TippyDB: Geographically-Aware Distributed NoSQL Key-Value Store

[1]Iskandar Setiadi, [2]Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
[1]iskandarsetiadi@students.itb.ac.id, [2]imam@informatika.org

Abstract— **The need of scalable databases has raised rapidly in these past few years. Researchers and developers have tried to develop several techniques to create replications and partitions in non-relational databases. However, high latency between distributed data centers has imposed a costly bottleneck to the system's performance. In this paper, we present a system design for non-relational database – TippyDB – that provides better response time for users who are accessing our service. We enhance our system with the important observation that interactions between people almost always happen in a limited geographical area. We demonstrate that, in contrast with other NoSQL databases, our system offers a considerable improvement of performance in a distributed environment.**

Distributed Systems, TippyDB, Database Design, NoSQL, Key-Value Store, Geolocation, Raft, MongoDB

I. Introduction

Data play an important value in our daily life, such as economics, politics, educations, and other fields. In the past, SQL databases are widely used to support daily activities in storing data. As the needs of scalable storage increased, the strict relational schema in SQL databases is too costly for distributed system. The increasing demands from industries start a new age of non-relational database. In 2009, Eric Evans introduced a new term which is known as NoSQL [1]. NoSQL offers a non-relational database which provides a scalable and distributed design in order to maintain availability to global users. As opposed to the ACID rule of traditional SQL database, NoSQL needs to take BASE and CAP Theorem into consideration [2] [3]. Big companies such as Google have developed a number of alternatives to their distributed database system: BigTable, MegaStore, and Spanner [4].

Relationship between people has transitioned from onsite relation to internet-based relation such as social media and online chat. In spite of borderless characteristics from the Internet, it is important to note that the deployment of data centers is based on geographical location. A comparative study from Facebook shows that 70 - 80% of interactions between people occur in the range of 100 - 500 miles [5]. It is believed that an accurate prediction of users' behavior can significantly improve user experience. This observation is not limited to the technology of NoSQL. Another research has tried to develop a distributed system for static pages in a website based on geographical observation [6]. Based on the same observation, the akamai system indicates that a local access may reduce network latency 100 times as compared to multi-continent access [7].

On the other hand, even if a depth analysis to replication and partition techniques in NoSQL has been researched [8] [9] [10] [11], there is no robust system which takes users' geographical relationship to the consideration. NoSQL databases tend to use a specific rule in distributing its data such as range-based and hash-based partitioning. These rules restrict the distribution of data in geographical deployment of data centers. Nowadays, several NoSQL databases such as MongoDB start to realize the importance of geographical distribution and begin to offer location-aware reads across multiple data centers [12]; however, these services are not automatically scalable and limited to read operation. As an example, MongoDB requires a user-specified configuration that associates shard key ranges with shards hosted on specific node. Upon failure, MongoDB doesn't provide additional mechanism for the automatic failover process.

Our contribution in this paper is to demonstrate that we can improve the performance of non-relational and distributed database system by considering users' geographical location. Specifically, we provide a scalable implementation of NoSQL databases which is named as TippyDB. TippyDB always tries to store users' data in the nearest location with the writer. Our system also provides consistency in stable environment and prioritizes availability during network partition. In this manner, TippyDB is flexible and requires minimum intervention from users in its operational phase. We also demonstrate that our system design is able to outperform other popular NoSQL databases (MongoDB with range-based sharding) in providing lower average response time (Section IV).

While our system design is not omnipotent to all cases, we believe that the relationship property of users in limited geographical area can be applied to the large number of modern applications. Our system design offers a new alternative solution with substantial benefits to the design of geographically distributed data centers.

II. Related Work

Technology has grown in a fast-paced manner, where computing powers and network bandwidth have increased significantly in the last decade. Using web performance as the study case, Sundaresan et al. (2013) observe that page load times stop improving beyond a specific point, as latency becomes the main performance bottleneck [13]. Latency is bounded to speed of light and client's performance will surely stagnate at a certain level, which is independent to hardware performance. Over the years, content delivery networks such as Akamai [7] have been used to minimize the bottleneck of fetching first byte in web pages. The idea behind content delivery networks (CDN) is to store data in the nearest geographical mirror with users' location.

NoSQL databases are usually deployed in a distributed environment. This simple fact implies that latency will give a substantial cost to the overall performance of NoSQL databases. On the further note, there are two popular ways in distributing data throughout multiple machines: range based and consistent hashing [10]. Several popular NoSQL databases such as MongoDB and HBase use range-based partitioning, while Voldemort, Riak, and Cassandra use hash-based partitioning. These techniques ensure the balanced distribution of data across multiple data centers. However, social proximity of human relationships does not follow the same distribution rule [5]. In several use cases, a closer storage may increase client's performance significantly.

In 2014, MongoDB starts to offer a tag-based partitioning, where MongoDB will store its data in the specified node [12]. This feature can be considered as one step ahead towards geographically-aware NoSQL database. However, this schema requires administrator intervention to constantly maintain the distribution rule. In addition, it is hard to guarantee closest proximity under node failure in the following schema.

III. System Design and Architecture

In this section, we present the system design of TippyDB. Our prototype is implemented in C++ with additional support from LevelDB [14] as its database backend and Apache Thrift [15] [16] as its RPC protocol. LevelDB as a key-value storage library is used to support basic functions such as insert (Put), read (Get), and delete (Delete).

In TippyDB, each region may have multiple nodes (replicas), whereas each node may have multiple shards with predefined size. Each node stores the information of other nodes and its relative distances to those nodes. The complete architecture can be seen in Figure 1.

Key-value Store. Each of the stored data will be assigned with a shard key and a logical clock. A shard key consists of three main components:

- First 4 bytes: denotes the region where the datum is firstly written
- The next 4 bytes: denotes the node where the datum is firstly written
- The last 8 bytes: denotes the auto increment key as unique identifier from each of the nodes

A logical clock (timestamp) is used to differentiate the newest version of datum in the secondary node. In addition, a global metadata is also used to identify the primary and secondary responsibilities in storing data. This metadata will be updated and propagated throughout all active nodes in order to ensure the availability of our system. A metadata can be said as a metadata collection of shard keys, which has the following specification:

- id: denotes the first 8 bytes from a shard key, which shows the respective range of data
- primary: denotes the responsible primary node for the specified range (id)
- secondary: denotes the responsible secondary node(s) for the specified range (id)

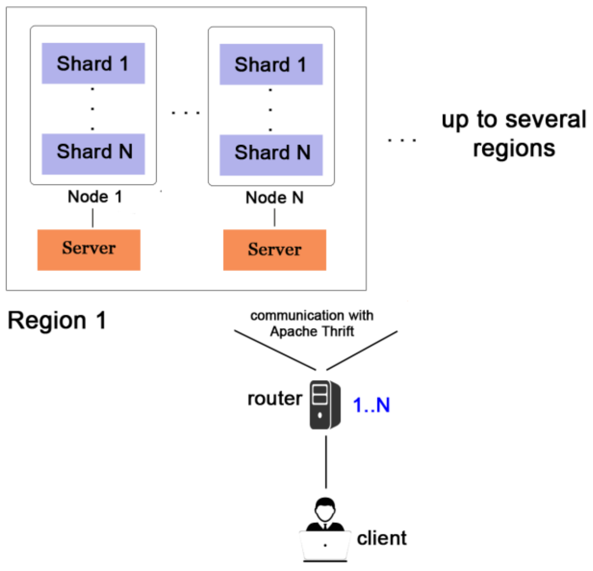

Figure 1. General Architecture of TippyDB.

For example, a datum with shard key "0001 0002 00000001" will be stored in all responsible nodes under id "0001 0002". Because of network partition, it is possible to have two different versions of metadata. In resolving this issue, our proposed system chooses availability over consistency which implies a possibility of data loss due to inconsistencies.

Node Coordination. We implement Raft consensus algorithm [17] [18] for coordinating changes in the global metadata. In short, there are 3 main roles in Raft consensus: leader, candidate, and follower. Under node failure and recovery, system leader will update its metadata and propagate the newest information to all followers. By using Raft consensus algorithm, it implies that the system needs more than $N/2$ nodes to operate.

Write and Replication. Initially, clients store their data in the nearest node with their geographical position. In this point, we assume that clients will interact with people which are located in the same geographical area. The responsible node will propagate its data to several secondary nodes. These nodes are specified automatically in the metadata. Figure 2 illustrates the writing and replicating process in TippyDB with replication factor equals to 3.

Sharding. In order to minimize the amount of unbalanced data, TippyDB also offers internal balancing across all nodes in one region. Each shard has a predefined size in the configuration file. In our implementation, we define 32 MB as the default size. Since we want to store our data in the nearest position with the writers, we will limit our schema such that writes will be redirected to one of the node in the same region. A redirection will occur if the current size of the node is greater than twice of the preconfigured shard size.

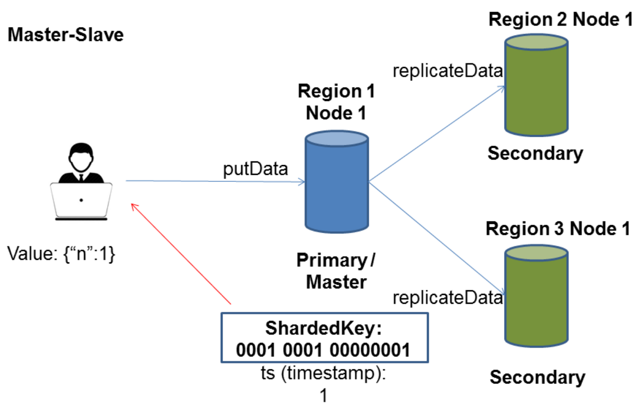

Figure 2. Write and replication in TippyDB.

Query Distribution. As stated above, 70 – 80% of users' interaction occurs in the effective range of 100 – 500 miles. If each data center is separated more than 500 miles, we can make an estimate that 70 – 80% read operation will retrieve data from the nearest node (Figure 3). In this manner, it is expected for clients to have lower latency in the average requests.

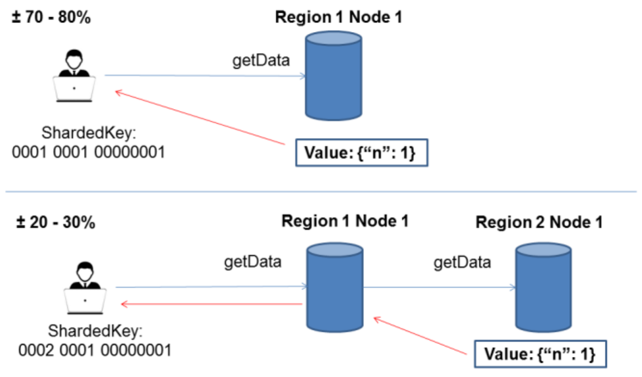

Figure 3. Query expectations in TippyDB.

Failover Mechanism. In TippyDB, failover is divided into 2 main phases. The first phase occurs when one of the active nodes detects the failure of the other node. TippyDB sends heartbeat to other nodes in the specified timespan. If the sender does not receive any reply in the configured timespan, TippyDB will start its first phase of failure detection. Upon failure detection, one of the active nodes will send a proposal to the current leader to update its metadata. After accepting a proposal, the respective secondary node will be selected as a temporary primary node. This phase ensures that the system is still available in term of node failure. Figure 4 depicts the illustrative diagram of the first phase of failover mechanism.

The second phase starts with choosing the nearest node to the failed node. This information is initially stored in the configuration file; therefore, each node knows its relative distance to the other nodes. The current primary will replicate its data to the chosen node in the background process (Figure 5). Finally, the global metadata will be updated and the nearest node to the failed node will take the primary responsibility of the failed node.

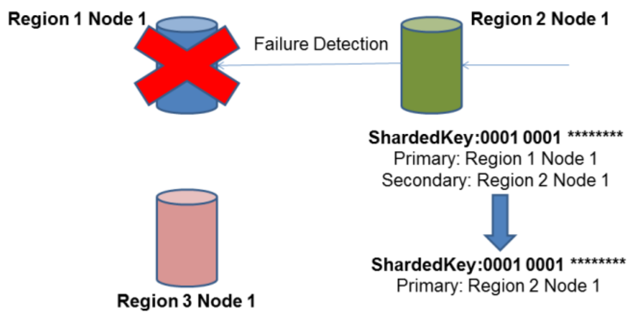

Figure 4. Failover process from secondary.

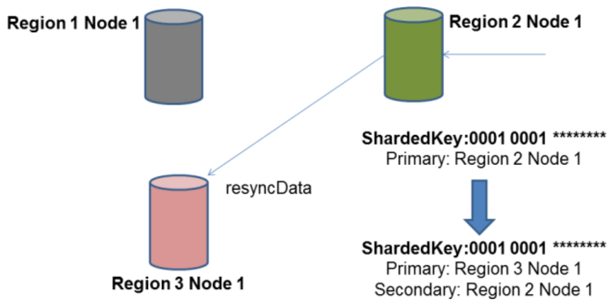

Figure 5. Resynchronization to the nearest node.

Recovery Process. As depicted in Figure 6, a recovered node will broadcast a join request to all nodes in order to find the current active leader. Afterwards, this node will send a pull request to the respective primary node in order to retrieve the newest stored data. Upon completion, this node will take over the responsibility of its primary section and send a proposal to the current leader to update its metadata. Finally, this node will be recognized as an active node throughout all regions.

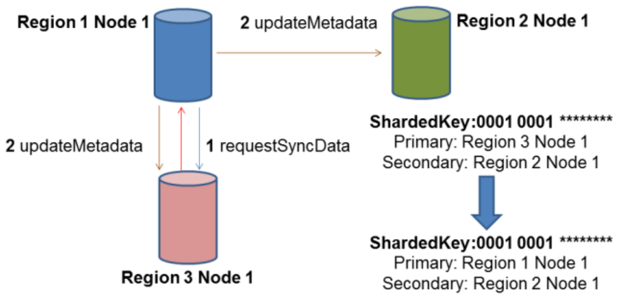

Figure 6. Node recovery from failure.

IV. Experimental Evaluation

In this experimentation, we evaluated CRUD performances of TippyDB as compared to the well-known NoSQL database, MongoDB. We use RDTSC and RDTSCP method for measuring time performance in TippyDB.

We evaluate each operation in 2 access-point deployments, Singapore and N. Virginia to receive requests from remote users. We employ *t2.micro* instances on Amazon EC2, with 2 replicas are configured in Singapore and 1 replica is configured in N. Virginia. To demonstrate the effect of geographically-aware aspect of users' behavior, each experiment is done with 80% write operation to Singapore (the nearest node) and 20% write operation to N. Virginia (the farthest node).

Request Performance. We measure the average response time from insert, read, update, and delete operations from TippyDB and MongoDB. The first experiment is done with 2,000 operations of 100 KB data size and 200 operations of 1,000 KB data size in a stable environment. The second experiment is done with the same specifications, unless 1 replica in Singapore will be unavailable.

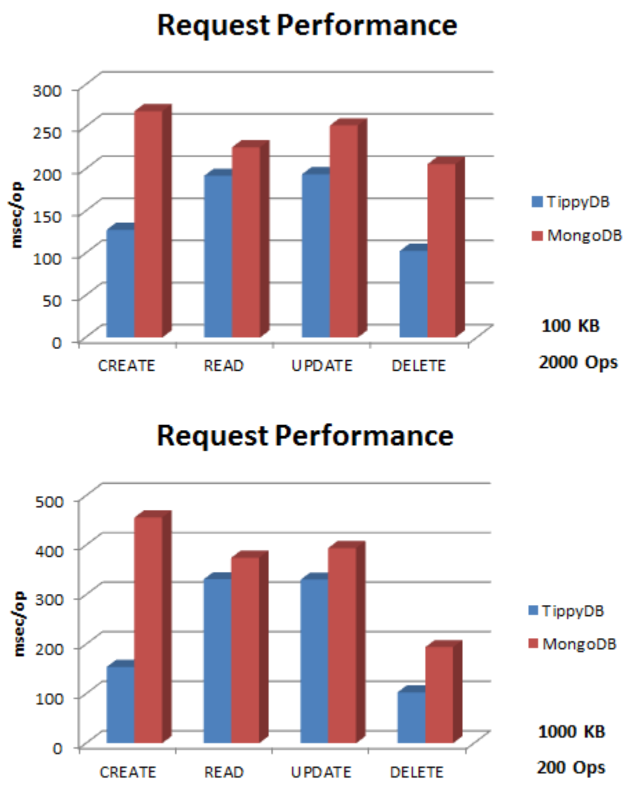

Figure 7. Latency of CRUD operations under varying data size and number of operations in a stable environment

As plotted in Figure 7, TippyDB achieves better performance in all aspects. One of the main reasons is 80% of data in TippyDB are stored near users' location. On the other hand, MongoDB only stores 66.7% of data near users' location, since we use 2 replicas in Singapore and 1 replica in N. Virginia. From [19], it is known that the average RTT between Singapore and N. Virginia is 253.5 ms, so we should expect approximately 30 – 50 ms improvement in average response time. Meanwhile, write operation in MongoDB performs quite slow and increases proportionally to data size. We believe this penalty is incurred due to sharding mechanism in MongoDB.

Figure 8 depicts system performance under 1 replica failure in Singapore. In this scenario, we configure TippyDB to use specific timeout for its failure detector. If an active node cannot be contacted in the configured timespan (60 seconds), we will use the assumption that the respective node has failed. In this timespan, TippyDB will start a consensus to determine the newest metadata, which allows a secondary node to take the responsibility over the failed node. Nevertheless, TippyDB still outperforms MongoDB in the average response time, since MongoDB also requires 30 – 60 seconds to finish its failover process.

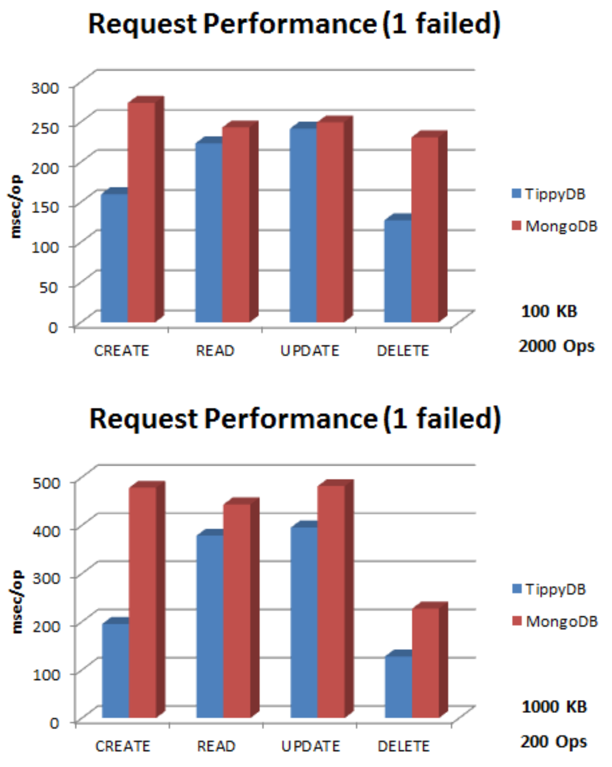

Figure 8. Latency of CRUD operations under varying data size and number of operations in 1 failed node.

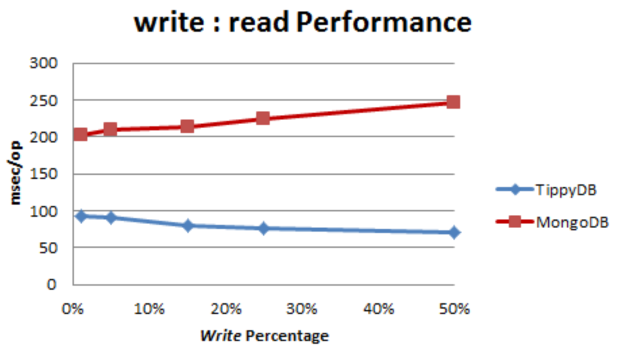

Figure 9. Average latency across varying write : read workloads.

Write : Read Proportion. We measure the difference in performance with different proportion of write and read operations. This test uses the same environment with the first one. Each experiment is done with 10,000 requests and 1,024 bytes data. As seen in Figure 9, TippyDB tends to have faster performance in handling write operation since TippyDB will always store its data in the nearest node with the client. In terms of overall performance, TippyDB is approximately twice as faster as MongoDB in this experimentation.

Basic Performance. We compared the basic performance of TippyDB and MongoDB in handling CRUD operations. Each operation is done 100,000 times with 100 bytes data size. Figure 10 depicts the basic performance of TippyDB and MongoDB in handling local requests. From our observation, the usage of RPC in Apache Thrift is quite costly, which an empty RPC needs 466 microseconds / operation in average (compromises 80% of total time). In short, MongoDB still has better local performance than TippyDB in handling number of requests.

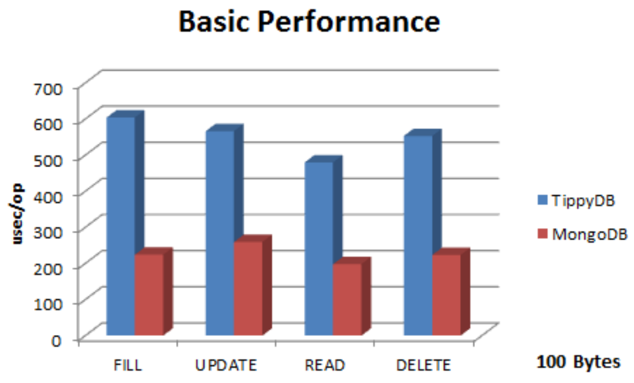

Figure 10. Local performance in handling CRUD operations.

V. Conclusion

This paper shows that we are able to construct a NoSQL database which offers lower average latency based on an observation that people tend to communicate with others who are living in the same region. We define our proposed system design – TippyDB – that always tries to store and rebalance users' data in the nearest node with the original writer. In handling requests from users, TippyDB outperforms MongoDB in create, read, update, and delete operations for 52.48 – 66.16%, 11.63 – 15.01%, 16.21 – 23.07%, and 47.01 – 50.15% respectively. Nevertheless, it should be noted that TippyDB are not appropriate for all applications; TippyDB is specifically designed for data which are sensitive to the location of users, such as social media, document editing, etc.

In terms of future work, there are many rooms for improvements in TippyDB. These include optimizing resynchronization algorithm, adding scheduler for failover checking, and implementing new features such as indexing, etc. Furthermore, a further study of system's overhead and its mitigation is also preferred.

References

[1] Eric Evans, NoSQL 2009, available at http://blog.symlink.com/2009/05/12/nosql_2009.html, 2009.

[2] F. Labs, "CAP Theorem: Its importance in distributed systems," available at http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter, 2014.

[3] G. Seth, N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News, 33(2), pp. 51-59.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, et al., "Spanner: Google's globally distributed database," ACM Transactions on Computer Systems (TOCS), 31(3), 8, 2013.

[5] L. Backstrom, E. Sun, C. Marlow, "Find me if you can: improving geographical prediction with social and spatial proximity," Proceedings of the 19th international conference on World wide web, 2010, pp. 61-70.

[6] V. Cardellini, M. Colajanni, P. S. Yu, "Geographic load balancing for scalable distributed systems," Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2000, pp. 20-27.

[7] E. Nygren, R. Sitaraman, J. Sun, "The akamai network: a platform for high-performance internet applications," ACM SIGOPS Operating Systems Review, 44(3), 2010, pp. 2-19.

[8] B. G. Tudorica, C. Bucur, "A comparison between several NoSQL databases with comments and notes," Roedunet International Conference (RoEduNet), 2011, pp. 1-5.

[9] M. Indrawan-Santiago, "Database research: Are we at a crossroad? Reflection on NoSQL," Network-Based Information System (NBiS), 2012, pp. 45-51.

[10] R.Hecht, S. Jablonski, "NoSQL Evaluation," International Conference on Cloud and Service Computing, 2011.

[11] I. Katsov, "Distributed algorithms in NoSQL databases," available at http://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/, 2012.

[12] MongoDB, "MongoDB Multi-Data Center Deployments," MongoDB Whitepaper, 2014.

[13] S. Sundaresan, N. Feamster, R. Teixeira, N. Magharei, "Measuring and mitigating web performance bottlenecks in broadband access networks," ACM Internet Measurement Conference, 2013.

[14] A. Dent, Getting started with LevelDB. United Kingdom: Packt Publishing Ltd, 2013.

[15] M. Slee, A. Agarwal, M. Kwiatkowski, "Thrift: scalable cross-language services implementation," Facebook White Paper 5, 2007.

[16] A. Prunicki, "Apache Thrift," available at http://jnb.ociweb.com/jnb/jnbJun2009.html, 2009.

[17] D. Ongaro, J. Ousterhout, "In search of an understandable consensus algorithm," Proc. USENIX Annual Technical Conference, 2014, pp. 305-320.

[18] H. Howard, ARC: Analysis of Raft Consensus. University of Cambridge Computer Laboratory, 2014.

[19] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: virtues and limitations (extended version)," Proceedings of the VLDB Endowment 7, no. 3, 2013.