Journal, Advances in Mathematical & Computational Sciences
Vol. 8 No. 1, March 2020
www.mathematics-computationaljournal.info

Article Citation Format
Obiora, Desmond; Ochei, L. C., Wobidi, E. (2020): Modelling the Performance of Interoperable Service-Oriented Systems. Journal of Advances in Mathematical & Computational Sc.
Vol. 8, No. 1. Pp 97-108

Article Progression Time Stamps

Article Type: Research Article
Manuscript Received9th Jan, 2020
Final Acceptance: 21st March, 2020

Query Performance of Binary Data Object (BLOB) in Database Transactions

[1]Desmond, Obiora, [2]Ochei, L. C., [3]Wobidi, E.
Department of Computer Science
University of Port Harcourt
Port Harcourt, Nigeria.
E-mail: [1]desmondoboira@gmail.com, **[2]**laud.ochei@gmail.com, [3]echebs62@yahoo.com

ABSTRACT

It is common to encounter BLOB data (e.g., images, audio, video, etc.) in applications that save data in a database or perform data reading from a database. Testing the performance of BLOB data is important to avoid degradation in database performance. It is usually assumed that databases efficiently handle large numbers of small objects, whereas filesystems are more efficient with large objects. This paper aims to evaluate the performance of saving BLOB data in a relational database when the number of users and iterations increases. A Java program was written to save BLOB data multiple times in the database. Thereafter, the custom method of the Java program was imported into JMeter to test the performance operation. Results of extensive evaluation show that BLOB data can impact heavily on the database when multiple users save BLOB data repetitively in the database especially in a situation where the workload is unpredictable (i.e., sudden increase) and aggressive.

Keywords: Query, Performance, Binary Large Data Object, Database, Transactions, Testing

1. INTRODUCTION

Query performance is one of the most important aspects while designing a database for enterprise systems. At a high level, query performance can be defined as the rate at which a query returns information from database/repository to users. The goal of query performance is to minimize the response time of your queries by making the best use of the resources of the system (IBM, 2021). There is always a tremendous amount of load on the database server as numerous users will be accessing the database server simultaneously through various queries. The time it takes for a query response plays a vital role in the functioning of an enterprise and taking strategic decisions. Databases for storing business data should be designed such that the response time of the query is minimized and at the same time load on the server is reduced (Domdouzis, 2021).

The query operations used by the application consist of select, insert, update, and delete. Each operation is run by DBMS then the result is transferred to the business application. All data in DBMS are stored in a file on a physical device such as a disk device. Assume one file consists of many records and the user want to retrieve a single record based on a particular criterion, the disk device can go directly in the middle of a file to retrieve the record. To accomplish that, the system needs time to move the disk device and retrieve the requested record using many procedures compilation in DBMS. To reduce the time needed by the system, the DBMS must be tuned according to the executed query.

It is common to encounter BLOB data (e.g., images, audio, video, etc.) in applications that save data in a database or perform data reading from a database. Testing the performance of BLOB data is important to avoid degradation in database performance. Storing BLOB data in a database can make it easier to utilize existing DBMS functions to access the database. However, when the database grows bigger over a short time, managing the tables gets slower as the dataset grows.

It is usually assumed that databases efficiently handle large numbers of small objects, whereas filesystems are more efficient with large objects. This paper aims to evaluate the performance of saving BLOB data in a relational database when the number of users and iterations increases.

The main contributions of the paper are:

(a) An approach for simulating the process of repeatedly saving BLOB data into a database using the JMeter testing tool.
(b) An algorithm for evaluating the performance of querying BLOB data.
(c) Measuring the performance of saving BLOB data under the two conditions:
    (i) increasing number of users
    (ii) increasing the number of iterations

JMeter was configured to simulate the process of multiple users repeatedly saving BLOB data into a database. Results of extensive evaluation show that BLOB data can impact heavily on the database when multiple users save BLOB data repetitively in the database. The rest of the paper is organized as follows: Section 2 discusses Binary Large Object, query processing and techniques for improving query performance, Section 3 is the evaluation and describes the experimental setup and procedures. Section 4 is the results and discussion. Section 5 concludes the paper with future work.

2. QUERY PERFORMANCE OF BINARY DATA OBJECT (BLOB) IN DATABASE TRANSACTIONS

In this section, we present the binary large object (BLOB) and techniques for improving query performance

2.1 Binary Large Object

A Binary Large Object (BLOB) is a collection of binary data stored as a single entity in a database management system. Blobs are typically images, audio or other multimedia objects, though sometimes binary executable code is stored as a blob. (Patnaik, K. K., & Nagarjun, B., 2011). Database support for blobs is not universal. Blobs were originally just big amorphous chunks of data invented by Jim Starkey at DEC, who describes them as "the thing that ate Cincinnati, Cleveland, or whatever" from "the 1958 Steve McQueen movie", referring to The Blob.

Later, Terry McKiever, a marketing person for Apollo, felt that it needed to be an acronym and invented the backronym Basic Large Object. Then Informix invented an alternative backronym, Binary Large Object.

The data type and definition were introduced to describe data not originally defined in traditional computer database systems, particularly because it was too large to store practically at the time the field of database systems was first being defined in the 1970s and 1980s. The data type became practical when disk space became cheap. This definition gained popularity with IBM's DB2. The term is used in NoSQL databases, especially in Key-value store databases such as Redis. The name "blob" is further borrowed by the deep learning software Caffe to represent multi-dimensional arrays. In the world of free and open-source software, the term is also borrowed to refer to proprietary device drivers, which are distributed without their source code, exclusively through binary code; in such use, the term binary blob is common, even though the first letter in the blob abbreviation already stands for binary.

2.2 The Query Processor

There are three phases that a query passes through during the DBMS' processing of that query: parsing and translation, optimization, and evaluation. Most queries submitted to a DBMS are in a high-level language such as SQL. During the parsing and translation stage, the human readable form of the query is translated into forms usable by the DBMS. These can be in the forms of a relational algebra expression, query tree and query graph. Consider the following SQL query:

> **select** *make*
> **from** *vehicles*
> **where** *make* = "Ford"

This can be translated into either of the following relational algebra expressions:

> *make* = "Ford" (*make (vehicles))*
> *make (make* = "Ford" *(vehicles))*

After parsing and translation into a relational algebra expression, the query is then transformed into a form, usually a query tree or graph that can be handled by the optimization engine. The optimization engine then performs various analyses on the query data, generating several valid evaluation plans. From there, it determines the most appropriate evaluation plan to execute. After the evaluation plan has been selected, it is passed into the DMBS' query-execution engine (also referred to as the runtime database processor), where the plan is executed, and the results are returned.

2.2.1 Parsing and Translating the Query

The first step in processing a query submitted to a DBMS is to convert the query into a form usable by the query processing engine. High-level query languages such as SQL represent a query as a string, or sequence, of characters. Certain sequences of characters represent various types of tokens such as keywords, operators, operands, literal strings, etc. Like all languages, there are rules (syntax and grammar) that govern how the tokens can be combined into understandable (i.e., valid) statements. The primary job of the parser is to extract the tokens from the raw string of characters and translate them into the corresponding internal data elements (i.e. relational algebra operations and operands) and structures (i.e. query tree, query graph). The last job of the parser is to verify the validity and syntax of the original query string.

2.2.2 Optimizing the Query

In this stage, the query processor applies rules to the internal data structures of the query to transform these structures into equivalent, but more efficient representations. The rules can be based upon mathematical models of the relational algebra expression and tree (heuristics), upon cost estimates of different algorithms applied to operations or upon the semantics within the query and the relations it involves. Selecting the proper rules to apply, when to apply them and how they are applied is the function of the query optimization engine.

2.2.3 Evaluating the Query

The final step in processing a query is the evaluation phase. The best evaluation plan candidate generated by the optimization engine is selected and then executed. Note that there can exist multiple methods of executing a query. Besides processing a query in a simple sequential manner, some of a query's individual operations can be processed in parallel—either as independent processes or as interdependent pipelines of processes or threads. Regardless of the method chosen, the actual results should be same.

2.3 Database Transaction

A transaction can be defined as a group of tasks. A single task is the minimum processing unit that cannot be divided further. Consider a simple transaction that involves a bank employee transferring £1000 from account X to Account Y. This transaction is comprised of at least five(tasks) different low-level tasks as shown in Table 1. The tasks are shown in line numbers.

	Account X	Account Y
1	Open_Account(X)	Open_Account(Y)
2	Old_Balance = X.balance	Old_Balance = B.balance
3	New_Balance = Old_Balance - 1000	New_Balance = Old_Balance + 1000
4	X.balance = New_Balance	Y.balance = New_Balance
5	Close_Account(X)	Close_Account(Y)

A transaction is a unit of work that results in either success or failure and fulfils Atomic, Consistent, Isolated, and Durable (ACID) principles. The ACID principles ensure that there is accuracy, completeness, and data integrity.

Atomicity

This property states that a transaction must be treated as an atomic unit, which means that either all or none of its operations must be executed. In a database, there must be no state where a transaction is left partially completed.

Consistency

After any transaction, the database must remain consistent. No transaction should have an adverse effect on the database's data.

Isolation

The property of isolation states that in a database system where multiple transactions are being executed concurrently and in parallel, all transactions will be carried out and executed as if they were the only transaction in the system. No transaction will have an impact on the existence of another transaction.

Durability
The database should be durable to retain all its most recent updates even if the system fails or restarts. If a transaction commits after updating a chunk of data in a database, the database will retain the modified data. If a transaction commits but the system fails before the data can be written to disk, the data will be updated when the system restarts.

2.4 Techniques for Improving Query Performance
Queries Performance technique is extremely general and applies to almost all database systems and as such to most generic systems. In this section, we consider precomputation, data compression, surrogate processing, bit vector filters, and specialized hardware. Recently proposed techniques that have not been fully developed are not discussed here, e.g., “racing” equivalent plans and terminating the ones that seem not competitive after some small amount of time.

2.4.1 Computational and Derived Data technique
It is trivial to answer a query for which the answer is already known—therefore, pre-computation of the frequently requested information is an obvious idea. The problem with keeping preprocessed information in addition to base data is that it is redundant and must be invalidated or maintained on updates to the base data. Pre-computation and derived data such as relational views are duals. Thus, concepts and algorithms designed for one will typically work well for the other. The main difference is the database user’s view: computed data are typically used after a query optimizer has determined that they can be used to answer a user query against the base data, while derived data are known to the user and can be queried without regard to the fact that they must be derived at run-time from stored base data. Since data are likely to be referenced and requested by users and application programs, computation of derived data has been investigated both for relational and object-oriented data models.

Indices are the simplest form of computed data since they are redundant and, in a sense, computed selection. They represent a compromise between a non-redundant database and one with complex pre-computed data because they can be maintained relatively efficiently. The next more sophisticated form of pre-computation is inversions as provided in System Rs “Oth” prototype [Chamberlain et al. 1981a], view indices as analyzed by Roussopoulos ([1991), two-relation join indices as proposed by Valduriez [1987], or domain indices as used in the ANDA project (called VALTREE there) (Deshpande and Van Gucht 1988) in which all occurrences of one domain (e.g., part number) are indexed together, and each index entry contains a relation identification with each record identifier. With join or domain indices, join queries can be answered very fast, typically faster than using multiple single-relation indices.

2.4.2 Data Compression technique
Several researchers have investigated the effect of compression on database systems and their performance (Graefe and Shapiro 1991; Lynch and Brownrigg 1981; Ruth and Keutzer 1972; Severance 1983). There are two types of compression in database systems. First, the amount of redundancy can be reduced by prefix and suffix truncation, in particular, in indices, and by use of encoding tables (e.g., the colour combination “9” means “red car with black interior”). Second, compression schemes can be applied to attribute values, e.g., adaptive Huffman coding or Ziv-Lempel methods (Bell et al. 1989; Lelewer and Hirschberg 1987). This type of compression can be exploited most effectively in database query processing if all attributes of the same domain use the same encoding, e.g., the “Part-No” attributes of data sets representing parts, orders, shipments, etc. because common encodings permit comparisons without decompression. Most obviously, compression can reduce the amount of disk space required for a given data set. Disk space savings has several ramifications on 1/0 performance.

First, the reduced data space fits into a smaller physical disk area; therefore, the seek distances and seek times are reduced. Second, more data fit into each disk page, track, and cylinder, allowing more intelligent clustering of related objects into physically near locations. Third, the unused disk space can be used for disk shadowing to increase reliability, availability, and 1/0 performance [Bitten and Gray 1988].

2.4.3 Bit Vector Filtering Technique

In parallel systems, bit vector filters have been used very effectively for what we call here "probabilistic semi-joins." Consider a relational join to be executed on a distributed-memory machine with repartitioning of both input relations on the join attribute. It is clear that communication effort could be reduced if only the tuples that actually contribute to the join result, i.e., those with a match in the other relation, needed to be shipped across the network. To accomplish this, distributed database systems were designed to make extensive use of semi-joins [Bernstein et al. 1981].

A faster alternative to semi-joins, which, as discussed earlier, requires basically the same computational effort as natural joins, is the use of bit vector filters [Babb 1979], also called Bloomfilters [Bloom 1970]. A bit vector filter with N bits is initialized with zeroes, and all items in the first (preferably the smaller) input are hashed on their join. For each item, one bit in the bit vector filter is set to one; hash collisions are ignored. After the first join input has been exhausted, the bit vector filter is used to filter the second input. Data items of the second input are hashed on their join key value, and only items for which the bit is set to one can possibly participate in the join.

3. EVALUATION

This section explains the experimental setup and procedure.

3.1 Experiment Setup

Apache JMeter was used to conduct the experiments. Apache, a java-based application can be used as a load testing tool for analyzing and measuring the performance of a variety of services and applications. JMeter can be used as a unit-test tool for JDBC database connections, FTP, LDAP, web services, JMS, HTTP, generic TCP connections and OS-native processes. Apache JMeter requires Java 8. The experiments aim to evaluate the performance of multiple users saving multiple BLOB data (i.e., image files) simultaneously to a database. The experiments simulate fifty (20) users simultaneously inserting fifty (50) images into a database. The key metrics to evaluate in the experiment is **response times**, and **throughput**.

The algorithm was written using Java programming with Apache NetBeans 12.2. All experiments have been carried out on the same computation platform, which is a Windows 10 running on a SAMSUNG Laptop with an Intel(R) CORE(TM) i7-3630QM at 2.40GHZ, with 8GB memory and 1TB swap space on the hard disk. The Apache JMeter settings for the experiments are shown in Table 1.

Table 1. Parameters in JMeter used for the experiments.

Settings	Values
No. of Requests (Program)	50
No. of threads/users (Apache)	20
Ramp-up period	5
Loop count	20

The BLOB data used for the experiment is an image file. The properties of the image file is shown in Table 2.

Table 2. Properties of the Image used for experiments.

Property	**Value**
Filename	Fish
Type of file	JPG File (.jpg)
Open with	
Location	C:/Users/desomond/Documents/dataset/fish.jpg
Size	3.36 (3444 bytes)
Size on disk	4.00 (4096 bytes)
Created	November 9, 2019
Attribute	Not read only, Not Hidden
Security	Safe and Unblocked

Database Configuration
The name of the database and table used for the experiment is given below:
Database name: test   and Table name: datastore

Table 3 Datastore Table Specification

Field name	Data type	Default	
Msg	Varchar(45)	NULL	
Store binary	BLOB	NULL	

3.2 Program for Saving BLOB data
The program for measuring the execution time for performing the query performance was developed in Java. The program was compiled to create a JAR and then used in JMeter to automate and simulate the process of repeatedly saving BLOB data in the database by multiple users. The logic used in the Java program is shown in Algorithm 1.

Algorithm 1: Algorithm to calculate query response time.

Input:
db_Name, db_Path, blob_File, N
A set of N database queries (q_1, q_2, …, q_m)
Output:
resp_Time, throughput
Start:

```
1   {
2       capture time at start of query
3         for each query
4         {
5             invoke method for query operation
6             if(error is found in query)
7             {
8                  end query execution
9             }
10        }
11      capture time at end of query
12      calculate response time, throughput
13      store result in repository
14  }
15  Report query performance result
```

The purpose of the above program is to measure the execution of inserting 50 images into a database. For example, the database query operation might be to connect to a MySQL database and insert 50 image files into a table. The total execution time is measured in nanoseconds. Execution time is calculated as follows:

$$Executiontime = (endTime - startTime)$$

The result of the execution would be a response time for inserting the specified number of images in the database. This value could be 7346451500 seconds. For this program, BLOB data was inserted 20 times.

3.3 Experimental Procedure

The procedure for this experiment is as follows:

1. Download and install JMeter on your computer
2. Open NetBeans IDE and compile the Java program written to insert image files into the database.
3. Copy the compiled java program to create a JAR file to the following path in the JMeter folder – JMETER_HOME/lib
4. Start JMeter by opening the folder that contains Apache JMeter. Click on jmeter.bat which can be found in the following path: …/bin/jmeter.bat
5. Setup JMeter and include the BeanShell sampler
6. Run JMeter to test the program.

The custom method in the Java program has to be executed in JMeter. The BeanShell sampler looks as follows:

```
import jmetertest.JMeterTest;
JMeterTest test = new JMeterTest(); // instance of class JMeterTest
test.methodToTime(); // method call
```

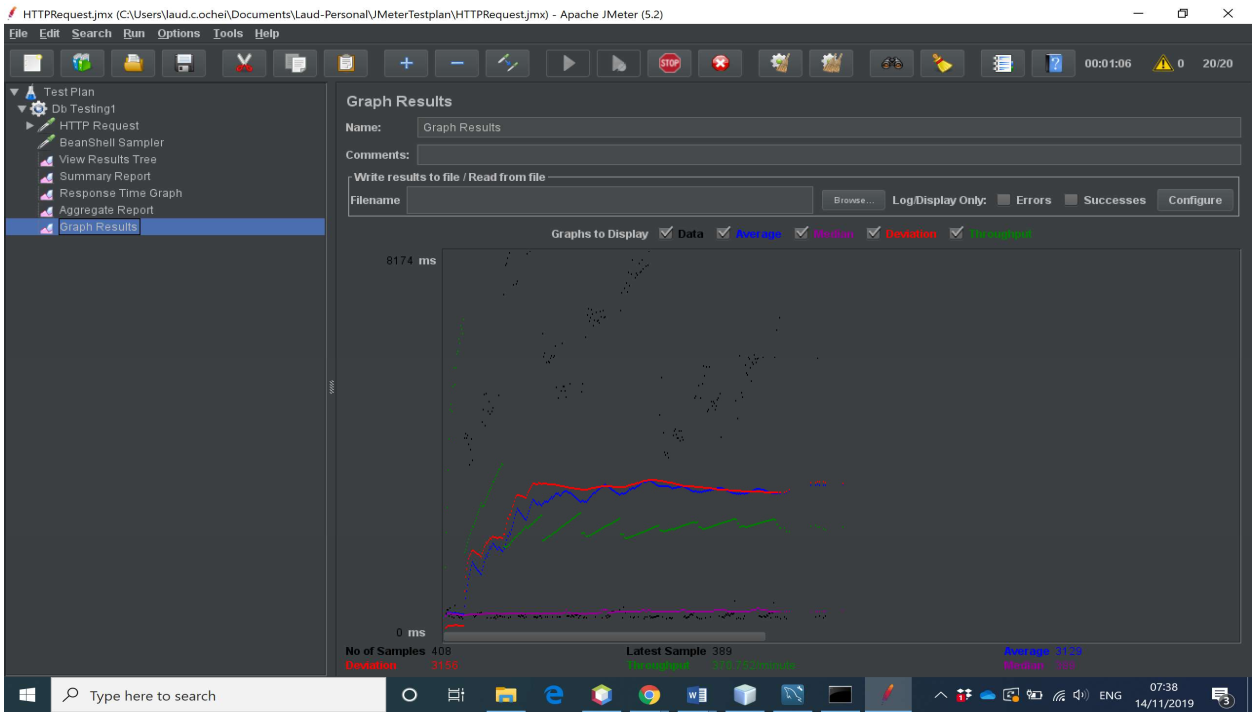
4. RESULTS AND DISCUSSION

In this study, two experiments were performed to measure the performance of the above program under three main scenarios.

Experiment 1: Increasing the number of users.

Experiment 2: Increasing the number of times the same operation is performed.
When testing a program that saves data in a database or performs data reading from a database, it is often necessary to encounter objects of the BLOB type. BLOB, Binary Large Object, is where the program can store images, video, text, and other information that is stored in a binary form. In addition to writing data to the BLOB, the program can also read data from the **BLOB.**

The results of the experiments are summarized below. Figure 1 shows the Graph results of the experiments in JMeter.



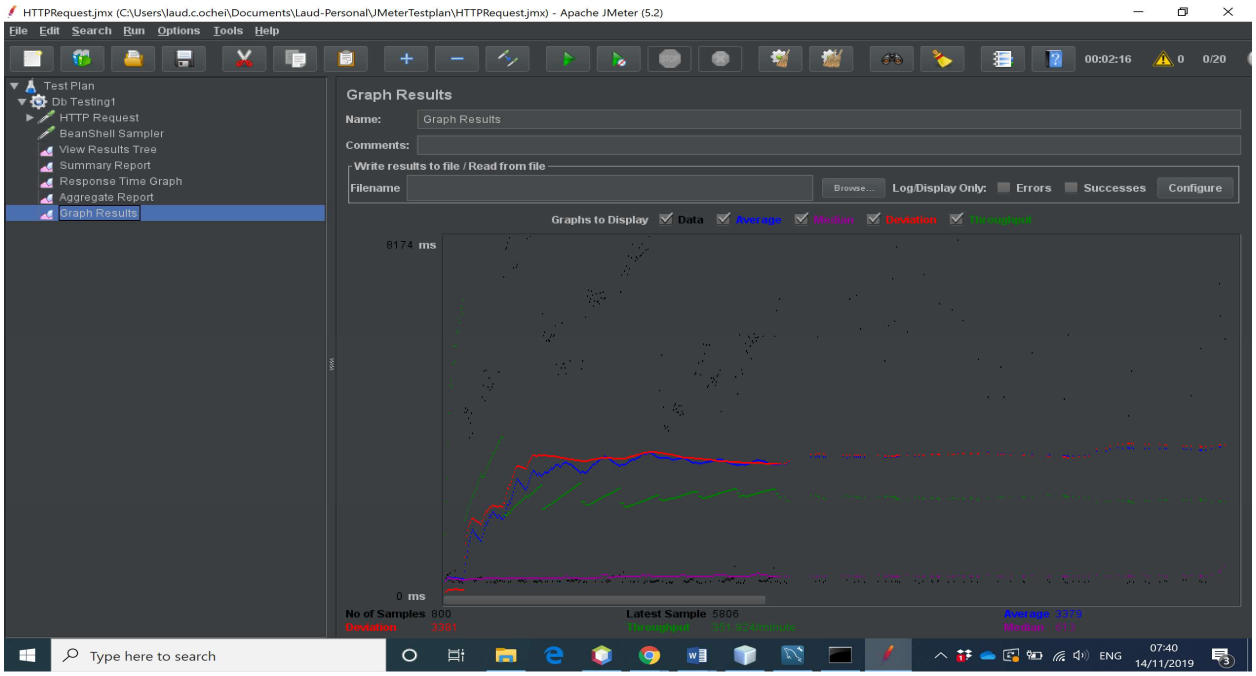
Graph Results

Name: Graph Results

Comments:

No of Samples 408
Deviation 3156
Latest Sample 389
Throughput 370.752/minute
Average 3129
Median 389

Figure 4. Graph result of query performance test (1)



Graph Results

Name: Graph Results

Comments:

Write results to file / Read from file

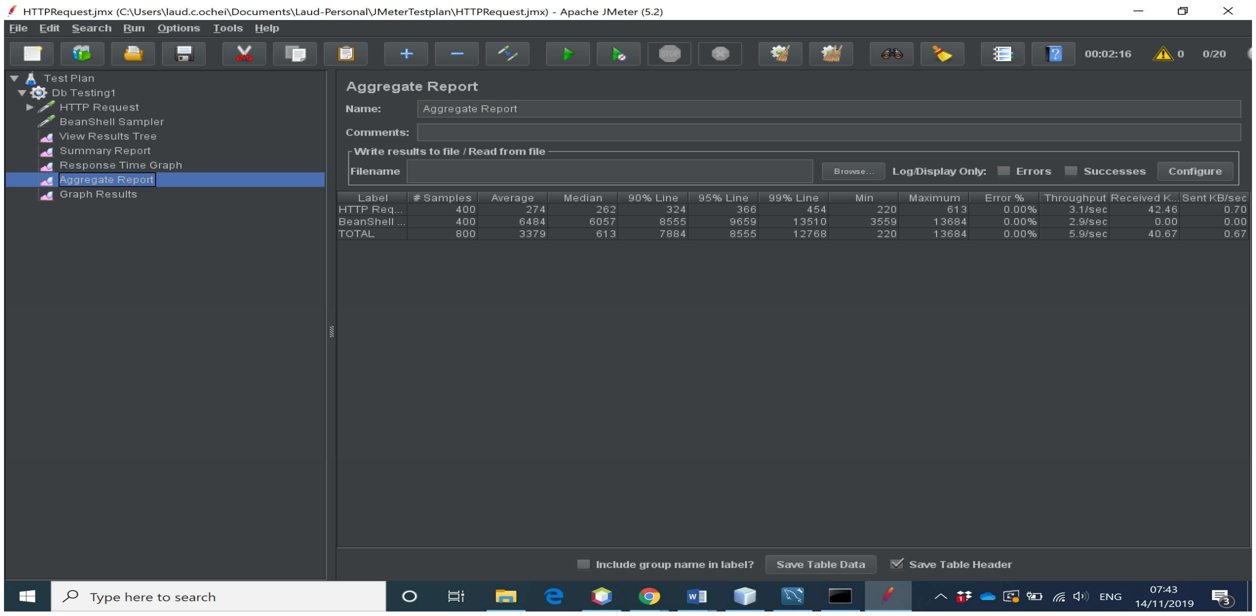
Filename

Log/Display Only: Errors Successes

No of Samples 800
Deviation 3381
Latest Sample 5806
Throughput 351.924/minute
Average 3379
Median 613

Fig 4. Graph result of query performance test (3)

The aggregate report of the performance test showing the number of samples tested, the average, median and throughput of the results.



Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
HTTP Req...	400	274	262	324	366	454	220	613	0.00%	3.1/sec	42.46	0.70
BeanShell ...	400	6484	6057	8555	9659	13510	3559	13684	0.00%	2.9/sec	0.00	0.00
TOTAL	800	3379	613	7884	8555	12768	220	13684	0.00%	5.9/sec	40.67	0.67

Figure 5. Aggregate Report of JMeter Test

This study has demonstrated how to measure the performance of saving BLOB, data (that is, images) into a database under varying conditions. The above results show that when a small number of BLOB files (e.g., images) are inserted into a database, the execution time will continue to increase steadily up to a certain point as more users are repeatedly saving data into the database. Beyond this point, the execution time will increase very slowly and then flatten. If further insertion is done, then the database may not cope with the increased load.

The purpose of this study is the performance of multiple users repeatedly saving BLOB data into a database, the same experimental setup and procedure can also be used to compare the performance of different processes. Generally, if the throughput of process A is less than that of B, it means that the performance of process B is better. If the deviation of process A is more than that of process B, it means that the performance of process B is better.

5. CONCLUSION

This paper discusses the query performance of Binary Data Object (BLOB) in database transactions. An algorithm for evaluating the performance of querying BLOB data has been presented. The algorithm has been implemented in Java and then used in JMeter to simulate multiple users repeatedly saving BLOB data into the database. The extensive evaluation shows that BLOB data can impact heavily on the database when multiple users save BLOB data repetitively into the database, especially when the data is saved in the database due to a sudden and aggressive workload experienced by the system. It is recommended that BLOB data should be used with caution especially when saving multiple BLOB data in a scenario that represents a heavy load burst. In future, we will compare the performance of BLOB and LOB in MySQL and PostgreSQL.

REFERENCES

1. Alom, B.M., Henskens F. and Hannaford, M. (2009), Query processing and optimization in distributed database systems, *International Journal Computer Science and Network Security(IJCSNS'09)*
2. Garcia Luna Aceves, J.J.(2014), *Diffusing Update Algorithm, Wikipedia;*
3. Pop, F. and Cristea V.(2011), Scheduling optimization based on resources state prediction in large scale distributed systems, *Journal of Control Engineering and Applied Informatics (CEAI),* Vol.13, No.4,2011.
4. Rahimi, S.K., Haug, F.S.,(2010), Distributed database management systems: a practical approach, *Wiley Publication*, ISBN: 047040745X, IEEE Computer Society; Truica, C.O. , Boicea, A. and Radulescu, F. (2013),Asynchronous replication in Microsoft SQL Server, PostgreSQL and MySQL, *International Conferenceon Cyber Science and Engineering (CyberSE'13)*
5. T, Boicea, A. and Radulescu, F. (2014),Performance time for e-learning applications with multiple databases, *International Scientific Conference*
6. Rupley Jr, M. L. (2008). Introduction to query processing and optimization. *Indiana University at South Bend.*
7. Oktavia, T. (2014). Implementing data warehouse as a foundation for decision support system (perspective: technical and nontechnical factors). *Journal of Theoretical & Applied Information Technology, 60(3).*
8. Alom, B. M., Henskens, F., & Hannaford, M. (2009). Query processing and optimization in distributed database systems. *IJCSNS, 9(9), 143. Greenspan, J., & Bulger, B. (2001). MySQL/PHP database applications. John Wiley & Sons, Inc.*.10

9. Oktavia, T., & Sujarwo, S. (2014). Evaluation of sub query performance in SQL server. *In EPJ Web of Conferences (Vol. 68, p. 00033).* EDP Sciences.

10. Roussopoulos, N., Faloutsos, C., & Sellis, T. (1988). An efficient pictorial database system for PSQL. *IEEE transactions on software engineering,* 14(5), 639-650.
11. Deshpande, A., & Van Gucht, D. (1988, August). An Implementation for Nested Relational Databases. In *VLDB* (pp. 76-87).
12. Graefe, G., & Shapiro, L. D. (1991, April). Data compression and database performance. In *[Proceedings] 1991 Symposium on Applied Computing* (pp. 22-27). IEEE.
13. Lynch, C. A., & Brownrigg, E. B. (1981, September). Application of data compression to a large bibliographic data base. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7* (pp. 435-447). VLDB Endowment.
14. Severance, D. G. (1983). A practitioner's guide to data base compression tutorial. *Information Systems,* 8(1), 51-62
15. Bernstein, P. A., Bernstein, P. A., & Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR),* 13(2), 185-221.
16. Lelewer, D. A., & Hirschberg, D. S. (1987). Data compression. ACM Computing Surveys (CSUR), 19(3), 261-296.
17. Domdouzis, K., Lake, P., & Crowther, P. (2021). Database Performance. In *Concise Guide to Databases* (pp. 281-322). Springer, Cham.
18. IBM. (2021). Database performance and query optimization. IBM. Retrieved from https://www.ibm.com/docs/en/i/7.1?topic=database-performance-query-optimization
19. Patnaik, K. K., & Nagarjun, B. (2011). Extending Binary Large Object Support to Open Grid Services Architecture-Data Access and Integration Middleware Client Toolkit. *Journal of Computer Science*, *7*(6), 832.