Towards scalability in systems with write operations in relational databases

Leonardo José Gomes da Silva
Faculdades Integradas Teresa D'Ávila, FATEA
Lorena, Sao Paulo, Brazil

Leandro Guarino de Vasconcelos
Brazilian Institute for Space Research
Sao Jose dos Campos, Sao Paulo, Brazil

Glauco da Silva
Institute of Aeronautics and Space
Sao Jose dos Campos, Sao Paulo, Brazil

Luiz Eduardo Guarino de Vasconcelos
Flight Test Research Institute
Sao Jose dos Campos, Sao Paulo, Brazil

***Abstract*— The scalability of systems is essential in scenarios where there is a lot of concurrent users. Some systems have peak access during certain times; others have a lot of concurrent users regularly. Thus, these systems must be able to perform equally as a growing number of requests. The study of system scalability that manipulate database considers read operations in database systems. However, in this paper we present an approach for horizontal scalability on systems that perform write database operations. The proposed approach uses queueing theory and differs from commonly developed solutions like clustering databases and elasticity in the cloud. A case study was conducted with an application of persistent logs, and two versions of the application have been implemented in order to compare the efficiency of the use of queues. Results show the efficiency of the use of queueing theory implemented to allow scalability for applications with a large amount of writing database operations.**

Keywords - scalability, queueing theory, distributed systems, relational database.

I. INTRODUCTION

Scalability is a desirable feature in any software, and it indicates the ability of a system uniformly to handle an increasing amount of work.

That the greater the number of users of the system, more resources will be required to process their requests. Systems that suffer technical losses (e.g. loss of performance and increased latency) as the number of simultaneous accesses increases are not scalable or does not maintain its features regardless of the load accesses. Great news portals and e-commerce has a lot of concurrent users and the need to manage scalability issues in order not to impair the services provided.

Currently, two concepts are used to scale systems, vertical and horizontal scalability.

The concept of vertical scalability is based on the improvement of the hardware on which software is running, for example, addition of memory, adding processors, addition of cores or disk space. Vertical scalability is a simpler concept to be applied, but in the long term can be a very expensive solution, and depend on technological advances in hardware. According to Fisher and Abbot [2], when large companies do not scale their systems vertically, the only option is to buy better and faster hardware systems. When they reach the limitation of faster and better hardware supplied by more expensive supplier in question, they will be in big trouble.

The horizontal scalability, according to Fisher and Abbot [2] is a duplication of services or databases to distribute the load of transactions, in addition to being an alternative to buying larger and faster servers.

The concept of horizontal scalability differs from the vertical scalability model when thinking about the availability of system resources. While the vertical scalability increases the processing power of the machine running the application, The horizontal scalability is based on replication of resources and the use of a responsible mechanism for scheduling of the requests.

Duplication of system resources (e.g., application and database) is the initial distribution model. In this model, the resources are distributed and scheduled. But this method generates a very large resource redundancy, and it creates an architecture rather cohesive, in which every application and every duplicate databases in architecture do all type of operation required.

In this paper, we present an approach applicable to systems that perform a lot of writing database operations. The goal of the approach is to help increase the performance and scalability of these systems through a web application that provides the necessary scenario for the presentation., analysis and solution of bottlenecks.

To validate the approach, this paper presents a case study that exposes bottlenecks in existing systems that perform a lot of concurrent write operations in a relational database. For the case study, a prototype for managing logs was created (i.e. data generated by applications at runtime, e.g., error messages). This prototype simulates thousands of clients simultaneously sending new logs to be saved by the application. The project was implemented in two versions, both with the same technologies (Java, Spring, Hibernate, Maven). But in the second version, concepts and techniques to increase performance and scalability have been implemented, such as (i) implementation of an intermediate layer between the application and the database, (ii) inserting objects batch in the database, (iii) application distribution and (iv) use of cache queries to the database.

The results of the comparison between the two versions show the efficiency of the techniques used, eliminating the loss of logs without drastically impacting the running time.

978-1-4799-8828-0/15 $31.00 © 2015 IEEE
DOI 10.1109/ITNG.2015.49

This paper is organized as follows: Section II discusses the concepts of distributed systems, scalability for relational databases, and queueing theory. In the Section II are shown the test scenarios conducted with each version of the application. Results of comparisons between versions are also presented in this section. Section IV presents the conclusion and future work.

II. THEORETICAL

A. Distributed Systems

For Tanenbaum and Steen [11], “a distributed system is a set of independent computers accessed by users as a single coherent system”. Another approach is given by Fisher and Abbot [2] which reports that the word "distribute" refers directly to grid computing. Grid computing is the concept of dividing tasks into smaller chunks of work that can be performed by two or more computers, each of which performs a portion of the task required for the final result".

Instead of duplicate resources it is possible distribute them at the level of business logic, where every database must manage only information related to the actions performed by the system that uses it. When deploying an application in business modules it is necessary think in a way to make them accessible and usable by other modules. According to Tanenbaum and Steen [11], "the main goal of a distributed system is to facilitate the users, and applications, remote access to resources and sharing it through controlled and efficient manner". Thus, the architecture of distributed systems do not only mean the distribution system among several computers but also the distribution of functions into components (e.g., services and resources) and their integration. Figure 1 shows the architecture of distributed components.

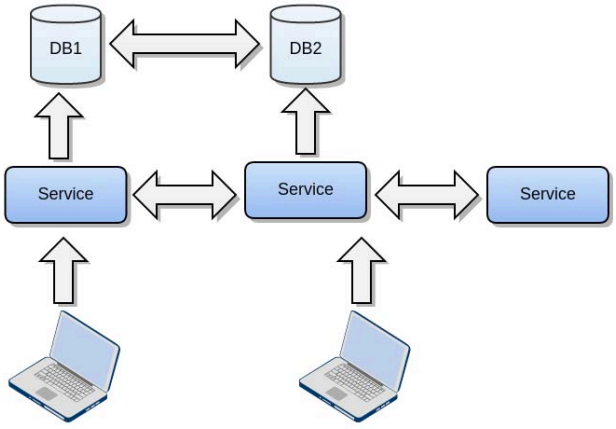

Figure 1. Architecture of distributed components

In the architecture of Figure 1, the components are distributed considering their business models, and there is an exchange of information between them.

With the service layers in distributed system, it is possible to have greater visibility and flexibility to scale the system both horizontally and vertically. The cohesive services allow identifying of the application's bottlenecks more quickly. Also identify the services most used to optimize, replicate and scale only the modules that have greater demand.

Besides the concepts that directly affect the scalability of systems, there are other techniques. The correct use of threads, I/O non-blocking, store information without the use of session, and cache servers to manage temporal information of the application may help scalability of systems. Following sections explain these concepts.

B. Scalability in Relational Databases

Horizontal and vertical scalability can also be applied to the database, but with some singularities. Online systems can be scaled horizontally replicating and scaling the client requests; however access to the database becomes a problem.

A simple approach to increase scalability in databases is the master-slave concept. According to Fisher and Abbot [2], the application can direct the writing and updating data operations for the master instance and read operations for instances slaves.

The implementation of the concept of master-slaves should be used on any system that performs a large amount of read operations on databases. When reading information via slave instances can increase the number of slaves as needed, thereby delivering search operations across multiple databases. This method performs horizontal scalability in the database. Figure 2 shows an example of a master-slave architecture.

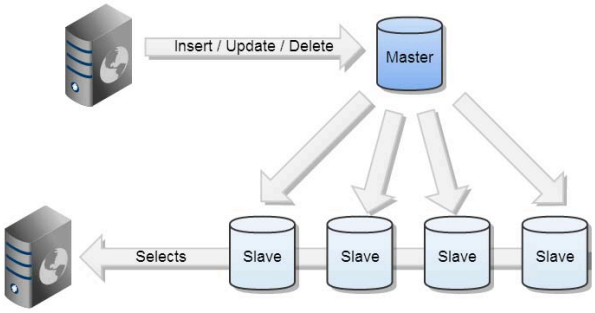

Figure 2. Architecture Master-slave

In databases, the problem of master-slave concept is that the slave instances only work with read operations. For applications that perform many insert, update and delete operations, this approach does not offer an increase of horizontal scalability. Because, to change information on the slave, this change is not replicated in the master instance.

For Fisher and Abbot [2], because of the guarantee of the ACID properties (Atomicity, Consistency, Isolation and Durability), scaling a RDBMS (Relational Database Management System) is more challenging than other forms of data storage. The RDBMS offer several advantages over ACID, e.g., transactional operations, removing the effects of partial database transactions arising from system failure, unexpected situation or an interrupted transaction. However, the RDBMS also brings impacts on system scalability when working with many concurrent operations on the database,

Identify applicable sponsor/s here. *(sponsors)*

especially if they are write operations. Currently, there are several solutions for scalability of systems, such as the clustering of instances of databases, the use of non-relational databases [9] (e.g., Cassandra [1], MongoDB [7]), MapReduce-based systems (e.g., HadoopDB [10]), performance tuning available for each RDBMS, database replication [12], hardware upgrade, among other.

Plattner and Alonso [13] proposed the Ganymed, a database replication middleware intended to provide scalability without sacrificing consistency. The main idea is to use a novel transaction scheduling algorithm that separates update and read-only transactions.

Lin et. al. [14] analyzed how database replication can be used Multi-player Online Games (MOGs), that require a high degree of fault tolerance, scalability and performance.

Moon et. al. [3] proposed Archival Information Management System (AIMS) that achieves scalability by using advanced storage strategy based on relational technology, suitable temporal indexing and clustering techniques and temporal query optimizations.

The approach proposed in this paper is based on minimum possible infrastructure restricted the use of relational databases. The proposed approach is a distributed architecture that uses queueing theory, cache queries to database and batch insertion of records. Our solution is a low-cost efficient for applications with a large amount of write operations in relational databases.

C. Queueing Theory

According to Gross et. al. [4], queueing theory has been developed to provide models that predict the behavior of systems that meet demands arising randomly.

An implementation of queueing theory was used in our work to control the amount of writes in the database of the application.

The implementation works with a responsible component for inserting new objects in the queue. Several components are responsible for removing objects from the queue and insert their into databases. When controlled the amount of components that remove objects from the queue, it is possible adjust the access I/O of database, and enable the implementation of process improvements, such as the implementation of batch insert of records.

The queueing theory has some essential features such as the request arrival pattern, pattern of service servers, queue discipline, system capacity, number of service channels and number of service stages. These characteristics are defined as:

- The arrival pattern defines the way that the queue will manage new objects to be inserted.
- The pattern of service servers defines that the service must be unique or batch, and should increase or decrease the processing according to the queue growth.
- The discipline of the queue determines the process used to insert and retrieve the objects from the queue.
- The system capacity determines the maximum number of objects that can be managed by a queue.
- The number of service channels determines that the queue may be distributed in different input and output channels, and can, therefore, benefit from access by several clients in different parallel channels.
- The number of service stages determines a process with a single stage. However, in a multistage process, each stage must be worked in a different queue.

Among these features, a unique feature not implemented in our architecture is the system capacity. If the queue is limited by a number of objects, the application should provide other ways to store objects that were not inserted in the queue due to overflow.

III. CASE STUDY

To conduct the case study, we developed a test application that manages application logs. This type of application was chosen due to the large number of write operations in databases and because it is possible to simulate many concurrent users. Validating the use of queueing theory, two versions of the log manager were implemented, and only one of them uses queueing theory.

After selecting the application to be tested, three test scenarios were defined, considering the amount of concurrent users (clients), the number of write operations and the time to create the users (clients).

In the test scenarios, 18,000 users (clients) were created in order to performs 100 concurrent write operations per client. For each scenario there was a variation of time to create the clients, ranging from 10, 5 and 2 minutes. The purpose of this change was to simulate the intensity of concurrent operations.

Each scenario was run twice in order to calculate the average of execution times. Each scenario was run using the Apache JMeter tool [5], a tool that allows the simulation of various clients in memory and allows you to perform different types of requests to the server.

The features of the infrastructure used to execute the test scenarios are listed in Table I.

TABLE I. INFRASTRUCTURE ENVIRONMENTAL TESTING

CPU	Intel(R) Core(TM) i3-2348M CPU @ 2.30GHz
CPU cache	3072 KB
RAM memory	6 GB
Operating system	Linux Mint 16 Petra, 64bits
Server	Jetty. Initial memory 512MB e maximum memory 2048MB
RDBMS	Postgres. Number of maximum connections: 500
Test tool	Apache JMeter 2.11 with 512MB

No improvement was implemented in the RDBMS; all improvements were implemented in the log manager.

The architecture implemented in the first version of log manager is shown in Figure 3, and it works as follows: (i) the client system sends the log to a Restful web service [8] and

JSON (JavaScript Object Notation) [6]; (ii) the manager authenticates user access; (iii) the manager validates the received log and, if it is valid, persists it in the database.

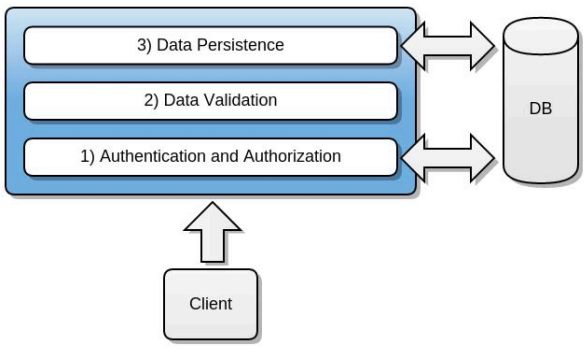

Figure 3. Log manager - first version of the architeture

Considering the first version of the architecture implemented, the persistence of the logs in the database generates some performance bottlenecks and scalability shortcomings. When inserting an individual log, a performance problem occurs because each insertion generates a transaction and, consequently, a commit, and it is computationally costly. Additionally, after each commit, all indexes on the table are updated. When there are thousands of concurrent users, an issue of scalability appears. The system has no flow to manage the logs to limit the number of insertions in the database. Thus, all the logs sent by client systems after authenticated and validated, are inserted directly in the database. The RDBMS has a processing limit, according to the released memory and processing for its use. Thus, problems of database timeout occur in the application, because the DBMS can not handle so many logs at once.

Considering bottlenecks generated by the first architecture, which is limited to the RDBMS, the architecture of the proposed approach was implemented as shown in Figure 4.

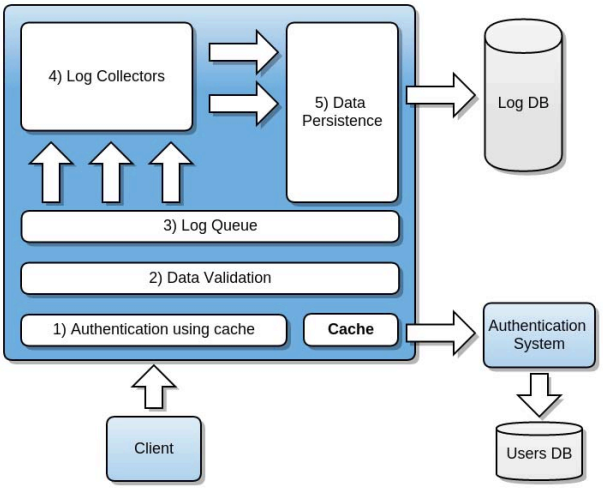

Figure 4. Second version - architecture of the proposed approach

The second version of the log manager works in a distributed way. One database is used only to authenticate users. In addition, the log manager has a cache model on the results generated by the authentication system (i.e. it saves authenticated users in memory). After authentication, the log is validated and inserted in queue logs, that works with FIFO (First In First Out).

Logs inserted in queue logs are captured by collectors, where each object until collects thousand logs at once and send them to the RDBMS for persistence. This technique is known as persistence batch. With persistence batch, the RDBMS creates a transaction and generates a commit every thousand new logs, reducing the performance bottleneck.

With the responsibility of managing logs being attributed to the log queue and collectors, it is possible to control the amount of logs sent to the RDBMS to fulfill the demand without loss of performance and scalability.

In the test scenarios presented below, two time parameters were defined: timeout and the time to create the clients.

The timeout refers to the time taken by clients to simulate a real environment, where an application could not wait a long time to get a response from a server.

Time for creation the clients is used by JMeter, which distributes the time of the creation of all client systems within this defined time, thus providing an ideal way to measure the scalability of the system.

D. Test Scenario # 1

In the first scenario, clients 18,000 were created in 10 minutes, sending logs to the server simultaneously, each system sent 100 logs.

Table II shows the results of the scenario executed in the first architecture. The average execution time was 10m01s. In this run, the percentage of lost logs (over 99%) reflects the inefficiency of architecture.

TABLE II. RESULTS OF THE FIRST SCENARIO IN THE FIRST ARCHITECTURE

Description	First run	Second run
Time to create 18,000 clients	10 minutes	10 minutes
Timeout	5 seconds	5 seconds
Logs sent	1.800.000 logs	1.800.000 logs
Logs saved	11.811 logs	11.869 logs
Time	10:02:418 minutes	10:00:768 minutes
Average of the saved logs per minute	1.178,250989108 logs	1.185,989160325 logs
Average of the saved logs per second	19,637516485 logs	19,766486005 logs
Percentage of data loss	99,343833333%	99,340611111%

Table III shows the results of the scenario I executed in the architecture that uses queueing theory. The average execution time was 10m22s. Although the running time is a few seconds higher, there was no loss of any log record.

TABLE III. RESULTS OF THE FIRST SCENARIO IN THE PROPOSED ARCHITECTURE

Description	First run	Second run
Time to create 18,000 clients	10 minutes	10 minutes
Timeout	5 seconds	5 seconds
Logs sent	1.800.000 logs	1.800.000 logs
Logs saved	1.800.000 logs	1.800.000 logs
Time	10:16:835 minutes	10:27:523 minutes
Average of the saved logs per minute	177.019,870480461 logs	175.178,560479911 logs
Average of the saved logs per second	2.950,331174674 logs	2.919,642674665 logs
Percentage of data loss	0%	0%

E. Test scenario #2

The second scenario, similar to the first, also created 18,000 customers, and every customer sent 100 logs to the log manager. However, the time to create the clients has been reduced by half (from 10 to 5 minutes), with the aim of intensifying simultaneous operations.

Table IV shows the results of the scenario executed in the first architecture. Again, the percentage of loss shows the inefficiency of the first architecture. The average execution time was 10m42s.

TABLE IV. RESULTS OF THE SECOND SCENARIO IN THE FIRST ARCHITECTURE

Description	First run	Second run
Time to create 18,000 clients	5 minutes	5 minutes
Timeout	5 seconds	5 seconds
Logs sent	1.800.000 logs	1.800.000 logs
Logs saved	22.116 logs	22.657 logs
Time	10:41:249 minutes	10:43:523 minutes
Average of the saved logs per minute	2.123,987634082 logs	2.171,202743016 logs
Average of the saved logs per second	35,399793901 logs	36,186712384 logs
Percentage of data loss	98,771333333%	98,741277778%

Table V shows the results of the scenario executed in the second architecture. The average execution time was 10m47s, compared to the first scenario is slightly higher (25s), and the percentage of logs stored is 100%.

TABLE V. RESULTS OF THE SECOND SCENARIO IN THE PROPOSED ARCHITECTURE

Description	First run	Second run
Time to create 18,000 clients	5 minutes	5 minutes
Timeout	5 seconds	5 seconds
Logs sent	1.800.000 logs	1.800.000 logs
Logs saved	1.800.000 logs	1.800.000 logs
Time	10:56:480 minutes	10:38:849 minutes
Average of the saved logs per minute	170.377,101317583 logs	173.268,684861804 logs
Average of the saved logs per second	2.839,618355293 logs	2.887,811414363 logs
Percentage of data loss	0%	0%

F. Test scenario #3

In the third scenario, the time for creation of 18,000 clients was reduced to 2 minutes.

Table VI shows the results of the scenario executed in the first architecture. The execution time was the smallest of the three scenarios, however the loss of logs remained greater than 99%, again showing the inefficiency of the first architecture.

TABLE VI. RESULTS OF THE THIRD SCENARIO IN THE FIRST ARCHITECTURE

Description	First run	Second run
Time to create 18,000 clients	2 minutes	2 minutes
Timeout	5 seconds	5 seconds
Logs sent	1.800.000 logs	1.800.000 logs
Logs saved	9.119 logs	9.152 logs
Time	09:50:466 minutes	09:48:799 minutes
Average of the saved logs per minute	959,424114066 logs	964,587863183 logs
Average of the saved logs per second	15,990401901 logs	16,076464386 logs
Percentage of data loss	99,493388889%	99,481555556%

Table VII shows the results of the scenario executed in the second architecture. Although average running time was more than 2 minutes compared to the other scenarios, the percentage of persistence remained at 100%.

TABLE VII. RESULTS OF THE THIRD SCENARIO IN THE PROPOSED ARCHITECTURE

Description	First run	Second run
Time to create 18,000 clients	2 minutes	2 minutes
Timeout	5 seconds	5 seconds
Logs sent	1.800.000 logs	1.800.000 logs
Logs saved	1.800.000 logs	1.800.000 logs
Time	12:35:770 minutes	12:28:585 minutes
Average of the saved logs per minute	145.658,172637303 logs	146.510,009482453 logs
Average of the saved logs per second	2.427,636210622 logs	2.441,833491374 logs
Percentage of data loss	0%	0%

IV. CONCLUSIONS

In this paper, we presented a distributed architecture that uses queueing theory to achieve the scalability of systems that perform a lot of write operations on the database.

In the case study, the comparison of test scenarios in a non-distributed architecture and the proposed architecture allows to identify: (i) the scalability of the system; (ii) the performance of the log manager to operations per second, and (iii) efficiency in the persistence of logs in the database. These improvements are due to the distributed architecture and the batch insert.

To the system manager logs distributed into two components, one designed to manage and save the logs and another specifically for authenticating users, it was possible to delete the contents of FK (foreign key) of the log table. Moreover, it has been provided a good practice of development known as cohesion systems, where each component has only a responsibility. An implementation of a queue and collectors for managing logs provided the greatest control over all information received by client applications. This implementation allowed for better handling of the information, such as the persistence batch up to a thousand logs in the same transaction.

In future work, we intend to implement sharding technique and changing the log queue with MoM (oriented Middleware Messages). In addition, we intend to develop a benchmarking on performance and scalability on write operations between relational databases and databases non-cycle.

V. REFERENCES

[1] Cassandra. "The Apache Cassandra Project". Available at http://cassandra.apache.org/

[2] Michael T. Fisher, and Martin L. Abbot. "Scalability Rules". Boston: Pearson Education 2011.

[3] Hyun Jin Moon, Carlo A. Curino, and Carlo Zaniolo. "Scalable architecture and query optimization fortransaction-time DBs with evolving schemas". In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, p. 207-218, 2010.

[4] Donald Gross, John F. Shortle, James M. Thompson, and Carl. M. Harris. "Fundamentals of queueing theory". Hoboken, New Jersey: John Wiley & Sons, Inc., 2008.

[5] JMeter. Available at http://jmeter.apache.org/.

[6] JSON. Introducing JSON. Available at http://json.org/.

[7] MongoDB. Available at http://www.mongodb.org/.

[8] Leonard Richardson, and Sam Ruby. "RESTful web service", O'Reilly Media, 2007.

[9] Jaroslav Pokorny. "NoSQL databases: a step to database scalability in web environment". In Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services. ACM, New York, NY, USA, 278-283, 2011.

[10] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads". Proc. VLDB Endow. 2, 1 (August 2009), p. 922-933. 2009.

[11] Andrew S. Tanenbaum, and Marten van Steen. "Distributed Systems: Principles and Paradigms", Boston: Pearson Education. 2nd Edition. 2008.

[12] Bettina Kemme and Gustavo Alonso. "Database replication: a tale of research across communities". Proc. VLDB Endow. 3, 1-2 (September 2010), p. 5-12. 2010.

[13] Christian Plattner and Gustavo Alonso. "Ganymed: scalable replication for transactional web applications". In Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware '04). Springer-Verlag New York, Inc., New York, NY, USA, p. 155-174. 2004.

[14] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. "Applying database replication to multi-player online games". In Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (NetGames '06). ACM, New York, NY, USA. 2006.