IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 19, 2021, accepted April 23, 2021, date of publication May 5, 2021, date of current version May 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077680

Highly Scalable Distributed Architecture for NoSQL Datastore Supporting Strong Consistency

ADAM KRECHOWICZ, STANISŁAW DENIZIAK, (Member, IEEE), AND GRZEGORZ ŁUKAWSKI
Faculty of Electrical Engineering, Automatic Control and Computer Science, Kielce University of Technology, 25-314 Kielce, Poland
Corresponding author: Adam Krechowicz (a.krechowicz@tu.kielce.pl)

ABSTRACT Introducing a strong consistency model into NoSQL data storages is one of the most interesting issues nowadays. In spite of the CAP theorem, many NoSQL systems try to strengthen the consistency to their limits to better serve the business needs. However, many consistency-related problems that occur in popular data storages are impossible to overcome and enforce rebuilding the whole system from scratch. Additionally, providing scalability to those systems really complicates the matter. In this paper, a novel data storage architecture that supports strong consistency without loosing scalability is proposed. It provides strong consistency according to the following requirements: high scalability, high availability, and high throughput. The proposed solution is based on the Scalable Distributed Two–Layer Data Store which has proven to be a very efficient NoSQL system. The proposed architecture takes into account the concurrent execution of operations and unfinished operations. The theoretical correctness of the architecture as well as experimental evaluation in comparison to traditional mechanisms like locking and versioning is also shown. Comparative analysis with popular NoSQL systems like MongoDB and MemCached is also presented. Obtained results show that the proposed architecture presents a very high performance in comparison to existing NoSQL systems.

INDEX TERMS NoSQL, NewSQL, SD2DS, consistency.

I. INTRODUCTION

Contemporary applications more and more frequently store and process large volumes of data. In the case of very large data sets or data sets operated by many clients simultaneously, the limits of classical SQL Relational Database Management Systems (RDBMS) are usually exceeded. The problem arises in the case of Internet applications where billions of people have access to the database. Moreover, a rapidly growing number of big data applications requires efficient database architectures supporting scalability and availability for data analytics.

In such cases, simpler but more efficient datastores are widely used instead of RDBMS systems. A lot of distributed datastore systems have been developed so far and they are usually referred as NoSQL. Such systems usually lack strong relational data models, ACID transactions (Atomic, Consistent, Isolated and Durable) and strong consistency models, but offer much higher throughput, efficiency and high scalability instead.

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

Consistency is one of the most important features of all information systems [1]. Its role is especially visible in distributed systems where its lack may lead to serious problems (e.g., data loss). However, preserving consistency in all situations may be a very difficult task. It is important to remark that some inconsistencies may be tolerable in certain situations (or for some time). Because of that, a concept of consistency model was introduced. A consistency model describes the system tolerance to inconsistencies. A weak consistency model implies that the system may tolerate some inconsistencies. A *strong strict consistency* model implies that the system does not tolerate inconsistencies at all.

Database scalability is the ability to scale depending on the workload. Thus, a highly scalable datastore should efficiently handle a large amount of data, great volume of requests, and large sizes of requests. The database may support the scalability by vertical or horizontal scaling. Vertical database scalability is adding more capacity to a single machine. In horizontal database scaling, the capacity is increased by adding more machines. Only the horizontal scalability may expand beyond the limit of the single machine, but it requires a distributed database architecture. Commonly used

techniques of horizontal scaling are: data replication, data partitioning, or distributed processing of requests. However, as it was proved in [2], to achieve horizontal scalability, availability, and partition tolerance, database systems cannot impose strong consistency.

In this paper, a highly scalable datastore supporting strong consistency is presented and evaluated. The datastore is based on Scalable Distributed Two-layered Data Structures (SD2DS). SD2DS concept was used to build a scalable and efficient key–value datastore. Because of the fact that the data stored in SD2DS may be split into multiple parts and/or duplicated, the problem of keeping the data consistent is essential in this case. This paper presents an analysis of the consistency model of SD2DS. Moreover, two different approaches to improving the efficiency of the datastore are presented and evaluated.

The main contributions of this paper are two consistency models that are used with basic SD2DS architecture. They allow ensuring strong consistency without affecting the scalability. The first of the developed model ensures the of all operations that are executed on the system. The second model is responsible for scheduling only those operations that affect the content of the datastorage. The contribution of this paper is as follows:

- analysis of all consistency issues that may arise in SD2DS based datastore;
- design of scheduling algorithms for supporting strong consistency in SD2DS;
- theoretical proof of the correctness of those algorithms;
- practical implementation of datastorages based on those algorithms;
- performance evaluation of those datastorages with comparison to the well-known NoSQL systems (mongoDB and MemCached).

It is also worth to notice that our goal was to augment the existing SD2DS datastorage conception with mechanisms that support strong consistency without affecting the scalability rather than to develop a complete system from scratch.

In the following section, the problem of consistency in NoSQL is discussed further. Sections 3 and 4 describe the basic properties of Scalable Distributed Data Structures in single and two–layer design, respectively. Section 5 is focused on the proposed strong consistency model developed for SD2DS. In the next section, the enhanced SD2DS is evaluated and compared with a few existing NoSQL datastores. The paper ends with conclusions.

II. PROBLEM STATEMENT

The Relational Database Management Systems have been a standard way of storing data for many years. Nowadays, they are actively used even in many newly developed systems. Databases require the definition of a custom data schema which must follow some consistency constraints. The assurance of these constraints is guaranteed by the *ACID transactions* [3]. The ACID abbreviation stands for Atomicity, Consistency, Isolation and Durability which indicates four main features of transactions. Each transaction can be considered as a set of operations that needs to be performed together on a set of data. The transaction is *committed* when all operations have been properly executed. If some operations could not be executed (when they violate the consistency constraints) the transaction needs to be *rollbacked* and all operations within that transaction need to be reverted [4]. Generally, the transactions in RDBMS can be realized with one of two basic strategies: optimistic and pessimistic. In the optimistic approach, it is assumed that each transaction will not violate the consistency. This assumption requires transaction verification and if the consistency constraints cannot be satisfied, the transaction needs to be reverted [4]. In contrast, the pessimistic approach assumes that each transaction can violate the consistency constraints. Because of that, each transaction needs to acquire a lock on a data item to ensure proper execution [4]. While reducing the number of rollbacks, the pessimistic approach has serious drawbacks concerning performance issues and deadlocks. The deadlocks can be avoided to some extent by utilizing the two-phase locking protocol, which indicates that after even one unlocking further lock acquisition is impossible [5]. In typical RDBMS systems the ACID transactions are performed by a transaction manager which usually does not work efficiently when the whole system is distributed [3].

One of the most significant limitations of RDBMS systems is the data capacity [6]. In many modern systems, the sets of data are so large that they need to be distributed to many nodes within the cluster [7]. It is one of the main reasons to give up the RDBMS for the NoSQL systems. In their basic form, most of the NoSQL systems do not provide a strong consistency model. The CAP theorem [8] tells that ensuring strong consistency, availability, and partition tolerance in a distributed system at the same time is impossible. Because of that, the strong consistency model is often sacrificed in favour of the two other above mentioned features [9].

The goal of this work is to develop a highly scalable architecture of NoSQL datastore providing strong consistency, according to the following requirements:

1) High scalability in terms of data capacity - this may be satisfied only by distributed architecture, where data will be distributed among nodes of the cluster (horizontal scalability).
2) High scalability in terms of the volume of requests - this may be achieved by distributed/parallel processing of requests.
3) Strong consistency model - the strong consistency should be proven for all basic operations: READ, WRITE, DELETE and UPDATE.
4) High availability - the architecture should ensure that each request will result in a response.
5) High throughput - in many existing approaches high efficiency is achieved through applying weaker consistency models. The main goal of our approach is to

provide high throughput without giving up the strong consistency of the datastore.

III. RELATED WORKS

NoSQL systems may be assigned depending on the utilized model of data to one of four main categories: column-oriented (e.g., Cassandra [10]), key–value (e.g. memcached [11]), document (e.g. MongoDB [12]) and graphs (e.g. Neo4j [13]). Most of the NoSQL systems support a very weak consistency model which follows the concept of Basically Available, Soft state, Eventually consistent (BASE) [2] model. Eventual Consistency (EA) means that the state of the system will become consistent if no further modifications occur [14]. This implies that at some points in time the state of the system may be inconsistent. The models based on eventual consistency are usually developed on the top of the anti-entropy protocols [15] (like epidemic algorithms [16]) which try to minimize the changes between the state of datastore nodes. Causal Consistency [17] is a stronger model than EA but it is usually used for maintaining multiple replicas of data [18]. A good overview of consistency models of NoSQL databases can be found in [19].

Nowadays, the NoSQL data storages are used in more and more applications. The weak consistency models start to be insufficient in many real–world implementations. For example, Facebook needs to change the internal data storage [10] in order to achieve better consistency [20]. The need to develop stronger consistency models is increasingly visible [21] even for social network systems [22]. The justification created by the CAP theorem becomes insufficient for many applications and leads to many critical considerations because the lack of consistency is more and more severe [23]–[25]. This led to the emergence of a new trend in NoSQL world called NewSQL [26]. NewSQL systems allow to create Online Transactional Processing (OLTP) solutions on the base of distributed datastorages [27]. The first NoSQL systems that support stronger consistency models (like Calvin [28], Megastore [29] and Spanner [30]) were mostly built on the top of the already developed systems (for example by adding Multiversion Concurrency Control [31]). The new standard of consistency model was developed and is described as Basic Availability, Scalability, Instant Consistency (BASIC) [32]. It eliminates the eventual consistency model in favour of better consistency models. Already developed datastores use different algorithms for keeping data sets consistent, e.g., locking, versioning, or write-through strategy. In each case, introducing those mechanisms has a strong influence on the performance of the whole system. Additionally, many of those systems cannot be run on commodity hardware due to special hardware demands (like GPS signals or access to atomic clocks) for performing synchronization [30].

One of the most recognizable examples of NewSQL, ensuring a strong consistency model, is VoldDB [33]. However, it can only execute predefined operations in the form of stored procedures. It was also optimized only for very simple transactions (so-called single–node and one–shot transactions). Furthermore, it does not provide any advanced processing methods and does not ensure scalability. RubatoDB [34] is one of the most promising system. It can support BASE, BASIC as well as ACID consistency models. It uses timestamp–base concurrency as well as two–phase commit protocol to ensure consistency. The performance of the RubatoDB is comparable to other NewSQL systems, but is much lower than the other commonly used systems. Furthermore, the performance is obviously correlated with the chosen consistency model. In spite of the fact that those systems are distributed, they may now be considered as RDBMS [27].

Consistency in distributed systems can be provided by one of the existing protocols that were designed during many years of research [35]. Two–phase commit [4] is one of the most recognizable examples. It allows to ensure the same system state between many participants. It consists of two stages. In the first stage, all participants perform voting. The second stage is executed only if every participant is ready to accept changes. The two–phase commit protocol was successfully implemented in many systems despite some serious limitations. One of the main drawbacks of this protocol is the risk of failure. The protocol fails even if one of the participants does not work correctly. Furthermore, it requires a special element, called the coordinator, to perform the whole operation. Failures in the coordinator node could really affect the whole system. The three–phase commit protocol [36] was designed to eliminate some of those drawbacks concerning faulty participants. It introduces the third stage that allows to resolve the state of the system even if some participants do not work correctly. The Paxos protocol [37] is the extension of such protocol that allows to maintain consistency while tolerating faults. It is a distributed consensus algorithm that allows to make a commit, even in a faulty environment. Despite the correctness and many additional variants of this protocol [38], [39] it is rarely used in developed systems due to its complication. Because of that, its simplified versions were developed for easier implementation [40], [41]. However, the complexity of those algorithms makes them suitable only for coarse-grained configurations, like in ZooKeeper [42] or Chubby [43], and they are not well suited for ensuring consistency for a single data item.

In the last years, there have been many attempts to build a fully scalable database that could provide an acceptable level of consistency known previously from RDBMS systems. Currently used systems such as MIDDLE-R [44], Postgres-R [45] or RocksDB [46] seriously suffer from a trade-off between consistency and scalability. In that case, consistency is often sacrificed. Many different definitions and approaches to scalability further complicate this matter. The term workload scalability is defined as the ability to scale relative to the number of clients simultaneously operating on the system. Its variant elasticity represents the ability to handle changing workload. Additionally, the term data scalability means that it is possible to scale relative to the total size of the stored data. Current state of the art systems solve the problems related to scalability using replication.

Mostly they are developed using existing RDBMS engines with specialized replicated back-end storage [46] or additional middleware level [47]. Those replication-based solutions may be organized both as Master-slave or Multi-master systems [47]. Consequently, more specialized consistency mechanisms are needed to improve/support those systems. Multi-master systems need to use complex quorum algorithms. However, they can still suffer from problems with changing configurations (adding and removing slave nodes). Synchronization of replicas can be organized both as eager or lazy. The lazy approach is not typically suited for preserving strong consistency. On the other hand, eager approach may drastically influence performance as well as scalability. It is often the case that slaves are updated sequentially and that what drastically influences scalability [48]. The communication overhead between replicas (for data and synchronization messages) also has a negative impact on scalability (e. g. group communication) [44]. The synchronous updates may also cause deadlocks with probability proportional to x3, where x is a number of replicas. All in all, to provide strong consistency (with eager synchronization), only few replicas can be used [49]. To sum up, the currently available solutions are based on extending existing solutions. Unfortunately, some of their drawbacks are left without proposing suitable corrective solutions, which could allow to maintain consistency at a higher level. Typically, the existing non-replicated transaction manager can be still a serious bottleneck [46]. Most of the currently developed systems are optimized for a very narrow class of applications and are far away from the universality known from traditional RDBMS systems. According to [49] a very promising solution to preserve both scalability and consistency is to prepare dedicated algorithms to perform client scheduling. Such a novel approach to maintaining consistency without loosing scalability for Big Data will be the subject of this paper. To the authors' best knowledge, this work is the first one that deals with inconsistency issues in multi–layered datastore architecture. Moreover, this work provides first complete solution to those problems maintaining a very good efficiency and data scalability at the same time.

IV. SCALABLE DISTRIBUTED DATA STRUCTURES

Scalable Distributed Data Structures (SDDS) [50] were developed for storing and processing large sets of data. This is mainly achieved by storing the data in RAM of a distributed system. The data distributed among many multicomputer nodes may also be efficiently processed by many machines simultaneously.

The data stored in an SDDS are divided into *records*. Each record is uniquely identified with a *key*, usually in the form of unsigned integer value. Each node stores a limited number of records in RAM, in a fixed–size container called *bucket*. The capacity of each bucket is equal. A node may store multiple buckets if it has enough RAM. All buckets form a single, consistent structure called *file*.

Records of the SDDS are addressed using various methods like Linear Hashing (LH*) [51], Range Partitioning (RP*) [52] or tree-based structures [53]. One of the most efficient addressing methods is LH*, which may be used in the two-layer version of the SDDS. It is presented in section V.

For the sake of scalability, SDDS uses a distributed directory instead of a centralized one. Each node accessing the data, called *a client*, maintains its own local directory, called *a file image*. Because the number of buckets in a file changes during the evolution of the file, clients' images may become outdated. In such a case, a client may commit *an addressing error* while trying to access a wrong bucket. Incorrectly addressed messages are redirected by buckets using so called *forwarding*. In the worst case, LH* will allow to reach the correct destination bucket after two additional messages sent between buckets. More details of LH* are given in [51].

Basic SDDS supports simple data operations such as INSERT, UPDATE, RETRIEVE and DELETE. Since a single bucket holds all information about a record, all basic operations may be considered as atomic. As more and more data are inserted into the SDDS file, a bucket may become overfilled. Before it happens, a new, empty bucket is created and about half of the capacity of the overloaded bucket is moved into a new one. Such a process is called *a bucket split*. Analogously, buckets may be *merged* if more and more records are deleted and the space of the bucket is not used efficiently.

The decision whether a bucket should be split (or it should be merged with another bucket) may be undertaken in two ways. SDDS RP* uses a decentralized algorithm, where a bucket decides whether to split on its own [52]. On the other hand, SDDS LH* uses an additional file component called *a Split Coordinator (SC)*. In the latter case, an overloaded bucket sends to the SC a special message called *collision*. Next, the SC decides whether and which bucket should split.

The main drawback of the SDDS is the performance during the split operation. Because in each split the half of the bucket content must be transferred, at least 33% of all data transfers is consumed by these operations [54]. It can be a serious issue in applications with big data velocity that requires a lot of splits during operations. Additionally, performing a split on large records also consumes a lot of network transfer.

V. SCALABLE DISTRIBUTED TWO-LAYERED DATASTORE

The Scalable Distributed Two-layered Data Structures (SD2DS) [54] were introduced to overcome the disadvantages of single-layered structures. They allowed to develop an efficient datastore for large files [55], [56]. Apart from improving the split performance, they also allow to introduce other features like throughput scalability [57], [58] or anonymity [59], [60]. We have already applied the SD2DS concept into content search datastore [61]–[63] and scalable IoT system [64]. The datastore is continuously integrated with many Big Data analysis mechanisms [65], [66]. But all mentioned above applications of SD2DS did not support the data consistency.

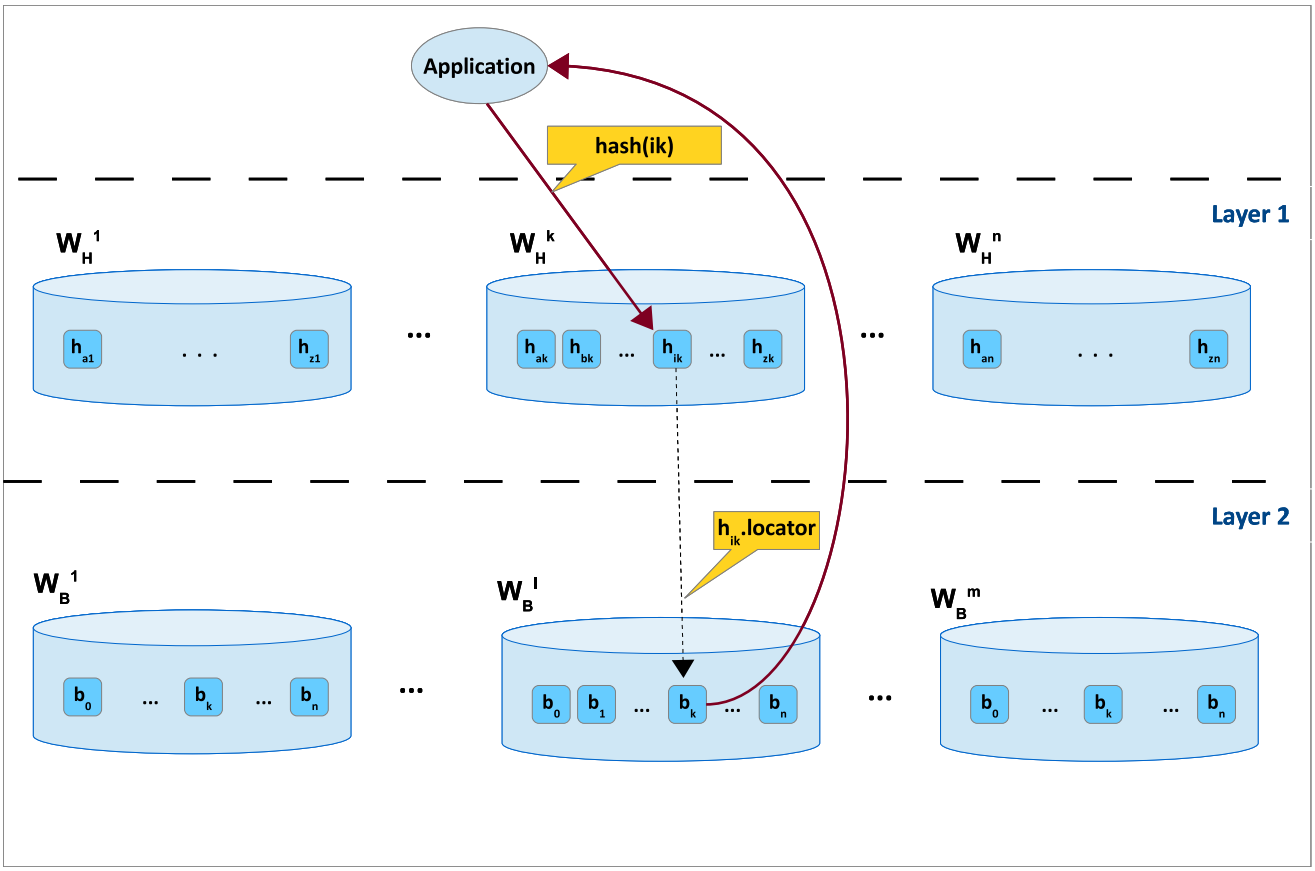

FIGURE 1. Architecture of Scalable Distributed Two-layered Datastore.

The overall architecture of SD2DS is presented in Fig. 1. The data items, stored in SD2DS, are called *components* and are divided into two elements: component *header* and component *body*. The single component is defined as follows:

$$c_k = (h_k, b_k) \tag{1}$$

The header is stored in the first layer of the structure and contains metadata of the designated component. The most important part of this metadata is *locator* which points to the actual data stored in the component body in the second layer of the structure. Each component body consists of the stored data while the component header is described as:

$$h_k = (key, locator) \tag{2}$$

The whole sets of headers and bodies are defined as follows:

$$H = \{h_0, h_1, \ldots, h_n\} \tag{3}$$

$$B = \{b_0, b_1, \ldots, b_n\} \tag{4}$$

All components that are stored in SD2DS, create a *file* which is defined as:

$$F = \{c_0, \ldots, c_n\} \tag{5}$$

The bucket is divided into two completely separate containers: a first layer bucket W_H^i and a second layer bucket W_B^i. The W_H^i buckets are responsible for storing the headers while the W_B^i are responsible for storing bodies. It is essential to outline that the first layer and the second layer in SD2DS might be totally separated and located on different infrastructures. The first and the second layer buckets are described as follows:

$$W_H^i = \left\{ h_0^i, h_1^i, \ldots, h_n^i : \underset{j=0,\ldots,n}{\forall} (h_j^i \subset H) \right\} \tag{6}$$

$$W_B^i = \left[b_0^i, b_1^i, \ldots, b_m^i : \underset{j=0,\ldots,m}{\forall} (b_j^i \subset B) \right] \tag{7}$$

Operations that are performed on SD2DS, require access to both layers of the structure. The Distributed LH* scheme is used to correctly address components in the first layer of SD2DS. It allows to identify the first layer bucket which contains the header for a given key. All functions, that operate on SD2DS, need to identify the receiving first layer bucket by using $Hash(k)$ function which has to fulfill the following condition:

$$\underset{c_k \in C}{\forall} \left(h_k \in W_H^{Hash(h_k.key)} \right) \tag{8}$$

The second layer bucket is identified by a $h_k.locator$. The locator has to satisfy the following condition:

$$\underset{c_k \in C}{\forall} \left(b_k \in W_B^{h_k.locator} \right) \tag{9}$$

To properly address the bodies in the second layer, the $Addr(b_k)$ function was introduced, which follows the condition:

$$\underset{k}{\forall}\left(Addr(b_k) = h_k.locator\right) \tag{10}$$

It is also worth to notice that headers and their corresponding bodies can be stored in the same server. That variant can be considered as equal to the original SDDS described in the previous section. However, dividing headers and bodies in such a way that they are managed separately allows to overcome a transfer issue that arises when the whole components are moved during the split operations. In this form, LH* addressing scheme is applied only to the first layer that contains relatively small headers. The location of the component bodies, which are typically much larger than component headers, never changes. Even when the location of the header changes, the locator field always points to the same location of the body. Because of the fact that in typical implementations the component header is much smaller than the component body, the transfer needed for split is seriously reduced. Consequently, the time of split operation is also significantly reduced and the whole performance of the storage is greatly improved.

It is also worth to notice that while a single server can contain both the first layer bucket and the second layer bucket, there is no guarantee that both the header and the body of a single component are located on the same server. Due to the split operation, the component headers may change their location (are moved between buckets), while the component bodies will always reside in the same location.

The basic functionality that should be provided to operate on SD2DS is based on the following primary operations:

- $PUT(k, b_k) = CH \rightarrow Z_1 \rightarrow Z_2 \rightarrow S$ defined in Algorithm 1. Operation is responsible for inserting a new component into SD2DS. The function $new(b_k)$ was used to create the new locator for the new component. The definition of this function is not relevant to this analysis and is strictly related to the implementation. The $MAX\_CAPACITY\_H$ represents the capacity of the first layer bucket while $Split(W_H^i)$ performs the split of the bucket which follows the steps described in [50].
- $GET(k) = CH \rightarrow O_1 \rightarrow O_2$ defined in Algorithm 2. Operation is responsible for retrieving the existing component from the SD2DS.
- $DEL(k) = CH \rightarrow U_1 \rightarrow U_2$ defined in Algorithm 3. Operation deletes the existing component.
- $UPDATE(k, b_i') = CH \rightarrow O_1 \rightarrow U_2 \rightarrow Z_2$ defined in Algorithm 4. Operation performs modification of the existing component.

Thus, the set of primary operations that can be applied on SD2DS is defined as:

$$\Omega = \left\{PUT(k, b_k), GET(k), DEL(k), UPDATE(k, b_k')\right\} \tag{11}$$

Primary operations defined in the Algorithms 1–4 consist of atomic operations, which are executed in sequence. The specification of sequences uses the "$\rightarrow$" operator which means "happened before", as it was introduced in [67]. The order of atomic operations has to be strictly preserved. However, it is possible to reverse the order of U_2 and

Algorithm 1 $PUT(k, b_k)$ Inserting Component in SD2DS

$$\begin{aligned}
&CH: i \leftarrow Hash(k)\\
&Z_1: \begin{cases} if\ \left(\exists_x (h_x \in W_H^i \wedge h_x.key = k)\right)\ then\\ \quad STOP\\ end\ if\\ h_k.key \leftarrow k\\ h_k.locator \leftarrow new(b_k)\\ W_H^i = \{h_k\} \cup W_H^i \end{cases}\\
&Z_2: W_B^{h_k.locator}.append(b_k)\\
&S: \begin{cases} if\ \left(\left|W_H^i\right| > MAX_CAPACITY_H\right)\ then\\ \quad Split(W_H^i)\\ end\ if \end{cases}
\end{aligned}$$

Algorithm 2 $GET(k)$ Getting Component From SD2DS

$$\begin{aligned}
&CH: i \leftarrow Hash(k)\\
&O_1: \begin{cases} if\ \left(\exists_x (h_x \in W_H^i \wedge h_x.key = k)\right)\ then\\ \quad h \leftarrow h_x\\ else\\ \quad STOP\\ end\ if \end{cases}\\
&O_2: result \leftarrow W_B^{h_k.locator}.find(b : (h, b) \in F)
\end{aligned}$$

Algorithm 3 $DEL(k)$ Deleting Component From SD2DS

$$\begin{aligned}
&CH: i \leftarrow Hash(k)\\
&U_1: \begin{cases} if\ \left(\exists_x (h_x \in W_H^i \wedge h_x.key = k)\right)\ then\\ \quad h \leftarrow h_x\\ \quad W_H^i = W_H^i \backslash \{h\}\\ else\\ \quad STOP\\ end\ if \end{cases}\\
&U_2: W_B^{h.locator}.delete(b : (h, b) \in F)
\end{aligned}$$

Algorithm 4 $UPDATE(k, b_i')$ Updating Component in SD2DS

$$\begin{aligned}
&CH: i \leftarrow Hash(k)\\
&O_1: \begin{cases} if\ \left(\exists_x (h_x \in W_H^i \wedge h_x.key = k)\right)\ then\\ \quad h \leftarrow h_x\\ else\\ \quad STOP\\ end\ if \end{cases}\\
&U_2: W_B^{h.locator}.delete(b : (h, b) \in F)\\
&Z_2: W_B^{h_k.locator}.append(b_k)
\end{aligned}$$

Z_2 within primary operation $UPDATE(k, b'_i)$. The proposed order was chosen to avoid the storing of two versions (the old and the new one) of component bodies simultaneously. In case of many simultaneous updates, these old versions of bodies, still stored in buckets, may cause bucket overflow, which consequently leads to the allocation of a new bucket in the second layer of the SD2DS. This new bucket will become useless after completion of all operations. As a result, the overall performance as well as memory utilization level may seriously deteriorate. However, in the later sections the case with changed order of these operations will also be considered.

Each operation is initiated by SD2DS client and performs some actions on both layers of the structure. Because of that, some atomic operations are executed on the client side, others are executed on the first layer, and finally others of them are executed on the second layer buckets. The set of atomic operations that are executed on the client side is defined as:

$$Op_K = \{CH\} \tag{12}$$

The set of atomic operations that are executed on the first layer is defined as:

$$Op_1 = \{Z_1, O_1, U_1, S\} \tag{13}$$

The set of atomic operations that are executed on the second layer is defined as:

$$Op_2 = \{Z_2, O_2, U_2\} \tag{14}$$

VI. CONSISTENCY MODEL

Introducing a second layer into SDDS may lead to many problems concerning consistency. Because each function that operates on SD2DS, requires access to both layers, there is a serious risk of component inconsistency. We can define *components consistency* as follows:

Definition 1: The SD2DS components are consistent when

$$\underset{k}{\forall}\ \underset{h_k \in H}{\exists!}\ \underset{b_k \in B}{\exists!}\ \left((h_k, b_k) \in F\right)$$

Concurrent or unfinished executions of operations may lead to situations in which inconsistent components are created in the file. We identified three types of inconsistencies which are defined as follows:

Definition 2: Orphan Header Inconsistency (*OHI*) is defined as:

$$\underset{h_i \in H}{\exists}\ \underset{b_j \in B}{\forall}\ \left((h_i, b_j) \notin F\right)$$

Definition 3: Orphan Body Inconsistency (*OBI*) is defined as:

$$\underset{b_i \in B}{\exists}\ \underset{h_j \in H}{\forall}\ \left((h_j, b_i) \notin F\right)$$

Definition 4: Duplicated Body Inconsistency (*DBI*) is defined as:

$$\underset{h_k \in H; b_k, b'_k \in B}{\exists}\ \left((h_k, b_k) \in F \wedge (h_k, b'_k) \in F\right)$$

Complementary *Duplicated Header Inconsistency* will never occur in a properly working system because of the condition that is checked in Z_1 operation. Thus, this type of inconsistency will not be considered.

During concurrent execution of operations, a kind of temporal inconsistency, called here Deleted Component Inconsistency (*DCI*), may also occur. It is not considered as the SD2DS inconsistency, but if it occurs it does not allow to finish $GET(k)$ functions properly. *DCI* occurs during the $GET(k)$ function, when there was $(h_k, b_k) \in F$ during execution of O_1 operation and $(h_k, b_k) \notin F$ during the execution of O_2 operation.This problem is defined as follows:

Definition 5: Deleted Component Inconsistency (*DCI*) is defined as:

$$\underset{GET(k)}{\exists}\ \left(GET(k) = CH \rightarrow O_1 \rightarrow O_2\right) \wedge O_1 : (h_k, b_k) \in F \wedge O_2 : (h_k, b_k) \notin B$$

A. CONCURRENT EXECUTION OF OPERATIONS

Concurrency is considered as a vital feature for transactional execution of operations [27]. During sequential and completed executions of operations, it is not possible to cause any inconsistencies in the SD2DS. However, in real world implementation, it is not acceptable to execute all operations sequentially. The existence of many clients that operate simultaneously on SD2DS may cause that operations may overlap. These overlapping operations may cause *OHI*, *OBI*, *DBI* and *DCI*.

Inconsistencies may be transient or permanent. Transient inconsistencies arise temporarily during the execution of some operations and they disappear when the operation will be completed. Thus, such situations are critical only for operations executed concurrently with operation causing this type of inconsistency. Permanent inconsistencies cause that the SD2DS will be inconsistent after completing operations. Observations 1–4 summarize all critical situations caused by inconsistencies that occur during the simultaneous execution of any two SD2DS primary operations.

Observation 1: Assuming that all operations were executed completely during the concurrent execution of any two primary operations, the OHI is critical only in the following situations:

- OHI_1*: during concurrent execution of* $PUT(k, b_k) = CH' \rightarrow Z'_1 \rightarrow Z'_2 \rightarrow S'$ *and* $GET(k) = CH'' \rightarrow O''_1 \rightarrow O''_2$ *the transient OHI causes error in* O''_2*, if and only if* $Z'_1 \rightarrow O''_1$ *and* $O''_2 \rightarrow Z'_2$*, when* $c_k \notin F$ *before execution of* $PUT(k, b_k)$*;*
- OHI_2*: during concurrent execution of* $PUT(k, b_k) = CH' \rightarrow Z'_1 \rightarrow Z'_2 \rightarrow S'$ *and* $UPDATE(k, b'_k) = CH'' \rightarrow O''_1 \rightarrow U''_2 \rightarrow Z''_2$ *the transient OHI causes error in* U''_2*, if and only if* $Z'_1 \rightarrow O''_1$ *and* $U''_2 \rightarrow Z'_2$*, when* $c_k \notin F$ *before execution of* $PUT(k, b_k)$*;*
- OHI_3*: during concurrent execution of* $UPDATE(k, b'_k) = CH' \rightarrow O'_1 \rightarrow U'_2 \rightarrow Z'_2$ *and*

$GET(k) = CH'' \rightarrow O_1'' \rightarrow O_2''$ *the transient OHI causes error in* O_2''*, if and only if* $U_2' \rightarrow O_2'' \rightarrow Z_2'$*;*

- OHI_4*: during concurrent execution of UPDATE* $(k, b_k') = CH' \rightarrow O_1' \rightarrow U_2' \rightarrow Z_2'$ *and* $UPDATE(k, b_k') = CH'' \rightarrow O_1'' \rightarrow U_2'' \rightarrow Z_2''$ *the transient OHI causes error in* U_2' *(when* $U_2'' \rightarrow U_2'$*) or* U_2'' *(when* $U_2' \rightarrow U_2''$*), if and only if* $U_2' \rightarrow Z_2''$ *and* $U_2'' \rightarrow Z_2'$*.*

Observation 2: Assuming that all operations were executed completely, during the concurrent execution of two primary operations the OBI is critical only in the following situations:

- OBI_1*: during concurrent execution of* $PUT(k, b_k) = CH' \rightarrow Z_1' \rightarrow Z_2' \rightarrow S'$ *and* $DEL(k) = CH'' \rightarrow U_1'' \rightarrow U_2''$ *the permanent OBI is caused by* Z_2' *and the error appears in* U_2'' *if and only if* $Z_1' \rightarrow U_1''$ *and* $U_2'' \rightarrow Z_2'$*, when* $c_k \notin F$ *before execution of* $PUT(k, b_k)$*;*
- OBI_2*: during concurrent execution of* $DEL(k) = CH' \rightarrow U_1' \rightarrow U_2'$ *and* $UPDATE(k, b_k') = CH'' \rightarrow O_1'' \rightarrow U_2'' \rightarrow Z_2''$ *the permanent OBI is caused by* Z_2''*, if and only if* $O_1'' \rightarrow U_1'$ *and* $U_2' \rightarrow Z_2''$*, moreover error appears in* U_2' *(when* $U_2'' \rightarrow U_2'$*) or* U_2'' *(when* $U_2' \rightarrow U_2''$*).*

Observation 3: Assuming that all operations were executed completely, during the concurrent execution of the two primary operations the DBI is critical only in the following situations:

- DBI_1*: during concurrent execution of* $PUT(k, b_k) = CH' \rightarrow Z_1' \rightarrow Z_2' \rightarrow S'$ *and* $DEL(k) = CH'' \rightarrow U_1'' \rightarrow U_2''$ *the transient DBI occurs after* Z_2 *before* U_2*, if and only if* $U_1'' \rightarrow Z_1'$ *and* $Z_2' \rightarrow U_2''$*, when* $c_k \in F$*;*
- DBI_2*: during concurrent execution of* $PUT(k, b_k) = CH' \rightarrow Z_1' \rightarrow Z_2' \rightarrow S'$ *and* $UPDATE(k, b_k') = CH'' \rightarrow O_1'' \rightarrow U_2'' \rightarrow Z_2''$ *the permanent DBI is caused by operations* $Z_2\prime$ *and* $Z_2\prime\prime$*, if and only if* $Z_1' \rightarrow O_1''$ *and* $U_2'' \rightarrow Z_2'$*, moreover error appears in* U_2''*;*
- DBI_3*: during concurrent execution of* $UPDATE(k, b_k') = CH' \rightarrow O_1' \rightarrow U_2' \rightarrow Z_2'$ *and* $UPDATE(k, b_k') = CH'' \rightarrow O_1'' \rightarrow U_2'' \rightarrow Z_2''$ *the permanent DBI is caused by operations* $Z_2\prime$ *and* $Z_2\prime\prime$*, if and only if* $U_2' \rightarrow Z_2''$ *and* $U_2'' \rightarrow Z_2'$*, moreover error appears in* U_2' *(when* $U_2'' \rightarrow U_2'$*) or* U_2'' *(when* $U_2' \rightarrow U_2''$*).*

Observation 4: Assuming that all operations were executed completely during the concurrent execution of the two primary operations, the DCI is critical only in the following situation:

- DCI_1*: during concurrent execution of* $DEL(k) = CH' \rightarrow U_1' \rightarrow U_2'$ *and* $GET(k) = CH'' \rightarrow O_1'' \rightarrow O_2''$ *the DCI causes the error in* O_2''*, if and only if* $O_1'' \rightarrow U_1'$ *and* $U_2' \rightarrow O_2''$*.*

B. UNCOMPLETED OPERATIONS

In the previous section, we assumed that all primary operations were executed completely (the whole sequence of atomic operations were executed). In practice, we cannot fulfill this requirement because all atomic operations that are executed are initiated by client applications. Client processes may be unexpectedly terminated or even may work maliciously. Because of that, additional inconsistencies may occur due to incorrect working clients. All inconsistencies that may occur because of the uncompleted operations, are summarized in observation 5. The operator $\mapsto$ indicates the interruption of the execution of an operation in such a way that all atomic operations on the right of this operator are not executed.

Observation 5: In the case of uncompleted operations, the inconsistencies arise only in the following situations:

- END_1*: in the case of uncompleted* $PUT(k, b_k)$ *operation OHI arises if and only if* $CH \rightarrow Z_1 \mapsto Z_2 \rightarrow S$*;*
- END_2*: in the case of uncompleted* $DEL(k)$ *operation OBI arises if and only if* $CH \rightarrow U_1 \mapsto U_2$*;*
- END_3*: in the case of uncompleted* $UPDATE(k, b_k')$ *operation OHI arises if and only if* $CH \rightarrow O_1 \rightarrow U_2 \mapsto Z_2$*.*

C. PRESERVING THE STRONG CONSISTENCY IN SD2DS

Consistency in SD2DS may be provided using classical methods. In our previous work [68] we applied locking and versioning mechanisms. Both methods provide consistency only to some extent. Neither locking nor versioning deals with uncompleted operations. Moreover, these methods have severe drawbacks. The locking may lead to very serious problems like client starving and deadlocks. On the other hand, versioning consumes a lot of memory. The next versions have to be stored and in most cases they are not used at all. This might be a serious problem in cloud computing environments with pay–per–use model.

Due to the shortcomings of existing techniques, we decided to develop dedicated methods, especially suitable for SD2DS, that deal with the consistency problems caused by concurrent execution of operations as well as uncompleted operations. The main idea is based on proper scheduling of atomic operations. Based on our consistency analysis presented in previous sections, we developed scheduling methods that avoid the critical situations identified in Observations 1-4.

D. SCHEDULING OF ATOMIC OPERATIONS

To prevent from inconsistencies caused by concurrent execution of operations, we introduced a scheduling mechanism that is based on the sequence numbers (SN^k). The sequence number is generated separately for each component in the first layer of the structure according to the Algorithm 5. Because only one bucket is responsible for managing each component, the Algorithm 5 will return monotonic and rising values which can be used during scheduling operations in the second layer. The first operation that is executed on the component always is $PUT(k, b_k)$, so for this operation SN^k equals 0.

The second layer follows the Algorithm 6 to schedule all operations that have to be executed to process the component

Algorithm 5 Sequence Number Generation in the First Layer of SD2DS: $SEQ(k, \omega)$

Input: k - key, ω - primary operation
Output: *result* - sequence number

1 : *if* $(\omega = PUT(k, b_k))$ *then*
2 : $SN_H^k \leftarrow 0$
3 : $result \leftarrow SN_H^k$
4 : *else if* $(\omega = UPDATE(k, b'_k))$ *then*
5 : $SN_H^k \leftarrow SN_H^k + 2$
6 : $result \leftarrow SN_H^k - 1$
7 : *else*
8 : $SN_H^k \leftarrow SN_H^k + 1$
9 : $result \leftarrow SN_H^k$
10 : *end if*

Algorithm 6 Scheduling in the Second Layer of SD2DS

Input: k - key, SN - sequence number, op_2 - atomic operation ($op_2 \in Op_2$)
Output: *result* - ERROR or OK or WAIT

1 : *if* $(SN = 0 \wedge op_2 = Z_2)$ *then*
2 : *execute* op_2
3 : $SN_B^k \leftarrow 0$
4 : *GOTO NEXT*
5 : *else if* $(SN \neq 0 \wedge SN = SN_B^k + 1)$ *then*
6(EXEC) : *execute* op_2
7 : $SN_B^k \leftarrow SN_B^k + 1$
8(NEXT) : *if* $\left(\exists_{(x,o) \in Q^k} (x = SN_B^k + 1) \right)$ *then*
9 : $sn \leftarrow x$
10 : $op_2 \leftarrow o$
11 : $Q^k \leftarrow Q^k \backslash \{(x, o)\}$
12 : *GOTO EXEC*
13 : *else*
14 : $result \leftarrow OK$
15 : *end if*
16 : *else if* $(SN \neq 0 \wedge SN > SN_B^k + 1)$ *then*
17 : $Q^k \leftarrow Q^k \cup \{(SN, op_2)\}$
18 : $result \leftarrow WAIT$
19 : *else*
20 : $result \leftarrow ERROR$
21 : *end if*

body in sequence ordered by SN^k. The basic idea is to preserve the same order of execution of operations, both in the first and the second layer of the SD2DS. To achieve this, the SN^k should be stored in the first (SN_H^k)as well as the second layer (SN_B^k) of SD2DS. Algorithm 6 checks if the current operation can be executed. It is possible to execute the current atomic operation if all previous atomic operations that were initiated in the first layer were completed. The operation is stored in the queue Q^k if it should be postponed. Otherwise, the operation is completed and also the next operations (if any), with the following SN^k will be executed immediately after.

Theorem 1 proves that using Algorithms 5 and 6 it is possible to avoid all critical situations caused by inconsistencies *OHI*, *OBI*, *DBI* and *DCI*. At this point, we assume that all primary operations are executed completely, so all corresponding atomic operations are properly executed.

Theorem 1: Let $\omega', \omega'' \in \Omega$ *be primary operations that will be executed on* c_k, *and* $SEQ(k, \omega') = SN_H'^k$, $SEQ(k, \omega'') = SN_H''^k$. *All critical situations caused by OHI, OBI, DBI and DCI inconsistencies will be eliminated if the operations are executed completely and are scheduled using the Algorithm 6.*

Proof: According to Observation 1 *OHI* inconsistencies may cause four critical situations OHI_1–OHI_4:

- Assume that $\omega' = PUT(k, b_k)$ and $\omega'' = GET(k)$. If $Z_1' \rightarrow O_1''$ then $SN_H'^k < SN_H''^k$, therefore according to Algorithm 6 $Z_2' \rightarrow O_2''$, this means that OHI_1 will never occur.
- Assume that $\omega' = PUT(k, b_k)$ and $\omega'' = UPDATE(k, b'_k)$. If $Z_1' \rightarrow O_1''$ then $SN_H'^k < SN_H''^k$, therefore according to Algorithm 6 $Z_2' \rightarrow U_2''$, this means that OHI_2 will never occur.
- Assume that $\omega' = UPDATE(k, b'_k)$ and $\omega'' = GET(k)$. If $U_2' \rightarrow O_2''$ then $SN_H'^k + 2 \leq SN_H''^k$. After executing U_2' the SN_B^k will be equal to $SN_H'^k$. If the next operation issued for execution will be O_2'' then, according to Algorithm 6 this operation will be added to queue Q^k as (SN, O_2'') where $SN > SN_B^k + 1$. Hence, the operation Z_2' with assigned $SN = SN_H'^k + 1 = SN_B^k + 1$ will be executed first, i.e. $Z_2' \rightarrow O_2''$. This means that OHI_3 will never occur.
- Assume that $\omega' = UPDATE(k, b'_k)$ and $\omega' = UPDATE(k, b''_k)$ and that $\omega'$was issued for execution as the first. Thus, according to Algorithm 5: $SN_H'^k + 2 \leq SN_H''^k$. After issuing operations U_2' and U_2'', operation U_2' will be scheduled for execution as the first, and $SN_B^k = SN_H'^k$. Assume that the operation U_2'' will be issued before Z_2'. Then, it will be added to the Q^k as (SN, U_2'') where $SN > SN_B^k + 1$. Since Z_2' will be scheduled as $(SN_H'^k + 1, Z_2')$ regardless of the order of the request, thus the only possible order is $U_2' \rightarrow Z_2' \rightarrow U_2'' \rightarrow Z_2''$. Similarly, when ω''will be issued for execution as the first, the only possible order is $U_2'' \rightarrow Z_2'' \rightarrow U_2' \rightarrow Z_2'$. This means that OHI_4 will never occur.

Therefore, the above analysis proves that the Algorithm 6 eliminates all critical situations that may be caused by *OHI* inconsistencies.

In a similar way, we may prove that the Algorithm 6 eliminates all critical situations identified in Observations 2-4, i.e. that in all cases the only possible orders of execution will be the following:

- OBI_1, if $Z_1' \rightarrow U_1''$ then $Z_2' \rightarrow U_2''$;
- OBI_2, if $O_1'' \rightarrow U_2'$ then $Z_2'' \rightarrow U_2'$.
- DBI_1, if $U_1'' \rightarrow Z_1'$ then $U_2'' \rightarrow Z_2'$;
- DBI_2, if $Z_1' \rightarrow O_1''$ then $Z_2' \rightarrow U_2''$;
- DBI_3, if $U_2' \rightarrow Z_2''$ then $Z_2' \rightarrow U_2''$.
- DCI_1, if $O_1'' \rightarrow U_1'$ then $O_2'' \rightarrow U_2'$;

Thus, the Algorithm 6 eliminates all inconsistencies in SD2DS caused by concurrent executions of primary operations. □

Algorithm 6 schedules all atomic operations, even when there is no possibility to violate SD2DS consistency. Usually in NoSQL data storages, the operations of component retrieving are the most frequently used. As far as the $GET(k)$ functions do not introduce any inconsistencies they do not have to be scheduled in such a way. In those cases the performance of the whole scheduling mechanism may be significantly increased. Thus, we modified the scheduling method to eliminate unnecessary ordering of operations. The Algorithm 7 performs scheduling only operations that modify components in SD2DS. To achieve this we introduced a sequence number of the last modification of the specified component (SNM^k). It indicates the last operation that performs a component modification (adding, modifying and deleting). If all operations that modify the component are completed, the getting operations may be executed in any order. To perform this the SNM^k should be stored in the first layer (SNM^k_H) as well as in the second layer (SNM^k_B). $GET(k)$ operations that should be postponed are rejected to eliminate any locks.

E. RESTORING UNFINISHED PRIMARY OPERATIONS

Unfinished operations may violate the consistency of SD2DS, but they may also cause blocking of the execution of Algorithms 6 and 7. To prevent those problems, additional mechanisms for detecting and restoring unfinished operations were introduced. To detect unfinished operations the SN^k numbers are used. After predefined time the first layer of the SD2DS asks the second layer if the operation corresponding to SN^k number has been properly executed. If not, the first layer is responsible for restoring the unfinished operation. In the case of the $GET(k)$ function, if operation O_2 has not been executed, no additional actions are required. All other operations have to be restored whenever any critical situation identified in Observation 5 occurs. In the case END_1, if the operation Z_2 has not been executed, the U_1 operation has to be executed to cancel $PUT(k, b_k)$ and to revert to a previous state. In the case END_2 the U_2 operation has to be executed again, to finish $DEL(k)$. It should be noted that the case END_3 is indivertible. But it will be possible to make $UPDATE(k, b'_k)$ reversible, if the sequence of atomic operation is changed into $UPDATE'(k, b'_k)$ as follows:

$$CH \rightarrow O_1 \rightarrow Z_2 \rightarrow U_2 \tag{15}$$

Because of modifications of $UPDATE(k, b'_k)$ introduced by equation (15) the END_3 situation will never occur, but the new situation may cause inconsistency, according to the following observation:

Observation 6: In the case of uncompleted $UPDATE'(k, b'_k)$ the inconsistency arises only in the following situation:

- END'_3: DBI arises if and only if $CH \rightarrow O_1 \rightarrow Z_2 \nmapsto U_2$.

Algorithm 7 Partial Scheduling in the Second Layer of SD2DS

Input: k - key, SN - sequence number, SNM - sequence number of the last modification, op_2 - atomic operation ($op_2 \in OP_2$)

Output: *result* - ERROR or OK or WAIT

```
1          : if (SN = 0 ∧ op2 = Z2) then
2          :   execute op2
3          :   SN_B^k ← 0
4          :   SNM_B^k ← 0
5          :   GOTO NEXT
6          : else
7(EXEC)    : if (SN ≠ 0 ∧ SNM_B^k = SNM ∧
8          :     op2 = O2) then
9          :   execute op2
10         :   SN_B^k ← SN_B^k + 1
11         :   GOTO NEXT
12         : else if (SN ≠ 0 ∧ SN_B^k + 1 = SN ∧
13         :     (op2 = Z2 ∨ op2 = U2)) then
14         :   execute op2
15         :   SN_B^k ← SN_B^k + 1
16         :   SNM_B^k = SN_B^k
17         :   GOTO NEXT
18         : else if (SN ≠ 0 ∧ SN > SN_B^k + 1) then
19         :   Q^k ← Q^k ∪ {(SN, op2)}
20         :   result ← WAIT
21         : else
22         :   result ← ERROR
23         :   STOP
24         : end if end if end if end if
25(NEXT):  if ( ∃ (x = SN_B^k + 1)) then
                (x,o)∈Q^k
26         :   SN ← x
27         :   op2 ← x
28         :   Q^k ← Q^k \ {(x, o)}
29         :   GOTO EXEC
30         : end if
31         : result ← OK
```

It may be noted that END'_3 is the only situation which has to be reverted in case of uncompleted $UPDATE(k, b'_k)$ operation. Thus operation U_2 has to be executed to finish $UPDATE(k, b'_k)$ and revert to a consistent state. Algorithms 6 and 7 will prevent from any inconsistencies in the case of concurrent function execution even when the implementation of $UPDATE(k, b'_k)$ will be replaced with $UPDATE'(k, b'_k)$.

Algorithm 8 provides the full procedure for detecting and restoring the unfinished operations. It has to be executed after each primary operation that was executed by the first layer of the SD2DS and before the next operation that refers to the same component.

Theorem 2 proves that Algorithm 8 eliminates all inconsistencies caused by unfinished operations.

It is also worth to notice that in all cases, the restoring of unfinished primary operation according to 8 consists of

Algorithm 8 Checking of Unfinished Functions in the Second Layer of SD2DS

Input: k - key, SN - sequence number, $\omega \in \Omega$ -
Output: *result* - FIXED or NOTHING

```
1        : if (∃_x (h_x ∈ W_H^i ∧ h_x.key = k)) then
2        : j ← x
3        : else
4        : STOP
5        : end if
6        : if (SN_B^k < SN ∧ ω = PUT(k)) then
7        : execute U_1 on W_H^{Hash(h_j.key)}
8        : GOTO FIX
9        : else if (SN_B^k < SN ∧ ω = DEL(k)) then
10       : execute U_2 on W_B^{Addr(h_j.locator)}
11       : GOTO FIX
12       : else if (SN_B^k = SN − 1∧
13       :   ω = UPDATE'(k, b'_k)) then
14       : execute U_2 on W_B^{Addr(h_j.locator)}
15       : GOTO FIX
16       : else if (SN_B^k < SN ∧ ω = GET(k)) then
17 (FIX) : SN_B^k = SN
18       : result ← FIXED
19       : else
20       : result ← NOTHING
21       : end if
```

just a single operation that is performed on the second layer bucket of SD2DS. That is why this operation will be always faster than a single primary operation performed by the client. It does not need to contain an address calculation (*CH* operation) as well as communication between the client and the first layer bucket. In the most optimistic variant, when no restoration is needed, this algorithm will only need to ensure communication between the first and the second layer buckets. This communication is performed by a very simple message that is far smaller than a typical message used in SD2DS that contains the component's body.

Theorem 2: Let $\omega' \in \Omega$ be primary operation that will be executed on c_k, and $SEQ(k, \omega') = SN'^k_H$. All critical situations caused by *END* inconsistencies will be eliminated if the operations are checked using the Algorithm 8.

Proof: According to the Observations 5 and 6 inconsistencies may cause three critical situations END_1, END_2 and END'_3:

- Assume that $\omega' = PUT(k, b_k)$. If $CH' \rightarrow Z'_1 \mapsto Z'_2 \rightarrow S'$ then $SN'^k_B < SN'^k_H$, therefore according to Algorithm 8 additional U_1 operation will be executed on $W_H^{Hash(k)}$, this means that END_1 will never occur.
- Assume that $\omega' = DEL(k)$. If $CH' \rightarrow U'_1 \mapsto U'_2$ then $SN'^k_B < SN'^k_H$, therefore according to Algorithm 8 additional U_2 operation will be executed on $W_B^{Addr(h_k.locator)}$, this means that END_2 will never occur.
- Assume that $\omega' = UPDATE'(k, b'_k)$. If $CH' \rightarrow O'_1 \rightarrow Z'_2 \mapsto U'_2$ then $SN'^k_B = SN'^k_H - 1$, therefore according to Algorithm 8 additional U_2 operation will be executed on $W_B^{Addr(h_k.locator)}$, this means that END'_3 will never occur.

□

VII. PERFORMANCE ANALYSIS

In this section, we have carried out a theoretical analysis of the performance of our SD2DS data store. By analysing Algorithms 1–4 we have concluded that the computational complexity of primary operations is influenced mostly by three factors: message passing time (T), processing time of the component on the first layer bucket (P_1) and processing time of component on the second layer bucket (P_2). In the case of a T we assumed that it is an average value. This transmission time is not dependent on the internal data store structure and is the same for data storages of any kind.

Firstly, we analysed the time complexity in terms of the overall number of components that are stored in SD2DS. By analysing Algorithms 1–4 we concluded that the time complexity of $PUT(k, b_k)$ operation can be described as: $3T + P_1 + P_2$ in the most optimistic case. In the case that the *Split* event was notified, the time complexity will be described as: $5T + P_1 + P_2$. In the case of $GET(k)$ and $DEL(k)$ operations, the time complexity is expressed as $3T + P_1 + P_2$. In the case of $UPDATE(k, b'_i)$, the time complexity is defined as $3T + P_1 + 2P_2$. Assuming that the size of buckets is constant, the time of P_1 and P_2 is independent from the overall number of components, so the time complexity is expressed $O(1)$ in this matter.

Next, we performed the analysis of the impact of the whole *Split* operation. It is worth to notice that this operation is not frequent in the typical SD2DS functioning. It only applies to the situation when the whole structure is growing. Assuming W is a size of the first layer bucket, the *Split* operation is performed after inserting $W/2$ components (in the case of the first *Split* it will be performed after W insertions). In that case, to insert an overall number of C components, the *Split* operations will be executed $\frac{2C}{W} - 1$ *Split* times. During each *Split* operation $W/2$ component bodies are transmitted, so the total number of transmissions will be $C - W/2$. Thus, the time required for split operations is linearly dependent on the size of the datastore and the computational complexity of each *Split* is $O(1)$.

Different versions of the presented SD2DS vary only in terms of the time of P_1. Because of that, the differences in their performance are basically independent on the C.

On the other hand, the performance complexity can be analysed in terms of the number of clients that simultaneously operate on the data store (N). Assuming that each bucket is located on a different server and the overall number of servers is equal S, the average load can be expressed as N/S. This expression applies when $N \gg S$, which indicated that each server is loaded with many requests. In another case, most servers handle requests with maximal performance.

Since the presented analysis shows that all complexities are at most linear, SD2DS can provide a decent level of horizontal scalability. It is also worth to notice that SD2DS can also provide vertical scalability by fine tuning the size of each buckets to the server resources (their memory capacities in particular). Alternatively, it is also possible to run multiple buckets on the same server. The fact that bigger buckets contain more components can influence the time of performance of the split operation (there will be more body buckets to transfer). More importantly, bigger buckets may cause the performance to drop faster as the overall number of clients and the processing time grow. Additionally, the processing time will become slower as more time will be needed to search the component. Those two drawbacks are common in other architectures. That is why in most NoSQL and NewSQL horizontal scaling over vertical scaling is preferred. That is why in our work we focused mostly on this issue.

VIII. EVALUATION

The developed architecture was implemented as a prototype datastore. Both scheduling methods were implemented as $SD2DS_{BS}$ (Algorithm 6) and $SD2DS_{PS}$ (Algorithm 7). For evaluation purposes SD2DS datastores based on component locking ($SD2DS_{CL}$) and component versioning ($SD2DS_{CV}$) were also developed. We ran these implementations on a multicomputer cluster that consisted of 16 nodes. Each machine served one first layer and one second layer bucket. Additional 10 machines were used to run the client processes. These nodes were connected through 1GiB/s Ethernet. While faster network solutions are typically available in modern clusters, we decided to choose a more pessimistic, slower option in our experiments and used Ethernet connection in all experiments. Current applications of SD2DS are not limited to the environment that runs on a single cluster. Both SD2DS layers can be divided into two separate infrastructures. Additionally, in our previous works we also utilized SD2DS in peer-to-peer networks [59], [60]. To cover all of these applications, we decided to perform all experiments on commonly available 1GiB/s Ethernet.

We took special precautions to avoid any distortions that could influence our experimental results. All experiments were repeated at least once to make sure that they were not dependent on any internal or external factors. The cluster that was used to carry out the experiments was isolated for the time of evaluation in such a way that no other services were running during the experiments. Because of that, we minimized not only the additional resources load on nodes but also other network transfers.

In the experimental section, we focused mostly on the comparison and evaluation of our main contribution, which is our scheduling mechanisms. Those mechanisms are used on each operation on the datastore and affect all its functionality. On the other hand, the mechanism for restoring unfinished primary operations in non-malicious environment is not frequent. This mechanism may affect the data store only due to some problems on the client side. Additionally, due to its nature, restoring unfinished primary operation does not need the calculation of component address and operation performed on the first layer, so it affects data store less than typical client's operations.

The first experiment concerned the comparison of the efficiency of the proposed scheduling methods with traditional approaches. To eliminate the split and merge operations, experiments were limited to UPDATE and GET operations. Figures 2 and 3 present an average access time to components of two sizes: 1 MiB and 10 MiB. The comparison was performed using 50 client applications simultaneously accessing the datastore. Each client was performing either $GET(k)$ or $UPDATE'(k, b'_k)$. The number of clients performing $GET(k)$ and $UPDATE'(k, b'_k)$ operations are designated as C_r and C_u, respectively ($C_u = 50 - C_r$).

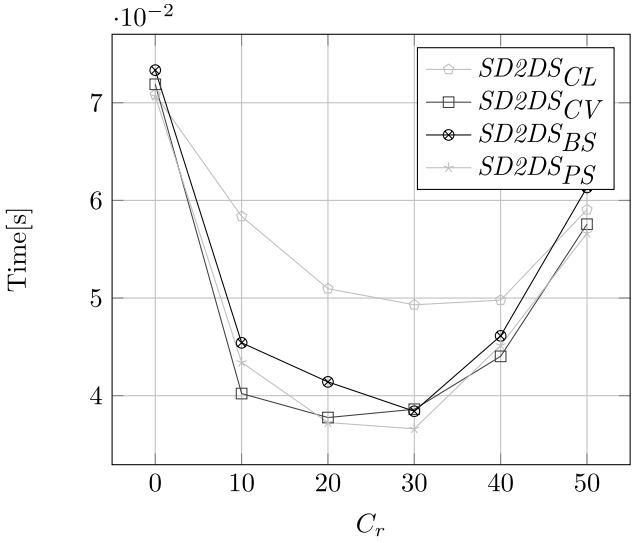

FIGURE 2. Access time for components of 1MiB.

In all cases, the architecture $SD2DS_{PS}$ achieved better performance than $SD2DS_{BS}$. This proves that partial scheduling gives significant improvement to the basic scheduling. In the most cases the $SD2DS_{CL}$ architecture is least efficient. This is caused by the fact that operations lock buckets and in case of heavy load they can wait really long time to execute. So, the locking mechanism significantly decreases the performance of datastores. The performances of $SD2DS_{PS}$ and $SD2DS_{CV}$ are comparable, but in the most cases the best performance was obtained for $SD2DS_{CV}$. The slight vantage of $SD2DS_{CV}$ is visible mainly for large components. In the most cases the best performance was obtained when values of C_r and C_u were similar. Because the $UPDATE(k, b'_k)$ operations perform longer than $GET(k)$ in the case of the high value of C_u, the most clients perform updates which slow down $GET(k)$ as well. In the case of the high value of C_r faster $GET(k)$ functions may cause an overload of the single bucket.

The next experiments concerned the scalability of SD2DS architectures. The Figure 4 presents the results of the

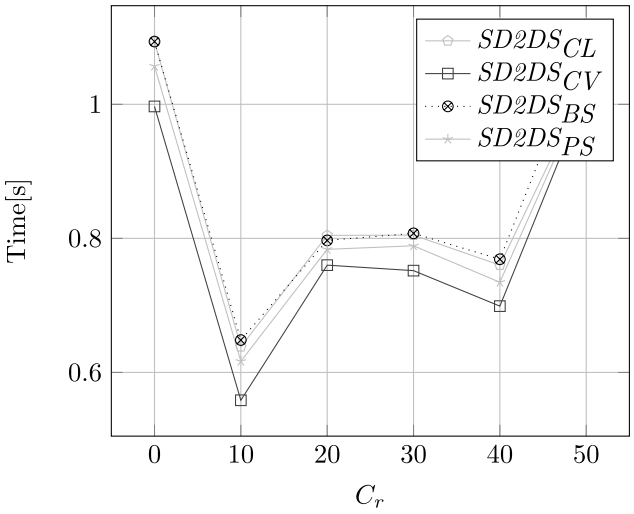

FIGURE 3. Access time for components of 10MiB.

scalability evaluation of the previously compared architectures. In this experiment, the access time was measured for datastores consisting of different number of components and different number of buckets. Each bucket that was used in these experiments contained 512 components of size equal 10MiB. As it is presented in Figure 4, each proposed architecture provides very good data scalability. Attaching more and more buckets with components does not have a negative influence on the performance of SD2DS. This trend is visible in all tested architectures, regardless of the consistency mechanism applied. Those results are also confirmed by the performance analysis presented in the previous section, since it is clearly visible that the total number of components does not influence the access time.

Finally, the most important experiment showing the advantages of the proposed architecture is the comparison with existing solutions in terms of the performance. Thus, the last experiment was performed to compare the performance of the $SD2DS_{BS}/SD2DS_{PS}$ with other well-known NoSQL systems for storing big sets of data. The comparison was made with the MongoDB and MemCached systems. Both databases were developed in C++, just like SD2DS, and allow to store data in a distributed environment. Additionally, MongoDB also provides a mechanism to scale the data store in a very similar way to SD2DS. The results showing the access time for datastores with components of 1MiB are presented in the Figure 5 while the results of accessing components of 10MiB are presented in Figure 6. To use MongoDB as a distributed datastore, a special query router is used (so called *mongos*) as an interface between the application and the sharded cluster. In our tests we used a different number of *mongos* (1, 2 and 5) to evaluate their influence to the overall performance. The *mongos* can become bottlenecks so the performance of the MongoDB is far worse than SD2DS even with the 5 *mongos* instances. In a typical application, MongoDB stores data in the form of structured documents that need to be smaller than 16MiB, while SD2DS and MemCached store raw blocked data. To allow reliable comparison and overcome this difference between those structures, a special API called GridFS was used. It allowed to store raw files inside MongoDB data store. The MemCached system is known for its very good performance. However, the MemCached is optimized for storing components that are smaller than 1MiB. While it enables to store components of arbitrary sizes, MemCached rejects some requests if they cannot be executed within a specified time. Experimental results showed that the proposed architecture outperforms both existing solutions.

Figure 6 shows an interesting trend that presents that all SD2DS variants become slightly slower after the number of clients exceeds the number of used buckets. After crossing this point, the presented dependence becomes almost linear, what confirms the theoretical analysis of the performance.

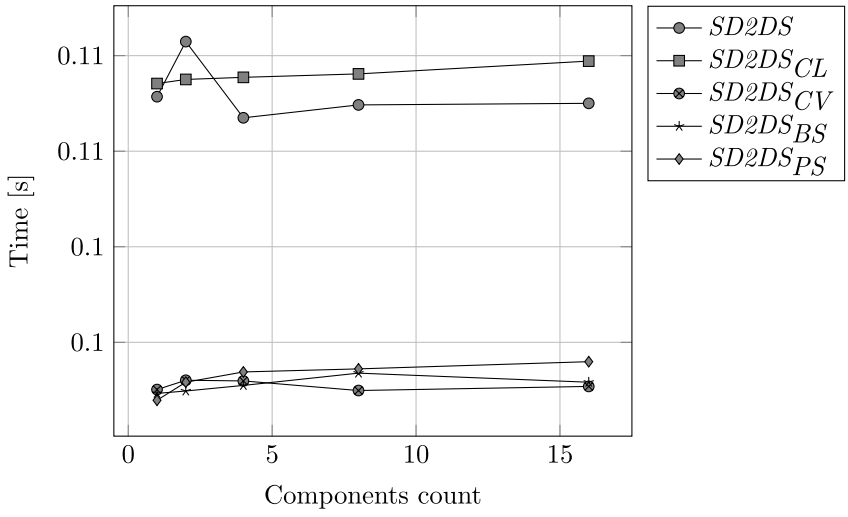

FIGURE 4. Scalability comparison while getting components of 10MiB.

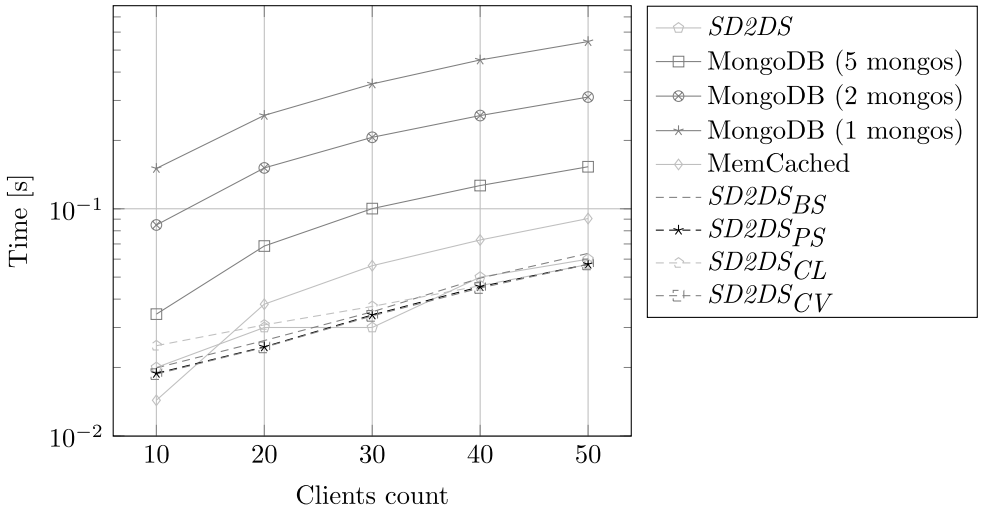

FIGURE 5. Efficiency comparison while getting components of 1MiB.

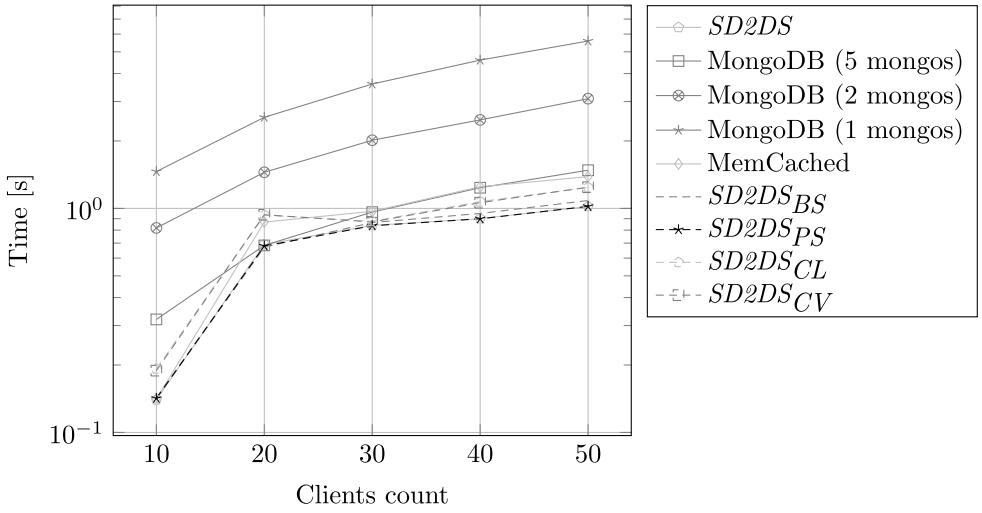

FIGURE 6. Efficiency comparison while getting components of 10MiB.

In the case of the MongoDB, the slower processing time was mainly caused by the overload of the mongos routers. One can expect that in the case of a MongoDB, the time complexity is estimated as N/m, where m is the number of mongos and $m \ll S$. In the case of a MemCached, we can expect that the time complexity is also estimated as N/S. However, it utilizes the locking mechanism to ensure consistent access [69], what has a visible impact on its performance.

Figure 7 presents the efficiency comparison in terms of different component sizes (1MiB, 2MiB, 5MiB, 10MiB). In the case of all data stores (both SD2DS versions as well as compared MongoBD and MemCached) there are linear dependencies between the processing time and the size of the component. Additionally, the processing time of all SD2DS versions, even those that provide consistency, grows slower with the increasing component size in comparison to MongoDB and MemCached.

All above-mentioned experiments allow to formulate the benefits and limitations of our proposed solutions. Analysis of the Figures 5 and 6 shows that the best performance of SD2DS was achieved for components of relatively large size. The performance for the small components is worse. It is caused by the influence of the indirect component access caused by the separation of layers. Additionally, SD2DS in its current form allows to store only raw blocked data identified by a key. This organization makes it difficult to perform advanced data processing and aggregation. On the other hand, our proposed architectures have a number of advantages. Full horizontal scalability is one of the most important issues that should be highlighted here. Additionally, our architectures

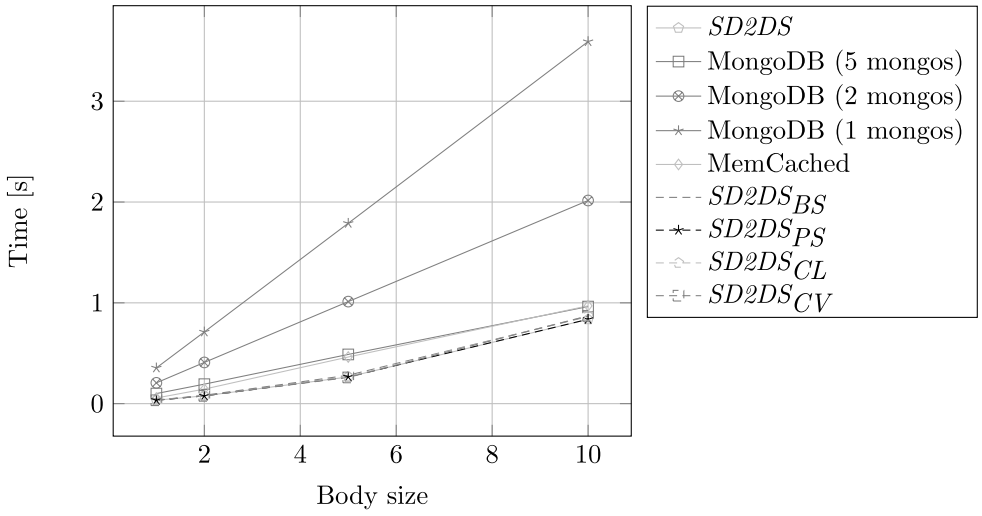

FIGURE 7. Efficiency comparison in relation to component body size.

provide high efficiency even in an environment that is heavily loaded with clients.

IX. CONCLUSION

In this paper, we proposed a novel architecture for NoSQL datastores, called SD2DS. We defined all inconsistencies that can occur during concurrent access to the datastore and caused by unfinished operations. The following inconsistencies were identified and explained: Orphan Header Inconsistency (*OHI*), Orphan Body Inconsistency (*OBI*), Duplicated Body Inconsistency (*DBI*) and Deleted Component Inconsistency (*DCI*). We also grouped inconsistencies into two types: durable and transient. The durable inconsistencies are connected with the internal state of the SD2DS while the transient ones may appear only during concurrent execution of operations.

To cope with the inconsistency in SD2DS we decided to introduce the novel scheduling mechanisms. By using Algorithms 5, 6 and 7 it is possible to prevent all inconsistencies caused by the concurrent function execution. Additionally, a special algorithm (Algorithm 8) was developed to detect unfinished functions and to restore the SD2DS into a consistent state. It was shown that the proposed scheduling method is more efficient than component locking and overcomes the limits of component versioning (by eliminating both the need of applying locks and storing an additional component body version) while still preserving comparable performance.

Our prototype implementation of the proposed architecture gave us very good experimental results. First of all, the experimental results proved that preserving consistency does not affect the scalability of the datastore. Moreover, our prototype implementation can still seriously concur with the popular NoSQL datastores (like MongoDB and MemCached) as far as the performance is concerned. Efficiency comparison results that were obtained show that all the proposed consistency methods outperform popular and well-known NoSQL systems. Additionally, the experimental results confirmed that the proposed mechanisms allow very good data scalability. They also outperform traditional methods like component locking and versioning.

To sum up, our work resulted in the development of a complete NoSQL datastore solution with the following advantages:

- elimination of all inconsistencies;
- preserving scalability;
- outperforming traditional methods (locking and versioning);
- outperforming other NoSQL systems (MongoDB and MemCached).

In our future work, we plan to apply SD2DS architecture to develop an efficient scalable distributed datastore. The main goal of our research is to develop a highly optimized datastore dedicated to high performance big data analytics.

REFERENCES

[1] J. Gray and A. Reuter, *Transaction Processing*. Burlington, MA, USA: Morgan Kaufmann, 1993.

[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[3] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall Press, 2008.

[4] J. N. Gray, *Notes Data Base Operating System*. Berlin, Germany: Springer, 1978.

[5] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[7] J. Gantz and D. Reinsel, *The Digital Universe Decade—Are You Ready*. Needham, MA, USA: IDC, 2010. Accessed: Jan. 26, 2014. [Online]. Available: http://www.emc.com/collateral/analyst-reports/idc-digital-universe-are-you-ready.pdf

[8] E. A. Brewer, "Towards robust distributed systems," in *Proc. PODC*, 2000, p. 7.
[9] H. Yu and A. Vahdat, "Building replicated Internet services using TACT: A toolkit for tunable availability and consistency tradeoffs," in *Proc. 2nd Int. Workshop Adv. E-Commerce Web-Based Inf. Syst.*, Oct. 200, pp. 75–84.
[10] Cassandra. (2017). *Apache Cassandra (TM) 2.0, the Documentation*. Accessed: Dec. 5t, 2019. [Online]. Available: http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_about_transactions_c.html
[11] Memcached. *Memcached—A Distributed Memory Object Caching System*. Accessed: Dec. 5, 2019. [Online]. Available: http://memcached.org
[12] E. Plugge, T. Hawkins, and P. Membrey, *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, 1st ed. Berkely, CA, USA: Apress, 2010.
[13] J. Webber, "A programmatic introduction to neo4j," in *Proc. 3rd Annu. Conf. Syst., Program., Appl., Softw. Humanity*, 2012, pp. 217–218.
[14] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009, doi: 10.1145/1435417.1435432.
[15] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *Proc. ACM Symp. Oper. Syst. Princ.*, 1997, vol. 31.
[16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proc. 6th Annu. ACM Symp. Princ. Distrib. Comput.*, 1987, pp. 1–12.
[17] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 761–772.
[18] H. S. Aldin, H. Deldari, M. H. Moattar, and M. R. Ghods, "Strict timed causal consistency as a hybrid consistency model in the cloud environment," *Future Gener. Comput. Syst.*, vol. 105, pp. 259–274, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X19321053
[19] M. Diogo, B. Cabral, and J. Bernardino, "Consistency models of NoSQL databases," *Future Internet*, vol. 11, no. 2, p. 43, Feb. 2019, doi: 10.3390/fi11020043.
[20] D. Borthakur, "Apache Hadoop Goes realtime at facebook," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2011, pp. 1071–1080.
[21] Z. Wei, G. Pierre, and C.-H. Chi, "CloudTPS: Scalable transactions for Web applications in the cloud," *IEEE Trans. Services Comput.*, vol. 5, no. 4, pp. 525–539, Jun. 2012.
[22] K.-Y. Whang, I. Na, T.-S. Yun, J.-A. Park, K.-H. Cho, S.-J. Kim, I. Yi, and B. S. Lee, "Building social networking services systems using the relational shared-nothing parallel DBMS," *Data Knowl. Eng.*, vol. 125, Jan. 2020, Art. no. 101756.
[23] S. Gilbert and N. Lynch, "Perspectives on the CAP theorem," *Computer*, vol. 45, no. 2, pp. 30–36, Feb. 2012.
[24] M. Stonebraker, "In search of database consistency," *Commun. ACM*, vol. 53, no. 10, pp. 8–9, Oct. 2010.
[25] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, May 2013.
[26] M. Stonebraker. *New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps*. Accessed: Dec. 5, 2016. [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext
[27] A. Pavlo and M. Aslett, "What's really new with NewSQL?" *ACM SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, Sep. 2016.
[28] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 1–12.
[29] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. CIDR*, vol. 11, 2011, pp. 223–234.
[30] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, and C. Frost, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, p. 8, 2013.
[31] P. A. Bernstein and N. Goodman, "Multiversion concurrency control-theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, 1983.
[32] L. Wu, L.-Y. Yuan, and J.-H. You, "BASIC: An alternative to BASE for large-scale data management system," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Feb. 2012, pp. 5–14.
[33] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, Jun. 2013.
[34] L.-Y. Yuan, L. Wu, J.-H. You, and Y. Chi, "A demonstration of rubato DB: A highly scalable newSQL database system for OLTP and big data applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 907–912.
[35] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, p. 19, 2016.
[36] D. Skeen, "Nonblocking commit protocols," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1981, pp. 133–142.
[37] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.
[38] L. Lamport and M. Massa, "Cheap paxos," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2004, pp. 307–314.
[39] L. Lamport, "Fast paxos," *Distrib. Comput.*, vol. 19, no. 2, pp. 79–103, Oct. 2006.
[40] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.
[41] T. Pankowski, "Lorq: A system for replicated NoSQL data based on consensus quorum," in *Proc. Int. Conf. Eval. Novel Approaches Softw. Eng.* Cham, Switzerland: Springer, 2015, pp. 62–79.
[42] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, vol. 8, 2010, p. 9.
[43] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. 7th Symp. Oper. Syst. Design Implement.*, 2016, pp. 335–350.
[44] M. Patiño-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "MIDDLE-R: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 375–423, Nov. 2005.
[45] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proc. VLDB*, 2000, pp. 134–143.
[46] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in RocksDB," in *Proc. CIDR*, vol. 3, 2017, p. 3.
[47] M. A. Georgiou, A. Paphitis, M. Sirivianos, and H. Herodotou, "Towards auto-scaling existing transactional databases with strong consistency," in *Proc. IEEE 35th Int. Conf. Data Eng. Workshops (ICDEW)*, Apr. 2019, pp. 107–112.
[48] M. Ronstrom and L. Thalmann, "Mysql cluster architecture overview," MySQL, Tech. Rep., vol. 8, 2004.
[49] C. Plattner and G. Alonso, "Ganymed: Scalable replication for transactional Web applications," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.* Berlin, Germany: Springer, 2004, pp. 155–174.
[50] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH—A scalable, distributed data structure," *ACM Trans. Database Syst.*, vol. 21, no. 4, pp. 480–525, 1996.
[51] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH: Linear hashing for distributed files," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 327–336, Jun. 1993.
[52] W. Litwin, M.-A. Neimat, and D. Schneider, "RP: A family of order preserving scalable distributed data structures," in *Proc. VLDB*, vol. 94, 1994, pp. 12–15.
[53] R. Devine, "Design and implementation of DDH: A distributed dynamic hashing algorithm," in *Foundations of Data Organization and Algorithms*. Berlin, Germany: Springer, 1993, pp. 101–114.
[54] K. Sapiecha and G. Lukawski, "Scalable distributed two-layer data structures (SD2DS)," *Int. J. Distrib. Syst. Technol.*, vol. 4, no. 2, pp. 15–30, Apr. 2013.
[55] A. Krechowicz, S. Deniziak, M. Bedla, A. Chrobot, and G. Łukawski, "Scalable distributed two-layer block based datastore," in *Proc. Int. Conf. Parallel Process. Appl. Math.* Cham, Switzerland: Springer, 2015, pp. 302–311.
[56] A. Krechowicz, A. Chrobot, S. Deniziak, and G. Łukawski, "SD2DS-based datastore for large files," in *Proc. Federated Conf. Softw. Develop. Object Technol.* Cham, Switzerland: Springer, 2015, pp. 150–168.
[57] K. Sapiecha, G. Łukawski, and A. Krechowicz, "Data and throughput scalable service-oriented datastore," in *Proc. 21st Euromicro Int. Conf. Parallel, Distrib., Network-Based Process.*, 2013, pp. 1–9.
[58] K. Sapiecha, G. Lukawski, and A. Krechowicz, "Enhancing throughput of scalable distributed two–layer data structures," in *Proc. IEEE 13th Int. Symp. Parallel Distrib. Comput.*, Jun. 2014, pp. 103–110.

[59] A. Krechowicz, "Scalable distributed two-layer datastore providing data anonymity," in *Proc. Int. Conf., Beyond Databases, Archit. Struct.* Cham, Switzerland: Springer, 2015, pp. 262–271.
[60] A. Krechowicz and S. Deniziak, "SD2DS-based anonymous datastore for IoT solutions," in *Proc. DEStech Trans. Comput. Sci. Eng. (WCNE)*, Oct. 2016, pp. 316–320.
[61] S. Deniziak, T. Michno, and A. Krechowicz, "The scalable distributed two-layer content based image retrieval data store," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Oct. 2015, pp. 827–832.
[62] S. Deniziak, T. Michno, and A. Krechowicz, "Content based image retrieval using modified scalable distributed two-layer data structure," *Int. J. Comput. Sci. Appl.*, vol. 13, no. 2, pp. 106–120, 2016.
[63] T. Michno and A. Krechowicz, "SD2DS database in the direction of image retrieval," *Appl. Inf. Technol.-Theory Pract.*, vol. 11, pp. 259–270, Oct. 2015.
[64] S. Deniziak, G. Łukawski, M. Bedla, and A. Krechowicz, "A scalable distributed 2-layered data store (SD2DS) for Internet of Things (IoT) systems," *Meas. Autom. Monitor.*, vol. 61, no. 7, pp. 382–384, 2015.
[65] A. Krechowicz and S. Deniziak, "Business intelligence platform for big data based on scalable distributed two-layer data store," in *Proc. Communiation Papers Federated Conf. Comput. Sci. Inf. Syst.*, Sep. 2017, pp. 177–182.
[66] A. Krechowicz and S. Deniziak, "Hierarchical clustering in scalable distributed two-layer datastore for big data as a service," in *Proc. 6th Int. Conf. Enterprise Syst. (ES)*, Oct. 2018, pp. 138–145.
[67] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[68] A. Krechowicz, S. Deniziak, G. Łukawski, and M. Bedla, "Preserving data consistency in scalable distributed two layer data structures," in *Beyond Databases, Architectures Structure*. Cham, Switzerland: Springer, 2015, pp. 126–135.
[69] N. Gunther, S. Subramanyam, and S. Parvu, "Hidden scalability gotchas in memcached and friends," in *Proc. Velocity Perform. Oper. Conf.*, 2010, pp. 1–6.

ADAM KRECHOWICZ was born in Kielce, Poland, in 1987. He received the M.S. degree in engineering in computer science from the Kielce University of Technology, Poland, in 2011, and the Ph.D. degree in computer science from the Warsaw University of Technology, in 2018. From 2011 to 2018, he was a Research Assistant with the Kielce University of Technology. In 2014, he was a Visiting Researcher with the Delft University of Technology, The Netherlands. Since 2018, he has been an Assistant Professor with the Department of Information Systems, Kielce University of Technology. He took part in several research grants connected with information systems which were implemented in cooperation with the industry. He has authored or coauthored 13 research articles connected with distributed and parallel systems. His research interests include distributed systems, big data systems, natural language processing, and bioinformatics and software engineering. He was a member of the Programme Committee and a reviewer of several international conferences on computer science.

STANISŁAW DENIZIAK (Member, IEEE) graduated from the Faculty of Electronics and Information Technologies, Warsaw University of Technology, in 1988. He received the Ph.D. degree from the Gdansk University of Technology, in 1994, and the D.Sc. degree from the Warsaw University of Technology, in 2006. From 2012 to 2019, he was the Vice Dean for Research and Promotion of the Faculty of Electrical Engineering, Automatics, and Computer Science, Kielce University of Technology, where he has been the Dean of the of Faculty of Electrical Engineering, Automatics and Computer Science, since 2020. He is currently a Professor of computer science with the Department of Information Systems, Kielce University of Technology, Poland. He is also the Head of the Division of Computer Science, Kielce University of Technology. He has published more than 120 research articles in various journals and books and more than 120 research papers in various conferences. His research interests include design of embedded systems, the Internet of Things, logic synthesis for FPGAs, and big data. He is a member of the IEEE Computer Society. He is an Active Reviewer and an Editorial Member of many international journals, like *Microprocessors and Microsystems*, *Sensors*, *Remote Sensing*, IEEE Access, *Applied Sciences*, *Multimedia Tools and Applications*, *Journal of Systems and Software*, *Computing*, and others. He is also a member of program/scientific committees and a Reviewer of many scientific conferences, like IEEE DDECS, RUC, SETIT, IEEE DAC, EPMCCS, and others.

GRZEGORZ ŁUKAWSKI received the Ph.D. degree in computer science from the Gdansk University of Technology. He is currently an Assistant Professor with the Department of Information Systems, Kielce University of Technology. His research interests include distributed systems, load balancing, and fault tolerance.