InvaliDB Implementation

Group7

b07902014 蔡承濬　　b07902048 李宥霆

b07902124 鄭世朋　　b07901043 沈信樺

1. INTRODUCTION

In the generation network has become indispensable, shopping or performing bank operations online is fairly normal in our daily life. However, the rise of these kinds of web services demands nowadays databases or servers to detect and publish changes with low latency. Given that traditional databases are mostly pull-based, which would output data only when receiving queries. Since they are excelling in dealing with slowly-changing large amounts of data, not rapidly-changing small amounts of data. Therefore, real-time databases that can handle high frequency data change and publish changed data in a short time are getting more popular these days. Some real-time databases such as MeteorDB, RethinkDB and Firebase have received great reputation and are widely-used today. However, the query mechanisms of these databases are encountered with the challenge of read and write scalability and a lot of research has been done managing to fix this bottleneck.

Seeing the problem listed above, we focus on soft real-time databases which accept a deadline violation, and try to implement a real-time database that can support push-based queries based on a pull-based database, called InvaliDB. We claim that it is a solution to the scalability problem. Our implementation is a software solution instead of the hardware method our referenced paper proposed. In addition, we design our experiments, profiling the scalability of software InvaliDB.

2. Method comparison

2.1. Nowadays' real-time DB

There are two main methods adopted by today's real-time databases, i) poll-and-diff and ii) change log tailing. But, these two methods have their own shorthands.

i) **poll-and-diff**. In order to solve the condition that a server can detect the changes that are caused by the clients subscribing to the particular server, it can not detect the changes which occur on different servers. Thus, the server needs to send a query to the database periodically (poll), comparing the new result of the query with the old result (diff), and eventually pushes the update back to the client. However, the period time has a lower bound of about ten seconds, the frequency that the clients get the updates is limited. Besides, in the case that there are many clients subscribing with many read queries, the server should re-send a large number of queries to the database, and the latency may increase exponentially. This causes congestion on read, and there's a read scalability problem.

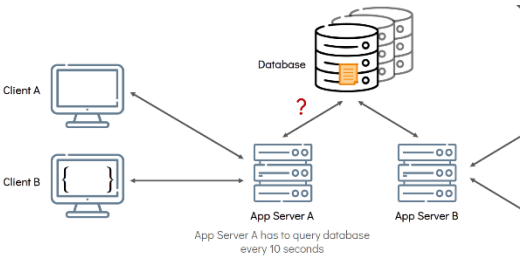

Figure 1. poll-and-diff

ii) **change log tailing**. The database is sliced to shards, and the servers subscribe to all the shards and change logs. Whenever a write request is sent to a server, the server distributes the update data to shards of the database. Then, the database broadcasts the change to all the servers, and servers push back the change of data which clients want. However, in the case that there are many write requests, the server will receive a

large broadcast of changes and take more time to deal with pushing. This may cause congestion on write, and there's a write scalability problem.

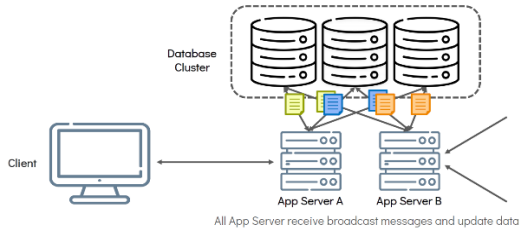

Figure 2. change log tailing

2.2. InvaliDB method

InvaliDB is based on a pull-based database, and it copes with updates by an additional cluster. The cluster uses 2-D parallel matching to speed up handling the large number of requests. The main structure of InvaliDB contains app server, database, cluster and is shown in Figure 3.

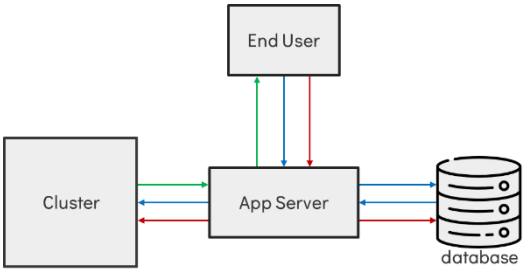

Figure 3. InvaliDB structure

End users can send two kinds of requests. One is a subscription request with a query criteria for reading and the other is a write request to update data.

App server acts as a communication bridge between user, server, database and cluster. It also deals with the data to make communication reasonable like translating the client request to database query. For communicating with cluster, app server partitions the subscriptions and write requests.

Cluster has $N \times N$ nodes as matching units. Every node stores a partition of data. Each row in Figure 4 is responsible for a part of the clients' subscription requests, and by collecting the data at the same row, we can get a complete copy of data that those subscription requests need. Besides, Every node in the same column shares the same vertical partition of data. With these properties that cluster owns, we discussed the detailed paths and partition ways of requests and data.

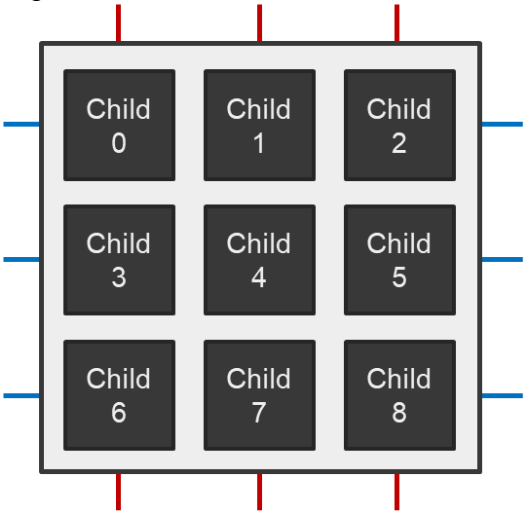

Figure 4. Detailed structure of cluster

For the path of read (blue arrows in Figure 3), assume a client sends a subscription request to the app server. The app server sends translated database queries to the database and gets the initial result. Then, the app server picks one row that will be responsible for this subscription request. separates the initial result, and sends the data that each node is responsible for to the nodes of the chosen row.

As for the path of write (red arrows in Figure 3), suppose a client sends a write request for the data change to the app server. Then, the app server sends translated database queries to the database to update the data in the database. Simultaneously, the write requests are partitioned vertically to the column of cluster.

As for the matching mechanism and the path of output (green arrows in Figure 3), when a subscription query and its initial data arrives at a row of the cluster, the result data is sent from the cluster through the app server to the subscribing client in the end user. When

a write request is sent to a column of cluster, the node in the column and in the subscribed row matches the data it stores with the newly changed data. Next, an update is sent from the cluster through the app server to the subscribing client in the end user. This 2-D parallel matching method.

3. IMPLEMENTATION

We choose MongoDB as our pull-based database since MongoDB is widely used in web applications. Besides, we use JavaScript to implement our program for convenience to access MongoDB.

Now, we see the end-user as the frontend user. There is a constraint on writing. We ask to provide the index of data so that we can do the hashing correctly.

App server is the backend server. We hash the requests and the data received from MongoDB by their index modulo n. The data and the write requests with the same remainder will be allocated to the same column of the cluster, and the subscriptions with the same remainder will be allocated to the same row of the cluster.

The nodes of the cluster are originally different computers to the storage data and match the update. Since we don't have enough hardware resources, we create the virtual cluster on software, and we use the resources of the CSIE workstation. For each node, we name it child and allocate a CPU resource and storage space to it.

4. EXPERIMENT

The target of implementing InvaliDB is to have good scalability. We design two experiments to test. One is for read scalability, and another is for write scalability. Also, we have tested for the relationship between the scalabilities and the cluster size. As for the implementing environment, the original paper implemented it on multiple hardware cloud computers, and each of the cluster nodes was installed on a computer; therefore, each of the nodes will not affect the others. However, since we don't have such a large number of computers, we then simplify the architecture of the cluster part. We combine the app server part and the cluster nodes part. Each of the cluster nodes is a child process of the app server. They communicate with each other by inter-process communication (IPC). This architecture was deployed on NTU CSIE workstation linux5, whose specification is x86_64 linux, with 24 Intel Xeon E5-2620 @ 2.00GHz cpus and 126G memory. As for the database part, we use the mongoDB cloud service.

For the two experiments, each of a write request contains a modification of one piece of data, and each of a read request contains a query of ten pieces of data.

In the experiment of read scalability, Client 1 fixes the frequency of write requests at 3 times per second, and Client 2 increases the frequency of read requests from 0 to 450 times per second, Client 3 pre-queries and pre-subscribes to 10000 pieces of data before the experiment starts, and we measure the latency of the read requests. Here, the latency of the read requests means the time interval between the app server receiving a request and the app server returning the result of the query. Also, we do the same test in different sizes of cluster.

The result is in Figure 5. Since the path of read needs to go through the database , awaiting for the results , and put results into the corresponding cluster nodes and subscribe them, and then return the data to client side. So the scalability is limited by the database, because each query has to wait in the queue of accessing the database. Therefore, the latency grows when the

frequency of read requests exceeds a threshold. However, we can observe that with the increasing size of the cluster, the threshold will also increase. This is because the latency of the path between the app server updates data to the cluster and the cluster returns the subscription of the data decreases as the cluster's size grows. More data can be parallely handled and returned. As you can see, the InvaliDB can improve the read scalability with more cluster nodes.

In Figure 5, the threshold gap between 4×4 nodes and 5×5 nodes is much smaller than the gap between 3×3 nodes and 4×4 nodes, and it is because of the limited CPU resource allocation, the benefit of parallel computing decreases as more threads are created. Another reason for the deviation result is the speed of network transmission. Since we measure the latency between the app server receiving requests and the app server returning data, the network access delay to the cloud mongoDB can vary. We've tried to solve this problem by using the local mongoDB database; however, the workstation doesn't have mongoDB and we don't have sudo privileges on installing it on the workstation.

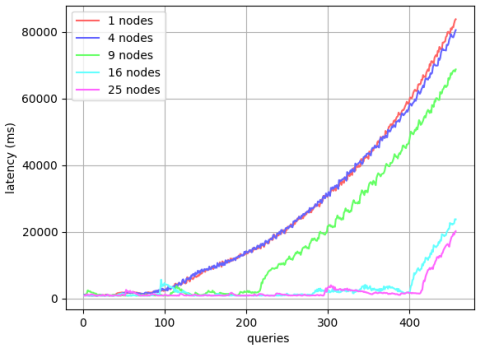

Figure 5. read scalability test

In the experiment of write scalability, we fix the frequency of read requests at 2 times per second, and increase the frequency of write requests from 0 to 450 times per second, and measure the latency of increasing write requests. Also, we do the same test in different sizes of cluster.

The result is in Figure 6. We can observe that the latency is always at some constant. It is because the app server asynchronously writes data to the database and the cluster, so it won't take time to wait for the write return. It represents that the method using clusters without accessing pull-based databases can provide data updates immediately. Thus, the InvaliDB can improve the write scalability.

In Figure 6, there are some high latencies occuring. We speculate that it is because there is network latency when connecting the workstation.

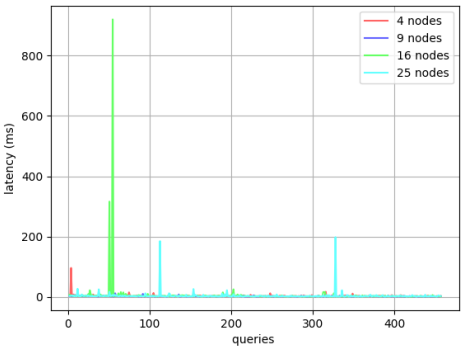

Figure 6. write scalability test

5. CONCLUSION

We implemented InvaliDB in a software method to overcome the read/write scalability issue of modern real-time databases. Despite the restriction of our testing environment, one can still easily learn from the experiment results that the larger the cluster grows, the better the system performs. In addition, thanks to the modular architecture, the part of MongoDB can be replaced by any other databases, in other words, all traditional pull-based databases can be treated as a real-time push-based database in our system. Furthermore, by keeping data synchronization on both sides of the cluster and the database, there is no limitation on any commands, users at the frontend can directly communicate with the database at the backend.

6. REFERENCES

Wolfram Wingerath, Felix Gessert, and Norbert Ritter. InvaliDB: Scalable Push-Based Real-Time Querieson Top of Pull-Based Databases (Extended). *PVLDB*, vol.13, no.12, pp.3032-3045, Aug, 2020

http://www.vldb.org/pvldb/vol13/p3032-wingerath.pdf

InvaliDB: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases – ICDE 2020, Dallas

https://www.youtube.com/watch?v=5N6zAa7zyfQ

7. DIVISION OF WORK

B07902048 李宥霆	前端 Client Coding、
B07902124 鄭世朋	MongoDB 資料、後
B07902014 蔡承濬	後端 Cluster Coding、
B07901043 沈信樺	投影片和 Report 主要