design:** Structuring the database schema to reduce redundant data and improve data retrieval.

- **Database Server Configuration:** Adjusting database server settings and parameters to meet the specific needs of your application.

- **Caching:** Implementing caching mechanisms to store frequently accessed data and reduce the need for repetitive database queries.

- **Hardware Upgrades:** Upgrading the hardware infrastructure, such as adding more memory or faster storage, to improve database performance.

- **Database Maintenance:** Regularly performing tasks like database vacuuming, reindexing, and analyzing to keep the database in optimal condition.

- **Connection Pooling:** Using connection pooling to efficiently manage and reuse database connections.

- **Load Balancing:** Distributing database workloads across multiple servers to prevent overloading a single database instance.

- **Profiling and Monitoring:** Continuously monitoring the database's performance, identifying bottlenecks, and making improvements based on performance profiling data.

- **SQL Execution Plans:** Analyzing and optimizing SQL execution plans to ensure that the database engine processes queries efficiently.

- **Query Tuning Tools:** Using specialized tools and utilities for database query analysis and optimization.

II. Security

Security in MySQL is a critical aspect of database administration. It's essential to protect your MySQL database from unauthorized access, data breaches, and other security threats. Here are some key security measures to consider when working with MySQL:

1. Authentication and Authorization:

- Use strong, unique passwords for MySQL user accounts.
- Avoid using the default "root" user for application access; create dedicated user accounts with appropriate privileges.
- Limit access to only the necessary IP addresses or hosts.
- Set strong password policies using the validate_password plugin.
- Apply the principle of least privilege – grant only the necessary permissions to users and roles.

2. **Encryption:**

- Encrypt data in transit using SSL (Secure Sockets Layer) or TLS (Transport Layer Security).
- Enable SSL support in MySQL server configuration.
- Use client-side SSL certificates for enhanced security.
- Encrypt sensitive data in the database using encryption functions or algorithms like AES.

3. **Network Security:**
- Secure your server's network by configuring firewalls to allow only necessary network traffic.
- Disable remote access to the MySQL server if it's not required.
- Bind MySQL to specific network interfaces and IP addresses.
- Use SSH tunnels for secure remote connections to the server.

4. **Updates and Patch Management:**

- Regularly update your MySQL server to apply security patches and bug fixes.
- Stay informed about security updates and subscribe to MySQL security mailing lists.

5. **Logging and Auditing:**

- Enable and review MySQL server logs to monitor activities and detect unauthorized access.
- Implement auditing plugins to keep a record of specific actions and changes in the database.

6. **Backup and Recovery:**

- Regularly backup your database and store backups securely.
- Test your backup and recovery procedures to ensure data can be restored in case of a security incident.

7. **Database Firewall:**

- Consider using a database firewall or intrusion detection system (IDS) to monitor and block malicious activities.

8. **SQL Injection Prevention:**

- Sanitize user input and use prepared statements or parameterized queries to prevent SQL injection attacks.

9. **User Account Management:**

- Disable or remove unused or unnecessary user accounts.
- Implement multi-factor authentication (MFA) for additional user account security.

10. **Monitoring and Alerting:**

- Set up monitoring tools to track system and database performance and detect anomalies.
- Configure alerts to notify you of security breaches or performance issues.

11. **Secure Configuration:**

- Follow security best practices when configuring MySQL.
- Disable features or options that are not needed.

12. **User Education:**

- Educate database administrators and developers on security best practices and potential threats.

13. **Third-Party Software and Plug-ins:**

- Carefully evaluate and secure any third-party software or plug-ins used with MySQL.

Conclusion

Database security is an important goal of any data management system. Each organization should have a data security policy, which is set of high-level guidelines determined by:

- User requirements
- Environmental aspects
- Internal regulations
- Government laws

Exercise:

Use ChatGPT to explore more Security in SQL

Put the below problem statement in the message box and see what ChatGPT says.

What security features and mechanisms are provided by the specific SQL database management system you are using, such as MySQL, PostgreSQL, SQL Server, or Oracle?