Blockchain-based Model for Social Transactions Processing

Idrissa Sarr[1], Hubert Naacke[2] and Ibrahima Gueye[1]
[1]*Université Cheikh Anta Diop, LID, BP. 16432, Dakar-Fann, Sénégal*
[2]*UPMC Sorbonne Universités, LIP6, 4, place Jussieu 75005, Paris, France*

Keywords: Transaction Processing, Load-aware Query Routing, Data Consistency.

Abstract: The goal of this work in progress is to handle transactions of social applications by using their access classes. Basically, social users access simultaneously to a small piece of data owned by a user or a few ones. For instance, a new post of a Facebook user can create the reactions of most of his/her friends, and each of such reactions is related to the same data. Thus, grouping or chaining transactions that require the same access classes may reduce significantly the response time since several transactions are executed in one shot while ensuring consistency as well as minimizing the number of access to the persistent data storage. With this insight, we propose a middleware-based transaction scheduler that uses various strategies to chain transactions based on their access classes. The key novelties lie in (1) our distributed transaction scheduling devised on top of a ring to ensure communication when chaining transactions and (2) our ability to deal with multi-partitions transactions. The scheduling phase is based on Blockchain principle, which means in our context to record all transactions requiring the same access class into a master list in order to ensure consistency and to plan efficiently their processing. We designed and simulated our approach using SimJava and preliminary results show interesting and promising results.

1 INTRODUCTION

Generally, social applications are read intensive whatsoever the write operations are so important. For instance, Facebook may face more than 1.6 billion reads per second and 3 million writes per second during peak hours. These read and write operations derive from users interactions. Users interactions are sequences of transactions and we assume that each transaction reads and writes data owned by one or several users. Even though, transactions are usually short, the volume of required data is extremely low regarding the size of the whole database. Moreover, many transactions may attempt to access the same dataset simultaneously (i.e., at the same time), which generates temporal load peaks on some hot data. Such a situation, more known as a net effect, has the drawback to slow user interactions.

Furthermore, it is worth noting that among the multiple requirements of social applications one may keep three: response time, scalability and availability. One way to achieve scalability and availability is to partition data and process independent transactions in parallel manner. However, partitioning data perfectly in such a way that each transaction fits only on one partition/node is quite impossible. Thus, multi-nodes or multi-partition transactions must be managed efficiently to avoid compromising the positive effect of splitting and/or replicating data. In a context such as social environment multi-partitions transactions may happen due to unbounded and unlimited interactions user may have with others.

We proposed in STRING (Sarr et al., 2013) a scheduling solution that uses various strategies to order (or group) transactions based on their access classes. The proposed approach for overcoming limits in existing solutions, is guided by the fact that grouping transactions with similar patterns of data access might save a significant amount of work. We leveraged on the ability to process a group of concurrent transactions that is faster than processing one transaction at a time and it reduces the number of messages sent to the database node. More precisely, if every application directly connects to the database, then the latter may become a bottleneck and overloaded. Rather, application requests can be gathered and grouped at a middleware stage and thereby, database connection requests are minimized.

The proposed solution works in two steps: a *scheduling step* followed by an *execution step*. The scheduling step aims at grouping all transactions accessing the same data into a block regardless where

Sarr I., Naacke H. and Gueye I..
Blockchain-based Model for Social Transactions Processing.
DOI: 10.5220/0005519503090315
In *Proceedings of 4th International Conference on Data Management Technologies and Applications* (DATA-2015), pages 309-315
ISBN: 978-989-758-103-8
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

they are issued. Afterwards, each transaction is executed at the nodes storing the required data. However, the grouping mechanism does not guarantee that a group of transactions are executed on the execution layer while respecting the order in which transactions are received. That is, the order in which transactions are sent by the scheduler may differ to the one the execution layer processes them, which has the drawback to induce inconsistencies or disorganize the social interactions. Moreover, we have assumed that multi-partitions transactions are infrequent and therefore had not been taken into account deeply.

This work in progress improves our previous solution. It aims at handling more efficiently multi partition transactions. It relies on a smarter description of the transactions requests, which allows for globally ordering concurrent transactions while providing efficient decentralized transaction execution. The main idea is to avoid imposing any specific ordering rules (such as processing multi-partition transactions before single-partition ones, or vice-versa) as it is proposed in previous work. Such ordering would unnecessarily delay some transactions. On the opposite, we aim to order the transaction as close as possible to the ”natural” ordering of the requests that are submitted to the system. Therefore, considering the local ordering of transactions arriving at each node of the system, we propose to describe them in a smart way to obtain a global ordering which is expected to bring a more efficient execution. Then, each node could follow that ordering to execute transactions efficiently an consistently.

With this in mind, transactions are ordered during the scheduling step in such a way that each transaction is followed or preceded by another one as in a blockchain model. Blockchain is a mechanism to validate the payment processing with Bitcoin currency (Barber et al., 2012). It states to keep a public transactions log shared by all users in order to record bitcoin ownership currently as well as in the past. By keeping a record of all transactions, the Blockchain prevents double-spending. The key idea of Blockchain is that each transaction is guaranteed to come after the previous transaction chronologically because the previous transaction would otherwise not be known. Once a transaction is positioned into the chain, it is quite impractical to modify its order because every transaction after would also have to be regenerated. These properties are what make double-spending of bitcoins very difficult.

Hence, we rely on Blockchain model for two reasons : i) ensuring consistency between transactions issued from anywhere and ii) being able to chain transactions with an order that cannot be changed whatever the replica on which the group of transactions will be executed. More precisely, by using the Blockchain, approach we are able to guarantee that a transaction chain will not be scheduled twice nor executed twice with a different order.

The remainder of this paper is organized as follows. In Section 2 we review some works connected to ours. In Section 3 we lay out our architecture of our solution and the communication model between the different pieces of it. We also describe how transactons are chained in blocks an routed. In Section 4 the transaction execution model for multi-partitions transaction. In Section 5 we present the preliminary results of our experiments while in Section 6 we conclude and present our future work.

2 RELATED WORK

Our work is linked to transaction processing for scalable data stores. Large scale solutions for managing data are facing a consistency/latency tradeoff as surveyed in (Abadi, 2012). Today several solutions relax consistency for better latency (Silberstein et al., 2012; Lakshman and Malik, 2010; Vogels, 2009) and do not provide serializable execution of concurrent transactions. Other solutions provide strong consistency but only allow transactions restricted to a single node (Oracle, 2014)[1] , (Chang et al., 2006).

In a multi-tenant database context we have ElasTraS (Das et al., 2013), which is a system providing a mechanism to face load peaks and avoid distributed transactions. In fact, ElasTraS uses a data migration mechanism couple with some load balancing schemes to face load peaks on database nodes. Moreover, ElasTraS uses a static partition mechanism called *Schema Level partitionning*, which statically group data expected to be accessed together in a single partition and allows to scale at the granularity of a partition. However, this partitioning scheme is not suited to the data we face with social media. ElastTraS uses some transaction semantics similar to the Sinfonia ones. Sinfonia (Aguilera et al., 2007) offers mini-transaction abstraction that ensures transaction semantics on only a small group of operations such as atomic and distributed compare-and-swap (Michael and Scott, 1995). The idea is to use the two phases of 2PC to perform simple operations. Operations are piggy-backed to the messages sent during the first phase of 2PC. The operations must be such that each participating site can execute them and respond with a commit or abort vote. The lightweight nature of a

[1] http://docs.oracle.com/cd/NOSQL/html/

mini-transaction allows the system to scale. In Sinfonia, no attempt has been made to absorb load peaks.

DORA (Pandis et al., 2010) is a system that decomposes each transaction to smaller actions and assigns actions to threads based on their access classes. This design is motivated by the need of avoiding the contention due to centralized lock manager. DORA promotes local-data access, since each thread to which is assigned some actions, requires as infrequently as possible the centralized lock manager. In fact, DORA is a locking-based system that partitions data and locks among cores, eliminating long chains of lock waiting to a centralized lock manager. The main problem of this design (partitioning) is that the performances of DORA can be worsen when we face transactions that access to many partitions. However, the authors of DORA propose PLP (Pandis et al., 2011), a work following DORA and in which they propose a mechanism to face transactions accessing to many partitions. In fact, they propose in PLP to organize the partitions through trees in such a way that a tree is managed by a single thread. However, even if this strategy mitigates the impact of transactions accessing many partitions, it still uses a contention point due to the necessity of maintaining a centralized routing table.

Finally, the two steps approach we propose for processing transactions has been demonstrated to be efficient for write intensive workloads made of short transactions (Thomson et al., 2012; Kallman et al., 2008). However, existing solutions following this approach assume that the workload is mostly composed of single node transactions (i.e. they mostly access independent data). Most of existing solutions are not designed to face a load peak of concurrent transactions. In such situation, they would suffer from communication overhead and high latency. Our work also relies on sequencing and scheduling to guaranty consistent processing of multi-node transactions, while better supporting high peak load.

3 SYSTEM OVERVIEW

The architecture is designed with two layers: the scheduling layer and the storage one (see Figure 1). The scheduling layer is made of a set of nodes called scheduling nodes (SN) while the execution layer contains execution nodes (XN) that rely on a datastore that we consider as a black-box. The motivation of doing so is to be able to tie our solution to any datastore that affords interfaces to manipulate data. As one can see, transactions may be sent to any point and afterwards they are gathered based on their access classes for execution. For example, transactions T_B, T_{BC} and T_{AC} are grouped on the first SN_1 even if they were received by SN_2 and SN_k. Moreover, it is worth-noting that SN nodes are structured over a ring for easing their collaboration.

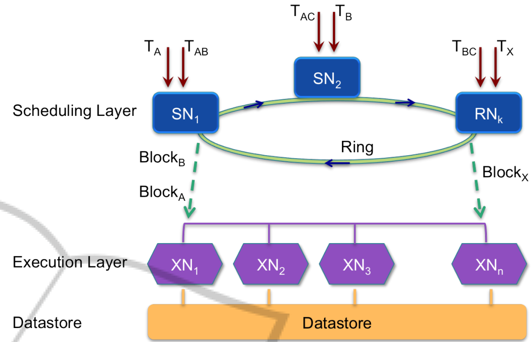
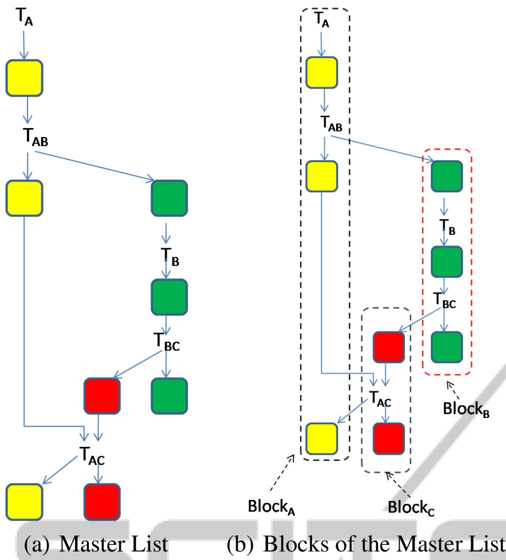


Figure 1: The layered architecture.

3.1 The Scheduling Layer

The scheduling layer is responsible to absorb the load peaks and serves as a front-end that isolates the underlying datastore nodes from the input load peak. The SN nodes that compose the scheduling layer receive any transaction request from applications or possibly from other SN nodes. Once transactions are received, SN nodes build a master list that records all transactions happening within a time window. The master list is a graph that contains two kind of information, namely, which transaction precedes another one, and what is the access class of each transaction. Figure 2(a) represents the master list of transactions described in Figure 1. One can see that T_A is the genesis transaction, *i.e.*, it is the first transaction on the list and it requires the access class A represented by the yellow square. Rather, T_{AB} requires two access classes A and B. In other words, a master list may hold transactions of different access classes. It is worth noting that each SN node maintains a master list that contains only transactions it receives from either the applications or its peers. That is, a transaction is exactly linked to one and only one master list, and when a SN node sends a transaction to another SN node, it removes it from its master list.

After the master list is established, the SN node identifies the different blocks that compose it. Each block is related to one access class. Figure 2(b) depicts the different blocks of the master list of Figure 2(a). $Block_A$ groups the transactions requiring the access class A, while $Block_B$ and $Block_C$ are respectively related to access classes B and C. For instance, $Block_A$ = {T_A, T_{AB}, T_{AC}}, $Block_B$ = {T_{AB}, T_B, T_{BC}}, and $Block_C$ = {T_{BC}, T_{AC}}. Moreover, one can see that



(a) Master List (b) Blocks of the Master List

Figure 2: Master List and Blocks.

the blocks are linked between them, which means that some transactions are requiring more than an access class. Hence, we observe that T_{AB}, T_{BC}, T_{AC} require several access classes in the example described in Figure 2(b).

Furthermore, the number of queries that a SN receives must remain bounded. To this end, we efficiently balance the load over all SNs regardless of the data requested by a transaction. We finally assess the number of SNs according to the entire workload submitted to the system, expressed in number of clients, and according to the number of transaction requests that a SN is able to handle in a unit of time.

3.2 The Storage Layer

This layer is a set of nodes called execution node (XN) that access directly to black-boxed datastore. Each XN node manages the access class that is a partition stored via the datastore. Plus, the XN node executes the block of transactions that are linked to them. Since transactions are ordered in a serial way within a block, the concurrency is already controlled at the scheduling layer. Therefore, the XNs guarantee consistent execution of transactions without locking. The XN node chosen for executing a block is obviously the one that is responsible of the access class required by transactions within the block. When blocks contain transactions requiring several access classes, thus a set of XN nodes are chosen, and Section 3.4 points out how the executions are done.

Considering a database divided into n partitions $\{p_0, p_1, \ldots, p_{n-1}\}$ such that $p_i \cap p_j = \emptyset$, we assign each p_i to at least one XN and each partition constitues a single access class. Each XN may be responsible of more than one partition, *i.e.*, an XN can hold many access classes. In this respect, each SN node may identify the XN that will be responsible of a transaction execution once the access class is found.

3.3 Communication Model

As in Blockchain model, transactions are grouped (or chained) and maintained in a decentralized manner. However, we use a more structured communication model than in the Bitcoin protocol where transactions are broadcasted to all nodes on the network using a flood protocol. In fact, we structure the SN nodes over a ring in such a way that their communication is directed and unicast. By doing so, we reduce the network overhead and it is more easy to locate where to send an incoming transaction in order to add it to the Blockchain list.

Basically, SN nodes are organized into a ring. Each SN node knows its successors and may communicate with them through a token that is a data structure. As stated in STRING (Sarr et al., 2013), we handle two distinct tokens: *processing token* and *forwarding token*. The *processing token* is used to serialize data access and the *forwarding token* is used to group transactions among SN nodes.

3.4 Building the Blocks of Transactions

Transactions are received by SNs, which handle them based on their access classes and the system status. To this end, a SN node needs to know where the data partitions are stored and, if such partitions are replicated on several XN, what is the less overloaded XN. The expected benefit is to minimize execution time. With this in mind, after receiving a transaction, T, modifying the partition p_i, a SN may process as follows:

- If τ_{p_i} is under its control, thus the SN reads metadata of p_i and assess where the T's latency will be the shortest regarding to its execution.
- else, it adds T in the block list till it acquires τ_{p_i} as well as it may decide to forward T to another SN for shortening T's latency.

When SN_1 decides to forward a transaction T to SN_2, thus SN_2 has to add T to the block list corresponding to the access class of T. SN_2 keeps T within a block until it gets τ_{p_i}.

Briefly, transactions are grouped when SNs forward transactions to others in order to reduce waiting time.

The blocks are built by using the grouping algorithms we described in STRING (Sarr et al., 2013).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

In that work, we proved that the time-based and ring-based approches produce groups with bigger size. Moreover, in a context where it is only matter of reducing the logs or datastore accesses, the time based is more suited followed by the ring approach. Therefore, we choose to rely on the ring-based approach for single partition transactions for low latency and the time-based for multi-partitions transactions to be able to handle them rapidly and to free ressources.

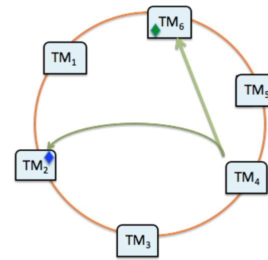


Figure 3: Inquiring several processing tokens.

4 DEALING WITH MULTI-PARTITION TRANSACTIONS

As pointed out early, processing tokens are used to synchronize concurrent transactions while avoiding starvation. We describe in the following subsections how we manage processing tokens to deal particularly multi-partition transactions. When a transaction requires several partitions, all corresponding tokens may be already hold elsewhere and thus, must be managed efficiently to shorten the response time. To this end, we propose two approaches to face such situations : an eager approach and a lazy one. The eager approach tries to handle multi-partition transactions as soon as possible they arrive, while the lazy approach delay their execution in order to manage efficiently their requirements in terms of access requests and to shorten the global latency.

4.1 Eager Approach

The eager approach gives a high priority to multi-partitions transactions. Therefore, once a multi-partitions transaction T is detected within SN_1's master list, SN_1 sends an alert to all other SN nodes to enter in idle time, *i.e.*, SN nodes have to push rapidly the corresponding tokens to SN_1 by suspending the execution of any transaction requiring the same partition as T.

To this end, the forwarding token is used not only for transferring transactions, but also for notifying SN nodes to not use any token required for handling T. Moreover, each SN manages its own forwarding token and we rely on the time-based grouping algorithm when forwarding transactions. To detail the steps followed when handling a multi-partition transaction, we consider the Figure 3. Assume that SN_4 receives a multi-partition transaction requiring both p_i and p_j. Thus, it identifies the SNs (SN_6 and SN_2 respectively) holding the processing tokens τ_{p_i} (green diamond) and τ_{p_j} (blue diamond). Once this identification done, SN_4 sends a specific message, called an *inhibiting* message, by using its forwarding token to SN_6 and SN_2. An *inhibiting* message states that all successors of SN_6 and SN_2 must not use (or hold) τ_{p_i} and τ_{p_j} even if they have in their pending list some transactions related to p_i and p_j. That is, SN_4 inhibits successors of SN_6 and SN_2 and will be the next one to hold and allowed to use both tokens τ_{p_i} and τ_{p_j}. The intuition behind such a strategy is to give priority to multi-partition transactions that are somewhat less frequent. Moreover, when a successor is inhibited, it has to forward transactions in its pending list by using the grouping algorithm. In other words, the inhibiting process leads to group transactions and thus, to shorten their execution time as pointed out earlier. However, when the number of multi-partition transactions becomes high, single transactions may suffer from starvation, and the response time is lengthened. To avoid starvation in case of a subsequent arrival of multi-partitions we propose a lazy approach that we describe in next section.

4.2 Lazy Approach

The lazy approach introduces a certain asymmetry in the SN nodes roles. In fact, one of the SN node is responsible of gathering and routing all multi-partitions transactions happening within a time window. It acts like a leader node regarding to other SN nodes that have to send to it all multi-partitions transactions they received. A time window lasts as long as the time required for a token to complete a round over the ring. This role of leading the execution of all multi-partitions transactions is not given statically to a given SN, but in a round robin fashion to all SN nodes. That is, a SN node gives the leader role to its successor once it finishes to route multi-partitions transactions received within a time period. The main reason of doing so is to balance the overall workload on all SN nodes and to avoid single point of failure for a kind of transactions. This approach uses only one forwarding token as in the ring-based algorithm described in (Sarr et al., 2013). The forwarding token

transfers multi-partitions transactions to the SN node leader. To explain step by step how the work is done, let us assume that SN_0 comes just to be elected as the leader. That is, the forwarding token is picking up multi-partitions transactions from SN_1, SN_2 and so on. Unless the forwarding token reaches again SN_0, each SN nodes removes every multi-partition transaction from its master list and adds it to the token. At the end of the time window, *i.e.*, the token is at SN_0, the leader requisitions all tokens required for processing multi-partition transactions. It is worth noting that a leader may predict the remaining time to get back the forwarding token. Moreover, it can start gathering the processing tokens related to the transactions that it has directly received from client application nodes before the forwarding token brings to it the remaining multi-partition transactions. This prediction is possible thanks to our time-based algorithm and it eases/accelerates the tokens requisition process.

The drawback of this approach is that the execution orders may differ from submission orders. Actually, between the reception of a multi-partition transaction and its execution, others single transactions may arrived and handled by others SN. However, since the delay between a reception and the processing of a transaction is short, even though with the lazy approach, it is worth noting that the number of incoming concurrent transactions is low. In the realm of social applications, this approach is suited since conflictual transactions are infrequent and the order of some interactions (say comments or Like) is not important.

5 VALIDATION

In this section we validate our approach through simulation by using SimJava (Howell and Mcnab, 1998), which is a toolkit (API Java) for building working models of complex systems. It is based on discrete events simulation kernel and includes facilities for representing simulation objects. We implement each of entities such as clients, SN nodes and XN nodes. Each entity is embedded into a thread and exchanges with others through events. To be as close as possible to a real system, each client that sends a query has to wait results before sending another one. To balance client requests over all SN nodes, clients use a round robin fashion to send a query to an SN node.

The main objective of our experiments is to assess the performances of our solution. To this end, experiments were conducted on an Intel duo core with 2 GB of RAM and 3.2 GHz running under Windows 7.

5.1 Evaluating Multi-partitions Transaction Latency

The main goal of this experiment is to evaluate and compare the two mechanisms proposed for facing multi-partitions transactions. We compare the two approaches in terms of latency as well as in terms of overhead. To this end, we set 10 SN nodes, 10 XN nodes (i.e. 10 partitions) and we vary the number of clients from 100 to 600. Moreover, we set a concurrency rate of 30% and 50% of transactions are multi-partitions. Figure 4 depicts the average response time when the workload increases. As one can see, the response time grows slightly and remains low for a small workload and it increases more rapidly when the workload becomes heavy. This is true for both approaches, either the eager approach or the lazy one.

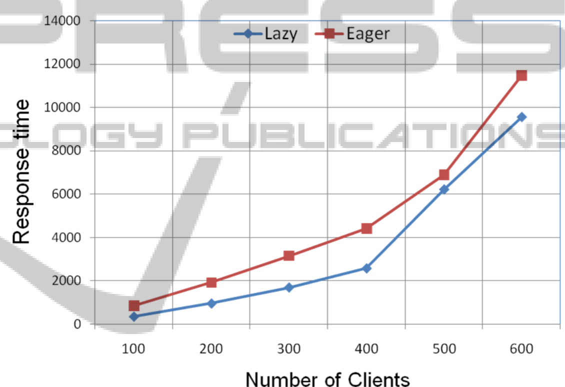


Figure 4: Response time vs. number of XN (or number of tokens).

However, the response time of the lazy approach stays low compared to the response time of the eager version. The main reason is that in the eager version, the SN nodes alternate the execution of the single transactions and the one of multi-partitions transactions. Basically, the execution of each multi-partitions transaction requires to suspend most of the single transactions, thus, when the number of multi-partitions is important, the waiting time of single transactions increases. The effect of this situation is attenuated in the lazy version where most of the single transactions are executed before the multi-partitions transactions are grouped and handled in a shot. In the lazy strategy, almost all tokens are available since single transactions are already proccessed before the leader starts executing multi-partitions transactions. Therefore, the overall response time is less important than in the eager version.

5.2 Overhead of the Eager and Lazy Approaches

Furthermore, we carried out an experiment to measure the network overhead of our solution, and to identify which approach costs more in terms of messages. We use the same configuration as previously, *i.e.*, we consider 10 SN nodes, 10 XN and a concurrency rate of 30%. We report in Figure 5 the results that unveil the high number of messages used by the eager approach where the lazy one requires less messages. This is due to the fact that the eager approach generates a set of messages for each multi-partitions transactions while the lazy waits a time window and aggregates/optimizes the total messages to send for routing and processing transactions.

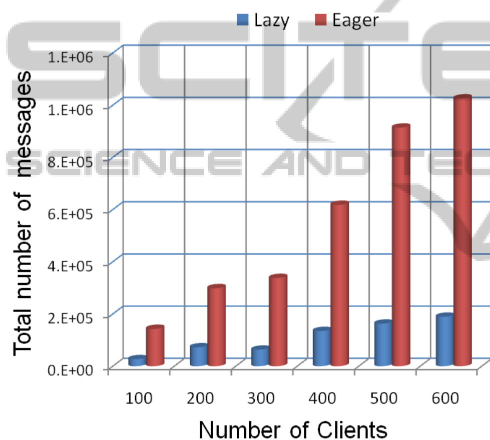


Figure 5: Number of messages vs. number of XN (or number of tokens).

6 CONCLUSION

In this paper, we propose blockchain-based model for routing social transactions. It is a two steps approach: a scheduling step followed by an execution step. The transactions are ordered during the scheduling step in such a way that each transaction is followed or preceded by another one within a block and based on their access class. Afterwards, each block of transactions is executed at the nodes storing the required data. Once a block is sent for execution, it remains unchanged and hence, the execution order stays identical for all the nodes involved. To reach our goal, we rely on the algorithms proposed in our previous works (Sarr et al., 2013) to reduce the communication cost. Moreover, we propose a lightweight concurrency control by using tokens that serve to synchronize simultaneous access to the same data. We focus specially on the case in which transactions require several access classes. We designed and simulated our solution using SimJava and we ran a set of experiments. Ongoing works are conducted to evaluate completely our solution in a cloud platform and to manage group transactions size for optimal execution.

REFERENCES

Abadi, D. (2012). Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *IEEE Computer*, 45(2):37–42.

Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. (2007). Sinfonia: a new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):159–174.

Barber, S., Boyen, X., Shi, E., and Uzun, E. (2012). Bitter to better — how to make bitcoin a better currency. In *FCDS*, volume 7397 of *LNCS*, pages 399–414.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: a distributed storage system for structured data. In *USENIX OSDI*, pages 15–15.

Das, S., Agrawal, D., and El Abbadi, A. (2013). Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM TODS*, 38(1):5–45.

Howell, F. and Mcnab, R. (1998). simjava: A discrete event simulation library for java. In *ICWMS*, pages 51–56.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. (2008). H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40.

Michael, M. M. and Scott, M. L. (1995). Implementation of atomic primitives on distributed shared memory multiprocessors. In *IEEE HPCA*, pages 222–231.

Oracle, C. (Retrieved on November 2014). Oracle nosql database, 11g release 2.

Pandis, I., Johnson, R., Hardavellas, N., and Ailamaki, A. (2010). Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939.

Pandis, I., Tözün, P., Johnson, R., and Ailamaki, A. (2011). Plp: Page latch-free shared-everything oltp. *Proc. VLDB Endow.*, 4(10):610–621.

Sarr, I., Naacke, H., and Moctar, A. O. M. (2013). STRING: social-transaction routing over a ring. In *DEXA*, pages 319–333.

Silberstein, A., Chen, J., Lomax, D., McMillan, B., Mortazavi, M., Narayan, P. P. S., Ramakrishnan, R., and Sears, R. (2012). Pnuts in flight: Web-scale data serving at yahoo. *IEEE Internet Computing*, 16(1):13–23.

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12.

Vogels, W. (2009). Eventually consistent. *Commun. ACM*, 52(1):40–44.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS