UDC 004.942 **doi: 10.26906/SUNZ.2020.2.095**

K. Rukkas, G. Zholtkevych

V. N. Karazin Kharkiv National University, Kharkiv, Ukrania

LOAD BALANCING CONSISTENCY IN A DISTRIBUTED DATASTORE

Abstract. The **subject** of the article's research is the CAP-guarantees of distributed datastore, particularly availability and consistency. The **goal** is to design an approach that will become an instrument to balance consistency CAP-guarantee for any business needs still maintaining appropriate availability guarantee. The algorithm could be integrated to datastore infrastructure as one of distributed datastore components that must stand on top of or integrated in database middleware standing on the path to node database instance and actual query execution. To achieve that, the following problems were solved in the paper: the simulation models for approaches have been implemented, actual possibility to implement an algorithm following specific approach has been investigated. The following **methods** were used to implement such solutions: UML modeling, computer model implementing the simulation of the designed algorithms, carrying experiments on the implemented models. Carried out experiments **resulted in** capability to estimate the complexity and possible performance and make conclusions choosing one of optimal approaches to be designed further. As a **conclusion**, the optimal designed and estimated approach of balancing consistency and availability is ready and it was the purpose of this paper. It could be applied as one of basic components on the design distributed datastores stage, so that balanced guarantees of distributed system reliability could be achieved at the earlier stage of business needs implementation.

Keywords: CAP-guarantees, load balancing, distributed databases, high availability, strong consistency.

Introduction

In modern times scaling technical software requires distributed systems it relies on to be resistant to node faults occurrence, fast enough response time and reasonable consistency reached on most of the distributed system nodes to avoid node conflicts. It is desired, the system must survive having any number of nodes in its infrastructure. In conditions of well-known CAP-theorem such requirements become hard to achieve. The current paper is devoted to the research of how the CAP-characteristics could be balanced to fulfill these needs. This could result in lots of opportunities: reaching needed value of consistency, maintaining reasonable value of availability, monitoring the consistency and availability current state of the system, and even achieving strict consistency in good enough network conditions.

Literature analysis and background. While designing any software architecture there is a need to make a choice: ACID or BASE model, that means strong consistency that results in weaker availability or basically available eventually consistent system where strong consistency is neglected. The comparing analysis for these models were made in [1]. It is reasonable solution for some business requirements, when, for example, the system has small number of nodes in its infrastructure, eventually consistency is achieved quickly and strong consistency is not important in such case. Or when database partitioning is settled and every dataunit can be found only on one node and replication is not needed. But for large distributed datastore with replication it is important somehow to achieve the best value of consistency and not to deteriorate availability and partition tolerance guarantees. Also, it is essential to keep track on these guarantees while system is working. CAP-guarantees had been deeply researched since the well-known CAP-theorem had been officially proven [2] and discussed 12 years later [3]. Some work had been done for consistency improvements that can tend to strict consistency [4]. But in this paper, the throughput is calculated only for number of nodes up to 16. Large systems were not considered in this work.

Background. In this paper we explore load balancing solutions to investigate the ability to balance across only consistent nodes that supports one or another dataunit. We are convinced that strong consistency or eventual consistency, that converges fast, could be achieved without losing availability since most of distributed system use epidemic algorithms to broadcast replicas (see [5], [6]) and the broadcasting can be faster than consistency disbalancing. For that we explore works for database load balancers ([7], [8]) and the possibility to contribute to existing load balancers to maintain the idea mentioned above. All existing load balancers for databases support well-known balancer algorithm like Round Robin, Least Connections etc. (see [7], [8]). One of the components in some of solutions, implemented in the paper, is load balancer dynamic API [9], [10]. This application allows some features, partially, can return the list of currently health checked nodes. Also, we need to mention that databases have parsers that allow parse database request [11] and there are many optimizations, like Microsoft has done, for example, in [12] and official doc is presented in [13]. Additionally, many of databases support caching compiled queries [14]. The last that we would like to mention is our previous paper [15], where we form the mathematical model with metrics for CAP-guarantees.

Core Material

Load balancing is the technique, that allows distributing requests between application instances or network devices where this application live. The purpose lies in optimization of resources usage, saving response time and providing system fault tolerance. The load balancing can be applied in different needs depending of system requirements and the desired result and there are also balancers for various database solutions [7]. A big problem in existing solutions is that data on nodes have to be the same. So there is no partitioning dataunits on different sets of servers. But some of the load balancers,

mentioned in [7], are opensource and algorithms for requests balancing technique can be enhanced or new algorithms are embedded. In this paper we investigate and estimate this ability in terms of imitation modeling.

In the paper we go deep into the research for the tradeoff of consistency and availability using load balancing algorithms enhancement. These algorithms are based on the idea to balance only across consistent nodes that have the given dataunit and on the idea that spreading replica is fast enough to maintain list of consistent nodes as big enough as needed. The several solutions has been explored:

1) load balancer custom algorithm (that may be contributed to opensource, so that needed solution is maintained);

2) hybrid balancing solution on the level of custom balancing algorithm that can be contributed to one of balancers solutions and some applications that are implemented in purpose to replace those features load balancer never supports;

3) own balancer database middleware component, that balances consistent nodes without any general load balancing features.

In this section we introduce these solutions in terms of class and sequence diagrams below and estimate effectiveness of each solution and discover pros and cons and perspectives of solutions.

Let us firstly describe main components that will be used in the approaches:

- Node object is a structure that contains the unique identifier of node (like host or IP-address), the time of last update and may be some other additional info that can be used for request forwarding (used in all represented approaches below)
- CNode hash table is the global mapping that each dataunit (as a key) associates with a binary tree of consistent Node objects that store given dataunit (used in all represented approaches below)
- DNode hash table is the global mapping that each dataunit (as a key) associates with a binary tree of Node objects that store given dataunit (used in all represented approaches below). This table will not be changed dynamically, only if the requirement to add new dataunits appears. But this case is another story and the separate interface can be simply implemented for that.
- Request is the request object that enhances general database request object may represent if it is write or read request, may contain the flag that identifies if it is forwarded from another node and the flag identifying if the given request is replica or request itself broadcasted.
- Load Balancer Dynamic API that allows to make some requests to Load Balancer. Here the ability to get the list of current nodes that can accept requests will be used.

Load Balancer Algorithm Enhancement. The first solution comes up with custom balancer algorithm that can be embedded into existing load balancer solution and enhance some opensource solution. We present a class and sequence diagrams (see Fig. 1 – Fig. 3). The class diagram shows that the solution is based on existing load balancer algorithm implementation (for example, on Round Robin to be more intuitive) and enhance this algorithm by replacing the node list that request can be forwarded to.

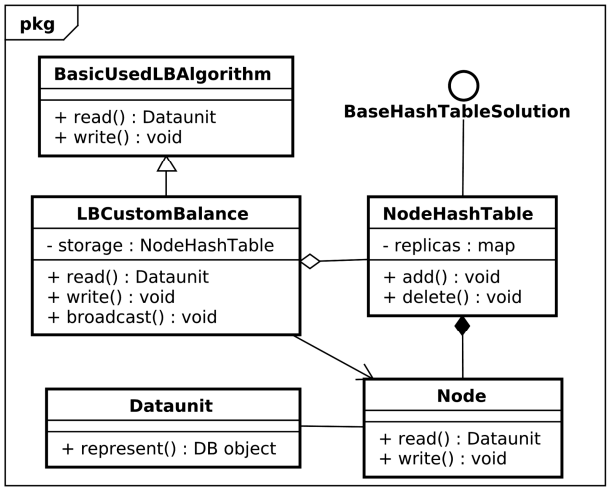

Fig. 1. Custom Load Balancer class diagram

The node list is based on mapping that associates each data unit to the list of *consistent* nodes that store it. Nodes are consistent in terms of that they have the up-to-date latest replica version. So once a write request came the association for this dataunit is updated with a new node set, the first node that actually changed own replica. That time all other nodes become inconsistent and with every broadcast the map of dataunits and nodes is growing again until the next write request. Our purpose in this paper was to find the ratio of number of nodes that store a dataunit to the amount of write requests. This would allow to show that during distributed database lifetime there are enough of consistent nodes that are able to maintain desirable availability. Basing on these structural and behavioral diagrams we created the imitation model that allows to execute experiments on simulated solution components. In the model a user can regulate the threshold that is the minimum amount of nodes that must respond at any moment of system lifetime.

Later we introduce the results of these executions as graphics with dataunits as abscissa axis and number of nodes that maintain a dataunit as axis of the ordinates. These experiments is run on 100, 1000, 5000 write requests for hundreds of nodes in a datastore for every approach presented in the paper. We emphasize that in this solution the load balancing algorithm itself is responsible for broadcasting the replicas across a datastore and interact with hash map, where the key is dataunit hash and the value is the list of consistent nodes that contains given dataunit and another hash map that differs only with the value of list of all nodes that contains given dataunit. This algorithm has pretty simple architecture to solve inconsistency problem.

But it has a lot of future problems related to the fact that algorithms of load balancers should solve completely other problems, such as, fault tolerance, health of the system etc. and it is not essentially to enhance these algorithms like that.

Hybrid Load Balancing. This solution still enhances general load balancer algorithms implementa-

tions, but built with the purpose of not additional responsibility put on load balancer algorithm itself. For that load balancer API is designed that should interact with load balancer algorithm and change the node hash table adding new nodes to a list for a dataunit or remove them once new request came.

So that load balancing algorithm just needs to interact with existing node hash table on every request to make decision on the list of nodes where a request can be forwarded. See the solution class diagram in Fig. 2.

So that current approach still has to implement custom algorithm that can inherit some of existing algorithms, but from point of architecture view it is still better than previous solution because the responsibility of filling the CNode Hash Table that the balancer should not definitely be responsible for is put on the separate API.

Own balancer solution. And finally, we want to introduce balancer approach that shall not touch the load balancer algorithms implementations at all. In this solution general load balancer is still used to maintain fault tolerance. This solution is a more complex one, so for that we need to introduce class diagram along with algorithm for write and read request cases. You can see class diagram in Fig. 3 and block schemas for read and write database requests in Fig. 4 and in Fig. 5.

Thus, now we can say that this approach have some specialties:

– it does not change the initial load balancer architecture and does not enhance its responsibility that cause load balancer to remain still essential mechanism to do; the application that stands on front of the database and can be understood as one of the database middleware, should understand request incoming to get a dataunit requested. Therefore this operation will depend on database implementation and optimization that is made to improve this part of functionality. Other operations will take: O(log n) on taking the nodes from a hash table and respond or O(log n) plus one more algorithm repeat when request came to the node that does not store given dataunit taking into account the operations to take fault tolerant nodes from dynamic load balancer api and intersect them, which will take O(1) as some constant time will be wasted and O(n) for each operation appropriately. So that in the worst case it will be O(log n) +O(1) + O(n) + + O(log n)+ O(1). Calculating this expression we get O(n) time complexity.

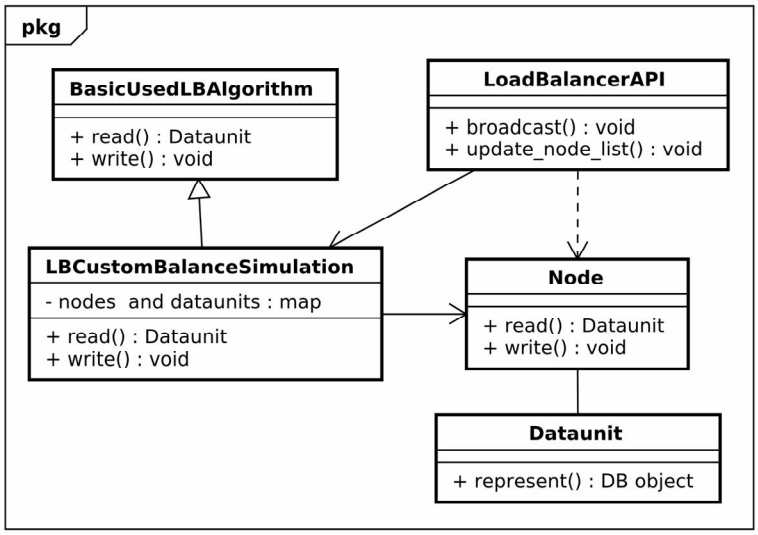

Fig. 2. Hybrid Load Balancer class diagram

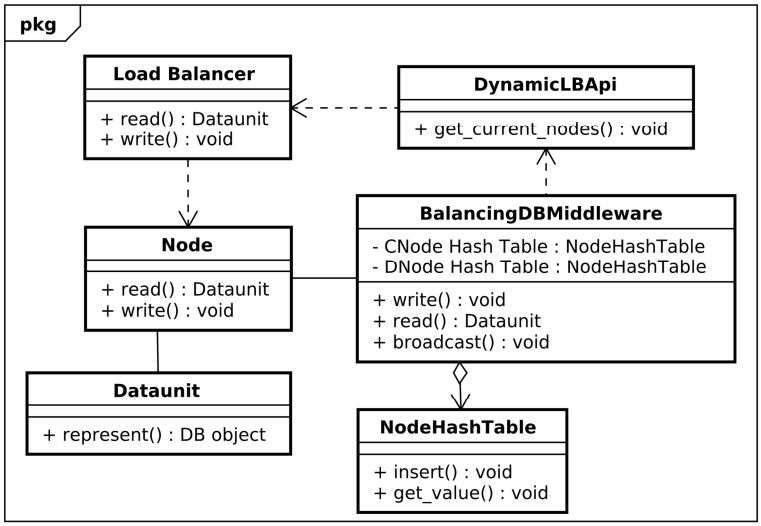

Fig. 3. Database Load Balancer Class Diagram

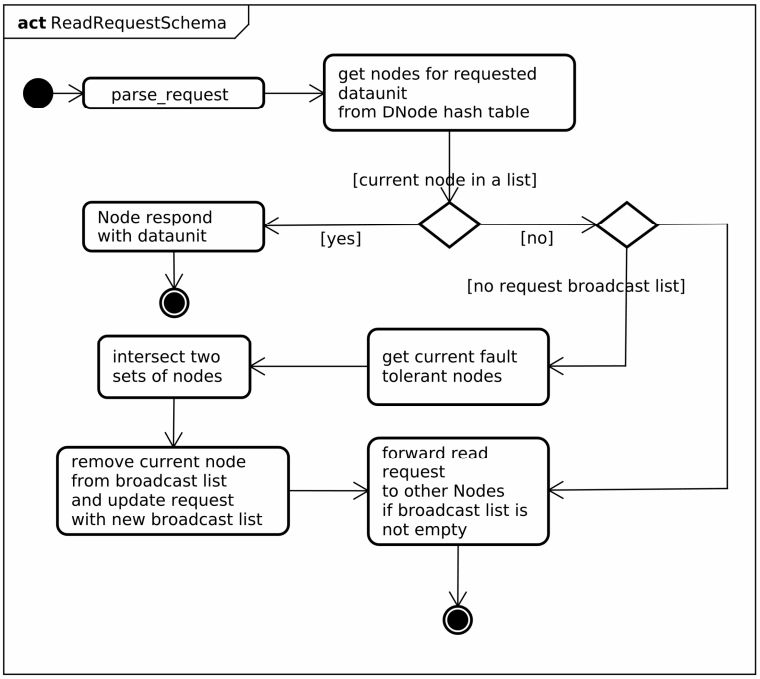

Fig. 2. Read request activity diagram for Load Balancing

Space complexity in the worst case will be O(n), because at maximum n nodes will be in memory at a time;

– basically, two other load balancer algorithms will have similar complexity since they will need to parse database request to make decision on what dataunit is requested and what nodes will be chosen to forward. Also, it shows the problems that can appear on load balancer side, when it will need to deal with database request somehow and it is not what it is designed for initially at all. In these terms the current approach should suit better in the architecture point of view, because database side is already optimized for understanding database queries and a lot of solutions for this already exist in a box.

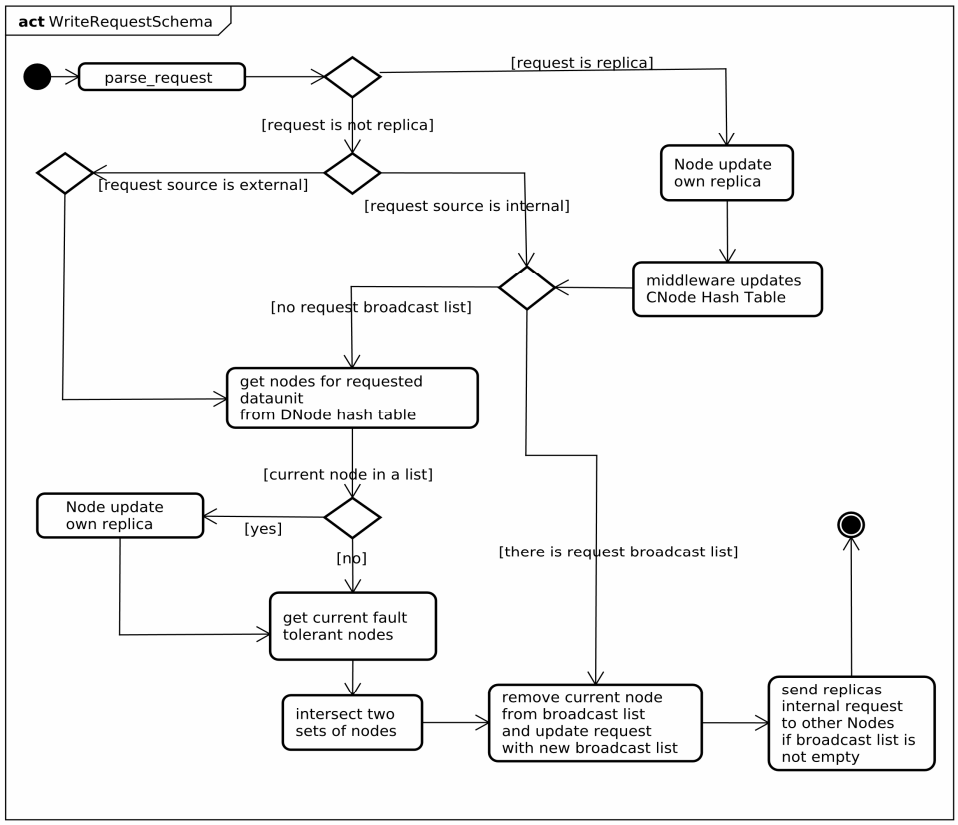

Fig. 3. Write Request activity diagram for Load Balancing

Case study. So since the simulation models for three of approaches have been implemented, we executed the set of experiments on these models in the conditions of 500 nodes in the network, 100 dataunits distributed across them at random and different number of write and read requests. The executions were run for: writes and reads, for both cases, when writes occur more often, and opposite cases, when reads occur more frequently than writes. All the simulation models have availability threshold that is a number of nodes that should stay available for any dataunit. The graphics represent the dataunits as axis and number of consistent nodes that stay consistent. In some graphics we can see the number of consistent nodes that can answer rarely reaches threshold, that means that sometimes the number of consistent nodes is less than threshold set as 30 nodes.

The solutions can be compared with the frequency of that number of consistent nodes outreaches the set limit. Look at the results of first approach simulation (custom load balancer algorithm) in Fig. 6 – Fig. 9. These figures represents the experiments run on small amount of writes and reads and on large quantity of read and write requests in order to show the way performance of algorithm changes.

The second set of figures (Fig. 10 – Fig. 13) represents results for custom algorithm with additional API for the same numbers of writes and reads. As it can be seen, this algorithm has extremely bad performance on large amount of requests as for both of the cases when write or read requests prevail. This makes implementation of this solution harder and not survivable for the real business needs.

Let's consider the last approach that simulation model and algorithm is implemented for. Look at the figures (Fig. 14 – Fig. 17) representing the state of our simulated system that experiments were run in conditions of the same amount of read and write requests. At the pictures we can clearly see that custom load balancer algorithm has best performance in the simulation.

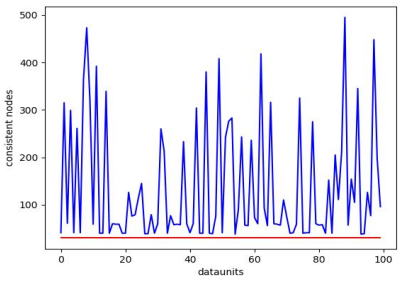

Fig. 4. Custom algorithm: 100 writes and 66 reads

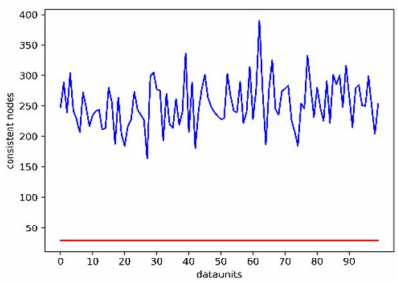

Fig. 5. Custom algorithm: 5000 writes and 3333 reads

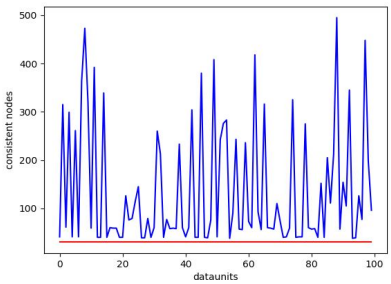

Fig. 6. Custom algorithm: 100 writes and 150 reads

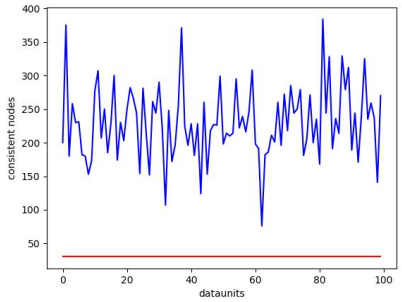

Fig. 7. Custom algorithm: 5000 writes and 7500 reads

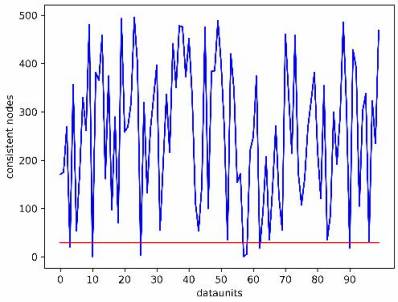

Fig. 8. Custom algorithm with API: 100 writes and 66 reads

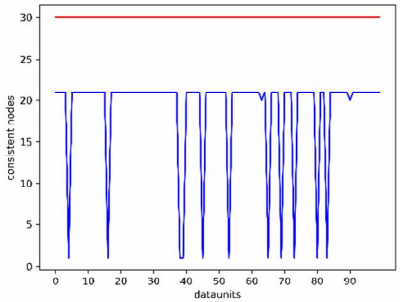

Fig. 9. Custom algorithm with API: 5000 writes and 3333 reads

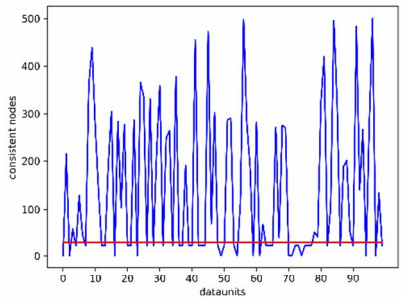

Fig. 10. Custom algorithm with API: 100 writes and 150 reads

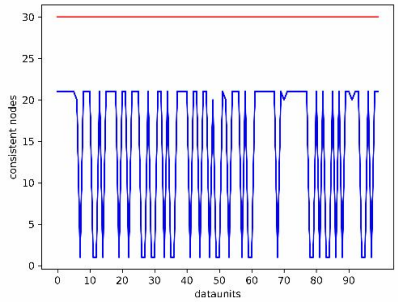

Fig. 11. Custom algorithm with API: 5000 writes and 7500 reads

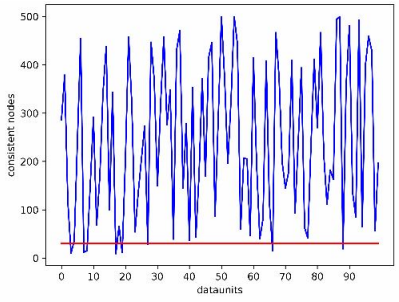

Fig. 12. Own balancing. 100 writes and 66 reads

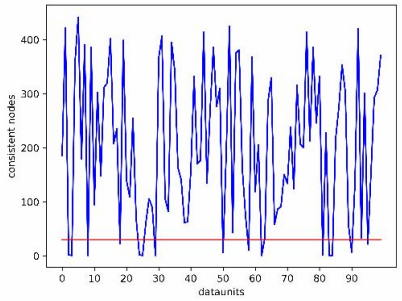

Fig. 13. Own balancing. 5000 writes and 3333 reads

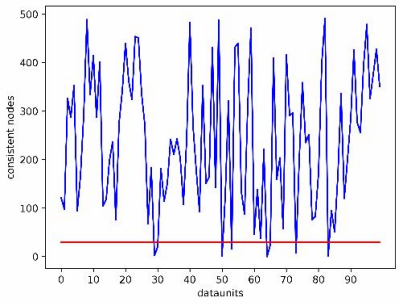

Fig. 14. Own balancing. 100 writes and 150 reads

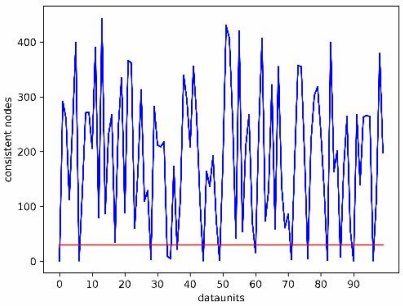

Fig. 15. Own balancing. 5000 writes and 7500 reads

But during implementation of this algorithm the following problems will appear and slow down performance:

- load balancer needs to know what dataunit is requested and needs to parse somehow the database request. It is not related to load balancer features and what it is designed for, so that it meets all the issues that databases had while optimizing parsers
- different databases have already implemented algorithms to parse the request and optimized them. Therefore, database has some options when understanding a request and some databases supports parsing a request partially, only a dataunit, for example, so that it will speed up the performance at the database side in compare with load balancer side solution
- load balancer will need to parse database request partially, but still database will need to parse the request again to execute the query.

These conclusions mean that the third approach is optimal after understanding some design details and limitations. The third approach has still good performance, a little bit worse, that the first one. But it can be clear that after implementation of these algorithm the first approach performance will be significantly decreased, because parsing of database request is not the load balancer problem. Thus, the database request will be parsed twice: at load balancer side to get a requested dataunit and database side: to execute the query. So that, the first approach loses in design complexity and impossibility to meet further issues. Also, we do not consider the second approach anymore since it has the worst performance of the algorithm. For now, the optimal solution is the third approach which is own load balancer as database middleware at the side of every node. For now, we have already tried to avoid some issues to be met, such as, selecting broadcast list in every request that will avoid flooding other with requests, optimizing number of requests to dynamic API of load balancer, removing current node from broadcast list to avoid cycling, replicas version implemented as timestamps to avoid conflicts.

Conclusions

The purpose of the current work was the formation of algorithm that will improve CAP-guarantees or balance them for specific business requirements. In order to achieve it three approaches were designed, investigated, compared and estimated using simulation computer model. During algorithm design all the inves-

tigated approaches have already shown their weaknesses and advantages and allowed to choose one of solutions basing on best practices analyzing technical weaknesses of every solution. As a result, when designing new software with distributed datastore or integrating distributed database with existing software, the architect could use the recommended algorithm as a distributed database middleware component so that there could be found the trade-off for CAP-guarantees that is necessary for specific software requirements.

REFERENCES

1. Banothu, N., Bhukya, S. and Sharma, K. (2016). Big-data: Acid versus base for database transactions. 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), available at https://ieeexplore.ieee.org/document/7755401.
2. Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), p.51, available at https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf.
3. Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. Computer, 45(2), pp.23-29, available at https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/.
4. Calder, B., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M., Wang, J., Haq, M., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., Rigas, L., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S. and Wu, J. (2011). Windows Azure Storage: a highly available cloud storage service with strong consistency. Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11, available at http://web.eecs.umich.edu/~mozafari/winter2014/eecs684/papers/azure.pdf.
5. Burmester, M., Le, T. and Yasinsac, A. (2007). Adaptive gossip protocols: Managing security and redundancy in dense ad hoc networks. Ad Hoc Networks, 5(3), pp.313-323, available at http://www.cs.fsu.edu/~burmeste/adhocjourn.pdf.
6. Haas, Z., Halpern, J. and Li Li (2002). Gossip-based ad hoc routing. Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societiesm, 3, pp.1707-1716, available at https://ieeexplore.ieee.org/document/1019424.
7. Veerman, G., Breuk, R., 2012. Database Load Balancing , Mysql 5.5 Vs Postgresql 9. Amsterdam: Universiteit van Amsterdam, System & Network Engineering, https://www.os3.nl/_media/2011-2012/courses/lia/rory_breuk_gerrie_veerman_-_report.pdf.
8. Joshi, S., Ameta, S., & Lavania, G. (2019). Balanced Load in Distributed System with NoSQL Middleware. International Journal of Emerging Technologies and Innovative Research (www.jetir.org), 6(5), pp.133-137, available at https://pdfs.semanticscholar.org/f6fd/7e1c441040ae0a022cb19d930df1ef9bd07b.pdf.
9. Mhedhbi, M., 2017. Dynamic Configuration With The Haproxy Runtime API – Haproxy Technologies. [online] HAProxy Technologies, available at https://www.haproxy.com/blog/dynamic-configuration-haproxy-runtime-api.
10. Dynamic Configuration Of Upstreams With The NGINX Plus API - NGINX Documentation. n.d. NGINX Docs [online], available at https://docs.nginx.com/nginx/admin-guide/load-balancer/dynamic-configuration-api.
11. Zelle, J.M., Mooney, R.J. (1996). Learning to Parse Database Queries Using Inductive Logic Programming. AAAI/IAAI, 2, pp. 1050-1055, available at https://pdfs.semanticscholar.org/1c9d/f99cce1903d34c53025e86e72331bbfbe08f.pdf.
12. Chen, X., Fang, H., Lin, T., Vedantam, R., Gupta, S., Dollár, P., Zitnick, C.L. (2015). Microsoft COCO Captions: Data Collection and Evaluation Server. ArXiv, abs/1504.00325, available at https://arxiv.org/pdf/1504.00325.pdf.
13. 2019, Optimize Cost And RU/S To Run Queries In Azure Cosmos DB – docs.microsoft.com [online], available at: https://docs.microsoft.com/en-us/azure/cosmos-db/optimize-cost-queries.
14. Patterson, R., Gibson, G., Ginting, E., Stodolsky, D. and Zelenka, J., 1995. Informed prefetching and caching. Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95, available at http://www.cs.columbia.edu/~nieh/teaching/e6118_s00/papers/p79-patterson.pdf.
15. Rukkas, K., Zholtkevych, G. (2015). Distributed Datastores: Towards Probabilistic Approach for Estimation of Dependability. 11th International Conference on ICT in Education, Research, and Industrial Applications, 1356, pp.523-534, available at https://pdfs.semanticscholar.org/5eb0/01632c6cd6da2e4ec92adbc288939de0f4f9.pdf.

Received (Надійшла) 26.03.2020
Accepted for publication (Прийнята до друку) 06.05.2020

Балансування узгодженості у розподіленому сховищі даних

К. М. Руккас, Г. Г. Жолткевич

Анотація. Предметом цієї статті є CAP-гарантії розподілених баз даних, зокрема, доступність та узгодженість. Метою є спроектоване рішення, яке стане інструментом балансування узгодженості як одної з гарантій надійного розподіленого сховища для будь-яких бізнес потреб і яке дозволить не погіршити значення доступності. Такий алгоритм міг би бути інтегрований у інфраструктуру розподіленого сховища даних і повинен бути одною з перших програм на шляху до виконання SQL запиту і може використовувати різні модулі проміжного програмного забезпечення бази даних на вузлу. Для досягнення цього були розроблені і порівняні три альтернативних рішення для балансування консистентних вузлів, досліджена фактична можливість реалізації кожного з рішень. Методами розробки стали такі інструменти, як UML моделювання, комп'ютерна модель, що реалізує імітаційні моделі для всіх розроблених рішень, яка дозволила провести набір експериментів на досліджених імітаційних моделях і оцінити складність та можливу швидкодію, зробити висновки, вибравши один з найоптимальніших підходів для подальшої розробки та розширення. Як висновок, готове оптимальне спроектоване і оцінене рішення для балансування узгодженості, що і було метою статті. Воно може бути застосоване у якості одного з базових компонентів проміжного програмного забезпечення розподіленої бази даних на етапі проектування будь-якого програмного забезпечення таким чином, що можуть бути досягнені збалансовані гарантії для надійного сховища на ранньому етапі імплементації бізнес потреб.

Ключові слова: CAP-гарантії, балансування навантаження, розподілені бази даних, висока доступність, жорстка узгодженість.