pH1: A Transactional Middleware for NoSQL

Fábio Coelho Francisco Cruz Ricardo Vilaça José Pereira Rui Oliveira
HASLab - High-Assurance Software Laboratory
INESC TEC and Universidade do Minho
Braga, Portugal
fabio.a.coelho@inesctec.pt
{*fmcruz,rmvilaca,jop,rco*}*@di.uminho.pt*

***Abstract*—NoSQL databases opt not to offer important abstractions traditionally found in relational databases in order to achieve high levels of scalability and availability: transactional guarantees and strong data consistency.**

In this work we propose pH1, a generic middleware layer over NoSQL databases that offers transactional guarantees with Snapshot Isolation. This is achieved in a non-intrusive manner, requiring no modifications to servers and no native support for multiple versions. Instead, the transactional context is achieved by means of a multiversion distributed cache and an external transaction certifier, exposed by extending the client's interface with transaction bracketing primitives.

We validate and evaluate pH1 with Apache Cassandra and Hyperdex. First, using the YCSB benchmark, we show that the cost of providing ACID guarantees to these NoSQL databases amounts to 11% decrease in throughput.

Moreover, using the transaction intensive TPC-C workload, pH1 presented an impact of 22% decrease in throughput. This contrasts with OMID, a previous proposal that takes advantage of HBase's support for multiple versions, with a throughput penalty of 76% in the same conditions

***Keywords*—NoSQL; Transactions; Snapshot Isolation;**

I. Introduction

Providing continual availability and support for millions of users – some of the core properties of the cloud computing paradigm – has been a major business opportunity for cloud service providers. The high availability goal along with the competitiveness of storage systems drew an increasing number of businesses to migrate their systems into the *cloud* and, thus reducing ownership and maintenance costs.

Up until recently, information systems relied almost exclusively in relational database management systems (RDBMS). These are well matured systems, which provide a standard SQL access interface with a programming abstraction that provides transactional semantics, usually with ACID guarantees. However, relational systems proved so far unable to sustain the non-functional characteristics of the cloud computing paradigm: high availability and scalability. This is mainly due to their rigid, monolithic and centralized architectures.

To cope with this demand, a new class of database systems emerged. These are non-relational, typically just based on a key/value data model, and usually, they do not provide a complex structured query language (just simple *put* and *get* primitives). These systems are often referred to as NoSQL databases. Regardless of its specific data model or API, a NoSQL database is expected to be highly available and scalable. Key to these two characteristics are their inherent distributed design, weak consistent data replication and transactionless operation.

However, the downside of using NoSQL databases is that much of the complexity now needs to be handled at the application level. Specifically, most of the highly efficient processing provided by SQL query engines inside RDBMS needs now to be done by the application, and the lack of a transactional programming model requires that hard problems such as concurrency control and failure recovery are now explicitly handled by the programmer. On the one hand this makes application programming much more demanding and error prone and, on the other, severely impairs the migration of legacy applications to the cloud.

In this paper we present the design and implementation of an elegant, non intrusive middleware system that is able to endow a typical NoSQL database with a transactional interface offering ACID guarantees. The proposed transactional middleware, *pH1*, preserves the interface of the underlying NoSQL database allowing operations to be bracketed in a transactional context offering Snapshot Isolation [1] as its isolation criterion. It relies on the optimistic execution of concurrent transactions that are certified at commit time. If by committing two concurrent transactions the isolation criterion would be violated then one of them is simply aborted. Each transaction runs on a private virtual snapshot of the database without any interferences of concurrent transactions. To achieve this, pH1 implements a multiversion distributed cache of the database, closely synchronized with the persistent NoSQL database. This is called the Non Persistent Version Store (NPVS) and along with the transactional context is the cornerstone of our approach.

The remainder of this paper is organized as follows: Section II and III offer an overview of some key concepts. Section IV describes the design and architecture of the main components of pH1 and Section V briefly describes its implementation. Section VI presents the assessment of the system. Section VII goes through related work and Section VIII concludes this work.

II. Transactional Systems

Transactional systems introduce the concept of *transaction*. A transaction is a sequence of operations whose execution traditionally satisfies the following ACID properties:

Atomicity: Either all or none of the operations within a transaction are successfully performed;

Consistency: Transactions preserve system constraints, taking the system from a valid state to another valid state;

Isolation: The execution of concurrent transactions preserves the semantics of the defined consistency criterion or isolation level;

Durability: The effects of successful transactions are durable even in the presence of faults.

A transaction that is able to successfully complete is said to *commit*, otherwise is said to *abort*.

A. Snapshot Isolation

Transactional systems allow the co-existence of concurrent transactions, which may lead to several incidents related to data being accessed and modified concurrently. Such events are called *anomalies* and are formally described in [1] as a characterization of four main isolation levels. Namely these isolation levels are: *read uncommitted*, *read committed*, *repeatable read* and *serializable* (these levels are here presented from the least to the most strict level).

A different isolation level, currently the default in most commercial RDBMS is called *Snapshot Isolation* [1], [2]. It uses both multi-version concurrency control and timestamps in order to avoid using locks [3], allowing a transaction to work over a consistent snapshot of data. This is presented as one of the main advantages of this isolation level, as a transaction is never blocked performing a read operation (which it is the case for concurrency control based on mutual exclusion), potentially increasing the level of concurrency.

Snapshot isolation works by creating two timestamps for each transaction. Upon the start of a transaction, a start timestamp (*Ts*) is assigned and, this particular transaction will observe all versions up to *Ts*. When the set of operations within a transaction ends, the transaction will try to commit, being assigned at that time a commit timestamp (*Tc*), if the transaction is allowed to commit. It should be noted that a transaction will only be allowed to commit if no other concurrent transaction has modified the same set of tuples (i.e a write-write conflict). From the moment a transaction is committed, all following transactions will be able to observe its modifications.

The Snapshot Isolation criterion avoids all anomalies described in the ANSI SQL standard [1], [3]. However, an anomaly called *write skew* may still occur. A write skew occurs when at least one safety feature for a system is disregarded, due to write-write synchronization problems, which causes Snapshot Isolation not to be serializable. Nonetheless, Snapshot Isolation is usually the most strict isolation level found in RDBMS, and for a wide array of applications it is possible to achieve a serial execution [4] or even to fully implement serializability [5].

B. Transaction ordering and certification

Snapshot Isolation requires the use of two modules: a timestamp generator to produce both the *Ts* and *Tc* timestamps, as well as, a certification authority to verify the existence of write-write conflicts and thus decide on whether a transaction should successfully commit or abort.

The timestamp generator issues new timestamps in an increasing monotonic way, which ensures total order as a stamp issued for transaction *Tj* is greater than the one for *Ti*, providing that $Ti \rightarrow Tj$.

The certification authority will use the timestamps that characterize a transaction to verify the existence of concurrency violations (i.e. write-write conflicts). Formally, two transactions characterized by $[Ts_A, Tc_A]$ and $[Ts_B, Tc_B]$ are said to be concurrent if $Ts_A \geq Ts_B$ or $Ts_B \geq Ts_A$ and $Ts_A < Tc_B$ or $Ts_{\leq} Tc_A$. In other words, when a transaction wants to commit, the certification authority will verify if there is a concurrent transaction operating over the same data items as the transaction waiting to commit. On the one hand if there is, and due to a write-write conflict, the certification authority will not allow the transaction to commit, causing it to be aborted. On the other hand, if no concurrent transaction conflicts the transaction successfully commits.

III. NoSQL databases

From a user's perspective, NoSQL databases differ from traditional relational databases by not encoding expressive data relationships, but being instead based in a simple key/value data model, and much as a consequence not providing a complex query language but generally minimal programming interfaces. In addition, typically, these new databases run in a distributed environment composed by hundred or thousands of computing nodes.

Actually, in order to promote high availability and scalability, NoSQL databases run in a more laid-back consistency criteria when compared with their relational counterparts. Some implementations use a relaxed consistency criteria called *eventual consistency* [6]. Running at this consistency level, requests are forwarded asynchronously, which may create periods where stale data can be read. For a comprehensive survey on the current NoSQL database offer, the interested reader is referred to [7]. For the purpose of this work we selected Cassandra and HyperDex, some of the most popular NoSQL databases.

A. Cassandra

Cassandra [8] is a distributed NoSQL database developed within Facebook to enable it to cope with large amounts of write operations while trying to achieve low latency read operations. Cassandra follows a fully decentralized architecture built on top of cheap commodity computing nodes that connect in the form of a logical ring. It follows a flat design in what concerns to the roles taken by each node. Cassandra adopts the BigTable's data model, being modelled in a multi-dimensional sorted map, without provision for multi-version tuples. Data is organized in structures called *Column Families*, which resemble a table in the relational model. The data objects held in Cassandra are accessible through an API that allows simple put and get primitives. With such API, there is no complex querying mechanism available.

Cassandra follows one of two possible partition strategies to shard data objects across replicas: random or ordered. By default, the random partitioning strategy is applied, which uses a horizontal partitioning strategy called *consistent hashing* [9]. The replication used by Cassandra depends on whether the

cluster is spread across a single or several data centers. With a single data center, Cassandra uses a *simple strategy* replication mechanism that is aware of a replication factor k.

When a tuple is inserted, it is firstly placed on the local node, being afterwards forwarded to the next $k-1$ nodes in the logical ring. Each Cassandra node can answer read and write requests. Whenever a client establishes a connection to a node, this specific node becomes the coordinator for that specific request. The coordinator uses its awareness of the replication strategy to decide what nodes should be contacted to fulfil the request. If it is a write request, the coordinator contacts all the online replicas (despite the replication factor k in place) that should hold the object according to the placement strategy. In the case of a read request, the coordinator contacts the k nodes imposed by the replication level. During the read procedure, if inconsistencies are detected, Cassandra uses the timestamp information present in each data object to provide the client with the most updated object. Afterwards, the coordinator re-establishes the inconsistent replicas by forcing them to update the specific data object (read repair request). This procedure may cause a situation where update operations might not be readable from every node. Therefore, during this period, a client read request over the data items involved in the read repair request may retrieve stale data.

Past versions of Cassandra (prior to version 2.0) did not offer any sort of transactional guarantees with ACID semantics. Newer versions of Cassandra introduce the concept of *Lightweight transactions* [10]. However, what actually is provided is a set of conditional read and put operations, trading isolation and atomicity for high availability and fast write performance. Thus, Cassandra's API still distances from a fully transactional API with atomicity and isolation at transaction level.

B. HyperDex

HyperDex [11] is a new distributed and scalable NoSQL database that is trying to change the pace in current NoSQL solutions by natively supporting queries using secondary indexes and strong data consistency guarantees, but still without transactions. While most NoSQL databases that support horizontal scaling either use a hashing function to map keys to computer nodes (like Cassandra) or partition the keyspace into several contiguous regions that are then assigned to several nodes (like HBase and BigTable); HyperDex uses a new object placement strategy called *hyperspace hashing* that accounts for several attributes when mapping objects to computing nodes.

Hyperspace hashing creates a multi-dimensional euclidean space, where each dimension corresponds to a single searchable attribute. The hyperspace is split into several regions that are then assigned to a single server. The task of choosing a holding server for an incoming request is led to a manager node called coordinator that identifies the server responsible for a specific region. The data partitioning implied by HyperDex significantly reduces the amount of servers that need to be contacted to answer a request. However, tying each key attribute to a dimension in the keyspace could become unmanageable as the hyperspace volume would grow exponentially with the number of dimensions. To avoid such problem, HyperDex partitions each space into smaller *subspaces*, each one accounting for several key attributes as a dimension. Applying this data partitioning strategy reduces the global dimensionality of the space and the subsequent number of servers contacted to fill in a request.

As other NoSQL databases, HyperDex achieves fault-tolerance by replicating regions in more than one server. Previous NoSQL systems would use eventual consistent update mechanisms, allowing a certain replica to accept updates concerning a key at a latter point in time. However, changes in the set of replicas from multiple concurrent updates could possibly result in inconsistencies across subspaces. To contradict this, HyperDex introduces a new replication strategy called *value dependent chaining* that is able to provide total order of replicas while performing updates on a given key.

As far as transactional operations is concerned, HyperDex does not natively provide support for such kind of operations. However, its data placement and data replication strategies are the basis to a an add-on called `Warp` [12].

IV. pH1

The pH1 middleware layer positions itself between the client and server of the NoSQL database, introducing transactional guarantees. It extends the client interface exported by the NoSQL database with commands to start and end transactions. After starting a transaction, the client will then execute a sequence of operations according to the NoSQL API but now in a transactional context. Once these operations are finished, the client will invoke an end transaction method, and pH1 will determine whether the transaction can successfully commit or should be aborted. pH1 offers Snapshot Isolation for which it will be providing a multi-version abstraction of the underlying NoSQL database. The pH1 was built in order to be generic and independent of the NoSQL database. That is, in principle, it can be used with any NoSQL database.

The architecture of pH1 is based on two different modules: (i) the Transaction Manager (TM) and (ii) the Non Persistent Version Store (NPVS). The TM (in the form of the transaction's write-set) and the NPVS represent two data sources used by the middleware. In addition, there is a third data source that is the NoSQL database. By data sources, we mean that data resides in these three modules, which will be accessed differently according to the operations being performed.

The middleware relies on another module the Timestamp Oracle (TSO). The TSO is an external certifier and timestamp generator reused from the OMID [13] project by Yahoo!. To be able to recover from failures, the TSO in OMID uses BookKeeper, a distributed logging service as a Write Ahead Log (WAL). The write procedure to this WAL is handled by OMID's TSO client, which we use in pH1. The articulation between the NPVS and the TSO modules will enable pH1 to provide Snapshot Isolation. While the first will keep all the versioned tuples of data, the TSO will act as an oracle, providing new timestamps to newly created transactions and, it will also keep track of the modifications done by each transaction, deciding whether a transaction should commit or otherwise abort in the case of conflicts occurring during concurrent execution.

In essence, the pH1 middleware should be able to scale with the number of clients by the adding more pH1 instances to

the system. Figure 1 depicts a configuration with two instances. Each instance shares access to the NoSQL database and to the same TSO module. The different NPVS instances communicate with each other by means of a group communication protocol. Over the following Sections, we describe the role of each module.

A. Transaction Manager.

The transaction manager is a central module of pH1, exporting and keeping the transactional context of each active transaction in the system and creating a proxy in-between modules. The transaction manager will provide to the client a simple interface comprised of two operations, which will enable it to (1) start a new transaction and (2) ask for the transaction to be terminated.

1) **Start Transaction***:* The request of a new transaction is brokered by asking the certifier module for a new timestamp (Ts). As previously introduced, each transaction will have a private snapshot of the data items being modified in that transaction (write, update and delete operations). This refers to the transaction's write-set, one of the three data sources used by pH1, so the transaction can read its own writes as defined by the Snapshot Isolation level.

2) **Try Commit***:* When the client wants to terminate the current transaction, it will ask the transaction manager to try commit the current transaction. The transaction manager forwards the modifications of the current transaction to the certifier module, which will decide if it can be committed or if it should abort. In case the certifier decides the transaction should commit, it will reply to the transaction manager with the commit timestamp and the transactional operation `flush` is executed.

B. Transactional Operations.

In the context of a transaction, the client can perform a group of primitive operations that comply with the operations exported by the API of most NoSQL databases. Namely, it can perform `read`, `scan`, `write` and `delete` operations. Additionally, there is a fifth operation, the `flush` operation that can only be executed by the transaction manager. Each one of these operations is executed in the context of one transaction. Besides the write-set, a single transaction contains a start timestamp (Ts) and a commit timestamp (Tc).

1) **Write Operation***:* The write operation, as depicted in Algorithm 1 will enable the client to insert or update a data item in the NoSQL database. However, when a client performs a write operation in the context of a transaction, that operation is not immediately flushed to the NoSQL database. Instead, it remains in the transaction's write-set and its kept there until the end of the transaction. If the transaction is committed then the modifications are persisted in the NoSQL database, or discarded if the transaction is aborted.

2) **Delete Operation***:* The delete operation enables the client to remove items from the NoSQL database. The unroll of a delete operation is similar to the write operation in what concerns to the steps of its execution, as it is actually only executed when the transaction can commit. Algorithm 2 depicts this operation. The major difference from the write operation lies in the fact that this operation adds a remove operation to the transaction's write-set.

Algorithm 1: TM: *Write* operation

Data: table, key, column, value
$searchKey \leftarrow table + key + column$
if $!WriteSet.contains(searchKey)$ **then**
  $WriteSet.insert(searchKey)$
$WriteSet.addWriteOp(searchKey, value)$

Algorithm 2: TM: *Delete* operation

Data: table, key, column
$searchKey \leftarrow table + key + column$
if $!WriteSet.contains(searchKey)$ **then**
  $WriteSet.insert(searchKey)$
$WriteSet.addRemoveOp(searchKey)$

3) **Read Operation***:* The read operation allows the client to read a data item. The algorithm behind this operation is key to ensure that the system complies with the Snapshot Isolation criterion. As previously mentioned, pH1 uses three different data sources: (i) the transaction's write-set, (ii) the NoSQL database and (iii) the NPVS. According to the Snapshot Isolation criterion, when reading an item, a given transaction is expected to read the most recent version up to its start timestamp (Ts). Therefore, the order in which the data sources are accessed is fundamental to determine the latest version of a given tuple. As a result of pH1's architecture, the latest version of a tuple will be found in the different data sources in this specific order:

1) **Transaction's write-set**: If it was modified by the current transaction;
2) **NoSQL database**: If the most recent transaction that modified that item had successfully committed prior to the beginning of the current transaction;
3) **NPVS**: If it was modified by a transaction that successfully committed after the start of the current transaction (i.e. a concurrent transaction).

The protocol of the *Read Operation* is depicted in Algorithm 3. First, it will verify if the given tuple exists in the transaction's write-set. If it does, it is returned because it is the latest version available. If it does not, then the NoSQL database is checked. If in fact the NoSQL database holds the tuple and the associated version check passes ($Ts \geq NoSQLcontent.Tc$), the tuple is returned. On the contrary, if the NoSQL database does not have the latest version ($ts < NoSQLcontent.tc$), the NPVS is checked and if the tuple exists there, the latest version according to Ts is returned. Please note that the NPVS may hold several versions for a given tuple, making this verification needed. Once the read operation can successfully retrieve a given tuple from one of the three data sources, all the left data sources are skipped.

Algorithm 3 establishes the standard procedure regarding a read operation. Nevertheless, the fact that we are dealing with NoSQL databases may cause issues related to the underlying consistency model (for instance eventual consistency) of some

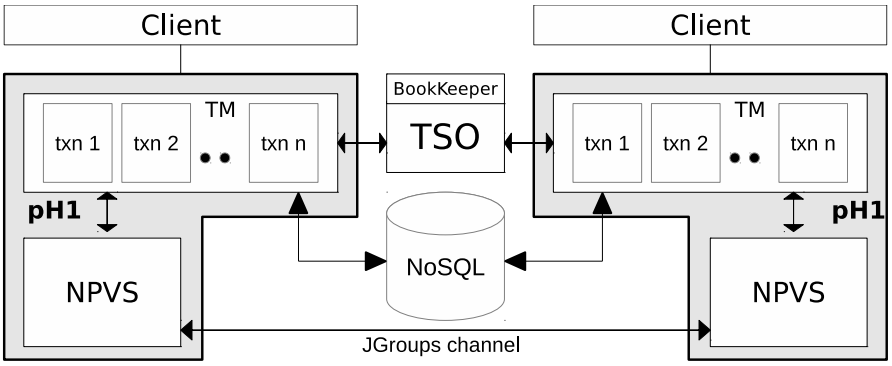


Fig. 1: Overview of a configuration with two pH1 instances.

Algorithm 3: TM: *Read* operation

```
Data: Table, Key, Column, Ts
searchKey ← table + key + column
if WriteSet.contains(searchKey) then
    return WriteSet.read(searchKey)
else
    NoSQLcontent ← NoSQL.read(searchKey)
    if NoSQLcontent != null then
        if Ts > NoSQLcontent.Tc then
            return NoSQLcontent
        else
            NPVScontent ←
            NPVS.read(searchKey, Ts)
            return NPVScontent
    else
        NPVScontent ←
        NPVS.read(searchKey, Ts)
        return NPVScontent
```

NoSQL databases. As previously stated in Section III, when executing a write operation, a NoSQL database may acknowledge the success of the write operation, while it may not have been successfully forwarded to all participating nodes. This behaviour may become a problem when reading a tuple from a node in the NoSQL database that has not yet been updated. Therefore, if during the read, the NoSQL database retrieves an empty result, the NPVS is checked to verify if the key exists. If it does, we may be in the presence of the illustrated issue and the version in the NPVS is returned if it belongs to a concurrent transaction. If it does not, the key never existed and no more than a null is returned.

4) **Scan Operation***:* The scan operation enables the client to perform a read operation comprised of a set of several keys. This operation must also comply with the Snapshot Isolation criterion, returning the latest version of every key in the scan operation according to its snapshot. The inherent algorithm behind this operation follows the exact same steps as the one for the read operation. The only difference lies on the fact that it is iteratively for each key instead of just one.

5) **Flush Operation***:* When the client finishes all the operations within a transaction, it will forward to the transaction manager a request to try commit the current transaction. As previously described, the transaction manager will forward the modifications of the current transaction (the transaction's write-set) to the TSO module, which will decide upon the commit or abort of the transaction. In the case the TSO successfully commits the transaction, it will issue a commit timestamp (Tc) and the transaction manager will invoke the flush operation to make changes durable.

When the flush operation is called, the transaction may hold several operations in its write-set that can invalidate themselves (e.g. write(x) and delete(x)). To avoid situations like such, the commit operation submits the transaction's write-set to a conciliation procedure that removes such issues.

The NoSQL database should hold the latest version of any tuple. As a result, before writing the new values of the transaction into the NoSQL database, the NPVS is updated with the existing tuples in the NoSQL database, which can be done asynchronously. When the NPVS acknowledges the writes, the transaction's write-set is flushed to the NoSQL database. This can actually be done asynchronously and not only at the time of flush. When the current transaction is allowed to commit and gets a Commit Timestamp, the TSO will not immediately update the current start timestamp (i.e. increment it). The TSO will wait until the current committing transaction flushes its modification to the NoSQL database, and only after that, update the current start timestamp so new transactions can observe the latest updates. In the meantime, the TSO provides a batch of timestamps that span from the start to the commit timestamp of the transaction that is performing the flush. If we didn't do so the Snapshot Isolation criteria could be violated. As a result, the TSO never blocks the timestamp acquisition, and thus no transaction is ever prevented from starting.

As transactions commit, the transaction manager no longer needs to hold information regarding those transactions. Thus, periodically, the transaction manager verifies which is the oldest active transaction and dictates the removal of all data regarding transactions prior to that.

C. Non-Persistent Version Store

As previously introduced, the Snapshot Isolation criterion requires the maintenance of versioned tuples. In order to support a wider range of NoSQL databases, specially those that

do not natively support multi-version tuples, we introduced the NPVS in pH1 to manage and store tuple versions.

The architecture of the Non-Persistent Version Store spreads along a group of individual and equal nodes, building a homogeneous distributed repository system. The NPVS can be configured to use any degree of replication. In the first place, the reason for using replication in the NPVS is due to fault tolerance. In other words, NPVS nodes always maintain their data in memory so if a NPVS node fails all its data is lost. By using replication (either full or partial) we could still access the data in other NPVS nodes. Moreover, in pH1 the user can configure the degree of replication it wants trading off performance for fault tolerance. When configured for full replication, each node is able to receive and process client requests. The individual nodes connect through a channel established by a group communication toolkit called JGroups [14]. The JGroups toolkit establishes a channel that enables the reliable and atomic exchange of messages among group members. This toolkit creates an abstraction called *view* that joins all the members in the channel and also manages its membership. Besides the management of the *view*, the toolkit allows for messages to be sent in a unicast or multicast fashion.

Implementation details:

The client of a NPVS node is a transaction manager instance. To enable the communication between nodes, the NPVS system defines two types of messages that will be sent through the established channel.

1) **Write message**: This message will hold the versioned tuples to be replicated across the nodes of the repository;
2) **Eviction message**: When a node receives an order to discard all versioned elements up to a given version, it uses this message to multicast that information to the remainder repository nodes.

Each NPVS node exports to the transaction manager a very simple API, comprised of only two operations:

1) **Write operation***:* Each and every node of the NPVS can handle a write operation. A write operation will hold the corresponding value and version to be associated to a given table, column and key. Upon the reception of such request, the node that received the request stores the versioned element and sends a write message through the established channel. When it receives the message from the group communication service (GCS), it informs the client that the request has been successfully completed.

2) **Read operation***:* The read operation will enable the transaction manager to access the versioned tuples. Due to the replication strategy in place, every node holds a full copy of versioned tuples within the system, which enables each node to answer this operation. During this operation, the node that is handling it only performs a lookup to the desired key, returning it if it exists.

D. Timestamp oracle and certifier

The timestamp oracle and certifier used in pH1 was developed by Yahoo! in the context of the OMID [13] project. Similarly to pH1, the OMID project was developed to allow NoSQL databases to be compliant with the transactional paradigm. The underlying NoSQL database used is HBase [15] and currently, this project only supports this implementation. Moreover, the system is highly coupled to the underlying NoSQL database and, thus assumes it already provides multi-versioning.

The OMID project uses a snapshot isolation compliant certifier, that does not use mutual exclusion primitives to enable the execution of concurrent transactions, but as pH1, uses timestamp based concurrency. The modularity of the OMID project, allowed us to re-use its certifier module, since it is actually decoupled from the data persistence layer. This module is responsible for two main functions: (i) generating new timestamps and (ii) certifying transactions.

The generation of timestamps must ensure that each timestamp is unique and, thus totally ordered. Specifically, the need for total order is key to the correct behaviour of the entire system, as described in Section II, guaranteeing that timestamp B is greater than A if A precedes B ($A \rightarrow B$). The timestamp increment is dictated by the transaction manager when it starts a new transaction, and also when the certifier determines a transaction to be committed.

The certification process is based on the information held in the certifier concerning all previously committed transactions and the modifications made by the transaction being certified. Only then, the certifier will be able to assess if the current transaction is concurrent with any transaction and if so, if there is a write-write conflict. If no conflict is detected, the certifier authorizes the transaction to commit. On the contrary, if the transaction conflicts with another previously committed or concurrent transaction, it is aborted and its changes are discarded.

In OMID, the TSO holds information regarding all committed transactions on the system, along with its modifications. To tolerate the failure of the TSO module, OMID introduces BookKeeper, a distributed logging service, that works as a Write Ahead Log (WAL), where it dumps the state of every committed transaction on the system. This allows to recover from a failure of the TSO without loosing information regarding previously committed transactions.

V. Implementation

The pH1 middleware, which includes all the aforementioned modules, was implemented in Java. Additionally, bindings to the two NoSQL databases were used, along with an event driven asynchronous communication toolkit called Netty, which allowed the link between the transaction manager and the TSO.

The communication among the nodes that build the Non-Persistent Version Store was achieved through the use of a group communication toolkit (JGroups [14]). As far as the versions used in each NoSQL database, we used Apache Cassandra version 1.0.10 and Hyperdex version 1.0.5.

As previously stated, pH1 uses the same timestamp oracle and transaction certifier as in OMID, the TSO. For the implementation of pH1, we used the same TSO configuration as in the OMID project, placing one Bookeeper node along with the TSO module, so that it could recover from a possible failure

without loosing information regarding previously committed transactions.

VI. Evaluation

Along this section we present the evaluation of pH1 over two different perspectives. Firstly, we quantify the overall throughput loss imposed by the introduction of transactional guarantees and a per operation latency analysis. For this matter, we used Yahoo!'s YCSB [16] benchmark. Secondly, we evaluate pH1's versatility under a TPC-C workload, comparing the results achieved for pH1 with OMID [13] by Yahoo!.

A. YCSB

The Yahoo Cloud Service Benchmark is a benchmark that allows the comparison among data stores designed for the cloud computing paradigm. In other words, *Yahoo!* developed this benchmark because both the paradigm and the access pattern of such data stores are quite different from the ones used by traditional benchmark systems, mostly designed for relational databases.

The YCSB benchmark starts by creating a defined number of concurrent clients that will try to perform a set of operations according to a pre-defined workload. The type of operations available include Read, Scan, Delete and Update operations.

We have modified the YCSB benchmark system to enable the existence of multiple update operations in a single operation, that is, a single operation run inside a transactional context. To do so, we introduced an operation called *Multi Update* in which we create an update transaction comprised of ten single update operations.

1) Configuration and tests: To perform the evaluation we relied in a configuration comprised of five identical machines, each of which is equipped with an Intel i3-2100-3.1GHz 64bit processor, 8GB of RAM and SATA II (3.0 Gbit/s) hard drives. These machines are interconnected by a switched Gigabit Ethernet and all of them ran Ubuntu 12.04 LTS as its operative system.

The five used machines were distributed as follows: each of two machines ran a NoSQL database node, one the TSO module and each of the remaining two ran a YCSB client, co-located with a pH1 instance. The NPVS nodes in each instance were configured to use a full replication strategy so that it is able to provide fault tolerance as previously described.

We divided the tests in two runs, a first one where we used the Cassandra NoSQL database and a second one where we used the Hyperdex NoSQL database. In each of these runs, we ran the benchmark with and without pH1, measuring the throughput and latency penalty registered by providing transactional guarantees.

Each run was divided in two stages: a first stage were the underlying NoSQL system was populated with data – 1,000,000 entries with about 1KB each – and a second stage where the benchmark client ran the specified workload. The workload was configured to run 25 concurrent clients in each machine, under a workload that amounted to 450 thousand operations according to the following distribution: 45% Read operations, 30% Scan operations, 12.5% Update operations and 12.5% Multi-Update operations. All experiments were performed under a uniform distribution of data in which all keys selected by the YCSB client are equally likely to be chosen.

2) Results: Figure 2 presents the results achieved for the described experiments. In what concerns the latency evaluation, the numbers depicted concern the average per-operation latency of both YCSB instances from five independent runs. Figure 2(a) shows that in all operations, independently from the underlying NoSQL database the introduction of pH1 caused an increase in latency, with the exception of the *Multi-Update* operation. All these operations were hampered by the cost of starting and ending each transaction, which justifies the differences. However, in both NoSQL databases used, the *Multi-Update* registered a smaller latency when pH1 was used. This is due to the fact that when using pH1, each single operation in the *Multi-Update* operation was performed in batch as opposed to the respective NoSQL database without pH1, where each *Multi-Update* operation is comprised of ten single *Update* operations.

	Cassandra	Hyperdex
Native (K Op/sec)	1.336	2.886
pH1 (K Op/sec)	1.164	2.655
Penalty (%)	13	9

TABLE I: Relative throughput loss

In both runs, the introduction of pH1 and thus the provision of transactional guarantees registered a very similar cost, measured in throughput loss. Table I depicts the registered difference where the use of Cassandra as the underlying database accounted to 13% throughput loss, in comparison to 9% when using Hyperdex.

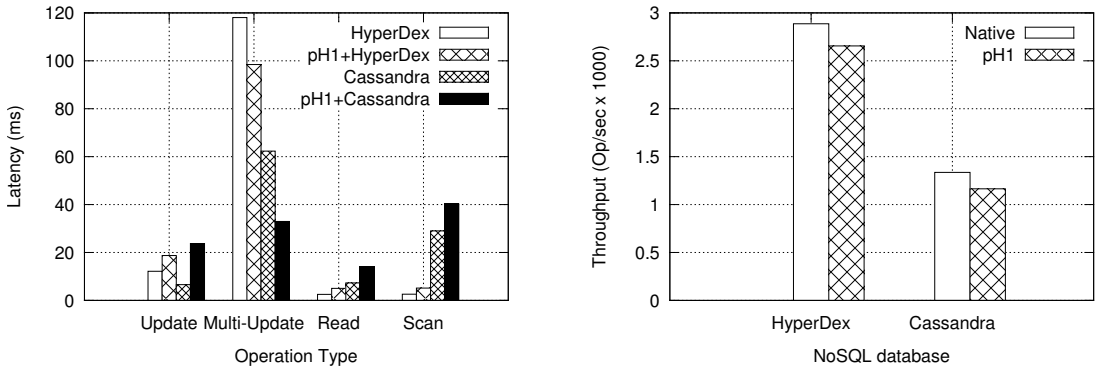
B. TPC-C

In order to verify the versatility of our middleware layer, we used a significantly different workload. For this purpose we used PyTPCC, an implementation of the OLTP standard workload TPC-C for NoSQL databases, which allowed us to make a comparison with Yahoo!'s OMID. We extended PyTPCC in order to provide an implementation for HyperDex.

This workload uses 5 different types of transactions to simulate a scenario where a company composed of several warehouses, distributed across several districts, processes orders placed by clients. The transactions performed by this workload are comprised of several read and update operations unlike the previous benchmark. TPC-C uses data scattered along 9 different tables, where only 8% of operations are read operations. The remainder 92% are update operations, which characterizes TPC-C's workload as write heavy.

The throughput of this workload is measured in *tpmCs* or in other words *transactions per minute of New-Order* transactions.

1) Configuration and tests: The evaluation presented in this section comes as a result of two different experiments, which enabled the comparison of the throughput penalty introduced by pH1 and OMID. To achieve this comparison, each experiment was divided in two settings, testing the



(a) Per Operation Latency comparison

(b) Throughput comparison

Fig. 2: Assessment of the penalization introduced by pH1.

underlying NoSQL store used by the system with and without the respective transactional systems.

Both experiments ran under a TPC-C workload characterized as follows: 45% New Order transactions, 45% Payment transactions, 5% Order Status transactions and 5% Stock Level transactions.

In both experiments, each run was also divided in two stages: a first stage where the TPC-C database was populated with 10 warehouses resulting in a database with roughly 5GB and a second stage in which each client ran the specified workload.

For the first experiment we considered OMID, which can only operate with HBase as its underlying NoSQL database. In this experiment we used a configuration comprised of 6 machines with the same specification as introduced for the YCSB setup. The HBase cluster was deployed using 4 of the 6 machines in a configuration where one machine ran the HBase Master node co-located with the HDFS Name Node and each of the remaining 3, a Region Server co-located with a Data Node. Besides the HBase cluster, one machine was devoted to OMID's transactional certifier and the remaining machine ran one instance of the benchmark co-located with OMID's transactional client. For this experiment, the single instance of the benchmark exercised a total of 100 concurrent clients, in runs that lasted for 60 minutes.

For the second experiment we considered pH1 using HyperDex as its underlying NoSQL database, in a configuration comprised of 7 machines. The HyperDex cluster was deployed in 4 of these 7 machines, in which one machine ran the HyperDex Coordinator node and each of the remaining 3 ran an HyperDex Daemon. As in the previous experiment, one machine ran the transaction certifier of pH1 (the same used in OMID), and each of the remaining two ran an instance of the benchmark co-located with pH1, in a total of 50 concurrent clients each, totalling 100 concurrent clients. Also, to provide fault tolerance, the NPVS nodes in these machines were configured to use a full replication strategy.

2) Results: The results depicted in Figure 3 concern the average of 5 independent runs for both experiments.

Concerning the first experiment, HBase without transactional guarantees ranked at about 7,740 tpmC or new order transactions per minute. As expected, there was a throughput penalty when OMID was introduced for the TPC-C workload, which ranked at about 1,857 tpmC or new order transactions per minute. This results amount to a throughput penalty of 76%.

In the second experiment, Hyperdex without transactional guarantees ranked at about 3,800 tpmC or new order transactions per minute. Also as expected, there was a throughput penalty when pH1 is introduced for the TPC-C workload. In detail, this second experiment ranked pH1 at about 3,000 tpmC or new order transactions per minute. This results amount to a throughput loss of 22%. The depicted results are consistent with those of YCSB, namely the introduction of transactional guaranteed incurred in a throughput penalty.

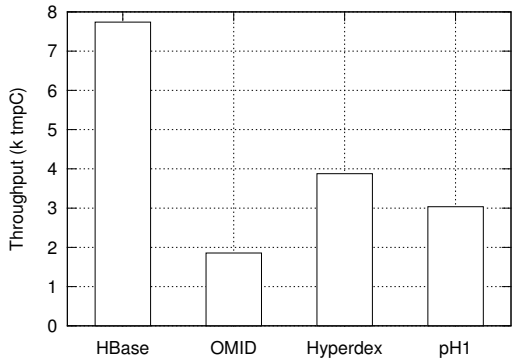


Fig. 3: Assessment of the penalization introduced by pH1.

When establishing a comparison with the results achieved during the YCSB experiment, the slightly higher throughput penalty registered in the current experiment is a clear consequence of the write-heavy nature of this benchmark. Please recall that while in the workload of the YCSB experiment only 25% were write operations; the transactions of the current workload are built from mostly read-modify-write operations, which makes this difference expected.

	HBase	HyperDex
Native (K tmpC)	7.740	3.876
OMID (K tmpC)	1.875	-
pH1 (K tmpC)	-	3.036
Penalty (%)	76	22

TABLE II: Relative throughput loss

The summary of results depicted in Table II shows that pH1 achieved a throughput penalty almost 4 times lower than OMID. This results are specially relevant since pH1 reuses the transaction certifier module of OMID. Also, it is worth noticing that a key difference between OMID and pH1 lies in how operations are orchestrated at commit time. On the one hand, in pH1 the instruction to flush the modifications of a transaction only comes after a transaction is successfully committed by the transaction certifier; while in OMID, write operations by a transaction are actually persisted in the NoSQL database at the time they are executed. With such mechanism, if the certification of the transaction dictates that it should abort, the respective modifications have to be rolled back, which does not happen in pH1. In addition, in OMID the various tuple versions are persisted in the NoSQL database, which implies a greater effort to manage and retrieve them. On the contrary, pH1 does not persist the older tuple versions in the NoSQL database: they are transiently stored in the NPVS module.

Even though we cannot claim on pH1's scalability due to the rather small test setup used, we believe that the underlying components of pH1, namely the TSO, do not currently harm the chances for scalability. In OMID, the TSO is able to scale to high transaction rates. Still our architecture is pluggable and would allows us to use a more scalable option as [17].

Therefore, we can claim that the better results achieved are the result of our contribution, which makes use of our multi-version distributed cache as a way to achieve Snapshot Isolation, surpassing the problems caused by not having to rollback transaction modifications and relieve the NoSQL database of the burden to persist and manage tuples versions.

VII. Related Work

There are currently several projects that aim at providing transactional and strong data consistency guarantees over NoSQL databases. Namely: CloudTPS [18]; MegaStore [19]; ElasTras [20]; Percolator [21]; OMID [13]; Warp [12]; Walter [22]; CumuloNimbo [17]. Also, projects as: SCORe [23]; GMU [24]; NMSI [25] have introduced protocols that leverage the state of the art in partial and replicated transactional systems.

The CloudTPS system offers transactional ACID guarantees over any NoSQL implementation. To do so, it introduces several local transaction managers (LTM). Each of them holds a copy of a partition of the data held in the NoSQL database; allowing several LTM instances to exist, and scale-out as demand grows. Each instance will be responsible to ensure the consistency in its own partition. With its multi-instance architecture, the CloudTPS system uses a *two phase commit* protocol to enable transactions that might hold data from different partitions. In pH1, we reused some architectural features of CloudTPS, such as the fact that is not tied a specific NoSQL database and the possibility to have several instances working together. However, pH1 distances from the CloudTPS system as each instance is not responsible for a specific data partition, but rather for the group of transactions that were initialized within that specific transaction manager. Also, pH1 does not use mutual exclusion, but instead, timestamp based concurrency.

The MegaStore and ElasTras follow a quite similar approach as the CloudTPS system. The main differences between these projects and the pH1 middleware are some architectural features and the isolation level used.

The Percolator system shares the same isolation level as our contribution (Snapshot Isolation) however, it does so by using a distributed approach over BigTable using mutual exclusion primitives. The use of mutual exclusion primitives eases conflict detection while performing write operations, however, the fact that read only transactions also need to acquire locks has a great impact in performance. pH1 steps away from Percolator by using timestamp based concurrency control and thus avoiding the use of locking primitives.

In contrast to Percolator, the OMID project implements *Snapshot Isolation* but does it over HBase [15]. Like pH1, OMID relies on timestamp based concurrency to offer Snapshot isolation over HBase, but the management of multi version elements is done natively by HBase. In pH1 the NoSQL database only hold the most recent data required by the isolation criterion in a durable way. The versions held in the NPVS only concern concurrent uncommitted transactions and can be removed as soon as these transactions commit.

Warp is a transactional toolkit that operates on top of HyperDex [11], providing a transactional context with ACID guarantees. Warp operates under an one-copy serializable isolation criterion, that is enabled by a new commit protocol named *linear transactions*. Briefly, this new commit protocol makes use of the underlying HyperDex deployment and its server chain construction to perform two passes across each chain. While on the first pass, servers ensure the readiness of every tuple within a transaction, on the second, the last server in the first chain decides if the transaction is to be committed or aborted, and then forwards its decision the the remainder servers in the chain.

SCORe proposes a partial replication technique along with the introduction of validation and abort-free life cycle for read-only transactions based on GMU. SCORe uses a 2PC protocol for validation and persisting write operations. The response to the client happens once all updates have been flushed. Read-only transactions can be executed right away and be exempted of any certification procedure. In comparison, pH1 uses deferred writes for update transactions and read-only transactions, although not blocking, may be executed in a previous snapshot.

Walter and NMSI also introduce new variations for the Snapshot Isolation criterion, enabling it to support replicated data sources. Walter introduces Parallel Snapshot Isolation (PSI) under which a transaction within a given replica is able to observe a consistent snapshot and use a common ordering. Between replicas, PSI ensures a causal ordering of transactions, which the authors claim to enable scalability while relying in asynchronous mechanisms to share transactions across replicas. NMSI ensures that read only transactions are never blocked, as in pH1, and genuineness as only replicas involved make steps to execute.

CumuloNimbo is a PaaS platform that aims to bring the transactional paradigm to cloud infrastructures. CumuloNimbo provides syntactic and semantic transparency, allowing applications that run in a centralized infrastructure to be ported with

no modifications to work on top of this PaaS, and still benefit from the high scalability and elasticity.

Although pH1 shares with the aforementioned projects, the objective of providing transactional guarantees, it tries to create a generic solution that may be used with any NoSQL database. pH1 reuses some key architectural features, such as: (i) multi-instance execution, (ii) timestamp based concurrency and (iii) the Snapshot isolation level. However, pH1 positively distances from projects like Percolator and OMID that are tied to specific NoSQL implementations, contributing to a truly generic solution, as it is able to provide Snapshot Isolation even when working with NoSQL databases that do not have support for versioned tuples.

VIII. Conclusion

In this paper, we presented pH1, a middleware layer that attempts to cover the lack of transactional guarantees of most NoSQL implementations today by proposing a non intrusive transactional middleware layer that can be used on top of a generic NoSQL database.

The approach is based on the client interface of the underlying NoSQL database extending it with the capability to perform operations in a transactional context. As the main features of this middleware layer, we highlight: (i) the possibility to execute ACID compliant transactions with Snapshot Isolation and (ii) the fact that by extending the simple NoSQL interface it has a minimal impact on the database clients.

We tested our prototype on top of Apache Cassandra and HyperDex using two different benchmarks. In order to evaluate the overall cost of adding transactional guarantees. Firstly, we used Yahoo!'s widely used NoSQL benchmark, YCSB, to evaluate the inherent cost of using pH1 in an overall and per-operation basis. The achieved results showed that the overall cost of introducing transactional guarantees is moderate, namely 11% on average.

Secondly, we used an implementation of the OLTP TPC-C workload to assess the versatility of our middleware, by exposing it to a significantly different workload, mainly composed by read-modify-write operations. We also established a comparison with Yahoo!'s OMID, which provides Snapshot Isolation transactions over HBase. The presented results show that we were able to achieve a significantly lower throughput penalty and still keep our middleware modular and generic to any NoSQL database, when comparing with OMID, which is tied to HBase. We are then led to conclude on the validity of our approach and the positive consequences to offering strong consistency over NoSQL stores.

Acknowledgements

We would like to thank D. Manivannan and the anonymous reviewers for their helpful comments. This work was part-funded by project CoherentPaaS: A Coherent and Rich PaaS with a Common Programming Model (FP7-611068) and ERDF– European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by national funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER- 037281.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *SIGMOD Rec.*, vol. 24, no. 2, May 1995.

[2] D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in PostgreSQL," *Proc. VLDB Endow.*, 2012.

[3] S. Revilak, P. O'Neil, and E. O'Neil, "Precisely serializable snapshot isolation," in *ICDE*. IEEE, 2011, pp. 482–493.

[4] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 492–528, Jun. 2005.

[5] M. Cahill, U. Röhm, and A. Fekete, "Serializable isolation for snapshot databases," ser. SIGMOD '08. ACM, 2008.

[6] W. Vogels, "Eventually consistent," vol. 52, no. 1. New York, NY, USA: ACM, Jan. 2009, pp. 40–44.

[7] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[8] A. Lakshman and P. Malik, "Cassandra - A Decentralized Structublue Storage System," in *LADIS'09*, 2009.

[9] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *STOCK*, New York, NY, USA, 1997.

[10] C. Team, "What's new in Apache Cassandra 2.0," Apache Cassandra, Tech. Rep., 2013. [Online]. Available: http://www.datastax.com/wp-content/uploads/2013/09/WP-DataStax-WhatsNewC2.0.pdf

[11] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 25–36.

[12] ——, "Warp: Multi-key transactions for key-value stores." HyperDex, Tech. Rep., 11 2013. [Online]. Available: http://hyperdex.org/papers/warp.pdf

[13] Y. M, Ferro.D, Kelly.I, Junqueira.F, and Reed.B, "Lock-free transactional support for distributed data stores," *ICDE*, 2014.

[14] N. Carvalho, J. Pereira, and L. Rodrigues, "Towards a generic group communication service," in *On The Move To Meaningful Internet Systems, International Symposium on Distributed Objects, Middleware, and Applications (DOA)*, R. Meersman and Z. Tari, Eds., vol. 4276, 2006, Proceedings Paper, pp. 1485–1502.

[15] L. George, *HBase: The Definitive Guide*, 1st ed. O'Reilly Media, 2011.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC'10*, 2010.

[17] R. Jimenez-Peris, M. Patiño-Martinez, K. Magoutis, A. Bilas, and I. Brondino, "Cumulonimbo: A highly-scalable transaction processing platform as a service," *ERCIM News*, 2012.

[18] W. Zhou, G. Pierre, and C.-H. Chi, "CloudTPS: Scalable transactions for web applications in the cloud," *IEEE Transactions on Services Computing*, vol. 99, no. PrePrints, 2011.

[19] J. Baker, C. Bondç, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. L´eon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *(CIDR)*, Jan. 2011, pp. 223–234.

[20] S. Das, D. Agrawal, and A. El Abbadi, "Elastras: an elastic transactional data store in the cloud," ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009.

[21] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *SOSDI*, 2010.

[22] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*, 2011.

[23] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Middleware*, 2012.

[24] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS*, 2012.

[25] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems," in *SRDS*, 2013.