

Exploration on Scalability of Database Bulk Insertion with Multi-threading

Boon-Wee Low, Boon-Yaik Ooi, and Chee-Siang Wong
Faculty of Information and Communication Technology, Department of Computer Science, Universiti Tunku Abdul Rahman, Jalan Universiti, Bandar Barat, 31900
Kampar, Perak, Malaysia.

{lowbw,ooiby,worgcs}@utar.edu.my

ABSTRACT

The advancement of database engine and multi-core processors technologies have enable database insertion to be implemented concurrently via multithreading programming. The objective of this work is to evaluate the performance of using multithreading technique to perform database insertion of large data set with known size to enhance the performance of data access layer (DAL) particularly on the bulk-insertion operation. The performance evaluation includes techniques such as using single database connection, multithreads the insertion process with respective database connections, single threaded bulk insertion and multithreaded bulk insertion. MySQL 5.2 and SQL Server 2008 were used and the experimental results show that the performance of databases do not scale linearly with the number of threads. This work justifies our intention in developing a smart data access layer that autonomously decides the number of threads to spawn in order to improve the performance of the DAL.

KEYWORDS

Database bulk insertion, multicore processor, multi-threading, data access layer and database technologies.

1 INTRODUCTION

With the fall in price of multi-core processors, it has made high-performance processors affordable to all. The steady increase of the number of cores in microprocessors has enabled parallel processing to be applied in systems such as enterprise resource planning (ERP) and customer relationship management (CRM) systems. For instance, the latest Intel Core i7 Gulftown microprocessor [1, 2] offers up to 12 logical cores when simultaneous multi-threading is enabled. Unfortunately, this increase of processing resources can only be utilized by a program if multi-threading or multi-processing techniques are used. One of such server-oriented application that may utilize multi-threading techniques is database system [3, 4]. Although, past research has shown that multi-threading is capable of improving the speed of database insertion, this has spark the question of how many threads would help in improving the database insertion performance. The scalability of such improvement with respect to various data sizes offers intriguing insight into providing overall improvement in database performance.

In this paper, we explore the scalability of performance improvement with respect to the size of the dataset, available cores and insertion techniques

such as bulk insertion. The relationships of the CPU (Central Processing Unit), the RAM (Random Access Memory), the I/O (Input/Output) transfer rate of system storages, and the performance of database bulk insertion are studied as well. The main contribution of this paper is to evaluate which insertion methods offer the best performance that suits different database bulk insertion environment.

The remaining parts of the paper are organized as follows: Section 2 discusses the related work in improving the performance of database bulk insertion by using multi-threading techniques and a short description of DAL. Section 3 details the methodology of the research, which includes experimental setup, threading methods used, and the systems utilization. The outcome of the evaluations is presented and discussed in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

The conventional way of inserting data into a database is by using the sequential SQL Insert method. In order to perform data insertion in bulk, database vendors have developed specific methods so that data can be inserted at a better rate. However, current DAL (Data Access Layer) is single-threaded and no attempts of multi-threading the DAL have been made so far as the authors are able to identify.

Previously, other research has shown that multi-threading can improve the performance of database insertion. This has set the trend of utilizing thread level parallelism and performance scalability in modern software development [3]. Previous works related to parallel database systems have also been studied. Özsu and Valduriez [19] introduced distributed and parallel Database

Management System (DBMS) that enables natural growth and expansion of database on simple machines. Parallel DBMSs are one of the most realistic ways working towards meeting the performance requirements of application which demands significant throughput on the DBMS.

DeWitt and Gray [17] shows that parallel processing is a cheap and fast way to significantly gain performance in database system. Software techniques such as data partitioning, dataflow, and intra-operator parallelism are needed to be employed to have an easy migration to parallel processing. The availability of fast processors and inexpensive disk packages is an ideal platform for parallel database systems.

According to Valduriez [18], parallel database system is the way forward into making full use of multiprocessor architectures using software-oriented solutions. This method promises high-performance, high-availability and extensibility power price compared to mainframes servers. Parallelism is the most efficient solution into supporting huge databases on a single machine. In a research to speedup database performance, Haggander and Lundberg [16] shows that by multi-threading the database application it would increase the performance by 4.4 times than of a single threaded engine. This research was done to support a fraud detection application which requires high performance read and write processes. Therefore they found that the process would be speed up by increasing the number of simultaneous request. Zhou et al. [15] shows that there is moderate performance increase when database is being multithreaded. He evaluated its performance,

implementation complexity, and other measures and provides a guideline on how to make use of various threading method. From the experiment results, multi-threading improves the database performance by 30% to 70% over single-threaded implementation. In this research, it is also found that Naïve parallelism is the easiest to implement. However, it only gives a modest performance improvement.

In 2009, Ryan Johnson shows that by increasing the number of concurrent threads it would also increase the normalized throughput of data into a database. But there is a limit on how many concurrent threads can be used. As the number of threads used gone pass the optimal figure, it will suffer from performance deterioration due to extra overheads initiated from additional context switching as a consequence of spawning excessive number of threads. The experiment was done based on different database engines; which are Postgres, MySQL, Shore and BDB. It can be concluded that different database engine has its respective optimal number of threads. The optimal number depends on how the database was being developed. This comes to show that a detailed study on different database system is required to get the best out of each database system. The research concludes that multi-threading does help in improving database insertion speed. The paper also discovers the bottlenecks that hamper the scalability. It is overcome by introducing Shore-MT, a multi-threaded version of Shore database engine which shows excellent scalability and great performance when compared to other database engines [3].

Reference [4] uses of .NET 4 Framework to parallelize database

access. From the test results, they have shown a significant increase of performance when there is a large amount of data. In contrast, there is only a slight increase of performance when the data size is small. The performance increases as much as 80.5% when it deals with a large amount of data. The experiment was done by inserting an amount of data into the database in parallel, and then retrieving the data from the database and storing it into an XML file. In this approach, multiple connections and threads access the database in parallel and all is controlled by the .NET 4 Framework.

From all previous research, we can see great potential in multi-threading database systems. It is proven that by parallelizing the system, it would have a moderate to significant gain in performance at a lower cost. Therefore with the right threading method and insertion method, we are able to improve database performance. This proves the potential in multi-threading database systems.

2.1 Data Access Layer (DAL)

DAL is a set of classes with functions for reading and writing to a database or other data storage medium. It does not contain any business logic for the application or user interface element and only. It's a background worker that interacts between the application and the database as a background worker. It i's a part of a multi-layer application design that would normally include items as shown in as below [20] and figure 1-2Fig. 1 shows where the DAL sits in an application [20].

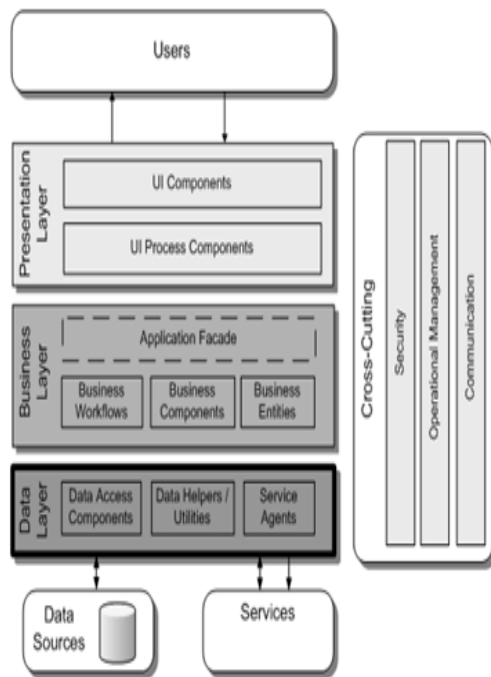


Fig. 1. A typical application showing where the DAL and other components [20].

- A User Interface Layer (UI) which contains screens and user interface components.
- A Business Logic Layer (BLL) which contains the business rules for the application.
- Data access logic components are abstract logic that are needed to access underlying data stores. Doing so would then centralize the data access function. It would then help make the application easier to configure and maintain.
- Data helpers/utilities are functions and utilities assist in data transformation and data access in the layer. It contains specialized API and routines that is designed to improve data access efficiency and reduce the need to develop logic components and service agents in the layer.
- Service agent is where business component uses functions that are exposed by external services. The service agent isolates your application from the idiosyncrasies of calling diverse services and additional services can be provided.

3 Methodology

This work focuses only on the INSERT operation of a DAL. The DAL will be multithreaded with multiple connections to the database engine. We evaluate the multithreaded performance of two different database systems, Microsoft SQL Server 2008 Enterprise and MY SQL 5.1 by Oracle to search for the most effective number of threads used depending on the available resources and data set size. All the evaluations done in this work were conducted on a same machine. We observe the performance of various insertion methods with multithreading implementation.

3.1 Environment Settings of the Experiment

The machine used in this work is a computer comprised of an Intel Core 2 Quad Q9400 2.66 GHz with 3.93 GB of RAM with Windows XP Professional Service Pack 3. The hard-disk used in this test bed is a 320GB Seagate Barracuda with rotational speed of 7200rpm (revolution per minute), and is capable of performing 78 MB/s data transfer rate [5]. At start the machine consumes 438 MB of RAM and 0% CPU utilization.

All the test programs were developed on .NET 4.0 Framework using C# via Visual Studio 2010. MySQL Connector .NET 6.2.4 adapter is being used to execute the InsertLoader for MySQL database. The test data consists of strings with 302 random characters each. It comprises alphabets, numeric and symbols. These strings are stored in flat file format and the file size ranges from 1 to 80,000 rows. The same set of files is used for the entire experiment. The database consists of one table with two

columns. The first column is an auto-increment numeric counter which is set to integer and the second column is to store the rows from the flat file which is set to VARCHAR(MAX). This is applied to both databases that are evaluated in this paper.

3.2 The Overview of the Experiment Process

The process begins with the data reading process. The data reading is done by using a single thread and subsequently distributes the data into multiple files with the same amount of rows. The reader writes the file into either flat file format or XML format depending on the requirement of the database engine.

Table 1. Evaluation on various insertion methods and the number of threads used respectively.

Test Number	Insertion Method	Number of Threads
1	Sequential SQL Insertion (SQL Server 2008 & MySQL 5.2)	1
		2
		3
		4
2	Import Loader	1
		2
		3
		4
3	Bulk Copy	1
		2
		3
		4

3.2.1 Threading Method

Throughout the experiments, threads are manually spawned in order to maintain a controlled environment. The codes below show how the program is being threaded where two threads are used.

```
//create and start threads
ThreadStart threadDelOne = new
ThreadStart
(insOne.RunInsertion);
ThreadStart threadDelTwo = new
```

```
ThreadStart
(insTwo.RunInsertion);

Thread threadOne = new
Thread(threadDelOne);
Thread threadTwo = new
Thread(threadDelTwo);
threadOne.Start();
threadTwo.Start();

//thread join
threadOne.Join();
threadTwo.Join();
```

3.2.2 sequential insertion

Sequential insertion is done by using the standard SQL insert command and each command would insert one row. Transaction is being used in this process where the whole block will be committed after the last data is inserted. Rollback is being used if there is an error [6]. The same code is being used for 1, 2, 4 and 8 threads. Before inserting the row, it is being formatted into compatible SQL command by replacing certain characters to work with SQL command formatting. For sequential, the test is done with and without transaction. The following is the code.

```
for (int i = 0; i <
dataList.Count; i++) {
    sqlStr = "INSERT INTO
TestTbl(DataCol) VALUES (N"+d
ataList[i]+")";

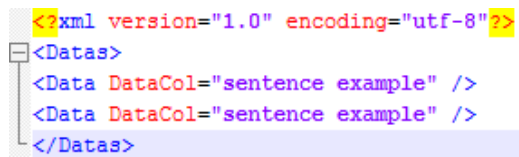
    sqlCmd = new
SqlCommand(sqlStr, conn,
transaction);

    sqlCmd.ExecuteNonQuery();
}

transaction.Commit();
```

3.2.3 SQLBulkCopy Insertion

SQLBulkCopy is a .NET4 function to insert data in bulks into SQL Server 2008 [8]. It receives XML's and inserts them. The format for the XML file is as shown in Fig. 1. This method is tested by using 1, 2, 4, and 8 threads; where the number of individual XML's are read according to the number of threads used respectively. For example, if 8 threads are used, they will read from 8 individual XML's.



```
<?xml version="1.0" encoding="utf-8"?>
<Datas>
  <Data DataCol="sentence example" />
  <Data DataCol="sentence example" />
</Datas>
```

Fig. 2. An example of XML formatting.

The codes below show how the SQLBulkCopy insertion is being done.

```
dataSet dataSet = new DataSet();
dataSet.ReadXml(xmlFileName);
sourceData = dataSet.Tables[0];

using (SqlConnection conn = new
SqlConnection(connStr)) {

    conn.Open();

    using (SqlBulkCopy bulkCopy
= new
SqlBulkCopy(conn.ConnectionString)) {

        bulkCopy.ColumnMappings.Add
("DataCol", "DataCol");
        bulkCopy.DestinationTableNa
me = "TestTbl";
        bulkCopy.WriteToServer(sour
ceData);

    }conn.Close();

}
```

3.2.4 MySQL Bulk Loader

Insertion

MySQL Bulk Loader from the MySQL .NET Connector 6.2.4 [7] is used for this experiment.

It receives flat files and inserts them using the MySQL import loader. The number of files created would depend on the number of threads used. This method is being tested with 1, 2, 4, and 8 threads. The following codes illustrate how the import loader is performed.

```
//perform import loader

try {

    MySqlBulkLoader myBulk
= new
MySqlBulkLoader(conn);
myBulk.Timeout = 600;
myBulk.TableName =
"testDatabase.testTbl"
;
myBulk.Local = true;

myBulk.FileName =
fileName;
myBulk.FieldTerminator
= "";
myBulk.Load();

}
```

3.2.5 System

Utilization

In the next experiment, the RAM, CPU and hard disk drive utilization are captured throughout the insertion period. This is done during 70,000 to 80,000 rows on all the insertion methods, number of threads and database engines as shown in Table 1. A sample is captured every 30 seconds and the average from the samples would be taken as the result [9]. Codes below illustrates the system utilization is being captured.

```
PerformanceCounter
cpuUsage = new
PerformanceCounter("Proce
```

```

ssor", "% Processor
Time", "_Total", true);

PerformanceCounter
memoryAvailable = new
PerformanceCounter("Memor
y", "Available MBytes");

PerformanceCounter
physicalDiskTransfer =
new
PerformanceCounter("Physi
calDisk", "Disk
Bytes/sec", "_Total",
true);

startMemory =
totalMemoryCapacity -
memoryAvailable.NextValue
();
    
```

4 Experimental Results

The test data ranges from 1 to 80,000 rows and we capture the elapsed time taken to insert the data. The same test data are being used on both database engines, SQL Server 2008 and MySQL 5.2, with the method discussed in Section 3.

4.1 SQL Server 2008

From the experiment, we confirmed that multi-threading has a significant improvement. At 50,000 rows, performance increase as much as 67% using multithreaded insertion method. Code with transaction increases the performance by 24%. Therefore multi-threading and transaction proves to improve the database insertion performance. Fig. 3 shows the performance increase between 1 and 8 threads using sequential insertion. But when the data size is small, the overhead of spawning the threads is too costly and the performance would deteriorate. In contrast with that, it implies that it is

best not to multithread the database insertion when the data size is small. This is reflected in Fig. 4.

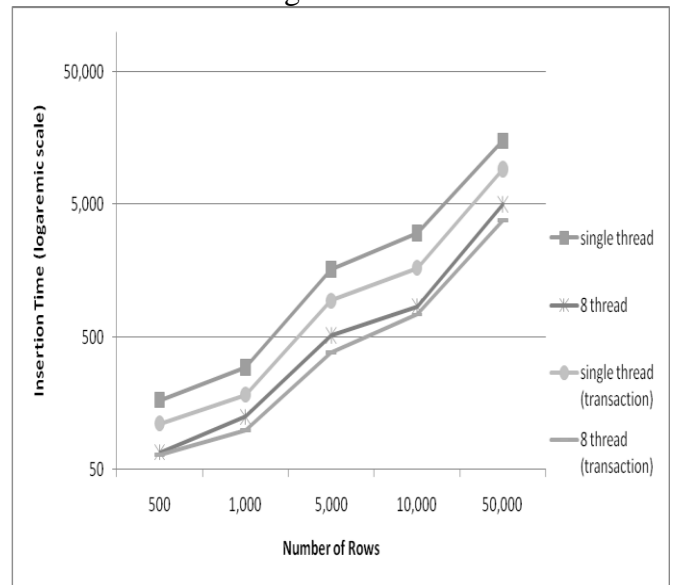


Fig. 3. Sequential insertion with and without transaction used.

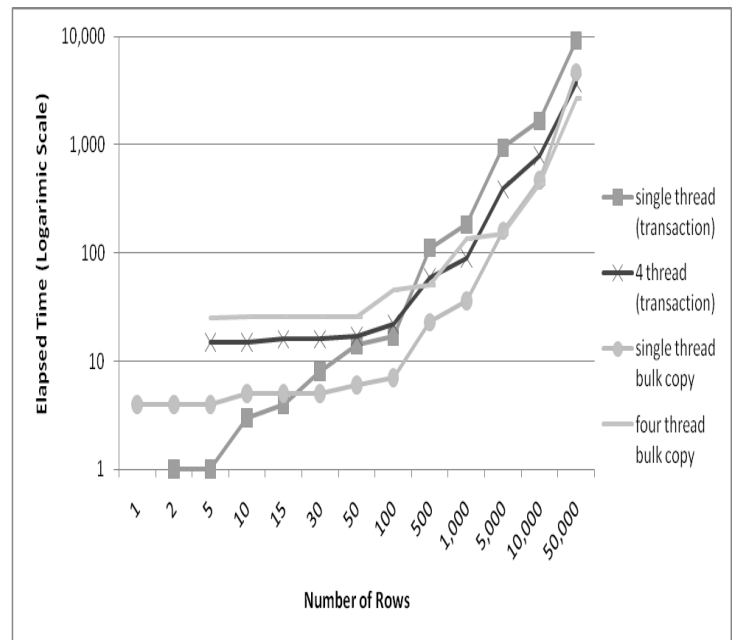


Fig. 4. Comparison between BulkCopy and sequential insertion with transaction.

From Fig. 4, the result showed that single threaded bulk copy can outperform the multithreaded insertion with transaction enabled. As the data grow larger, bulk copy is becoming more efficient. However, the performance of bulk copy can be further improved by using multiple threads. Fig. 5 shows the performance of BulkCopy compared to sequential insertion with a large data size. At 80,000 rows, the performance increase by 43% when threaded with eight threads.

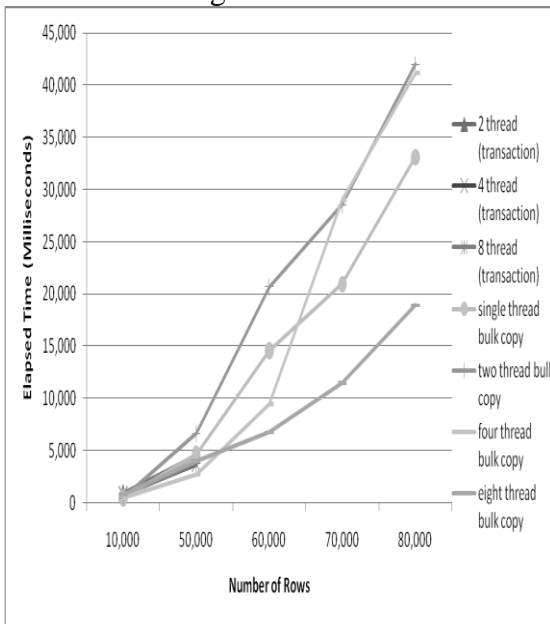


Fig. 5. Comparison between BulkCopy and sequential insertion.

The following figure shows the overall performance differences between having transaction and bulk copy on SQL server 2008. It shows that the performance of multithreaded database insertions do not scale linearly. Different dataset, database insertion method and number of threads will yield different insertion performance.

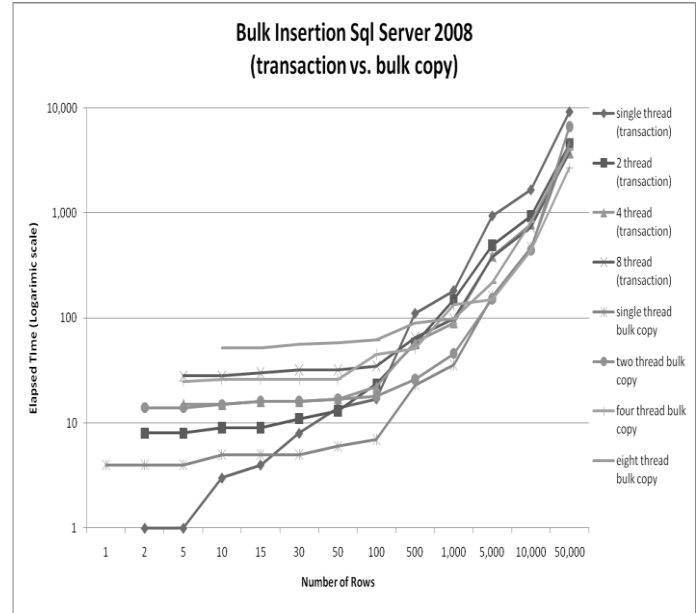


Fig. 6. Comparison between Transaction and BulkCopy insertion.

From the experiment, we observed that the performance of the insertion methods is dependent on the data size. The following table is the detail of the observation.

Table 2. SQL Server 2008 Insertion method for specific data size range.

Number of Rows	Threading Method
1 to 29	Single threaded sequential insertion with transaction
30 to 5,000	Single threaded BulkCopy
5,001 to 50,000	Four threads BulkCopy
50,000 to 80,000	Eight thread BulkCopy

4.2 MySQL 5.2

On the other hand, MySQL does not have significant performance improvement when being threaded. MySQL boost sequential insertion

performance by 99.5% when transaction code is being used. To insert 50,000 rows without transaction, it requires approximately 22 minutes compared to 6.3 seconds with transaction. Fig. 7 shows the sequential insertion performance.

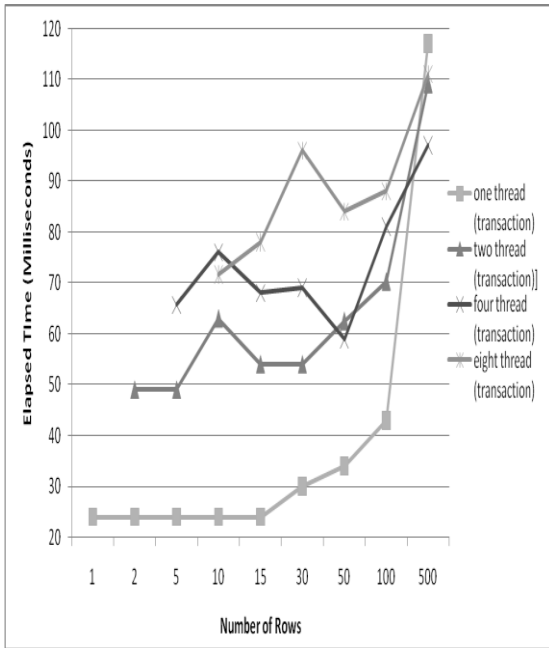


Fig. 7. Comparison between different numbers of threads using sequential insertion.

From Fig. 8, the multi-threading works well when data size is in the range of 501 to 5,000, in these range four threads improve the insertion performance by 42.5%. However, the performance plunges by 49% when it is being spawned with eight threads compared to single threaded for data size with 100 rows. Even with insert loader, single threaded still performs best. When insert loader is being spawned with eight threads, insertion performance plummet by 65.4% compared to single threaded at 80,000 rows. Insert loader performs best

when the data size is large and single threaded.

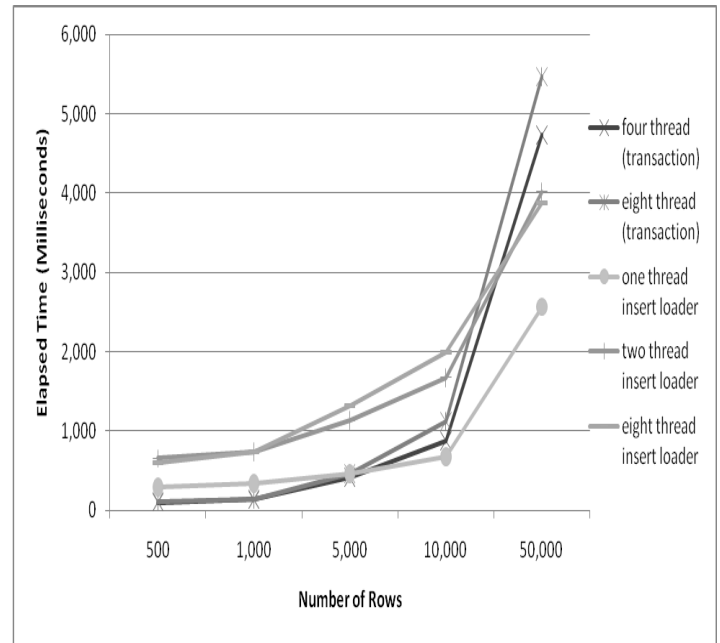


Fig. 8. Comparison between sequential insertion and insert loader.

The following figure shows the overall performance differences between having transaction and insert loader on MySQL 5.2. Similar to observation made in fig. 6, It shows that the performance of multithreaded database insertions do not scale linearly. Besides, different dataset, database insertion method and number of threads will yield different insertion performance; different database engines will have different overall behavior.

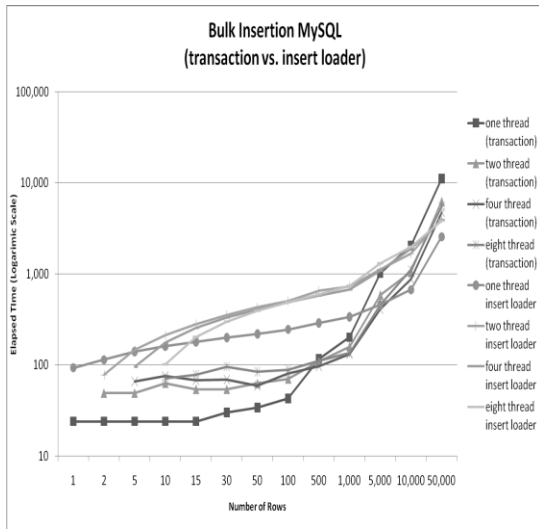


Fig. 9. Comparison between transaction and insert loader.

Therefore, from the experimental results the following is the observation made by from our experiment using MySQL 5.2. It is similar to MS SQL Server 2008 that the performance of the insertion methods is dependent on the data size. However, the insertion method varies.

Table 3. MySQL 5.2 Insertion method for specific data size range.

Number of Rows	Threading Method
1 to 500	Single threaded sequential insertion with transaction
501 to 5,000	Four Threads sequential insertion with transaction
5,001 and 50000	Single threaded insert loader

4.3 System Utilization

Besides that, we observe the system utilization when the insertion process is executed. Observations are made only with samples sizes ranging from 70,000 to 80,000 rows as they show the most significant system utilization.

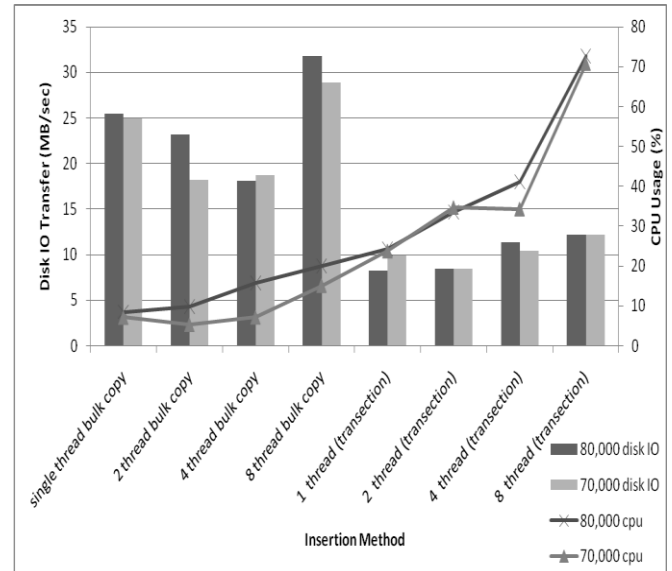


Fig. 10. System utilization between Disk I/O and CPU usage for SQL Server 2008.

Fig. 10 shows the system utilization of SQL Server 2008, we found that as the number of threads increase the overall machine utilization increase as well. In general, bulk copy has high IO traces while normal SQL insertion with transaction has high CPU traces relatively. Similar observation was seen with MySQL 5.2 in the following figure.

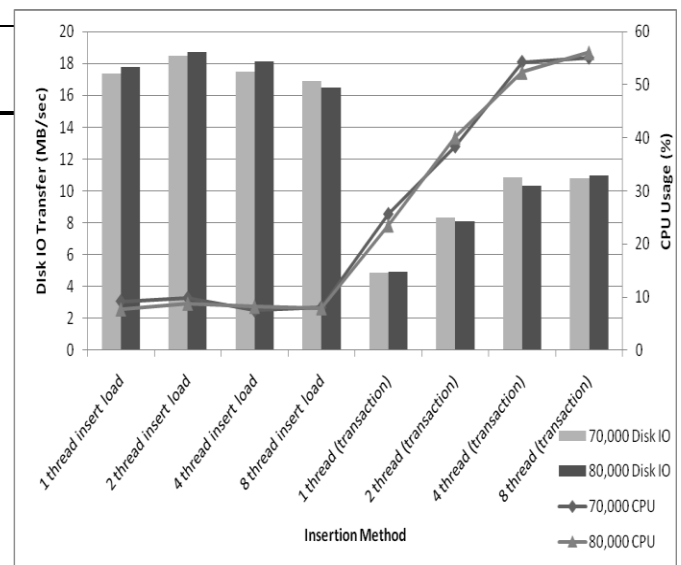


Fig. 11. System utilization between Disk I/O and CPU usage for MySQL 5.2.

5 Conclusion and Future Work

Although the advancement of multi-core processors is encouraging multithreaded application to be developed, we found that the performance of the insertion function of a database does not necessarily improve proportionately with the number of threads used. Multithreading did improve the performance of both of the databases' insertion function but the speed up is very dependent on the underlying architecture of the database system. Database architecture, RAM, CPU, and type of HDD do have an effect on each other. Different database engine react differently toward different approach into doing bulk insertion. Therefore, this work suggests that software developers should investigate the performance of multithreaded operations on databases before designing any system. From the experimental results, it shows that database threading must be carried carefully as it would have adverse effect if the database threading is overly done, it will cause a major drop in performance. Besides that, this work justify our intention to develop a smart DAL which capable of automatically select the most efficient threading methods depending on the type of hardware, available resource and the size of data set.

References

1. Intel® Core™ i7 Processor Extreme Edition, <http://www.intel.com/products/processor/corei7EE/index.html>
2. Intel® Core™ i7 – 920 Desktop Processor Series Product Specifications, <http://ark.intel.com/Product.aspx?id=37147>
3. Johnson, R., Ippokratis, P., Hardavellas, N., Ailamaki, A., Falsafi, B., Shore-MT: A Scalable Storage Manager for the Multicore Era. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, pp. 24--35, ACM, New York (2009)
4. Verenker, A., Using .NET4 Parallel Programming Model to Achieve Data Parallelism in Multi-tier Applications, MSIT, Microsoft Corporation (2010)
5. Seagate Barracude 7200.10 SATA 3.0Gb/s 320-GB Hard Drive, <http://www.seagate.com/ww/v/index.jsp?vgnextoid=2d1099f4fa74c010VgnVCM100000dd04090aRCRD>
6. TransactionScope Class, <http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx>
7. Using the Bulk Loader, <http://dev.mysql.com/doc/refman/5.1/en/connector-net-programming-bulk-loader.html>
8. SqlBulkCopy Class, <http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlbulkcopy.aspx>
9. Performance Counter Constructor, <http://msdn.microsoft.com/en-us/library/xx7e9t8e.aspx>
10. Bunn, J.J., Holtman, K., Newman, H.B., Object Database Scalability for Scientific Workloads. Technical report, California Institute of Technology (2000)
11. Thread Class, <http://msdn.microsoft.com/en-us/library/system.threading.thread.aspx>
12. Lui, D., Wang, S., Analysis of Database Workloads on Modern Processors. In: Proceedings of the 1st SIGMOD PhD Workshop on Innovation Database Research 2007, pp. 63--68, ACM, New York (2007)
13. Performance Monitoring, <http://www.csharphelp.com/2006/05/performance-monitoring/>

14. How to Create and Terminate Thread
(C# Programming Guide),
[http://msdn.microsoft.com/en-US/library/7a2f3ay4\(v=VS.80\).aspx](http://msdn.microsoft.com/en-US/library/7a2f3ay4(v=VS.80).aspx)
15. Zhou, J., Cieslewicz, J., Ross, K.A., Shah, M., Improving Database Performance on Simultaneous Multithreading Processors. In: Proceedings of the 31st International Conference on Very Large Data Bases 05', pp. 49--60, VLDB Endowment, Norway (2005)
16. Haggander, D., Lundberg, L., Multiprocessor Performance Evaluation of a Telecommunication Fraud Detection Application. In: ARTES Graduate Student Conference, Sweden (1999)
17. DeWitt, D., and Gray, J., Parallel Database Systems: The Future of High Performance Database Processing. Commun. ACM, 35, 85--98 (1992)
18. Valduriez, P., Parallel Database Systems: Open Problems and New Issues. J. Distributed and Parallel Databases. 1, 137--165 (1993)
19. Özsu, M.T., Valduriez, P., Distributed and Parallel Database System. J. ACM Computing Surveys. 28, 125--128 (1991)
20. Meier, J.D., et. al., Chapter 12: Data Access Layer Guidelines. Microsoft Pattern & Practice:
<http://apparchguide.codeplex.com/wikipage?title=Chapter%2012%20-%20Data%20Access%20Layer%20Guidelines>