Towards Scalability in Systems with write operations in relational databases

Leonardo José Gomes da Silva

Leandro Guarino de Vasconcelos

Glauco da Silva

Luiz Eduardo Guarino de Vasconcelos

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.
- The larger 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.

Vertical 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, when large companies do not scale their systems vertically, the only option is to buy better and faster hardware systems. When the companies reach the limitation of faster and better hardware supplied by more expensive supplier in question, they will be in big trouble.

Horizontal Scalability

- The horizontal scalability, according to Fisher and Abbot 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.

Paper Motivation

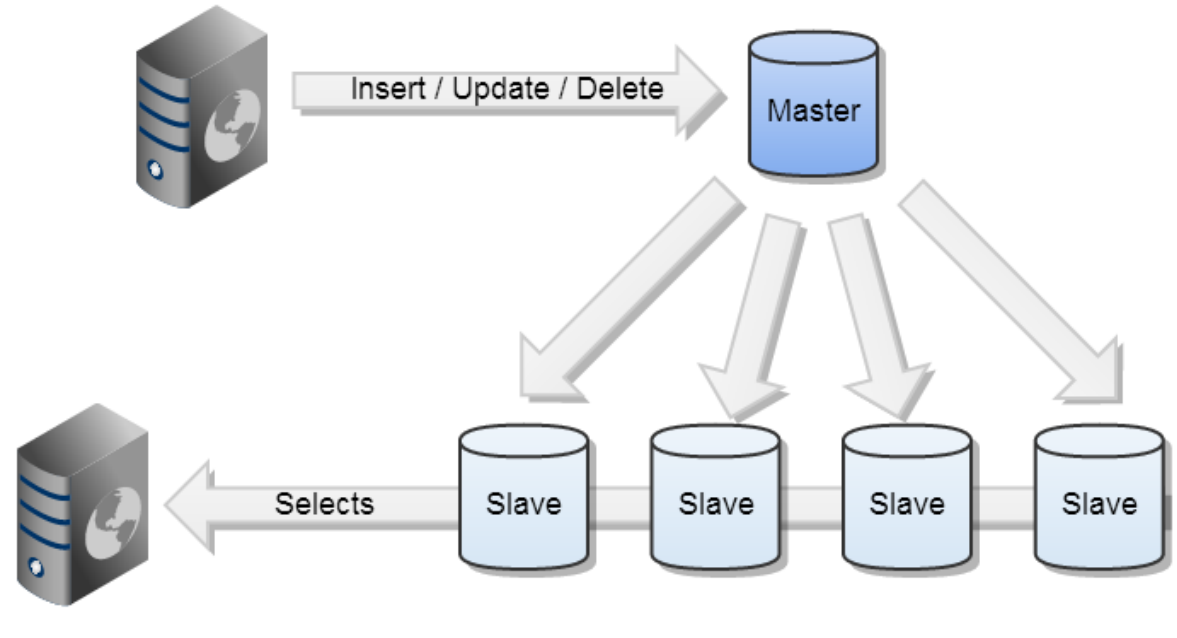
- Horizontal and vertical scalability can also be applied to the database, but with some singularities. Online systems can be scaled horizontally replicating and scaling the client requests; however access the database becomes a problem.
- A simple approach to increase scalability in databases is the 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 slaves.

Paper Motivation

- 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.

Paper Motivation

- Figure below shows an example of a master-slave architecture.



Paper Motivation

- For Fisher and Abbot, 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.
- However, the RDBMS also brings impacts on system scalability when working with many concurrent operations on the database, 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 (e.g., Cassandra, MongoDB), MapReduce-based systems (e.g., HadoopDB), performance tuning available for each RDBMS, database replication, hardware upgrade, among other.

Paper Proposal

- 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 queuing 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.

Paper Proposal

- 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.

Paper Proposal

- 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.

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 queuing theory, two versions of the log manager were implemented, and only one of them uses queuing 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).

Case study

- In the test scenarios, 18,000 users (clients) were created in order to perform 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 it allows you to perform different types of requests to the server.

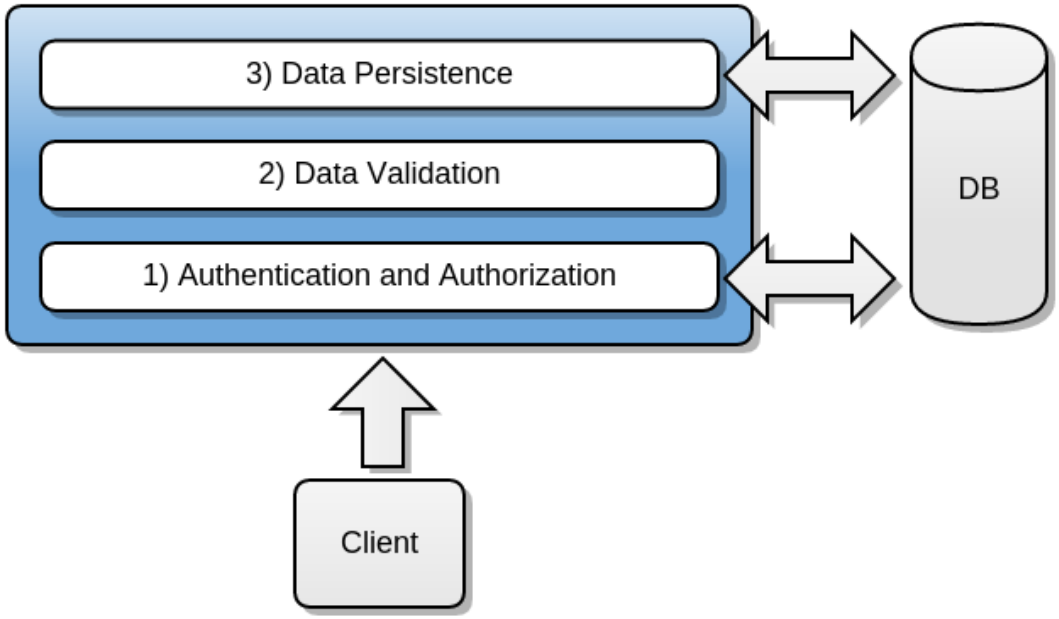
Case study

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

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

Case study

- 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 below.



Case study

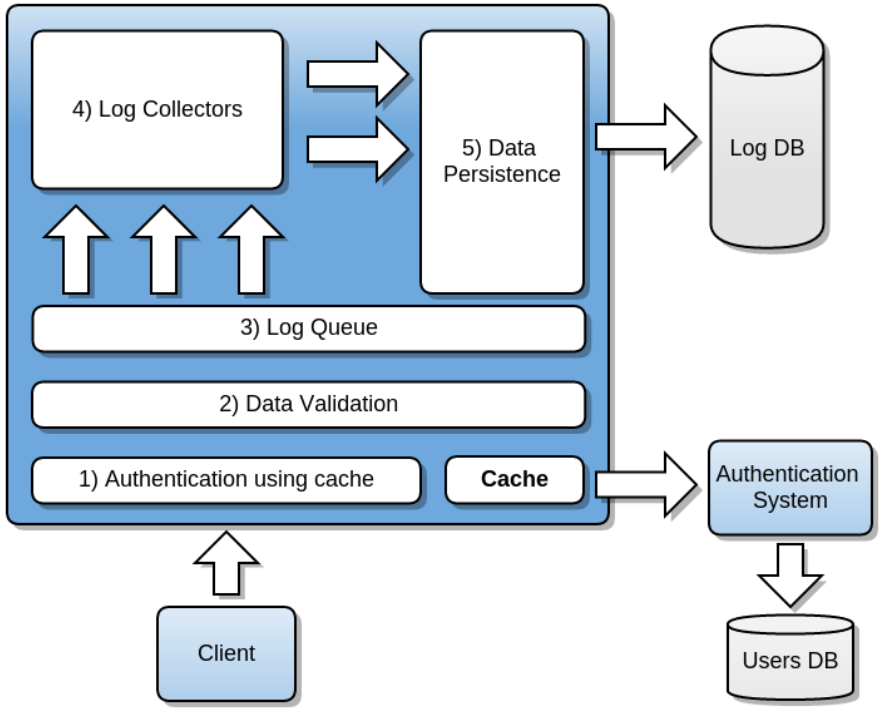
- Considering the first version of the architecture implemented, the persistence of the logs in the database generates some performance bottlenecks and scalability shortcomings.
- When inserting an individual log, a performance problem occurs because each insertion generates a transaction and, consequently, 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.

Case study

- 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.

Case study

- 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 below.



Case study

- 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.

Case study

- 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.

Paper Results

- In the first scenario, 18,000 clients were created in 10 minutes, sending logs to the server simultaneously, each system sent 100 logs.
- The table below shows the results of the scenario executed 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%

Paper Results

- In the first scenario, 18,000 clients were created in 10 minutes, sending logs to the server simultaneously, each system sent 100 logs.
- The table below shows the results of the scenario I executed in the architecture that uses queuing theory.

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%

Paper Results

- The second scenario, similar to the first, also created 18,000 clients, 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.
- The table below shows the results of the scenario executed 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%

Paper Results

- The second scenario, similar to the first, also created 18,000 clients, 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.
- The table below shows the results of the scenario I executed in the architecture that uses queuing theory.

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%

Paper Results

- In the third scenario, the time for creation of 18,000 clients was reduced to 2 minutes.
- The table below shows the results of the scenario executed 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%

Paper Results

- In the third scenario, the time for creation of 18,000 clients was reduced to 2 minutes.
- The table below shows the results of the scenario I executed in the architecture that uses queuing theory.

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%

Conclusion

- In this paper, we presented a distributed architecture that uses queuing theory to achieve the scalability of systems that perform a lot of writing 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.

Conclusion

- The log manager distributed into two components, one designed to manage and save the logs and another specifically for authenticating users, it allows to delete the FK (foreign key) of the log table.
- Moreover, a good practice of development known as cohesion systems was implemented , 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.