## Stay Away from Nesting Views

A nested view exists when one view calls another view, which calls more views, and so on. This can lead to confusing code for two reasons. First, the views are masking the operations being performed. Second, the query may be simple, but the execution plan and subsequent operations by the SQL engine can be complex and expensive. This occurs because the optimizer doesn't have time to simplify the query, eliminating tables and columns it doesn't need; instead, the optimizer assumes that all tables and columns are needed. The same rule applies to nesting user-defined functions.

## Ensure No Implicit Data Type Conversions

When you create variables in a query, be sure those variables are of the same data type as the columns that they will be used to compare against. Even though SQL Server can and will convert, for example, a `VARCHAR` to a `DATE`, that implicit conversion can prevent indexes from being used. You have to be just as careful in situations like table joins so that the primary key data type of one table matches the foreign key of the table being joined. You may occasionally see a warning in the execution plan to help you with this, but you can't count on this.

## Minimize Logging Overhead

SQL Server maintains the old and new states of every atomic action (or transaction) in the transaction log to ensure database consistency and durability. This can place tremendous pressure on the log disk, often making the log disk a point of contention. Therefore, to improve database performance, you must try to optimize the transaction log overhead. In addition to the hardware solutions discussed later in the chapter, you should adopt the following query-design best practices:

- Choose table variables over temporary tables for small result sets, less than 20 to 50 rows, where possible. Remember, if the result set is not small, you can encounter serious issues. The performance benefit of table variables is explained in detail in the "Using Table Variables" section of Chapter 18.

- Batch a number of action queries in a single transaction. You must be careful when using this option because if too many rows are affected within a single transaction, the corresponding database objects will be locked for a long time, blocking all other users trying to access the objects.
- Reduce the amount of logging of certain operations by using the Bulk Logged recovery model. This rule applies primarily when dealing with large-scale data manipulation. You also will use minimal logging when Bulk Logged is enabled, and you use the `WRITE` clause of the `UPDATE` statement or drop or create indexes.

# Adopt Best Practices for Reusing Execution Plans

The best practices for optimizing the cost of plan generation can be broadly classified into these two categories:

- Caching execution plans effectively
- Minimizing recompilation of execution plans

## Caching Execution Plans Effectively

You must ensure that the execution plans for your queries are not only cached but reused often. Do so by adopting the following best practices:

- Avoid executing queries as nonparameterized, ad hoc queries. Instead, parameterize the variable parts of a query and submit the parameterized query using a stored procedure or the `sp_executesql` system stored procedure.
- If you must use lots of ad hoc queries, enable the Optimize for Ad Hoc Workload option, which will create a plan stub instead of a full plan the first time a query is called. This radically reduces the amount of procedure cache used.
- Use the same environment settings (such as `ANSI NULLS`) in every connection that executes the same parameterized queries. This is important because the execution plan for a query is dependent on the environment settings of the connection.

- As explained earlier in the "Explicitly Define the Owner of an Object" section, explicitly qualify the owner of the objects when accessing them in your queries.

The whole idea is to ensure you have only the plans that you need in the cache and that you use those plans repeatedly rather than compiling new ones all the time. The preceding aspects of plan caching are explained in detail in Chapter 17.

## Minimizing Recompilation of Execution Plans

To minimize the unnecessary generation of execution plans for queries, you must ensure that the plans in the cache are not invalidated or recompiled for reasons that are under your control. The following recommended best practices minimize the recompilation of stored procedure plans:

- Do not interleave DDL and DML statements in your stored procedures. You should put all the DDL statements at the top of the stored procedures.
- In a stored procedure, avoid using temporary tables that are created outside the stored procedure.
- Prefer table variables over temporary tables for small data sets.
- Do not change the `ANSI SET` options within a stored procedure.
- If you really can't avoid a recompilation, then identify the stored procedure statement that is causing the recompilation, and execute it through the `sp_executesql` system stored procedure.

The causes of stored procedure recompilation and the recommended solutions are explained in detail in Chapter 18.

# Adopt Best Practices for Database Transactions

The more effectively you design your queries for concurrency, the faster the queries will be able to complete without blocking one another. Consider the following recommendations while designing the transactions in your queries:

- Keep the scope of the transactions as short as possible. In a transaction, include only the statements that must be committed together for data consistency.

- Prevent the possibility of transactions being left open because of poor error-handling routines or application logic. Do so using the following techniques:
  - Use `SET XACTABORT ON` to ensure that a transaction is aborted or rolled back on an error condition within the transaction.
  - After executing a stored procedure or a batch of queries containing a transaction from a client code, always check for an open transaction and then roll back any open transactions using the following SQL statement:

    ```
    IF @@TRANCOUNT > 0 ROLLBACK
    ```

- Use the lowest level of transaction isolation required to maintain data consistency as determined by your application requirements. The amount of isolation provided by the Read Committed isolation level, the default isolation level, is sufficient most of the time. If excessive locking is occurring, consider using the Read Committed Snapshot isolation level.

The impact of transactions on database performance is explained in detail in Chapter 20.

## Eliminate or Reduce the Overhead of Database Cursors

Since SQL Server is designed to work with sets of data, processing multiple rows using DML statements is generally much faster than processing the rows one by one using database cursors. If you find yourself using lots of cursors, reexamine the logic to see whether there are ways you can eliminate the cursors. If you must use a database cursor, then use the database cursor with the least overhead: the `FAST_FORWARD` cursor type (generally referred to as the *fast-forward-only cursor)*. You can also use the equivalent `DataReader` object in ADO.NET.

The performance overhead of database cursors is explained in detail in Chapter 23.

## Use Natively Compile Stored Procedures

In situations where you're accessing only in-memory tables, you have one additional performance enhancement open to you, which is to compile your stored procedures into a DLL that runs within the SQL Server executable. As was shown in Chapter 24, this has fairly radical performance implications. Just be sure that you call the procedures in the correct fashion passing parameters by ordinal position rather than by parameter name. Although this feels like you're breaking a best practice, it leads to better performance of the compiled procedure.

## Take Advantage of Query Store for Analytical Queries

Most applications using relational databases to store their information have some degree of analytical queries. Either you have an OLTP system with a few analytical queries or you have a data warehouse or reporting system with lots of analytical queries. Take advantage of columnstore indexes in support of the queries that do lots of aggregation and analysis. A clustered columnstore is best when the majority of the queries are analytical but doesn't work as well for OLTP point look up style of query. The nonclustered columnstore index adds analysis when the majority of queries are OLTP focused, but some of them need to do analysis. In this case, it's all about picking the right tool for the job.

# Summary

Performance optimization is an ongoing process. It requires continual attention to database and query characteristics that affect performance. The goal of this chapter was to provide you with a checklist of these characteristics to serve as a quick and easy reference during the development and maintenance phases of your database applications.

# Index

## A






G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2

## B

## C

## D

## E

## F

## G



## H



## I

## J



## K



## L

## M

## N



## O



## P