

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 has become a popular topic for research. This report seeks to implement a distributed database system that overcomes issues that centralized systems face such as bottlenecks and scalability. The 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 and 206-292 tpmC. 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. This establishes a good baseline result for the database. Further research will be performed in implementing the model on different programming language platforms as well as more sophisticated hardware.

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

I. INTRODUCTION

The need for sophisticated database architectures is rapidly increasing due to the large 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 database systems in regard to their database. In certain areas, using a centralized model can be beneficial. These database types provide many advantages such as data integrity maximization, accuracy, and consistency. 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, major problems that these architectures face include information leaking and meeting user requests for specific pieces of data [3]. If too many users are requesting various data from one centralized database, it can create a bottleneck effect that leads 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

downtime that can impact users if system failures or malicious attacks occur. Although data integrity is a defending point to this model, we must consider 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. A distributed database management system (DDMS) must be present 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 involving shared data between users do not conflict with each other. If data is shared among 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 approach is a locking mechanism. If two transactions 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. If a write lock is present, no other transaction can obtain another write or read lock on the same datum. A read lock can be obtained if other read locks are on the datum [7].

The distributed database architecture combats issues with bottlenecks. Users can request information 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 multiple servers which can free up the potential load on a user's end. Lastly, distributed systems are 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 follows a basic communication interface where a message can be sent between any two servers. Each server is represented as a physical node that 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 that is not under its domain, it will communicate with the node that has that data under its domain and request that node to perform the necessary transaction.

For the proposed application, 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 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 listens for potential connections from other nodes using a socket bound to a specific port. 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. Each node also keeps track of only its own partition of the data which is determined by a predetermined threshold value.

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 and perform the transaction. The transaction manager achieves this by breaking apart the string encoded message received and looks for keywords that indicate a transaction type that needs to be done. If the transaction manager does not

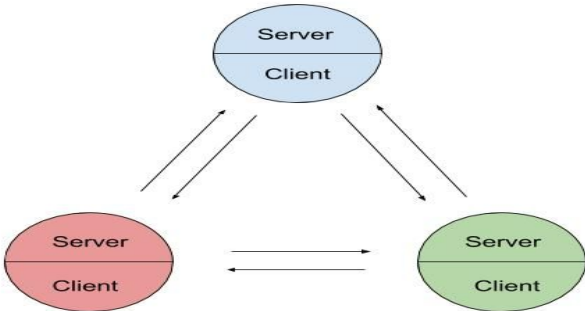


Fig. 1 Interconnected node network with server and client roles for each node where $n = 3$.

recognize a particular transaction type from the received message, it will indicate to the client node that the message was unrecognizable, returns false, and nothing was done to its partition of the database. If the transaction is valid, then the server performs it and sends a message respective to the transaction type back to the client node that originally had the transaction request.

B. Physical distribution of data

The database is simulated through a key-value store where the number of data items each node is responsible for is a parameter set by the user. Each node initiates its own domain which is tied to the data that the node is responsible for. Each time a new node connects, it determines the domain of data items it is responsible for based on the existing node's domains.

Let x_i represent the maximum threshold that a particular node N_i 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.

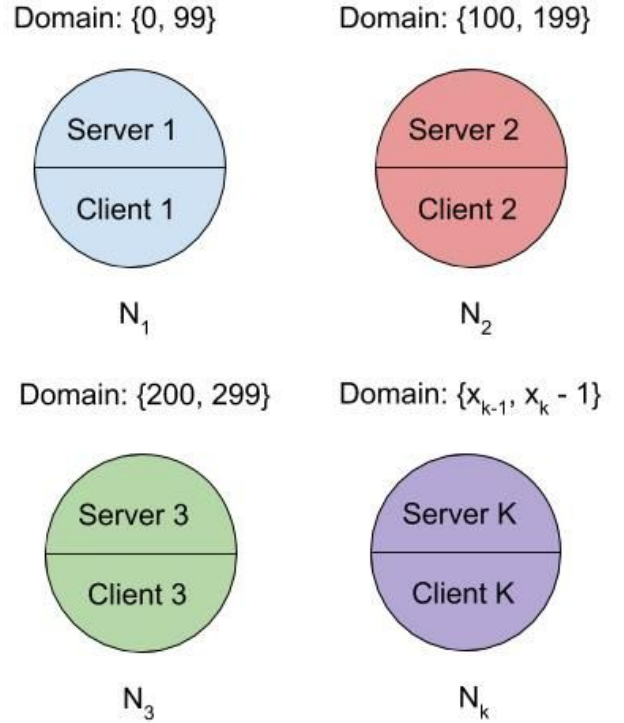


Fig. 2 Domain responsibility of $n = 3$ nodes and potential additional nodes.

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 it is the only node present. The single node will be able to perform operations on its own partition of the database

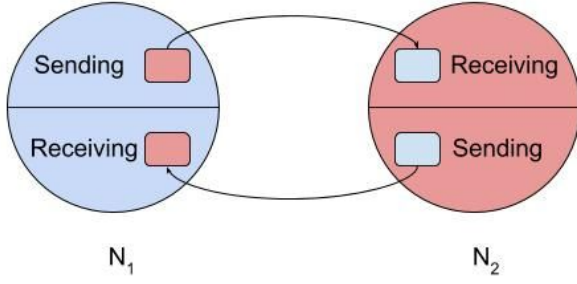


Fig. 3 Example of $n = 2$ nodes sending and receiving messages to each other. N_1 who is blue sends to red and N_2 who is red receives from blue. N_2 who is red sends to blue and N_1 who is blue receives from red.

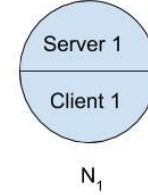
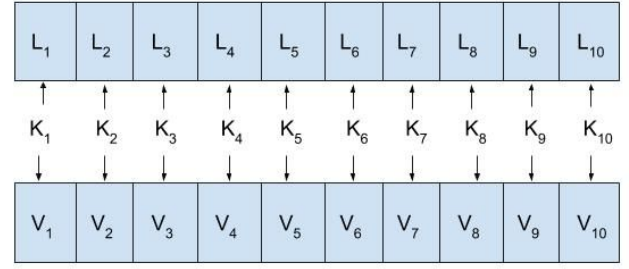
freely. This node's partition of the database will be equivalent to the entire current database.

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 if 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 in the transaction that affect data 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 maximum 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 a key and corresponding value, then it will have to also obtain the lock corresponding to that key. 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 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.



will obtain a lock for the data item it needs to read or write. The thread will perform the operation and then release the lock. The scenario where two or more transactions T_1 and T_2 are attempting to perform a read or write operation on the same data item is possible. 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 its domain, perform the read operation, and send the value back to the node who requested 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. It will then perform the write operation and send a message that the write was successful to the requester node.

F. Benchmark Evaluation

There is a need to test the distributed database prototype on a workload representing a more complex application environment in addition to the basic read and write operations mentioned above. The TPC Benchmark™ C (TPC-C) is characterized as an emulated online transactional processing (OLTP) workload that is considered the standard for evaluating database performance. TPC-C consists of five transaction profiles that consists of a mix of read-only and update-intensive transactions. These profiles are performed on a collection of nine relational tables.

The TPC-C benchmark represents a wholesale supplier company where the TPC-C database maintains information for nine separate and individual tables. These

tables contain various information such as unique IDs, variable text, fixed text, date and time, and signed and unsigned numerics. Each terminal server will represent a warehouse, where each warehouse is responsible for ten districts under it. Each of those districts serve three thousand customers. Each server has its own tables to keep track of their items and stock, as well as history for new orders. In our server client model, each server node is also acting as a client customer, where a given transaction profile can be executed. When a transaction profile is requested to be performed by the client portion, the server portion processes and performs the transaction and ultimately returns the result to the client portion.

The TPC-C benchmark consists of five transaction profiles. The *New-Order* transaction represents a medium intensity read and write transaction where a customer completes a new order for some items. The *Payment* transaction represents a light intensity read and write transaction where the balance information will be updated for a given customer based on a set of purchased items. The *Order-Status* transaction represents a medium read-only transaction in which a customer can query the status of their last order. The *Delivery* transaction represents a batch of read write transactions where new orders that are within a queue are processed. The *Stock-Level* transaction represents a heavy read only transaction where the stock levels that are below a specified threshold are identified.

In our model, each node maps to a single warehouse. The transaction input parameters randomly chooses whether to request a resource in the home warehouse or a remote warehouse based on a non-uniform probability distribution that is biased towards the home warehouse. This will make it such that a given server warehouse will need to request data that is located under another server's domain. The performance metric that is reported through this workload is the measurement of the number of orders processed per minute, which is expressed as transactions-per-minute-C (tpmC).

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. Lastly, the TPC-C benchmark implementation is run on varying number of n interconnected nodes and the throughput of TPC-C transaction profiles is obtained. 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 into 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 still be 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. If the randomization of input parameters happens to include portions of data that were within the lost partition, the nodes will also fail. One way to combat this is to back up the data for each node in a separate physical entity. A node would then try to send a message to the fallen node. If after some time frame there is no response from the fallen node, the node that needs the data would instead access the separate physical entity that has the data. In addition to this, 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.

E. TPC-C

The tpmC was measured with $n = 1$ to $n = 4$ interconnected nodes. Unlike the throughput reported in Part A, the tpmC does not change as much as the number of interconnected nodes increases. This is due to the individual nature of the transaction profiles in whether they are completed at home or remotely. Any transaction profiles that require data to be requested from a remote warehouse will add additional time required to complete the transaction. The amount of time spent sending requests for messages and receiving the data back will decrease the amount of available time to perform additional transactions.

The Order-Status transaction profile is always processed within the home warehouse and as a result does not cause any change in time required to process due to sending data requests from other warehouses. The Delivery transaction profile is a deferred queue that also is processed based off of the home warehouse. These two transaction profiles will not effect the tpmC throughput as the number of nodes increases.

The New-Order transaction profile only has a 1% chance that a remote warehouse ID will be generated as input. In most cases this transaction profile will be completely performed on the partition of the data of the server that called it and not be requested elsewhere. The Payment transaction profile has a 15% chance where a remote warehouse will process the payment of the customer. Lastly the Stock-Level transaction will occur for a randomly chosen warehouse, and may cause some added latencies in retrieving the data.

Table VI shows the tpmC for $n = 1$ to $n = 4$ nodes. When there is only one node present, the business throughput is seen at its highest at 292 tpmC. The difference in tpmC is minimal however once you have any number of interconnected nodes greater than one. When a transaction that requires remote data is performed, it will at most only have to send a message to one other server, and then retrieve data from that same server. This implies that regardless of the number of nodes, there will only be a slow down in terms of business throughput as long as the number of nodes is greater than one. This can be seen as the business throughput for $n = 2$ to $n = 4$ nodes is practically the same.

However as the number of interconnected nodes increases, the amount of time it takes to search for the proper node to request data from will increase. Another time penalty will incur once the node that has gathered the data needs to send the data back to the node that requested the data. A gradual decrease in business throughput is expected as the search space of connected nodes increases.

TABLE VI. TPC-C Results

# of Nodes	Business Throughput (tpmC)
$n = 1$	292
$n = 2$	212
$n = 3$	211
$n = 4$	206

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. The prototype also is successful and able to continue running the network if specific nodes fail, unlike a centralized model that only consists of one node.

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. In addition to this, future scalability tests will need to be performed on more sophisticated hardware such as a UNIX cluster. 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. Ozu 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. 1984.
- [9] Transaction Processing Performance Council, "TPC Benchmark C, Standard Specification, Revision 5.11.0", Feb. 2010.