Exploring a distributed database prototype

Adil Rahman
Department of Computer Science
University of California, Santa Cruz
California, United States
anrahman@ucsc.edu

***Abstract* — Transitions from centralized to distributed databases have become a popular topic for research. This report seeks to implement a distributed database system that overcomes some issues that centralized systems face such as bottlenecking and scalability. Initial prototype implemented in Python was created with serial and concurrent locking mechanisms. Throughput for varying number of interconnected nodes ranged from 2,290-11,681 operations/sec. Minimum latency values for n=4 interconnected nodes with concurrent execution for 1,000 and 10,000 operations were 0.2643 and 2.6296 sec respectively. Further research will be performed in implementing the model on different programming language platforms.**

Keywords — database, distributed, node, serial, serializability, concurrency, lock, scalability, throughput, fault-tolerance, latency

I. Introduction

The need for sophisticated database architectures is highly increasing due to the exponential amount of data being generated. In addition to this, more requests are being made for data by the increasing user demand on databases utilized by various applications.

Many applications currently implement centralized systems in regard to their database. These database types do provide many advantages such as data integrity maximization, accuracy, and consistency. In certain areas, using a centralized model can be beneficial. Centralized like databases have been created via database virtualization techniques and are shown to reduce workloads for data collection in data mining [1]. Other applications that want to have more protection in terms of security also may choose to utilize centralized databases. System administrators can control and restrict user access more easily when all data is stored in one physical location, as seen in some applications such as smart cards [2].

However, some major problems that these architectures face include information leaking and inefficiently being able to find specific pieces of datum that is being requested per user [3]. If too many users are requesting various data from one centralized database, it can create a bottleneck effect and lead to suboptimal efficiency in processing and managing transactions. Centralized approaches also face problems in terms of data transmission of files or resources. These approaches are not realistic to implement in terms of scalability in applications that have a large number of end users / systems. This can easily reach total data transmissions in the GB scale for data per hour [4]. In addition to this, centralized systems are more prone to having downtime that can impact users if system failures or malicious attacks occur. Although data integrity was a defending point to this model, in the case that a centralized system's security is not up to par, the integrity of the entire database can potentially be compromised. Delta Airlines was subject to this, causing numerous flights to be cancelled and impacted the business severely [5].

Distributed databases differ from their centralized counterparts in that there are now a collection of databases that are located at different sites, rather than one physical location. There is a distributed database management system (DDMS) that can take care of query processing and structured data organization on top of the functionality that a standard database management system would take care of [6]. When users send transactions to be performed, the DDMS ensures that the transactions of different users that may have shared data in their relative transactions to be used do not conflict with each other. If data is shared amongst the transactions, a scheduler is responsible for ensuring the database is still correct after concurrent accesses are performed by multiple transactions. This ensures concurrency control for both data integrity and serializability.

There are different ways that a DDMS can enforce concurrency control. One successful mechanism is a locking mechanism. If two transactions were to conflict, the scheduler can make one transaction wait for the other transaction to perform its operations first before performing its own. Read and write locks are placed on a piece of datum, where if a write lock is present, no other transaction can obtain another write or read lock on the same datum, whereas a read locks can be obtained if other read locks are on the datum [7].

The distributed database architecture combats issues with bottlenecking. User's who request information can request from specific servers that have the data that they need, without having to go through a central authority. Additionally, total data transmissions are now reduced and distributed across a number of servers which can free up the potential load on a user's end. Lastly, distributed systems are very robust and fault tolerant when system failures or malicious attacks occur. Distributed databases have shown to provide increased reliability and performance as seen in the relational model of distributed Ingres [8].

The primary objective of this report is to develop a distributed database system that will use a locking mechanism to enforce concurrency control for transactions. The prototype will be implemented in Python.

II. Implementation

The application that was implemented has a basic form where any message can be sent between servers. Each server is represented as a physical node. The node is able to send and receive messages between other established connected nodes. Each node is responsible for specific pieces of data. If a node needs to access a specific piece of data not under its domain, it will communicate with the node that has that data under its domain and request it to perform the necessary transaction.

Normally, each transaction is made up of either all read operations or all write operations. In this application's case, each transaction consists of a single read or write operation. Each transaction receives a lock corresponding to a piece of datum before it performs its transaction. If a piece of datum is currently being worked on by transaction T_1, then any other transactions trying to read or write it must wait until T_1 has committed and completed its operation. The following sections describe the process in more detail.

A. Nodes

Let n represent the number of interconnected nodes. Each node represents a server, which performs both server and client roles. Fig. 1 shows a diagram of a network of interconnected nodes where each node serves as both a server and a client. Each node has one socket that is bound to a specific port that is used to listen for potential connections of other nodes. Every node is capable of connecting to other nodes and maintains a list of node addresses and ports that it is currently connected to. A node remains connected to any number of nodes after the initial connection has been established. If a node fails or drops out, each node that it was connected to is notified and each node updates its list of connections. Each node also keeps track of only its own partition of the data by setting a value representative of the threshold of the data that it is responsible for.

The client component of a node will request for a transaction to be performed and send it to the node that is responsible for fulfilling that request. The server component of the node responsible for fulfilling the request will then receive that transaction. The server component of this node has a transaction manager that will process the nature of the transaction and perform it. The transaction manager achieves this by breaking apart the message received and looking for keywords that indicate a transaction type that needs to be done. If the transaction manager does not recognize a particular transaction type from the received message, it will send a message back to the node that sent the message. This indicates to the client node that the message was unrecognizable, returns false, and nothing was done to its partition of the database. Once the transaction has been performed, the server node will send a message respective to the transaction type back to the client node that originally had the transaction request.

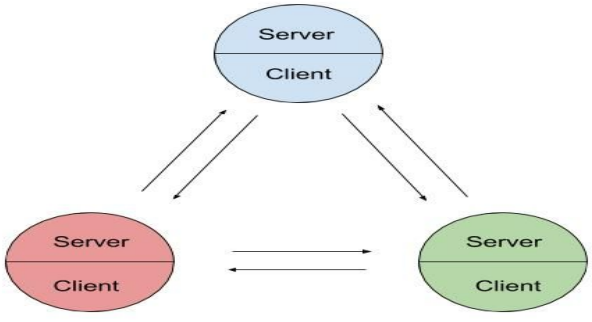

Fig. 1 Interconnected node network with server and client roles for each node where n = 3.

B. Physical distribution of data

The database is simulated through a key-value store where any node is responsible for 100 data items. This number can be increased or decreased accordingly to the owner's wishes. Each node initiates its own domain which is tied to the data that the node is responsible for. When a new node connects, it will receive information based off of the other existing nodes that it connects to for its domain responsibility of the data.

Let x represent the number of data items that a particular node N is responsible for. If n = 1, then $x_1 = 100$ and N_1 will have a domain of data items {0, 99}. If n = 2, then $x_1 = 100$, and $x_2 = x_1 + 100 = 200$ and N_1 will still have a domain of data items {0, 99} while N_2 will have a domain of data items {100, 199}. Thus when n = k, then $x_k = x_{k-1} + 100$ and N_k will have a domain of data items $\{x_{k-1}, x_k - 1\}$. In this way, data responsibility is distributed across any number of k nodes. Fig. 2 shows an illustrative diagram of how each node domain / responsibility appears.

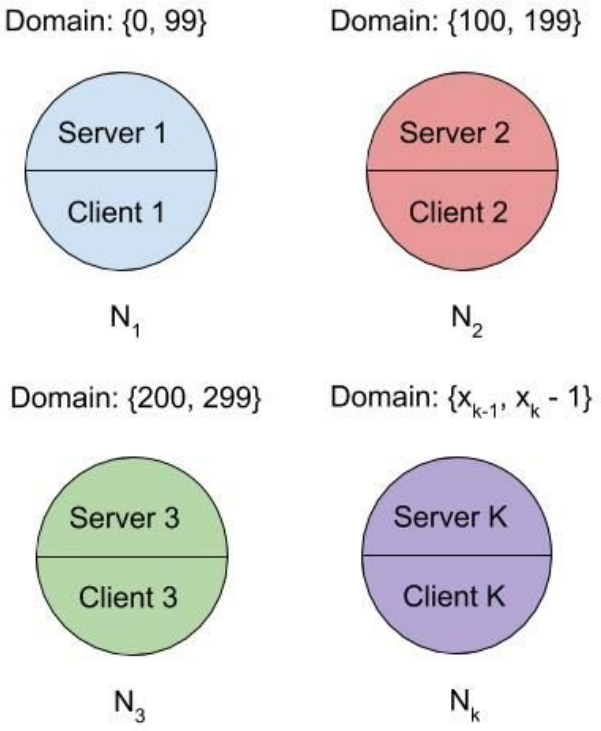

Fig. 2 Domain responsibility of n = 3 nodes and potential additional nodes.

C. Sending and receiving messages

Each node is capable of sending and receiving messages to other nodes. If n = 1, a node does not send or receive messages to other nodes, as there are no other nodes to perform these actions to. The single node will be able to perform operations on its own partition of the database

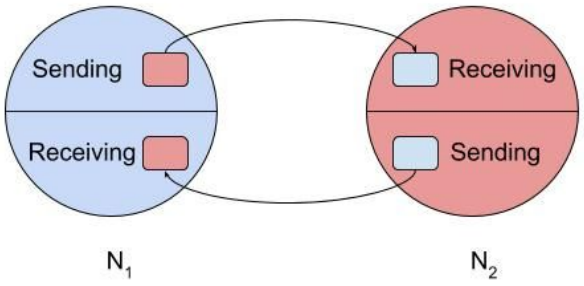

Fig. 3 Example of n = 2 nodes sending and receiving messages to each other. N_1 who is blue sends to red and N_2 who is red receives from blue. N_2 who is red sends to blue and N_1 who is blue receives from red.

freely. This node's partition of the database will be equivalent to the entire current database, as no other nodes exist to be responsible for other partitions in their respective domains. This single node will continue to listen for potential connections through its socket.

If n > 1, then a node will be able to send and receive messages to any node that it is connected to. Each node executes two threads, where the first thread is responsible for sending messages to other connected nodes, while the second thread is responsible for receiving messages from other connected nodes. Fig. 3 shows a diagram of two nodes sending and receiving messages via threads. Both of these threads loop continuously so that at any time any node including itself tries to perform a transaction, all nodes are able to attempt to perform it. If a client component of a node requests for a transaction to be performed, it will perform all of the operations within the transaction that it can do if those operations are affecting data that is in its own domain. It will then send any operations for data outside of its domain to the respective nodes that have the data in their domain. These nodes will then perform the operations and send a message back to the node who initiated the transaction to inform it that the operations were either committed or not.

D. Locking Mechanism

Before any transaction or operation is performed, a specific thread must obtain a lock on a particular data item. A list of locks is created for however many data items are present across n interconnected nodes. If there are n nodes, then there are also n number of partitions of the database. Each partition holds 100 data items, so the number of locks present at any time will be equal to 100n. Each lock and data item share the same key identity. If a transaction needs to perform an operation on a specific data item at <key, value>, then it will have to also obtain the lock corresponding to that key <key, lock>. Since the key identities for both lock and values are shared, the key value store and lock list exhibit a parallel characteristic. Fig. 4 shows the overarching parallel structure of each node's partition of the data and the corresponding locks.

Once a read or write operation needs to be performed, a thread will be created in the transaction that will obtain a lock for the data item it needs to read or write.

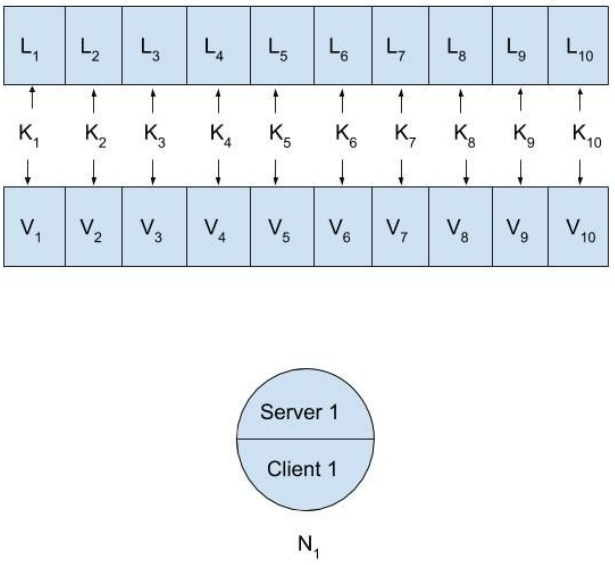

Fig. 4 Example of one nodes parallel structure of values and lock sharing the same key. Example only shows 10 lock and values, however it is the same structure for all 100 data items in a node's domain.

The thread will perform the operation and then release the lock. A scenario could occur where two or more transactions T_1 and T_2 are attempting to perform a read or write operation on the same data item. If T_1 obtains the lock first, then T_2 must wait for T_1 to commit its operation before T_2 can perform any operations on the specific datum. T_2 will indefinitely attempt to obtain the lock that T_1 currently has. Once T_1 has finished, it will release its lock, allowing T_2 to now obtain the lock and perform its operation.

E. Read and Write Operations

Each transaction will have either read or write operations. A read operation will take a key as input and return the corresponding value currently in the key value store. A write operation will take a key and a value that will overwrite the current value currently in the key value store. When a node performs a read operation that is outside its domain, the read operation request will be sent to the node who has the data item in their domain, perform the read operation, and send the value back to the node who requested for it. When a node performs a write operation that is outside its domain, the write operation request will be sent to the node who has the data item in their domain, perform the write operation, and send a message back to the node who requested for it telling them the write operation was successful.

III. Evaluation Study

The implementation above was tested for throughput, latency, scalability, and fault-tolerance. Tests were performed for three different scenarios. The first scenario runs the implementation with no locking mechanism implemented and is represented as read no locking (RNL) and write no locking (WNL). The second scenario implements a locking mechanism in which each operation is forced to act serially and is represented as read

test serial (RTS) and write test serial (WTS). The third scenario implements a locking mechanism that allows operations to act concurrently and is represented as read test concurrent (RTC) and write test concurrent (WTC). All scenarios were tested for read only and write only transactions. The following sections explore the results.

A. Throughput

Throughput was measured for how many read or write operations could be performed in one second for all scenarios with n = 1 to n = 4 interconnected nodes. The difference in the number of operations that are performed in one second at n = 1 is negligible. This is because a single node does not need to send any messages to other nodes which takes up more time. However as more nodes are interconnected, the number of operations that fall within the requesting node's domain decreases. As more and more nodes are involved in the reading and writing of various data items across several nodes domains, the number of operations per second decreases as more communication between other interconnected nodes is necessary in completing a transaction. Table I reflects the decreasing nature of throughput as n interconnected nodes increase.

TABLE I. Throughput

# of Nodes	Throughput (operations / sec)					
	RNL	RTS	RTC	WNL	WTS	WTC
n = 1	11,681	11,469	11,486	11,623	11,443	11,527
n = 2	3,741	3,437	3,631	4,951	4,492	4,909
n = 3	3,201	2,901	3,032	4,289	3,755	4,134
n = 4	2,885	2,290	2,725	3,794	2,760	3,761

B. *Latency*

Latency was measured for how long it took to perform 1,000 and 10,000 operations for all scenarios with n = 1 to n = 4 interconnected nodes. When only one node is present, the differences between no locking, serial execution, and concurrent execution of operations are negligible. However as you increase the amount of nodes interconnected to each other as well as increase the number of operations, the difference in latency is much more apparent, as seen in the tables below.

All scenarios that do not implement a locking system (RNL and WNL) take the least amount of time to perform. Since there is no lock to retrieve per data item, all threads doing read and write operations are free to perform their task without waiting for other threads that could be working on the same item.

All scenarios that implement a concurrent execution take slightly more time than the scenario of a no locking mechanism. In the concurrent execution, there may be instances where a thread th_2 is waiting to retrieve a lock that a pre-existing thread th_1 performing an operation on the same key currently has. This will increase the latency slightly as th_2 must wait for th_1 to release the lock so that it can acquire it. This scenario could occur where any p number of threads could all be trying to perform an operation on the same data item, in which the concurrency model will slow down.

Lastly in the serial execution, there is only one lock present and must be acquired before doing any operations whatsoever. This forces all threads to have to wait for the transaction before it to complete its operation before attempting to do their own. This drastically slows down latency. Tables II - V demonstrate the varying latencies of all scenarios as well as varying number of interconnected nodes.

TABLE II. Latency, n = 1

Number of Operations	Latency (sec)					
	RNL	RTS	RTC	WNL	WTS	WTC
1,000	0.0922	0.0935	0.0924	0.0891	0.0946	0.0943
10,000	0.8043	0.8131	0.8123	0.8163	0.8327	0.8317

TABLE III. Latency, n = 2

Number of Operations	Latency (sec)					
	RNL	RTS	RTC	WNL	WTS	WTC
1,000	0.2659	0.2812	0.2672	0.2026	0.2261	0.2033
10,000	2.6055	2.7567	2.6635	1.9426	2.1704	2.0080

TABLE IV. Latency, n = 3

Number of Operations	Latency (sec)					
	RNL	RTS	RTC	WNL	WTS	WTC
1,000	0.3031	0.3488	0.3171	0.2237	0.2626	0.2362
10,000	3.1392	4.7744	3.1596	2.2272	3.7824	2.3291

TABLE V. Latency, n = 4

Number of Operations	Latency (sec)					
	RNL	RTS	RTC	WNL	WTS	WTC
1,000	0.3489	0.3658	0.3510	0.2637	0.2744	0.2643
10,000	3.5833	4.9337	3.6785	2.5396	3.9570	2.6296

C. *Scalability*

The model has the capability to have an unlimited amount of nodes connect to each other and have their own partition of the data. In this implementation, each node was responsible for only 100 data items. However as seen in section A and B, with an increasing number of nodes, less operations can be performed because it takes more time to perform operations. This is due to read and write operations having to be sent to other nodes more frequently as the number of n interconnected nodes increase. If the domains of each individual node is small, then the likelihood a node will have to send an operation to another node is much higher.

Thus one way to make this model much more scalable is to increase the domain size of each individual node. In a real application setting, this number would be much higher than 100, and can be set to any number that is desired. By increasing the domain size, the likelihood that a

server would have to request operations to be done from another server is lower since there is a higher chance the operation it needs to do is already in its own domain. Depending on the application, data can be strategically placed in different servers so that users on the application level can retrieve information relevant to them much faster without having to rely on retrieval from other servers.

One example of this could be for a bank account database. If a user resides in Santa Cruz, California, then the user's account information would be stored to a server that is local to that site, either in Santa Cruz or closeby. The user would be able to retrieve and write information very quickly in regards to their own account without any other information needing to be accessed.

D. *Fault-Tolerance*

The current implementation has nodes keep track of which other nodes they are connected to. If a node is to run into system failures and become disconnected from other nodes, each node that is still connected is able to update its list of connections. All other nodes are still functional and are able to perform read and write operations for their own domains and request nodes that are still connected to perform operations on their respective domain.

However one side effect of a node failing is that its entire partition of the data will be completely lost. One way to combat this is to back up the data for each node in a separate physical entity. That way, when a fallen node is made available again, it can retrieve all of the information for data in its domain that it previously had.

Another side effect of direct partitioning of data is the scenario of a malicious attacker (MA). If a MA is able to edit information in the database wrongfully or take control of the node, there is no way to guarantee that the information being read from data in this node is accurate. A solution to this would to partially replicate data across different nodes. Instead of one node being only responsible for its domain, it would have partially overlapping domains with other nodes. If a malicious node were to be present, a consensus based decision making system could be put in place. The data the malicious node is presenting is compared to the same data that is present in overlapping portions in the other nodes. Then a consensus vote can be implemented to send the correct data where the information sent will be the value that is consistent and stored in the majority of nodes. In this scenario, there must be enough non-malicious nodes to outvote any amount of malicious nodes.

IV. Conclusion

The initial prototype of a lock mechanism driven distributed database network has been completed. This prototype does solve bottlenecking issues by partitioning data locally into servers that could cause latency issues in a centralized model. It also is successful and able to continue running the network if specific nodes fail, something a centralized database is prone to fail in.

The report does not address other potential issues that may be limiting throughput and increasing latency. Certain limitations may have been imposed by implementing the prototype in Python, which may sacrifice speed for its dynamic typing and interpreted language nature. Transitioning this model to C or Cython may provide performance increases due to these languages not relying on object oriented programming. Future extensions to the current prototype of this model and further research can be performed to further optimize the communication processes and transactions.

Acknowledgment

The author would like to acknowledge the support of Vishal Chakraborty for his help on portions of the programming.

References

[1] Y. Wada, Y. Watanabe, K. Syoubu, J. Sawamoto and T. Katoh, "Virtual Database Technology for Distributed Database," *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, Perth, WA, 2010, pp. 214-219.

[2] M. Mohandes, "A smart card management and application system," *2010 IEEE International Conference on Progress in Informatics and Computing*, Shanghai, 2010, pp. 1220-1225.

[3] M. Y. Jung and J. W. Jang, "Data management and searching system and method to provide increased security for IoT platform," *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju, 2017, pp. 873-878.

[4] M. B. Moss, "Comparing Centralized and Distributed Approaches for Operational Impact Analysis in Enterprise Systems," *2007 IEEE International Conference on Granular Computing (GRC 2007)*, Fremont, CA, 2007, pp. 765-765.

[5] cnn.com, 'Computer glitch grounds Delta flights', 2004. [Online]. Available: http://www.cnn.com/2004/TRAVEL/05/01/delta.delays/. [Accessed: 11- Mar - 2019].

[6] M. T. Ozsu and P. Valduriez, "Distributed database systems: where are we now?," in *Computer*, vol. 24, no. 8, pp. 68-78, Aug. 1991.

[7] B. Bhargava, "Concurrency control in database systems," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 3-16, Jan.-Feb. 1999.

[8] Stankovic, "A Perspective on Distributed Computer Systems," in *IEEE Transactions on Computers*, vol. C-33, no. 12, pp. 1102-1115, Dec. 198