The State and Progress of Distributed Database Systems

David Nguyen
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
davidng0123@live.com

ABSTRACT

We are currently observing immeasurable growth in both the amount of data we are producing and storing. Despite that, most enterprise storage systems are still using decades-old technologies that will not keep up with this growth in the coming years. As big data, cloud storage, and the "Internet of Things" proliferates, we are now challenged with discovering better ways to store and access such data. Distributed database systems (DDBS) are in a position to help solve this looming problem. Due to the distributed nature of such systems, DDBS enjoy the following advantages: improved reliability, higher availability, economical scalability, and excellent performance. Generally speaking, these allow for excellent data insertion and simple data access performance. However, DDBS are highly complex and are challenged to provide many features that centralized database systems possess. Features that DDBS have difficulty with include: replication, query optimization, concurrency control, and data consistency across the entire distributed system. As such, operations such as complex data retrieval and manipulation can be costly to perform, especially in high concurrency situation. Much database related research is being invested in minimizing these problems. Despite these disadvantages, commercially available distributed database systems are emerging and are enjoying relative success in the database market, such as: Cassandra, Clusterpoint, and FoundationDB. Note that each individual DDBS may be highly specialized, implement very different architectures, and sacrifice various features to achieve the performance and cost-effectiveness they possess. Every enterprising company or database administrator that wants to use the newest database technology in their business should be aware of the current state and progress of these distributed database systems.

1. INTRODUCTION

If one were to examine the history of the database system, they would notice that databases have dramatically evolved since the 1960s, when businesses were just installing their first computer systems and data storage was still in its infancy. The 60s and 70s are marked by the development of new data storage models. The first ever DBMS, General Electric's Integrated Data Store (IDS), was created in 1961 and was the first network database system [4]. IBM quickly followed suit and created the Information Management System (IMS) in 1966, which introduced the first hierarchical database system. In the early 1970s, IBM's Tedd Codd released a series of research papers detailing the relational data model. In a race to develop the first relational database system, IBM created SQL/DS in 1981 and Software Development Laboratories (now named Oracle) created Oracle Version 1in 1978. The 80s and 90s are marked by the rapid growth of relational database systems, the development of database standards, and an explosion in research. More specifically, these mark the "emergence of commercial relational DBMS (DB2, Oracle, Sybase, Informix, etc)" [4]. By the early 1990s, new and database-dependent applications were emerging and "DBMS features for spatial, temporal, and multimedia data … [and] standards for data query and exchange" [4] allowed our data-driven and data-hungry world to advance.

During that entire time, database systems more or less operated in a serial fashion and businesses had to maintain their databases in single, large mainframe computers. However, researchers quickly discovered that "relational [database] queries are ideally suited to parallel execution; [consisting] of uniform operations applied to uniform streams of data" [2] in the early 1990s. Of course, relational database developers began to take advantage of parallel architectures. These architectures allowed for incremental speedup and scalability, because they allowed the usage of "massive numbers of fast-cheap commodity disks, processors, and memories promised by current technology forecasts" [2]. A particular question naturally followed: can we separate these arrays of disks, processors, and memories by large physical distances? In a classical parallel database setup, a system's processors and storage arrays are likely to be in the same building or room. However, each processor or storage box can communicate just as well if they are separated by large distances, which leads us to distributed database systems.

2. DEFINITION

Dr. M. Tamer Oszu, a distributed database researcher, defines a distributed database as a "collection of multiple, logically interrelated databases distributed over a computer network [and] a distributed database management system is the software system that … makes the distribution transparent to the users" [11]. One can imagine many interconnected nodes, each of which has some form of database capability and can communicate with one another. This rather loose definition of a distributed database allows for many possible, implementable architectural models.

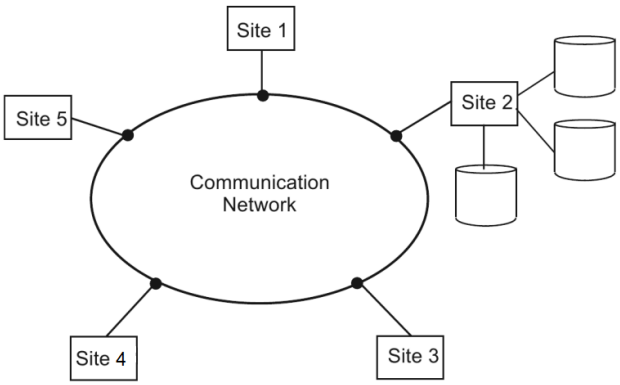

Figure 1. Not an example of a distributed database.

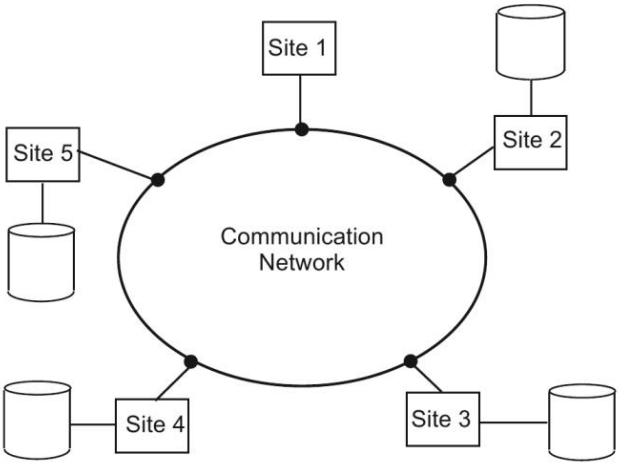

Figure 2. An example of a distributed database.

2.1 Node Properties

Distributed databases can be categorized and compared using the following three properties of each individual node: their autonomy, their distribution, and their heterogeneity. In this context, autonomy "indicates the degree to which individual DBMSs can operate independently" [11]. Each database node can be *tightly integrated*, where all the distributed data is presented to users as a single image, regardless of which node requested data. On the other hand, each database node can be semi-autonomous, where each node can operate independently but has the ability to share their local data. Another property to consider is a database's distribution, which refers to the level of control of data and functions each node possesses. Distributed databases can have a client/server distribution, in which a node can either be a client (that requests or inserts data) or a server (that serves and manages data). However, a database may have a peer-to-peer distribution, where a node can both be a client and server and every single node has the same capabilities. The last property deals with a database system's heterogeneity, which are basically the hardware and software differences between nodes in a database. In a highly heterogeneous distributed database, each node has a wide variation of hardware (hard drives, I/O, etc) and software (operating system, database version, etc). In a non-heterogeneous – or homogeneous – system, each node is practically identical in hardware and software.

2.2 Data Properties

Distributed databases can also be characterized on their data delivery methods and the following three dimensions can describe them: data delivery mode, delivery frequency, and communication method [11]. The data delivery modes include: pull-only, push-only, and hybrid. In pull-only mode, data transfer is initiated by the querying client. In push-only mode, data transfer is started by a server push and is broadcast to random or specific sets of nodes. Of course, hybrid mode combines pull-only and push-only modes. Delivery frequency is another characteristic that can be described as being: periodic, conditional, or ad-hoc. In periodic delivery frequency, data is transferred at scheduled intervals. In conditional frequencies, data is delivered whenever certain client and server conditions are met. In ad-hoc delivery frequency, data is only transferred when clients explicitly request it. The third and final data delivery characteristic is communication method, which can either be unicast or one-to-many. In the unicast method, one server communicates with one client at a time. In the one-to-many method, one server broadcasts to multiple nodes at a time.

3. ADVANTAGES

3.1 Expandability

Numerous advantages arise when a database and its data is distributed across a wide area. One such advantage is that these databases make for easy expansion and massive scalability. In the past, businesses had to maintain centralized, powerful mainframe servers that contained and managed all of their data. Expanding or updating this system usually meant either buying a brand new server or taking an outage to upgrade specific parts. Furthermore, the physical size of these databases were typically limited to how much available space a server room had. Today's mainstream parallel databases have similar problems, even if they utilize commodity hardware in greater numbers. However, in distributed databases, "major system overhauls are seldom necessary [and] expansion can usually be handled by adding processing and storage power to the network" [11]. In this situation, very little setup is required and new database nodes should immediately interoperate with the rest of the database. And since distributed databases can be highly heterogeneous, database nodes are allowed to greatly vary in terms of hardware and software. This is not unlike adding more processing cores or storage space in a typical desktop computer. In terms of scalability, companies will no longer be limited by how much available space there is in a server room, company campus, or local region. This is a boon for smaller companies that operate worldwide, use cloud storage, and/or don't want to be limited to one storage provider.

3.2 Availability

The ease of expansion, massive scalability, and distributed nature of this type of database all contribute to another advantage: extraordinarily high availability. This is primarily achieved through data replication, where "each logical data item has a number of physical instances" [10]. However, it is important to note that data consistency and replication performance is sensitive to whatever replication techniques are utilized. Most – if not all – current replication schemes can be categorized as either lazy or eager update. Eager update, or synchronous replication, is when "read and write operations of transactions are synchronized with respect to other operations in the entire [database] network" [6]. In this scheme, an update transaction immediately modifies every physical instance of a logical data item across the entire distributed database. This method offers absolute, or *tight*, consistency following a

transaction commit, but will negatively affect transaction performance when the database must update every data item's physical instance before commit completion. Either way, this may be useful for mission critical applications, such as banking applications or other finance systems that have strict consistency requirements. Users who are willing to sacrifice consistency in favor of better performance may opt for a more delayed replication.

Lazy update, or asynchronous replication, is when "transactions... are executed on the local replica [or local node and] the effects of locally executed transactions are propagated lazily to all copies in the system" [6]. While the term "lazy" is up for interpretation by database developers, the general idea is the same. Updates are committed on one data item's physical instance and the changes are saved away into a log or history. When an opportunity arises (i.e. the node is not very busy) or when replication is scheduled, the saved changes are propagated to other nodes. Note that these other nodes may also lazily propagate previously propagated changes. This method may be useful for content delivery networks (CDNs) or other systems that seldom update existing data. In any case, data replication makes for excellent availability in distributed databases and makes unplanned downtime much less of a problem.

3.3 Query Performance

Query processing and optimization in distributed databases exploit the power that many nodes possess altogether, which results in a significant advantage in performance and parallelism. To achieve this, distributed databases borrow from and extend upon the techniques that centralized DBMSs have been perfecting for decades. For centralized DBMSs, the "process typically involves two steps: query decomposition and query optimization" [10]. In query decomposition, a user's query is broken down, rewritten, and translated into machine-understandable, algebraic queries. Given the state of the data (how it is partitioned, if it is indexed, etc), query optimization understands that there are many algebraic queries that end in the same result and that some queries perform better than others. Query optimization attempts to calculate the processing cost of these possible queries and selects the one with least cost. The user's query is then processed based on the selected *plans* and hopefully the query result is returned within acceptable performance metrics.

Distributed databases typically expand query processing and optimization into four steps instead of two in order to account for and exploit their distance-separated nodes. The steps are: (1) query decomposition, (2) data localization, (3) global optimization, and (4) local optimization [8]. The first three steps are typically handled by a *control site*, which is usually the querying node or one better suited to coordinating many nodes at a time. Query decomposition operates the same as in centralized DBMSs: they convert user queries to algebraic queries. Data localization serves to "localize the query's data using data distribution information... and the query is transformed into one that operates on fragments rather than global relations" [10]. In this step, algebraic queries are transformed and organized into ones that individual database nodes can process independently of other nodes. These fragment queries are passed to the global optimizer, which considers the distributed nature of the data and nodes in order to generate and select the cheapest query processing plan to execute. The control site then sends the fragments to other database nodes, or *local sites*, for final processing. Each local site optimizes its fragment via local optimization, which executes the best fragment processing plan based on the state of the local data. Back at the control site, the results from each local site are compiled and processed further (data is joined based on global relations, etc) to create the final result that will be returned to the user.

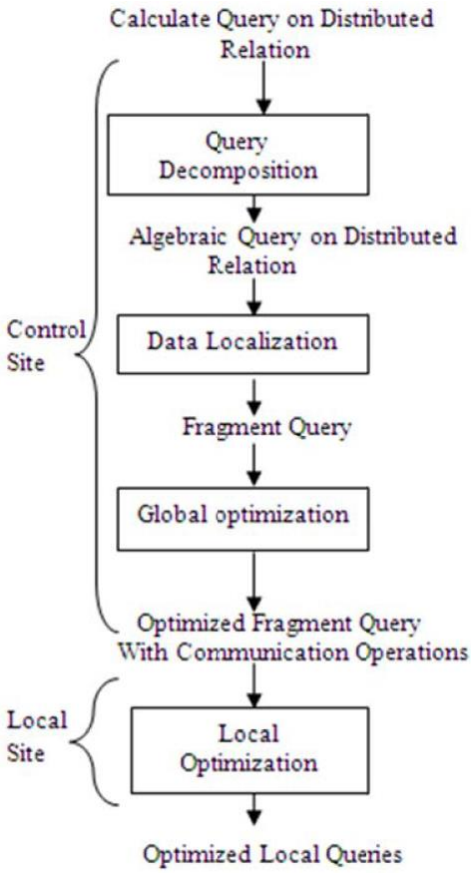

Figure 3. Distributed query optimization.

4. DISADVANTAGES

4.1 Consistency

However, it is important to note that distributed databases are not the de facto standard for data storage and management just yet. While they provide desirable advantages – such as better expandability, availability, and query performance – there are significant tradeoffs to be made. One disadvantage that distributed databases currently possess is the notion of loose consistency and deviation from traditional transaction commit. Earlier we discussed data replication and how synchronous replication allows for tight consistency. In real world applications, it turns out that true tight consistency is nigh impossible and that many distributed database administrators opt for some degree of loose consistency for practicality. Loose consistency allows "some of the [physical] copies [of a data item] to be inconsistent for some time [and] provides better responsiveness since the waiting operations associated with multi-site commit protocols are avoided" [14]. This means that following a commit for a data item update, queries on that data item may utilize an inconsistent copy and return outdated data. While not ideal, this is a tradeoff that many database administrators accept if their user requirements allow it. For users that cannot tolerate any loose consistency, then a centralized database would be the better approach for data storage and management.

The fact that distributed databases can possess varying levels of consistency mean that the database commit – the mechanism used to bring a database from a consistent state to another consistent state – has also changed. In centralized DBMSs, a transaction commit typically refers to two-phase commit. Two-phase commit is where a "transaction is not considered complete unless all of the programs involved complete their executions successfully" [16]. Obviously, this method consists of two phases: the prepare phase and the *commit or rollback* phase. In the prepare phase, all of the programs involved in an execution signal a transaction manager that they are ready to either commit or rollback changes. In the commit or rollback phase, the transaction manager signals all of the programs to rollback if there was an error or commit otherwise. In the context of a centralized database, changes to a logical data item are not visible to any queries until the system completes commit. For distributed databases, that would mean every single node on a database network would have to synchronously prepare and commit or rollback. This is extraordinarily costly and impractical because it would require that all involved nodes be online and connected to the network. And even if this was guaranteed, the communication overhead required to synchronize all of the operations may be too expensive and detriment overall performance. Distributed databases typically employ some other form of multi-phase commit protocol in lieu of the demanding two-phase commit. Again, this is a tradeoff that many administrators accept in favor of performance and practicality; however, they must be aware of what level of consistency and type of commit they're using in these distributed databases.

4.2 Concurrency & Deadlock

Supposing that a user set up a distributed database with very tight consistency and a close-to-two-phase commit protocol, this person would have to deal with significant concurrency and deadlock problems. The locking mechanism employed would have to locate all the data items and/or physical instances required and lock them. However, it is possible that "the resources that a transaction needs are potentially stored at different sites on different database servers [and lock] preacquisition will take a longer amount of time to complete due to communication delays" [15]. In a system with many transactions running concurrently, some transactions would have to wait for preceding ones to acquire global locks, commit their transactions, and release their locks. These exceptionally long wait times could result in unacceptable commit times or even transaction starvation. In addition to long waits and transaction starvation, complete blocking is also possible. Blocking is when a local or – worse – control site fails during a distributed transaction, especially after all participating sites have just acquired their locks [15]. If a local site fails, the entire system or network has to determine whether to continue with the transaction or release all locks, both options would take considerable time. If a control site fails, all local sites may end up waiting for a commit signal and eventually timing out. While promising protocols have been suggested to solve the concurrency and deadlock problems, we have yet to reach a satisfactory resolution that would make distributed databases the next data storage and management solution.

5. CASE STUDIES

5.1 Google Spanner

It may not come as a surprise that Google Inc. designed, built, and uses a distributed database called Spanner. Google is a worldwide company that needs to reach their audience anytime and anywhere, so the advantages that distributed databases possess make Spanner a perfect fit within the company. It specializes with two main features, the first of which allows "replication configurations for data [to] be dynamically controlled at fine grain by applications" [7]. To be more specific, it allows users to control options such as: data control per node group, data-to-user distances, data-to-replica distances, and replication factor. In most other distributed databases, only the system administrator has control over replication configuration and application programmers must request configuration changes. The second feature that sets Spanner apart is that it provides "externally consistent reads and writes, and globally consistent reads across the database" [7], which is typically extremely difficult to implement in distributed databases. This is achieved by assigning transaction commits accurate and meaningful timestamps. These timestamps are used to reflect serialization order and is in part guaranteed by Spanner's TrueTime API, which "directly exposes clock uncertainty" [7]. As with many other databases, Spanner has a rather unique server node setup and naming scheme. A Spanner deployment is grouped into zones of many nodes [7]. Each zone consists of one *zonemaster*, one *location proxy*, and all remaining nodes are *spanservers*. The zonemaster is responsible for assigning data to spanservers, which are responsible for serving data to clients. The location proxy is meant to help clients find spanservers to request data from. One whole Spanner deployment has one *universe master*, which compiles zone information, and one *placement driver*, which performs data load balancing. At its core, Spanner's data model resembles a key-value store wrapped in SQL-like semantics. Using syntax almost identical to SQL, data is inserted and returned in table and row form. However, each row is internally a key-value pair in the sense that users must specify a number of table columns to be unique and this *unique key* is mapped to the rest of a record's values. This unique key is used to determine data placement and replica locations at desired spanservers and zones.

5.2 RethinkDB (Open-Source)

Open-source distributed databases are quickly becoming popular and RethinkDB is one that currently stands out. Its ease of use, simple data & programming model, and understandable architecture make it popular for those interested in trying a distributed database or just distributing data. RethinkDB excels at ease of use, tight data consistency, and excellent query parallelism with table joins. First, while many databases mainly support system administration through command line interfaces and installable tools, RethinkDB can be managed via an easy-to-use web application featuring modern UI design elements. Complex database operations such as manual load balancing can be triggered with the slide of a scale and a click of a button. Database status and performance can be monitored in real time via interactive graphs in the same web application. Second, RethinkDB highly favors data consistency and guarantees it in most use cases. In order to achieve such tight consistency, the database will tend to sacrifice atomicity and write availability. In short, users may receive errors when attempting to commit update transactions during high concurrency situations. Finally, RethinkDB welcomes complex and intensive table join operations while most other distributed databases avoid them for performance reasons. This is achieved through complex and efficient query optimization. Queries are broken down into a stack of nodes, where the "bottom-most node of the stack usually deals with data access [and] nodes closer to the top usually perform transformations on the data" [17]. Machines in the cluster will

operate on the stack from top to bottom, with each machine working on one node and combining their results from bottom to top. RethinkDB is a NoSQL database that is built to store JSON documents, which contributes to the system's ease of use by allowing flexible database design and allows the query language to be simply embeddable in application code. RethinkDB's query language, ReQL, facilitates query construction via function calls in supported languages and allows users to chain multiple database commands together in one line of code [17].

5.3 Apache Cassandra

Apache Cassandra is an open-source NoSQL database that is currently popular with enterprise companies, especially those interested in performing analytics. The database was invented by Facebook, who wanted to "solve that company's inbox search problem, in which they had to deal with large volumes of data in a way that was difficult to scale with traditional methods" [3]. Cassandra specializes with decentralization, extraordinary flexibility, and excellent data write performance. Cassandra is decentralized due to its pure peer-to-peer distribution: all nodes are the exact same and they possess equal control over data. There are no master-slave relationships that many other distributed systems use to manage the entire system. This produces two key advantages: This database is highly available because the system does not depend on any particular nodes and it is linearly scalable because there is very little concentrated overhead in maintaining increasing node numbers. Cassandra also specializes in high flexibility in that system administrators can fine tune database consistency and replication levels. The consistency level "allows [users] to to easily decide the level of consistency [they] require, in balance with the level of availability … the replication factor lets [users] decide how much [they] want to pay in performance to gain more consistency" [3]. Basically, administrators can choose the tradeoff between availability, performance, consistency, and replication in one product – they do not have to consider the tradeoffs in a comparison of many distributed databases. Finally, Cassandra is optimized for excellent data write performance and may tend to prioritize it over read performance in high concurrency situations. This feature is meant for applications and systems that write more than they read, such as those involving "user activity updates, social network usage, recommendations/reviews, and application statistics" [3]. This is a boon for Internet websites that rely on crowdsourcing and user-generated content. While data tends to be stored for a long time – if not indefinitely – data is often actively accessed and read for only a short period of time after it has been created, such is the nature of today's web application revolution. Apache Cassandra's data model is similar to that of Google Spanner's: data is written and read in SQL, but the database is internally NoSQL. Data is inserted and retrieved as key-value pairs, with the keys defined on a table's unique columns.

6. SUMMARY/CONCLUSION

Of course, much database research is being devoted to improving or lessening the disadvantages that distributed databases generally possess. First and foremost, many recent papers have proposed new transaction commit protocols that look to improve the performance of transactions in tight consistency systems. Some attempt to go further by trying to detect and resolve *consistency conflicts* during transactions or after-the-fact. Secondly, research has also been devoted into making distributed table joins much more efficient. The distributed nature of these databases do not make table joins a cheap operation. However, complex enterprise-level applications tend to perform multi-way and/or nested table joins in very few queries. Although the same problem is more or less present in centralized databases, join performance is admittedly worse in distributed systems. Finally, a notable portion of distributed database research have been on new data models. Models range from those currently in use by centralized databases, such as graph-based models, to completely original and new ones, such as spatial models.

To conclude, the emergence of distributed databases in commercial and enterprise applications shows that users are willing to tolerate the aforementioned disadvantages in favor of excellent availability, scalability, and query performance. The fact that giants such as Apache, Google, and the general open-source community are throwing their support behind distributed databases may mark this as the next, go-to model for data storage. Of course, distributed databases have a long way to go in terms of mitigating its disadvantages and developing better architectures for simpler, effective usage. The author of this paper hopes that you – the reader – have gained valuable knowledge about the current state of distributed databases and that you may consider using them for your next software-related endeavor.

7. REFERENCES

[1] A. Danielsen. Class Lecture, Topic: "The evolution of data models and approaches to persistence in database systems". *Department of Informatics, University of Oslo*, Oslo, Norway. May 1998.

[2] D. DeWitt, & J. Gray. "Parallel database systems: The future of high performance database systems". *Communications of the ACM*, vol. 25, no. 6, pp. 85-98, Jun. 1992.

[3] E. Hewitt. *Cassandra: The Definitive Guide*, 1st ed. Sebastopol, CA: O'Reilly Media, 2011, pp. 1-28.

[4] F.L. Moore. "Database systems: A brief timeline". Unpublished.

[5] H.H. Darji, B. Shah & M.K. Jaiswal. "Concepts of distributed and parallel database". *International Journal of Computer Science and Information Technology & Security*, vol. 2, no. 6, Dec. 2012.

[6] J. Holliday, D. Agrawal, & A. Abbadi. "Database replication: If you must be lazy, be consistent". *Proceedings of the Eighteenth Symposium on Reliable Distributed Systems*. pp. 304-305, 1999.

[7] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford. "Spanner: Google's globally distributed database". *ACM Transactions on Computer Systems*. Vol. 31, no. 3, pp. 8:2-8:22, Aug. 2013.

[8] K. Jamsutkar, V. Patil, & B. Meshram. "Query processing strategies in distributed database". *Journal of Engineering, Computers, & Applied Sciences*, vol. 2, no. 7, pp. 71-77, Jul. 2013.

[9] M.T. Ozsu. "Distributed databases". Unpublished.

[10] M.T. Ozsu, P. Valduriez. "Distributed and parallel database systems". *ACM Computing Surveys*, vol. 28, no. 1, Mar. 1996.

[11] M.T. Ozsu, P. Valduriez. *Principles of Distributed Database Systems*, 3rd ed. New York, NY: Springer, 2011, pp. 1-125.

[12] M. Wiesmann, F. Pedone, A. Sciper, B. Kemme & G. Alonso. "Database replication techniques: A three parameter classification". *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pp. 206-215, Oct. 2000.

[13] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, & G. Alonso. "Understanding replication in databases and distributed systems". *Proceedings of the 20th International Conference on Distributed Computing Systems*. pp. 464-487, 2000.

[14] P. Chundi, D.J. Rosenkrantz, & S.S. Ravi. "Deferred updates and data placement in distributed databases". Data Engineering, 1996. *Proceedings of the Twelfth International Conference* on. pp.469-476, Mar. 1996. doi: 10.1109/ICDE.1996.492196

[15] S.K. Rahimi, F.S. Haug. *Distributed Database Management Systems: A Practical Approach*. Hoboken, New Jersey: John Wiley & Sons, 2010, pp. 254.

[16] "Two-phase commit". Internet: http://msdn.microsoft.com/en-us/library/aa754091.aspx. [Oct 29, 2014].

[17] "RethinkDB FAQ for programmers new to distributed databases". Internet: http://www.rethinkdb.com/docs/architecture/. [Nov 1, 2014].