**Reducing the Number of Network Round-Trips**

**To reduce the overhead of multiple network round-trips, consider the following techniques:**

**Execute multiple queries together.**

**Use SET NOCOUNT.**

**Execute Multiple Queries Together**

**It is preferable to submit all the queries of a set together as a batch or a stored procedure. Besides reducing the network round-trips between the database application and the server, stored procedures also provide multiple performance and administrative benefits. This means that the code in the application needs to be able to deal with multiple result sets. It also means your T-SQL code may need to deal with XML data or other large sets of data, not single-row inserts or updates.**

**Use SET NOCOUNT**

**You need to consider one more factor when executing a batch or a stored procedure. After every query in the batch or the stored procedure is executed, the server reports the number of rows affected.**

***(<Number> row(s) affected)***

**This information is returned to the database application and adds to the network overhead. Use the T-SQL statement SET NOCOUNT to avoid this overhead.**

SET NOCOUNT ON

<SQL queries>

SET NOCOUNT OFF

**Reducing the Transaction Cost**

**Every action query in SQL Server is performed as an atomic action so that the state of a database table moves from one consistent state to another. SQL Server does this automatically and it can’t be disabled. If the transition from one consistent state to another requires multiple database queries, then atomicity across the multiple queries should be maintained using explicitly defined database transactions. The old and new state of every**

**atomic action is maintained in the transaction log (on the disk) to ensure durability, which guarantees that the outcome of an atomic action won’t be lost once it completes successfully. An atomic action during its execution is isolated from other database actions using database locks.**

**Based on the characteristics of a transaction, here are two broad recommendations to reduce the cost of the transaction:**

**• Reduce logging overhead.**

**• Reduce lock overhead**.

**Reduce Logging Overhead**

**A database query may consist of multiple data manipulation queries. If atomicity is maintained for each query separately, then a large number of disk writes are performed on the transaction log. Since disk activity is extremely slow compared to memory or CPU activity, the excessive disk activity can increase the execution time of the database functionality. For example, consider the following batch query:**

--Create a test table

IF (SELECT OBJECT\_ID('dbo.Test1')

) IS NOT NULL

DROP TABLE dbo.Test1 ;

GO

CREATE TABLE dbo.Test1 (C1 TINYINT) ;

GO

--Insert 10000 rows

DECLARE @Count INT = 1 ;

WHILE @Count <= 10000

BEGIN

INSERT INTO dbo.Test1

(C1)

VALUES (@Count % 256) ;

SET @Count = @Count + 1 ;

END

**Since every execution of the INSERT statement is atomic in itself, SQL Server will write to the transaction log for every execution of the INSERT statement. An easy way to reduce the number of log disk writes is to include the action queries within an explicit**

**transaction.**

**The defined transaction scope (between the BEGIN TRANSACTION and COMMIT pair of commands) expands the scope of atomicity to the multiple INSERT statements included within the transaction. This decreases the number of log disk writes and improves the performance of the database functionality. To test this theory, run the following T-SQL command before and after each of the WHILE loops:**

DBCC SQLPERF(LOGSPACE);

**This will show you the percentage of log space used. On running the first set of inserts on my database, the log went from 2.6 percent used to 29 percent. When running the second set of inserts, the log grew about 6 percent.**

**The best way is to work with sets of data rather than individual rows. A WHILE loop can be an inherently costly operation, like a cursor . So, running a query that avoids the WHILE loop and instead works from a set-based approach is even better.**

DECLARE @Count INT = 1 ;

BEGIN TRANSACTION

WHILE @Count <= 10000

BEGIN

INSERT INTO dbo.Test1

(C1)

VALUES (@Count % 256) ;

SET @Count = @Count + 1 ;

END

COMMIT

**Running this query with the DBCC SQLPERF() function before and after showed less than 4 percent growth of the used space within the log, and it ran in 41 ms as compared to more than 2 seconds for the WHILE loop. One area of caution, however, is that by including too many data manipulation queries within a transaction, the duration of the transaction is increased. During that time, all other queries trying to access the resources referred to in the transaction are blocked.**

**Reduce Lock Overhead**

**By default, all four SQL statements (SELECT, INSERT, UPDATE, and DELETE) use database locks to isolate their work from that of other SQL statements. This lock management adds a performance overhead to the query. The performance of a query can be improved by requesting fewer locks. By extension, the performance of other queries is also improved because they have to wait a shorter period of time to obtain their own locks.**

**By default, SQL Server can provide row-level locks. For a query working on a large number of rows, requesting a row lock on all the individual rows adds a significant overhead to the lock-management process. You can reduce this lock overhead by decreasing the lock granularity, say to the *page level or table level*. SQL Server performs the lock escalation dynamically by taking into consideration the lock overheads. Therefore, generally, it is not necessary to manually escalate the lock level. But, if required, you can control the concurrency of a query programmatically using lock hints as follows:**

SELECT \* FROM <TableName> WITH(PAGLOCK) --Use page level lock

**Similarly, by default, SQL Server uses locks for SELECT statements besides those for INSERT, UPDATE, and DELETE statements. This allows the SELECT statements to read data that isn’t being modified. In some cases, the data may be quite static, and it doesn’t go through much modification. In such cases, you can reduce the lock overhead of the SELECT statements in one of the following ways:**

**• Mark the database as READONLY.**

ALTER DATABASE SET READONLY

**This allows users to retrieve data from the database, but it prevents them from modifying the data. The setting takes effect immediately. If occasional modifications to the database are required, then it may be temporarily converted to READWRITE mode.**

ALTER DATABASE SET READ\_WRITE

<Database modifications>

ALTER DATABASE SET READONLY

**• Place the specific tables on a filegroup, and mark the filegroup as READONLY.**

**--Add a new filegroup with a file to the database.**

ALTER DATABASE AdventureWorks2008R2

ADD FILEGROUP READONLYFILEGROUP ;

GO

ALTER DATABASE AdventureWorks2008R2

ADD FILE(NAME=ReadOnlyFile, FILENAME='E:\Data\adw\_l.ndf')

TO FILEGROUP READONLYFILEGROUP ;

GO

--Create specific table(s) on the new filegroup.

CREATE TABLE Tl (Cl INT, C2 INT)

ON READONLY FILEGROUP ;

CREATE CLUSTERED INDEX II ON Tl(Cl);

INSERT INTO Tl

VALUES (1, 1);

--Or move existing table(s) to the new filegroup

CREATE CLUSTERED INDEX II ON Tl(Cl)

WITH DROP\_EXISTING ON READONLYFILEGROUP ;

--Set the filegroup property to READONLY.

ALTER DATABASE AdventureWorks2008R2

MODIFY FILEGROUP READONLYFILEGROUP READONLY

**This allows you to limit the data access to only the tables residing on the specific filegroup**

**to READONLY but keep the data access to tables on other filegroups as READWRITE**.

**This filegroup setting takes effect immediately. If occasional modifications to the specific**

**tables are required, then the property of the corresponding filegroup may be temporarily**

**converted to READWRITE mode.**

ALTER DATABASE AdventureWorks2008R2

MODIFY FILEGROUP READONLYFILEGROUP READWRITE <Database modifications> ALTER

DATABASE AdventureWorks2008

MODIFY FILEGROUP READONLYFILEGROUP READONLY

**Use one of the snapshot isolations.**

**SQL Server provides a mechanism to put versions of data into tempdb as updates are**

**occurring, radically reducing locking overhead and blocking for read operations. You can**

**change the isolation level of the database by using an ALTER statement.**

ALTER DATABASE AdventureWorks2008R2 SET TRANSACTION ISOLATION LEVEL READ\_

COMMITTED\_SNAPSHOT;

• **Prevent SELECT statements from requesting any lock**.

SELECT \* FROM <TableName> WITH(NOLOCK)

**This prevents the SELECT statement from requesting any lock, and it is applicable**

**to SELECT statements only. Although the NOLOCK hint can’t be used directly on the tables referred to in the action queries (INSERT, UPDATE, and DELETE), it may be used on the data-retrieval part of the action queries, as shown here:**

DELETE Sales.SalesOrderDetail

FROM Sales.SalesOrderDetail sod WITH(NOLOCK)

JOIN Production.Product p WITH(NOLOCK)

ON sod.ProductID = p.ProductID

AND p.ProductID = 0

**This leads to dirty reads, which can cause duplicate rows or missing rows and is therefore considered to be a last resort to control locking. The best approach is to mark the database as read-only or use one of the snapshot isolation levels.**