Data Processing in a Database Management System Using Parallel Processing

Stephen Shears
sws30@uakron.edu

Please take a moment to share how this work helps you through this survey. Your feedback will be important as we plan further development of our repository.

Recommended Citation
https://ideaexchange.uakron.edu/honors_research_projects/1472

This Dissertation/Thesis is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Williams Honors College, Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.
Data Processing in a Database Management System Using Parallel Processing

Stephen Shears
Spring 2022

Honors Sponsor: Dr. Yincai Xiao
Honors Reader 1: Dr. Tim O’Neil
Honors Reader 2 and Faculty Advisor: Dr. Zhong-Hui Duan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td><strong>CHAPTERS</strong></td>
<td></td>
</tr>
<tr>
<td>I: INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II: DESIGN</td>
<td>4</td>
</tr>
<tr>
<td>III: IMPLEMENTATION</td>
<td>6</td>
</tr>
<tr>
<td>IV: EXPERIMENT</td>
<td>10</td>
</tr>
<tr>
<td>V: RESULTS</td>
<td>13</td>
</tr>
<tr>
<td>Insert Query Results</td>
<td>13</td>
</tr>
<tr>
<td>Select Top Query Results</td>
<td>16</td>
</tr>
<tr>
<td>Delete Top Query Results</td>
<td>18</td>
</tr>
<tr>
<td>VI: CONCLUSION</td>
<td>21</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>23</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Design Schema for movies table</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>PhpMyAdmin movies table</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Microsoft SQL Server Management Studio movies table</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Insert query creator program</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Insert query runtime results in milliseconds</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>Select top query runtime results in milliseconds</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>Delete top query runtime results in milliseconds</td>
<td>18</td>
</tr>
</tbody>
</table>
This research project will be focused on parallel processing as it is used with database management systems to process data. Specifically, the goal is to see if creating a database management system with parallel processing at the forefront of its data processing can offer enough of an efficiency increase to warrant using it against a sequential database management system and is it possible to make that system just as reliable as those databases without parallel processing. A parallel processed database will be created with a focus on monitoring its data reliability and consistency. It will then be compared to two sequential databases to compare their performance and determine the effectiveness of the parallel processed database.

This project has resulted in both the creation of a sequential database and a parallel processed database. Numerous tests have been run on both, and while it may be very beneficial to allow for a database to have parallel capabilities, the efficiency and speed up that a parallel database may have over a sequential do not warrant the use and creation of it with the technologies used at this time.
CHAPTER I

INTRODUCTION

A computer database is a completely computerized database that holds data. This data is often separated into tables. What will specifically be used within this project is what is known as a relational database. These databases are created with the idea in mind that each piece of data within a particular table are a series of related data. These tables often have a unique identifier to differentiate each entry in the table, and all the data within that entry can be found from that unique identifier otherwise known as a key. The data that is contained within a table can vary greatly, as do the strategies that can be used to create them. Whenever an action is taken on a database, this action is known as a query that is sent to the database to complete an action. These queries can range from inserts, searches, and deletes to name a few.

Parallel processing is the practice of using multiple processors or cores within a processor to complete a job or program. This is typically done by splitting up the data that must be processed into separate sections and giving each processor a different section. This must be done very carefully however, as these jobs can run into different problems, such as a race condition, where both processors are trying to access the same piece of data at the same time, and the one that gets it first may not be the one that should have it first. When this happens, it can cause whatever program is using parallelism within it to have unpredictable results. Safeguards must be put in place, and these
safeguards can cause the parallelized code to perform worse than if the code did not have it.

This is where the question of how efficient a parallel database can be was originally formed. Most databases are sequential databases, meaning that all the work done for each query is done in a sequential order and each individual query is done one after the other without any parallelism. Now this does not mean that a parallel database does multiple queries at the same time, and this is because it would be a very bad idea to do so. Suppose there is a database, and two queries are made at the same time. One query will insert a new item into a table within the database, and another will list out the data items within that database. If the insert is incomplete while the other query is run, then the new item will not be listed in, but if the insert completes before the list, then the new item will appear. It will be entirely up to chance which one of these outcomes will occur if both are attempted at the same time. Unpredictable results are not a sustainable practice, since no one is able to know what will occur when the queries are ran because they are in the race condition that was discussed previously. Instead, a parallel database is designed to split single queries apart and allow for each processor on the host machine to perform each part of the query that it was given. Theoretically this cuts the time to perform each query immensely since each processor is working at the same time as each other.

This project was intended to find a definitive answer to whether it was worth development time to create a parallel database over a sequential database. During this project two databases were created along with three separate tables across two different languages. Two of these tables were created to be entirely sequential based. One was
written in MySQL, a strictly sequential language that would display the runtimes for a
database that was optimized for sequential queries. The other was written in MSSQL, a
language that allows for both optimized parallel processing, and sequential queries, and
this table was created as a control for the last table which was created for specifically four
processor processing. This was to account for any quirks of the MSSQL language that
could have arisen during testing and to compare each tables’ data reliability and
consistency between a query with parallel processing, and one without within the same
language framework.
As was stated within the introduction, this project was created with three separate databases with each one having one table. Each table was a relatively simple table, that had five elements. To keep the idea that these are relational databases, a schema was created first that would show the design and flow of these tables. The exact schema diagram used is shown in figure 1. Take note that the name of the table is in the square at the center, and all attributes are contained within circles. The underlined attribute is the key to this table. This includes the film’s title, description, budget, and runtime. The fifth attribute is movieID, which serves as the key within the table for each film. All three tables that were created had the same schema as to make sure that they were as similar as possible.

Figure 1: Design Schema for movies table
This simple design for the table allowed for the project to be able to focus specifically on the performance on each query, while having a variety of datatypes and simulating what would be inside a possible real database. These datatypes are as follows, movieID is an int, title is a string, description is a string, runtime is an int, and budget is also an int. Implementing the key as well allowed for the quickest optimized select queries and better organized tables.

Besides the tables, there was one other element of the project that had to be designed separately. This other element was a separate program that was designed to create the queries that would be used during the project, such as a large number of inserts with unique data and create queries that are wrote in different languages. Exact implementation of this program will be gone into depth within the next chapter, but it was designed with ease of use and responsiveness at its forefront, since it would be used many times to create all the queries that were required during the project.
CHAPTER III

IMPLEMENTATION

As stated previously, two main languages were used to facilitate the implementation of the three tables. To start, MySQL was the first database to be created, and the one I had the most experience with going into this project. It is an optimized sequential query language that was essential in the testing process. It was very important that this project simulate real databases as close as possible to make the results applicable to real scenarios. To this end, these databases were hosted on virtual servers. To set up the server that would host the MySQL database on, I made use of WampServer, and PhpMyAdmin.

WampServer is an application that can be used to set up a virtual server on a local machine. It performs this through the machine’s localhost and allows the database to be accessed through a browser, in this project’s case, Microsoft Edge. As for PhpMyAdmin, it served as a direct line to the database. I was able to perform all the necessary queries through here, and the software offered the runtime of each query, which was essential to my research. Figure 2 shows an example of PhpMyAdmin.
For the parallel database, MSSQL was chosen as the language to be used, and it was created shortly after the MySQL database. The reason for using MSSQL was because it offered a lot of freedom when it comes to how to optimize queries, and whether to use a sequential system to process data or a parallel system to process data, a system that not very many SQL languages offer. Another benefit of using this language over another parallel language, was the amount of information that it could give you when interacting with a server that is ran with Microsoft SQL Server (MSS) in conjunction with Azure, and Microsoft SQL Server Management Studio (MSSMS) running as the interface to interact with the MSS and perform all the queries and tasks required of MSSMS to work with databases.

MSSMS offers a lot of tools for a developer to understand the exact performance of the server, but the one most focused on during this project was the client statistics options within the studio. It not only gave all the information on each individual connection being performed while executing a query, but it also had the exact execution and processing time that a query spent while it executes the query on the database. This studio is also where a lot of parallel processing options in MSSQL can be found. The sequential table was set up to specify that only one process could be running at a time, causing all queries to be
completely sequential, while the parallel table was set to make use of four processes and processors at a time, forcing it to always be processing data in a parallel manner. An example of the MSSMS interface can be seen in figure 3.

![Figure 3: Microsoft SQL Server Management Studio movies table](image)

The final piece that had to be created before this project could begin the testing phase, was that there had to be a way to create a series of test queries that would allow for large numbers of queries to be performed. The initial idea was to create a separate php page that would connect to each database and run the queries while outputting the runtimes for each query performed. However, this led to a myriad of issues, including incorrect runtimes that were caused by a delay in the page loading in the Edge browser. Instead, I decided to run the queries within both PhpMyAdmin, and MSSMS since they had accurate runtimes, and a better interface.

This created another issue however, since some of the queries were relatively easy to write out each time I needed to test in the database, such as the select and delete queries, but the insert queries were another matter. I needed to be able to insert up to 16,000 unique
pieces of data into the database with queries. This led to the creation of a rather simple C++ program, that would prompt the user if they would like to create a new insert query, and the number of movies they would like to insert. This new query would be placed into a text file, which then could be read into each database to be completed as the query. Upon completion, the program would output three files with the number of new movies specified. One file written in MySQL, and two files written in MSSQL, one for the sequential table, and one for the parallel table. A snippet of the code and program running can be found in figure 4.

```cpp
while(loop){
  std::cout << "Please enter whether you would like to write to mysql and mssql(1) or to quit(2)\n";
  std::cin >> choice;
  if(choice == "1"){
    bool valid = false;
    while(valid){
      std::cout << "Please enter the number of inserts you would like to input: ";
      std::cin >> N;
      if (std::cin.good()){
        valid = true;
      } else {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
        std::cout << "Invalid input; please re-enter.\n";
      }
    }
    filename = "testmysql";
    filename.append(std::to_string(N));
    std::ofstream datafile(filename);
    if(datafile.is_open()){
      datafile << "INSERT INTO movies (movieID, title, budget, description, runtime) VALUES\n";
      for(i = 1; i <= N; i++){
        if(count == 99) && i == (N-1)|
          datafile << (NULL, "movieID" << i << ",", "30000", "Test description", 90);\n;|
        datafile << ("movieID", "title", "budget", "description", "runtime") VALUES\n";
        count = 0;
      } else if(i == (N-1)){
        datafile << (NULL, "movieID" << i << ",", "30000", "Test description", 90);\n;|
      } else {
        datafile << ("movieID", "title", "budget", "description", "runtime") VALUES\n";
        count++;
      }
    }
  }
}
```

Figure 4: Insert query creator program
CHAPTER IV

EXPERIMENT

Once every database and program that would be needed was created, the next step was to run tests on the database that could measure the runtime of each query. With the runtimes listed out in a table, the efficiency and speed up or slowdown of running a parallel database against a sequential database could be found. These queries also had to be something that could done on real databases, as to simulate real situations. This also had to include variable large numbers, which would also give the scalability of each database and how it handles queries at small and large numbers. To this end, three different series of tests were ran on the three databases, and the efficiency and speed up was found for each.

The first series of tests were a large series of inserts into each database. The number of inserts into each table went as follows, 1,000, 2,000, 4,000, 8,000, and 16,000. With each query having this large number of movies being inserted into each database, this caused some issues with the MySQL database. With PhpMyAdmin, there is a limit to the amount of new data that could be inserted into the database within a query. This limit was 1,000 and could not be changed. To counter this, I separated each insert into multiple queries, with each having about 992 movies within the insert, for all three databases. This meant that there were two queries in the first test, three queries in the second test, five queries in the third test, nine queries in the fourth test, and seventeen queries in the final test. The reason for this choice was to make sure that even though MSSQL could run
more than 1,000 pieces of data within an insert at a time, that the process was the same for each database. This change is reflected in the insert query creation program and is made relatively simple because of it.

The second series of tests were a lot simpler than the requirements to make the insert program working and perform the insert queries. This series of tests involve the select top function within both MySQL and MSSQL, with only slight differences in syntax, with MySQL making use of the limit keyword. The test was to select the top 1,000, 2,000, 4,000, 8,000, and 16,000 movies within the database and display them. However, when I performed the first two queries on the MySQL server, the database was very slow to perform the operation, even though the actual processing time was not long at all. Then the final three queries could not be completed at all, with the server simply saying an error had occurred. I believe this error was caused by a lack of system memory being allocated to PhpMyAdmin and my browser, causing the query to try to squeeze all of the data from the table to be displayed into that memory. The memory would completely fill up, and then the query would time out and display the error that had occurred. Despite this failure in the PhpMyAdmin, MSSMS was able to perform these queries flawlessly, and this test also gave the opportunity to look over each input in the database and make sure that the data was reliable from the massive inserts as a secondary function. The results of these secondary findings will be discussed in the following chapter.

The final series of tests made use of the delete top function. Much like the last test, this test would select the top 1,000, 2,000, 4,000, 8,000, and 16,000 movies in the database, but it would delete those movies from the database in bulk. When performing
these queries on the MSSMS they went through flawlessly and quickly, with no failures and no issues on either table. Unlike the previous test, PhpMyAdmin also experienced no failures and was executed rather quickly.

After collecting this data in each server’s runtime, the question then becomes how to compare and gather useful information to the question of whether it would be worth it to implement a parallel database over a sequential database. The answer to this is to find the speed up and efficiency for using the parallel database over the sequential ones. The speed up can be found by dividing the runtime of the sequential query by the runtime of the parallel query. As for the efficiency, this can be found by dividing the speed up metric by the degree of parallelism that the parallel database used, which in this case is always four. Each of these concepts will yield a solid metric to look at when comparing each database and will reveal the actual benefit of using a parallel database over a sequential database.
CHAPTER V

RESULTS

The results of each test that was run on the three databases was separated into three different excel tables. This chapter will first look at the individual results of each table’s query runtimes. Then it will discuss the efficiency and speed up of the parallel table compared to each sequential database.

Insert Query Results

<p>| Insert Query Runtimes and Efficiency Results |  |</p>
<table>
<thead>
<tr>
<th>MySQL</th>
<th>4 Core</th>
<th>MSSQL</th>
<th>1 Core</th>
<th>MSSQL</th>
<th>MySQL Speed up</th>
<th>MySQL Efficiency</th>
<th>MSSQL Speed up</th>
<th>MSSQL Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.123</td>
<td>0.128</td>
<td>0.141</td>
<td>0.961</td>
<td>0.24</td>
<td>1.102</td>
<td>0.275</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>0.177</td>
<td>0.266</td>
<td>0.47</td>
<td>0.665</td>
<td>0.166</td>
<td>1.808</td>
<td>0.452</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>0.34</td>
<td>0.454</td>
<td>0.47</td>
<td>0.749</td>
<td>0.187</td>
<td>1.035</td>
<td>0.259</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>0.617</td>
<td>0.257</td>
<td>0.846</td>
<td>2.401</td>
<td>0.6</td>
<td>3.292</td>
<td>0.823</td>
<td></td>
</tr>
<tr>
<td>16000</td>
<td>1.27</td>
<td>0.438</td>
<td>1.71</td>
<td>2.9</td>
<td>0.725</td>
<td>3.904</td>
<td>0.976</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Insert query runtime results in milliseconds

When it comes to the runtime of MySQL, its runtime is rather linear. Each runtime is about double the runtime of the previous, as would be expected in a process
that is sequentially inserting data into the database. It is also ran very quickly for the first two small inputs of data. This trend will continue throughout the other tables, with the MySQL database, where the first two are very fast, and then the other queries will run in linear time.

With the sequential MSSQL database, an interesting anomaly occurs. Across the runtimes, they all run slower than the MySQL server, but the runtimes for 2,000 and 4,000 inserts were the same. I decided to test both a few more times to see if there was just some sort of slow down for the 2,000-insert query, but each time, they both ran at about 0.47 of a millisecond. Other than this oddity, there is little more to note about the runtimes for this set of tests.

The parallel MSSQL database for the first query runs very similar to the sequential database in MSSQL, and slightly slower than MySQL for the first three queries. However, it seems that once the number of inserts reached 8,000 and beyond, the performance of the parallel database went up dramatically. For both 8,000 and 16,000, it was able to achieve very small runtimes, even smaller than the runtimes of the same database for smaller inputs of data. It also outpaced both other sequential runtimes for 8,000, and 16,000. With this series of query tests in mind, this database is not very scalable. It gets a very big performance boost at large numbers, but for smaller queries, it does not run as well.

When the parallel database is compared with the MySQL database in terms of speed up and efficiency, it is not as beneficial as one would expect out of using parallelism as opposed to the sequential database. The average speed up of the parallel database is about 1.535 and its average efficiency is only about 0.384. What this data
means is that compared to MySQL, using the parallel database offers very little speed up and does not make full use of the four cores it was given compared to the sequential database. Once it is considered that the first three smaller queries had better performance on the MySQL server, it could potentially be detrimental to use the parallel database over the MySQL server in terms of performance.

As opposed to the sequential MySQL server, the parallel MSSQL server always had a better runtime than the sequential MSSQL server. The average speed up in this case was about 2.228, and the average efficiency was about 0.557. While these numbers may not seem very different from the previous comparison, they are actually very substantially different. These results mean that the parallel database is on average performing a little over double the speed of the sequential database but is still operating at only about half of the efficiency of the sequential database.

These results seem to be indicative of the lack of good scalability for the parallel database that was described earlier. At higher numbers of data being inserted, the parallel database was making full use of its available cores, at its highest point being 0.976 efficiency when being compared to the sequential MSSQL server. At its lowest efficiency it was at 0.259, however, which is not a substantial enough efficiency and speed up boost to warrant a preference for the parallel database.
As was stated in the previous chapter, this was the series of tests that had completely failed in the MySQL server past selecting the top 2,000 elements. Again, I believe this was due to an issue with available system memory that was allocated to the server that could not be changed. From what can be gathered from the first two results however, it ran the exact same runtime as the other two databases in the first query, and it ran faster than the other two queries in the second query. As was the case in the last set of query tests, this database ran its queries in about linear time.

Fortunately, the MSSQL server did not suffer the same issues as the MySQL server, and as such was able to complete the series of tests without issue. The runtimes of the sequential database seemed to be about on par with the runtimes for the parallel database. They seemed to go back and forth on which one ran faster. Much like the
MySQL server, these two databases also run in about linear time. This would make it appear as though these types of queries when executed on a database do not experience a very massive impact on performance when it is being run on a parallel database, or it is being run on a sequential database.

This would imply that this set of queries is implicitly sequential, as each database must gather the same amount of data by traversing the database. Both the speed up and efficiency of the parallel MSSQL database and the sequential database suggest this is the case as well. The average speed up for this comparison was at 1.014, and the efficiency of this comparison was about 0.254. With these two metrics in mind, both the parallel and sequential MSSQL databases have virtually the same performance, and runtime, when it completes this particular set of queries.

Once this data was found, two surveys over the parallel and sequential databases were also completed to check for data consistency and protection, as to make sure no data was lost during the inserting process for the parallel database. The first survey confirmed that both databases were identical and contained the exact same data in the same order. This meant that, at the least, the parallel database was just as consistent as the sequential database when it came to data consistency, and how MSSQL processes large inserts of data. The next survey was the comparison between the parallel database, and the original queries that were performed on it. Just like the previous survey, it yielded that both the original queries and the database were consistent, with no missing data. With this knowledge, it is safe to say that the parallel database, is just as consistent and can protect data just as well as a sequential database written in MSSQL.
As to whether this makes it seem better to use a parallel database over a sequential database, this series of tests further proves that a parallel database does is not a better option to the sequential database. There is next to no performance difference between the two different databases, and in some cases the parallelism hurt the performance of the query. In the few cases that the MySQL was able to run the tests, it ran faster the parallel database entirely, even with the memory issues as seen in its inability to fully finish this series of tests.

**Delete Top Query Results**

<table>
<thead>
<tr>
<th>MySQL Speed up</th>
<th>MSSQL Speed up</th>
<th>MySQL Efficiency</th>
<th>MSSQL Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.339</td>
<td>0.403</td>
<td>1.335</td>
<td>0.101</td>
</tr>
<tr>
<td>5.83</td>
<td>0.553</td>
<td>1.457</td>
<td>0.138</td>
</tr>
<tr>
<td>6.55</td>
<td>0.406</td>
<td>1.638</td>
<td>0.102</td>
</tr>
<tr>
<td>7.535</td>
<td>0.566</td>
<td>1.884</td>
<td>0.141</td>
</tr>
<tr>
<td>3.427</td>
<td>0.345</td>
<td>0.857</td>
<td>0.086</td>
</tr>
</tbody>
</table>

Figure 7: *Delete top query runtime results in milliseconds*

After the previous tests failures for large select top values, I had feared that the MySQL server would fail the same way again for the final set of tests, with the delete top queries. However, it was able to fully complete these series of tests, but at quite the runtime cost. It ran the slowest out of the three databases and had the slowest set of
runtimes for the entire series of tests. It appears that MySQL is not properly optimized for
the delete top functionality in its servers, and as such takes the longest to perform the
query.

Where the MySQL server fails is where the MSSQL succeeds. The sequential
database performs the delete top query extremely fast. The runtime for each query is
about on par with the other series of tests. It also has the shortest runtime at each query
amount when compared to the other two databases. Like the previous series of tests, this
seems to indicate that the delete top function is also implicitly sequential, since it must go
through the set number of elements within the database to delete from the database.

The parallel MSSQL server runs the second fastest out of the three databases.
Unlike the previous series of tests however, it does run much slower than its sequential
counterpart, where before it was either faster, or about the same runtime. As was stated
previously, this seems to be because the query is implicitly sequential. The reason it has
casted the database to be much slower this time is because the work required to make
each processor work together takes more time then just processing the data sequentially.

If just comparing the MySQL server and the parallel MSSQL, in this case, the
parallel database is very much worth using over the sequential database. The average
speed up being about 5.736 and the average efficiency being about 1.434. However,
when considering the likelihood of the MySQL server having an issue with optimizing
this query, and the sequential MSSQL database also having faster runtimes then the
MySQL server, this data does not determine much in the way of determining the benefits
of using the parallel database. The comparison between the sequential and parallel
MSSQL databases is, however. The average speedup was about 0.455 and the average
efficiency was at about 0.114. In this case, using parallelism was simply detrimental to
the database performing each query that was required of it.

In the case of the final test, using a parallel database would certainly not be
preferred. Parallelism actively made the query have a worse runtime then if it was simply
using a sequential method. It also was not making good use of the four processors, since
it had such a low efficiency compared to the sequential database. Now that the three
series of tests have been completed, the next chapter will discuss the final verdict on
using a parallel database over a sequential database.
CHAPTER VI

CONCLUSION

Over the course of this project, a lot has been accomplished. Three databases were created, two using MSSQL, and one using MySQL. An in-depth analysis was performed of how each database handles queries, and the performance of each. Comparisons of each were drawn, as were the weaknesses and strengths of each. With all this information, the original questions that sparked this project can now be answered. Does the benefit of having a parallel database outweigh its drawbacks, and can a parallel database be just as consistent and protect its data as much as a sequential database?

To the second question the answer was yes. Through testing it was shown that the parallel database created in this project was just as consistent as a sequential database. It was also shown that the parallel database was able to protect the data within its queries as well. As for whether a parallel database should be used over a sequential database, the answer is no. Through testing it was shown that a lot of database operations are typically implicitly sequential and using a parallel database either minimally assisted in speeding up the process of executing the queries as was the case in both the insert and select top series of query tests, or it actively slowed it down, in the case of the delete top series of query tests. Overall, it took a lot of work to get the parallel database working within MSSMS, and for this work there were very minimal effects on the performance of the database when compared to the sequential database.
While this project may have not exactly resulted in exactly what I expected from the parallel database, it has been a very insightful experience. I have learned a great deal about operating MSSMS to manage a server, and a lot of the more advanced features within MSSQL to perform the queries and operations necessary for parallel processing. It fascinated me a lot more than I expected it to with its intricacies, and in the future, I plan to continue working on personal projects using this system and learning as much about it as I can. I can also begin experimenting with other parallel technologies, such as PLINQ, a database query system designed by Microsoft to allow for more intricate control over parallel processed data within a database.
BIBLIOGRAPHY


