TRIGENT

Performance Management of SQL Server

Padma Krishnan
Senior Manager

Trigent ■ 2 Willow Street, Suite 201, Southborough, MA 01745 ■ 1-877-387-4436 ■ www.trigent.com

When we design applications, we give equal importance to the backend database as we do to the architecture and design of the application itself. A poorly designed database can become a bottleneck to an otherwise well designed application. And as a system is only as strong as the weakest link, we make sure that the database is well designed and tuned. This paper will examine some of the best practices followed by our teams using Microsoft SQL Server database in application development.

Introduction

Performance management of SQL Server is a vast topic and there are several distinct areas of focus. SQL Server Administration, SQL Server Design, SQL Server programming best practices, SQL Server tuning, SQL Server Disaster Recovery and Replication are some distinct areas. This paper will focus mostly on the design and programming best practices.

SQL Server Design

The key to a good design is a clear understanding of the business requirements of the application in question. You cannot effectively model what do not understand fully. This task is best done by a person who has a combination of business and technical savvy or a team comprising of a combination of business analysts and technical members.

Logical Data Model

While there are many tools to design a logical data model, we generally use Visio, the diagramming tool of SSMS (SQL Server Management Studio), Visual Studio and CA's ERWIN. The logical data model can be arrived at by understanding the different entities needed in the business, parent-child relationships and the relationships amongst various entities.

It is very important that the tables be normalized to the extent possible and desired by your application. The concept of normalization has been around for 30 years and is the basis of SQL and relational databases. In other words, SQL was created to work with normalized data structures. Normalization simply means that the tables are broken down into their constituent parts until each table represents one and only one "thing", and its columns fully describe only the one "thing" that the table represents. Under-normalization will cause excessive repetition of data, over-normalization will cause excessive joins across too many tables. Both of them will get poor performance.

Physical Data Model

After the logical data model is arrived at, the physical data model is generated.

Choose appropriate data types to store data as it helps to improve query performance. Avoid nvarchar and nchar unless you need to store Unicode data.

Partition large and unused/rarely used tables to different physical storages for better query performance. For example, if you have a very large table, say, in a banking environment where the current month of data is being constantly updated and the previous months' data is being constantly reported on, then you can partition this table by month. With partitioning, maintenance operations such as index rebuilds and defragmentation can be performed on the single month of write-only data, while the read-only data is available for online access.

Create clustered and non-clustered indexes since indexes help to access data faster. But be careful, more indexes on tables will slow the INSERT, UPDATE, DELETE operations. Hence try to keep a small number of indexes on a table, not more than 4-5 typically. Choose columns with the integer data type (or its variants) for indexing, varchar column indexing will cause performance degradation.

Store image and BLOB data in separate tables or a separate database with pointers in their related tables.

Use constraints (foreign key, check, not null, etc) for data integrity. Don't give whole control to the application code. An added advantage is that such constraints can easily be removed from the database versus changing, rebuilding and redeploying an application.

Programming Best Practices

The following are some of the best practices followed by our development teams in developing SQL queries used in stored procedures.

- Avoid * in SELECT statement since SQL Server converts the * to column names before query execution. Explicitly list the columns that you need in your SELECT query.
- Use LIKE clause properly. If you need an exact match use "=" instead.
- Using JOIN will perform better than sub queries or nested queries.
- Use stored procedures. They are faster, secure and easy to maintain.
- Use the latest features of SQL Server 2012 and beyond like Paging and Column Store Indexes.
- Use EXISTS to check existence instead of IN
- Avoid HAVING clause since it acts as a filter over selected rows. HAVING clause is required if you further wish to filter the result of an aggregations. Don't use HAVING clause for any other purpose.
- Use TABLE variable in place of TEMP table since TEMP table resides in the TEMPDB database. Hence use of TEMP tables requires interaction with TEMPDB database and executes slowly.

- Use schema name before SQL object name followed by "." since it helps the SQL Server for finding that object in a specific schema. As a result improves performance.

```
a. --Here dbo is schema name
b. SELECT col1,col2 from dbo.tblName
    i. Avoid
c. SELECT col1,col2 from tblName
```

- Keep transaction as small as possible since transaction locks the processing tables' data during its life. Sometimes long transactions may result into deadlocks.
- Use TRY-CATCH for handling errors in T-SQL statements. Sometimes an error in a running transaction may cause deadlock if you have not handled error using TRY-CATCH.
- Use comments for readability as well as guidelines for the next developer who comes to modify the same code. Write SQL keyword in capital letters for readability purpose.

SQL Server Tuning

The SQL Profiler is a tool to debug, troubleshoot, monitor, and measure your applications' SQL statements and stored procedures. SQL Profiler captures activity occurring in SQL Server, driven by requests from your client application.

The tool enables you to precisely select which events you want to monitor. For example, you might want to see when stored procedures are called, when they complete, how long they take to execute, how many logical reads occur during execution, and so on. You can also filter the trace, which is particularly useful when your database is under heavy load and a large amount of trace information is generated. SQL Profiler provides a set of templates with predefined event selection criteria. The templates are designed to capture commonly required events. You can modify and extend the templates or create your own. Trace data can be displayed interactively, or it can be captured directly to a trace file or database table.

Use the trace information to identify the bottlenecks in your queries. Create newer indexes or change the logic slightly to achieve desired outcomes.

Application Considerations

A database connection could be a scarce resource when several tens of thousands of users are accessing your application. The application should use SQL Server Authentication mode with pre-defined credentials that have the least privilege to perform the functionalities of the application. This allows for connection pooling and greatly aids scalability. The application should return database connections to the pool as soon as possible.

Long running data intensive operations can be batched as nightly jobs and executed separately if an immediate response is not required from the application's perspective.

Reporting

If the application is highly transactional and serves several thousands of users, then a separate reporting solution is recommended. A separate database which aggregates data from the transactional system for reporting purposes will provide an effective solution. A nightly job can be created to update this reporting database. This methodology will free up the transactional needs of the application from the compute intensive reporting needs. The reporting database will just be slightly behind the transactional system in terms of data accuracy, but will provide business users all the pertinent information along with a great user experience.

NoSQL Considerations

NoSQL is a class of DBMS that is being used in very large data systems and big data applications. It seeks to solve the scalability and big data performance issues that relation databases weren't designed for. It is especially useful when the application needs to analyze a large amount of unstructured data or data that is stored on multiple servers in the cloud.

A very popular NoSQL database is Apache Cassandra which was Facebook's proprietary database. MongoDB, HBase and Neo4j are some other choices. There are different types of NoSQL databases like Document databases to store documents, Graph stores to store information about social connections, Wide-column stores that are optimized for queries over very large datasets and which store columns of data together instead of rows etc.

NoSQL can be a consideration if your application fits into Bigdata analysis or has to deal with massive amounts of unstructured data.

About Trigent Software Inc.

Trigent is a privately held, professional IT services company and a Microsoft Gold Partner with its U.S. headquarters in the greater Boston area and its Indian headquarters in Bangalore with development centers in Boston, Bangalore and Pune. We provide consulting services in various technologies including Microsoft Solutions. Our operating model is to conduct sales, customer relationships and front-end consulting (e.g., business case, requirements, architecture) onsite with our clients and perform the detail design, development, integration, testing and quality assurance offshore at our world class development and support center in Bangalore. We are a SEI CMM Level 4 company and is ISO 9001:2000 TickIT certified organization.

For sales contact sales@trigent.com or call 508-490-6000.

Microsoft Partner
Gold Application Development
Gold Collaboration and Content
Silver Mobility