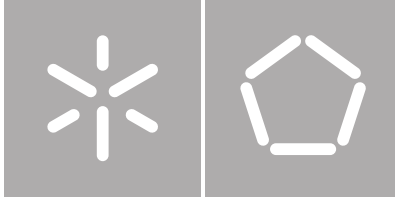**Universidade do Minho**
Escola de Engenharia

Fábio André Castanheira Luís Coelho

Implementation and test of transactional primitives over Cassandra

Outubro de 2013

Universidade do Minho
Escola de Engenharia
Departamento de Informática
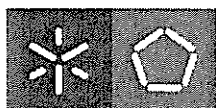
Fábio André Castanheira Luís Coelho

Implementation and test of transactional primitives over Cassandra

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor Rui Carlos de Oliveira
MSc Francisco Miguel Barros da Cruz

Nome: Fábio André Castanheira Luís Coelho

Nº Cartão Cidadão /BI: 13456866 4 ZY0   Tel./Telem.: 963428949

Correio eletrónico: faclc4@gmail.com / pg18776@alunos.uminho.pt

Curso: MEI   Ano de conclusão da dissertação: 2013

Área de Especialização: Sem área de especialização

Escola de Engenharia, Departamento/Centro: Departamento de Informática

TÍTULO DISSERTAÇÃO/TRABALHO DE PROJECTO:

Título em PT: "Implementação e teste de primitivas transacionais sobre Cassandra"

Título em EN: "Implementation and test of transactional primitives over Cassandra"

Orientadores: Professor Rui Carlos Mendes de Oliveira / Francisco Miguel Carvalho Barros da Cruz

Braga, 31 / 10 / 2013

Assinatura: _Fábio André Coelho._

To my parents

José Maria Coelho and Maria José Coelho

Acknowledgements

First of all, I would like to thank my advisor, Prof. Rui Carlos Oliveira that agreed to guide me through this thesis and since the first moment was a source of courage, guidance and help.

I would like to thank my friends Francisco Cruz and Ricardo Vilaça from the distributed systems group for all the talks, advices and patience that, without any doubt were key to improve my work. To all the other friends of the Distributed Systems Group, I leave my best regards and thank them for the wonderful work environment.

I thank my close friends and family that often against their will, were forced to understand my work. Their advice was indispensable to the completion of this work.

Finally, I thank my parents to whom I dedicate this work. Your support was essential.

Thank you.

vi

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. These limitations bring considerable complexity to the development of client applications and are therefore an obstacle to the broader adoption of the technology.

In this work we propose a middleware layer over NoSQL databases that offers transactional guarantees with *Snapshot Isolation*. The proposed solution is achieved in a non-intrusive manner, providing to the clients the same interface as a NoSQL database, simply adding the transactional context. The transactional context is the focus of our contribution and is modularly based on a *Non Persistent Version Store* that holds several versions of elements and interacts with an external transaction certifier.

In this work, we present an implementation of our system over Apache Cassandra and by using two representative benchmarks, YCSB and TPC-C, we measure the cost of adding transactional support with ACID guarantees.

Resumo

As bases de dados NoSQL optam por não oferecer importantes abstrações tradicionalmente encontradas nas bases de dados relacionais, de modo a atingir elevada escalabilidade e disponibilidade: garantias transacionais e critérios de coerência de dados fortes. Estas limitações resultam em maior complexidade no desenvolvimento de aplicações e são por isso um obstáculo à ampla adoção do paradigma.

Neste trabalho, propomos uma camada de middleware sobre bases de dados NoSQL que oferece garantias transacionais com *Snapshot Isolation*. A abordagem proposta é não-intrusiva, apresentando aos clientes a mesma interface NoSQL, acrescendo o contexto transacional. Este contexto transacional é o cerne da nossa contribuição e assenta modularmente num repositório de versões não-persistente e num certificador externo de transações concorrentes.

Neste trabalho, apresentamos uma implementação do nosso sistema sobre Apache Cassandra e, recorrendo a dois benchmarks representativos, YCBS e TPC-C, medimos o custo do suporte do paradigma transacional com garantias transacionais ACID.

x

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The advent of *cloud computing* is establishing new ways for users to store, access, share and process data [3]. All kinds of data created for personal or recreational use, be it text documents, presentations, photographs, videos are being stored online, "in the cloud" for immediate and continual availability. The storage, structuring and indexing of these huge amounts of data for subsequent efficient access and processing have been a major business opportunity but also a great challenge for cloud service providers. At the same time, the high availability standards and competitive offerings for cloud storage services and processing capabilities allure an increasing number of businesses to migrate their systems into the cloud, reducing maintenance and ownership costs. This adds up not only to the volumes of data to be managed by cloud providers but also to the diversity of solutions to be provided.

Cloud providers tend to use a whole set of logical storage technologies ranging from block devices and files to all types of database management systems. The latter are used internally as well as offered as services to the end users. While the choice of the right database paradigm (relational databases,

table or column based, non-relational databases, graph databases, etc.) is initially driven by the type of applications and workloads, cloud-based database systems bear non-functional characteristics crucial to leverage the cloud infrastructure that become key to this selection. Here we refer to the systems high availability and scalability.

Until recently, information systems have relied almost exclusively in relational database systems (RDBMS [20]). These are well matured database systems which provide a much consolidated access and programming interface based on SQL along with a remarkable programming abstraction that provides transactional semantics usually with ACID[1][14] guarantees. The use and provision of relational databases systems would thus seem the natural choice for cloud systems. However, so far, traditional RDBMS proved unable to sustain the required high availability and scalability of the cloud computing paradigm. 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, but typically just based on key-value pairs, and do not provide any form of expressive query language but very simple put and get interfaces. 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 [17]. Specifically, most of the highly efficient processing provided by SQL query engines inside

---

[1]ACID stands for Atomicity, Consistency, Isolation and Durability.

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 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 order to mitigate the difficulties of programming on top of NoSQL databases and therefore narrowing the gap between the RDBMS and NoSQL programming model, in this thesis we present the design and implementation of an elegant, non intrusive middleware system 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 [5] as its isolation criterion. The resulting transactional NoSQL database will be evaluated under representative workloads in order to show the overhead of the transactional layer as well as the impact of the transaction management on the system throughput.

Our approach is specific to the Snapshot 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 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 and is the cornerstone of our approach.

In the remainder of this dissertation we provide in Chapter 2 background

concepts and review related work, in Chapter 3 we present the concepts, architecture and implementation details of the pH1 middleware and then, in Chapter 4 its evaluation. Chapter 5 concludes this work.

Chapter 2

Background

In this chapter, we discuss two kinds of systems that are key to the design and development of pH1: NoSQL databases systems and transactional systems. Afterwards, we succinctly survey related work.

2.1 NoSQL databases

From a user's perspective, NoSQL databases differ from traditional relational databases systems (RDBMS) by not encoding expressive data relationships but, instead, being based on a simple key-value data model and, much as a consequence, not providing any kind of query language but minimal programming interfaces with put and get primitives.

Moreover, contrary to the usual centralized architecture of RDBMS, NoSQL databases tend to be much more flexible following a fully distributed architecture. Typically, NoSQL databases run in a distributed environment composed by hundreds or thousands of machines. Distributed processing in a scale like this introduces thus several challenges such as: data placement, data replication or agreement, that do not exist or are simpler in a central-

ized approach. NoSQL databases can have architectures that fall into one of the following main categories: fully decentralized, hierarchical or hybrid. In a fully decentralized architecture database nodes are typically organized in a logical ring. In a hierarchical architecture, there is a small set of modules that is responsible for maintaining data partitions and coordinate processing and storage modules.

When compared to the relational model, NoSQL databases adopt a fairly simple data model, where most solutions store data according to a key-value data model like Amazon's Dynamo [13]. There are however more complex data models, like the ones based on Google's BigTable's [9], where the notion of "keyspace" is created, holding several "column families". Each one of these column families is then populated with rows that have no constraint over the type or number of columns they may have, allowing several rows to have different columns. Still, despite the differences that may exist across different implementations, most NoSQL databases share concepts like the way requests are routed and processed, data storage and distributed systems techniques like replication, partitioning or failure detection.

Actually, in order to promote high availability and scalability, NoSQL databases need to run in a more laid-back consistency criteria when compared with their relational counterparts. Some implementations like Cassandra [16] use a relaxed consistency criteria called "eventual consistency". 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 [8]. For the purpose of this work we selected Cassandra, one of the most popular, open source, and mature NoSQL databases.

2.1.1 Cassandra

Cassandra is a NoSQL distributed key-value developed within the Facebook Inbox [16] project to enable it to cope with large amounts of write operations while trying to achieve low latency read operations. Its design was made in order to achieve a replicated structure of cheap commodity nodes placed geographically apart.

Cassandra follows a fully decentralized architecture where nodes form a logical ring. Cassandra has a flat model in what concerns to the roles taken by each node: each of them has the same responsibilities and thus the system has no single point of failure.

Cassandra is based on Bigtable's data model. Data is modelled in a multi-dimensional sorted map and allows objects to be grouped in structures called "*column families*". One column family enables the existence of several columns, which are then populated by row keys. Each column family resembles one table in the relational model. Cassandra does not impose any limitations concerning the total amount of columns each column family may have. One addition to Bigtable's data model, is the introduction of SuperColumnFamilies, where each attribute (a SuperColumn) may have a list or regular columns. Worth noticing is that Cassandra does not provide any support for versioned data.

Requests are handled by a *partitioner* running on each node that is in charge of choosing which nodes will actually store a specific row. Requests are handled as follows:

Forwarding requests

When a request arrives, that specific node becomes the *coordinator* for that specific operation. The *coordinator* acts as a proxy between the client

and the cluster. As data is spread along all nodes, the *coordinator* has
to decide which nodes should get the request, and how many node replies
it should wait until answering the client (each node issues one response).
This comes as a restriction imposed by the consistency level implemented.
Requests may be sent synchronous or asynchronously according to its type
and consistency level in place.

Read requests

Read requests are handled by the *coordinator* node for that specific op-
eration. The consistency level sets the amount of nodes that will get a direct
and synchronous request. The remainder nodes will afterwards receive an
asynchronous request to re-establish consistency if needed (read-repair).

Write requests

Write requests are handled by all replicas that own the data. If the node
is online, it will receive a direct and synchronous request. That is made
despite the consistency level configured; hence its effect will be translated to
the amount of replicas replying to the requests.

The partitioner runs accordingly to two different placement strategies:
Random and Ordered. The Random strategy uses *consistent hashing* [15]
over the keys, so that data is evenly distributed along the cluster. Each
Cassandra node in the cluster will hold a range of hash values. When a write
operation arrives, the partitioner will calculate the hash (MD5) of the key
to be written and will afterwards directs the request for the node with the
respective range assigned. The Ordered strategy also assigns each node a
range of keys. However, when a write operation arrives, the partitioner uses
the actual hexadecimal value of the key to be written, rather then calculating

its hash value; ordering them lexicographically. This placement strategy does not perform as best as the random strategy in what concerns to redistribute data across the cluster, when the need to reconfigure comes. Despite that, in theory, it allows the system to have better performance while executing range scans across rows, which can be a good feature for some workloads.

The replication strategy used by Cassandra depends on whether the cluster is dispersed by several or one data center. With just one data center, the replication strategy used ("Simple strategy") considers a replication factor k. Thus, when a data item is inserted, it is placed on the node chosen by the partitioner and on the following k nodes. If the cluster is scattered across multiple data centers, the replication strategy ("Network strategy") will use a similar strategy, chaining replicas of a key between the available data centers.

Cassandra operates under a tunable consistency model, which allows the consistency level to be chosen, in a per request basis. Some of the possible levels are: `one`, `quorum` or `all`. The selected consistency level will impact on the amount of nodes contacted in order to fulfil the consistency level for a specific request. As the name suggests, if the consistency level of `one` is chosen, only the replica that holds the item is contacted. For the `quorum` level, the number of nodes to be contacted depends on the replication factor in place (quorum: replication factor/2 +1 nodes), while for the `all` level, all the nodes are contacted.

The consistency model implemented by Cassandra creates the possibility for the existence of faults. The underlying fault model approaches two possible faults:

- *Fault A* : This fault occurs when Cassandra detects an error while performing a write operation. The write operation may have actually been performed in some nodes, but failed to reach all the nodes dictated

by the consistency level in place. Therefore, a read request may return
an empty result when it should have returned a non empty result.
For the key involved in this fault, there will be a time gap where this
mismatch may occur, in which Cassandra is recovering the inconsistent
replicas.

- *Fault B* : This fault occurs when a key is removed from Cassandra.
  Due to the consistency scenario it may take some time to the operation
  to be propagated to the entire cluster. A read request to a key like this
  may return a value that no longer exists.

As a result of this consistency level, it relaxes isolation and atomicity in
order to allow for high availability and performance executing write opera-
tions. Cassandra does not provide transactional contexts and therefore any
form of ACID guarantees.

In the following section, we discuss the main properties of transactional
systems from where we highlight the isolation property. Moreover, we discuss
how transactions are ordered and certified.

2.2   Transactional systems

Transactional systems introduce the concept of *transaction*. A transac-
tion is a sequence of operations whose execution traditionally satisfies the
following ACID [14] properties:

- **Atomicity**: Either all or none of the operations within a transaction
  are successfully performed.
- **Consistency**: Transactions preserve system constraints, as a whole
  any transaction takes the system from a valid state to another valid

state.

- **Isolation**: Concurrent transactions execution preserve the semantics of the defined consistency criterion or isolation level.

- **Durability**: The effects of a successful transaction are durable even in the presence of faults.

A transaction that performs successfully is said to "commit", otherwise it "aborts".

2.2.1 Isolation levels

A transactional system enables the coexistence of concurrent transactions which may lead to several incidents related to data being accessed and modified concurrently. These events called anomalies are described in [5] and they are used to characterize four isolation levels, formally described in the ANSI SQL standard. Namely these isolation levels are (in ascending order from the least to the most strict level): *read uncommitted*, *read committed*, *repeatable read* and *serializable*.

With the existence of concurrent transactions, transactional systems rely on concurrency control mechanisms to avoid those anomalies. These mechanisms can be based on mutual exclusion primitives, establishing "read locks" over data that will be read and, "write locks" over data that will be written. Thus, if a transaction acquired, for instance, a write lock over a data item; for the period of time that the lock is held, no other concurrent transaction will be able to acquire a write lock over the same piece of data. The different isolation levels define possible different behaviours regarding the acquisition of read or write locks over a data item by concurrent transactions, in order to avoid the different anomalies.

The most strict isolation level – *Serializable* – does not allow for any type
of anomalies to occur, as described in [5]. Please note that with a concurrency
control mechanism based on mutual exclusion, locks are not enough to ensure
the absence of anomalies, as in the Serializable isolation level. To provide
this isolation level, a protocol called *Two-Phase-Lock*[6] is usually used to
control the acquisition and lock release.

2.2.2   Snapshot isolation

A different level of isolation, currently the default on most RDBMS,
is the so called Snapshot Isolation. Snapshot Isolation was first presented
in [5] as a new isolation level. It uses both multiversion concurrency control
and timestamps in order to avoid using locks [19], 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 transac-
tion. 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 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. Thus, the start and commit timestamps limit the lifespan
of a transaction. From the moment a transaction is committed, all following
transactions will be able to read its modifications.

Anomalies

As seen in [5] and in [19], Snapshot Isolation avoids all anomalies described in the ANSI SQL standard. 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.
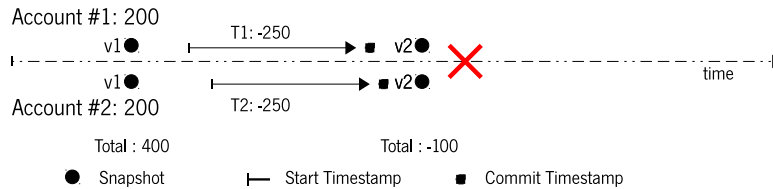
Figure 2.1: Snapshot isolation anomaly: write skew

Figure 2.1 shows a classic example of a bank system where a write skew anomaly occurs. Both transactions, T_1 and T_2 are performing a withdraw operation from two different accounts that belong to the same individual. Lets assume as a constraint for this system that the amount to be withdrawn can be higher than the available balance, if the sum of both account balances remain positive. Since when both transactions start, the system's constraint is met, both are allowed to carry on leaving the data in an inconsistent state, breaking a rule set for the system.

The consistency breach results from a write-write synchronization problem caused by the fact that both transactions operate over the same snapshot. Clearly this possible anomaly is what causes this isolation level not to be serializable as the strictest isolation level.

2.2.3   Transaction ordering and certification

Snapshot isolation requires the use of two separate 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.

Timestamp generator

With timestamp based concurrency, order is chosen before the actual transaction begins, by assigning a timestamp upon the creation of a transaction. The use of timestamp based concurrency implies that: (1) a timestamp has to be globally unique within the system and (2) at one particular instant, only one stamp can be generated. Most timestamps are based on the current system time, which makes them increasingly monotonic. This provides a notion of total order as a stamp issued for transaction T_j is bigger than the one for T_i, providing that $T_i \rightarrow T_j$.

In addition to the conditions stated before, there are other things to be considered when generating timestamps. Timestamps may be generated in a centralized fashion, in which there is a timestamp generator that assigns the stamps to all transactions; or, this task is carried out by multiple timestamp generators in the system. For the first case, no major problems arise as the timestamp generator in charge has full notion of the retrieved stamps and none of them will collide; while the second case will need a distributed agreement protocol to ensure the rules stated above remain.

Certification authority

The certification authority is bounded to work accordingly to a specific isolation level, ensuring that concurrency violations do not occur. With

snapshot isolation, the certifier will use the timestamps that characterize a transaction, the start timestamp (Ts) and commit timestamp (Tc) to verify the existence of concurrency violations (i.e. write-write conflicts).

Lets consider two transactions, the first characterized by $[Ts_A, Tc_A]$ and, the second characterized by $[Ts_B, Tc_B]$. These transactions are said to be concurrent if $Ts_A \geq Ts_B$ or $Ts_B \geq Ts_A$ and $Ts_A \leq Tc_B$ or $Ts_b \leq Tc_A$. In other words, when a transaction wants to commit, the certification authority will verify if there is a concurrent transaction that is operating over the same data items as the transaction waiting for certification. On one hand if there is, and due to the possibility of a write-write conflict, the certification authority will not allow the transaction to successfully commit, causing it to be aborted. On the other hand, if no concurrent transaction conflicts, the certification authority will ask the timestamp generator to issue a new timestamp Tc, for this transaction and the transaction can successfully commit.

2.3 Related work

There is nowadays a great deal of effort in trying to bring the transactional behaviour to several NoSQL implementations. Several projects offer already transactional guarantees and strong consistency, being supported by NoSQL solutions. In this section we will go over some of the projects, like CloudTPS [24], MegaStore [4], ElasTras [12], Percolator [18] and OMID [22].

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 them 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 to 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 it 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 therefore avoid the use of locking primitives, which specially impacts on read-only transactions.

In contrast to Percolator, the OMID project implements *Snapshot Isolation* but does it over HBase [2]. 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.

This contribution stands from OMID and other systems tied to specific NoSQL implementations, as it builds a transparent system that can be easily

adapted to work with NoSQL implementations that do not have support for multi version tuples.

Although pH1 shares with the remainder projects, the objective of providing transactional guarantees, it distances from them by trying to create a universal solution that may be used with any NoSQL database. PH1 relates with the aforementioned projects by reusing 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 universal solution, as it is able to provide Snapshot isolation even when working with NoSQL databases that do not have support for versioned tuples (MVCC: Multi-Version Concurrency Control).

Chapter 3

pH1

In this chapter we present the pH1 transactional middleware. We describe its architecture and then our prototype implementation in detail.

3.1 Architecture

The pH1 middleware layer positions itself between the client and 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 multiversion abstraction of the underlying NoSQL database.

The architecture of pH1 is based on three different modules: (i) the *TM: Transaction Manager*, (ii) the *NPVS: Non Persistent Version Store* and (iii)

the *TSO: Timestamp Oracle*. The NPVS and the NoSQL database are both
data sources used in pH1.

In Figure 3.1 is shown the architecture of pH1, and how the different
modules interact with each other. Particularly, in this figure it is depicted
a single instance configuration of pH1. In order to scale with the number of
clients, pH1 can be composed of several instances as shown in Figure 3.2.
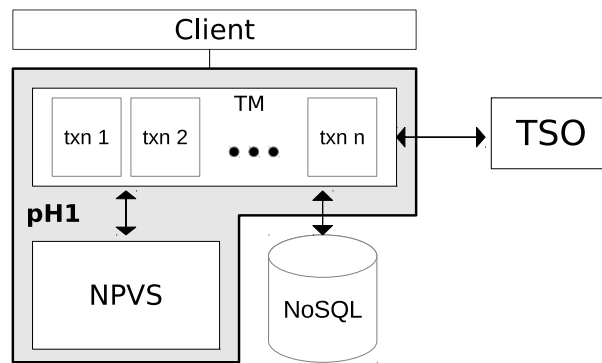
Figure 3.1: pH1 : single instance architecture

Each instance shares access to the NoSQL database and the same TSO
module. The NPVS nodes communicate by means of a group communication
protocol.

Next, we will describe in further detail the role of each module.

The **Transaction Manager** module is the central piece for this middle-
ware. It will be responsible for communicating with the other modules as
well as:

- Provide the transactional API to the clients;

- Maintain a view of all current active transactions within the system.

The **Non-Persistent Version Store** module is responsible for providing
a multiversion abstraction of the underlying NoSQL database to the clients.

Roughly, this is to be achieved by a distributed tuple cache persisted to the NoSQL database at the time transactions commit.

The **Timestamp Oracle** module is in charge of generating start and commit timestamps and certify transactions whenever requested by the *Transaction Manager.*
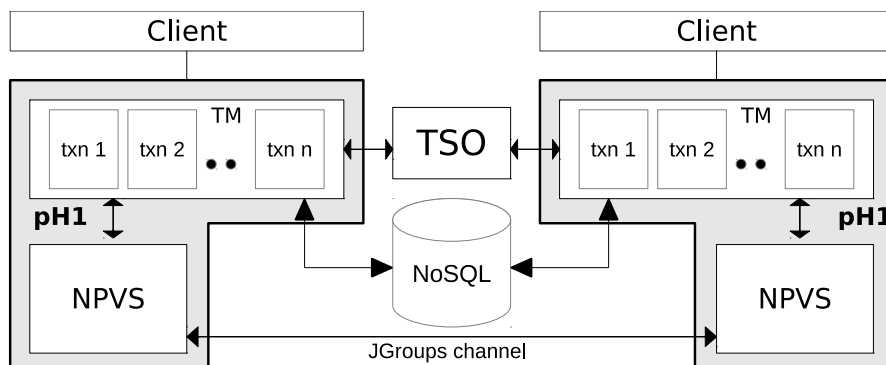
Figure 3.2: pH1 : two instances deployment

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 upon their success or failure in the case of conflicts occurring during concurrent execution.

3.2 Prototype

The pH1 was built in order to be generic and be independent of the NoSQL database. That is, in principle, it can be used with any NoSQL database. For the current prototype we used the Cassandra database.

In the following subsections, we will go through an extended presentation

of the actual implementation of each module that builds the pH1.

3.2.1   TM: Transaction Manager

The transaction manager module was devised to have a central role within the system. In what concerns the actual implementation, this module has the task of exporting and keeping the transactional context of each active transaction in the system, as well as to provide a proxy to the remainder modules.

Client interface

The *transaction manager* will provide to the client a simple interface comprised of two methods:

- **start transaction***:* The client will invoke this method to start a new transaction. This will enable the execution of Read, Scan, Write and Delete operations in the context of the transaction.

- **try commit**: The client will use this method whenever it wants to terminate a transaction. If successful, the TM will then send the modifications to the data layers.

Client interaction

When a client wants to read, write or delete something from the database, it will ask the TM to start a new transaction (Figure 3.3(a)). The TM will then direct a request to the TSO module, so the latter will generate a timestamp to initiate a new transaction.

The client may then perform read and write operations in transactional

context (Figure 3.3(b) and 3.3(c)). These operations will generate the respective interaction with the data source modules.

Whenever the client has finished the set of operations for a transaction, it will ask the TM to end the transaction by trying to commit the changes made by the transaction (Figure 3.3(d)). The commit request by the client will be forwarded to the TSO module, along with the transaction's write-set. In this case, after the TSO successfully certifies the transaction, the NPVS and Cassandra will reflect the modifications within the terminating transaction. The version present in Cassandra is placed in the NPVS, and the transaction's write-set is stored in Cassandra. After Cassandra acknowledge the success of the write operations, the TM notifies the TSO that the transaction succeeded. Only after receiving this notification will the TSO be able to generate new start or commit timestamps.

Transaction

A transaction is characterized by (i) the *start timestamp* (Ts) that represents the time of creation for the transaction; (ii) the *commit timestamp* (Tc) that represents the time at which the transaction is allowed to commit its changes and (iii) the transaction's write-set that will keep single write and delete operations for this transaction while it is not committed.

Transaction write-set

The transaction write-set holds write operations for a given transaction, write and delete operations. Figure 3.4 depicts its structure. Specifically, it sorts the operations by the corresponding table and key. Each key, will hold a list of columns to be written and deleted by the client in the current transaction.

(a) Client requests the start of a new transaction

(b) Transaction txn reads element y

(c) Transaction txn writes element y

(d) Client requests the commit of txn
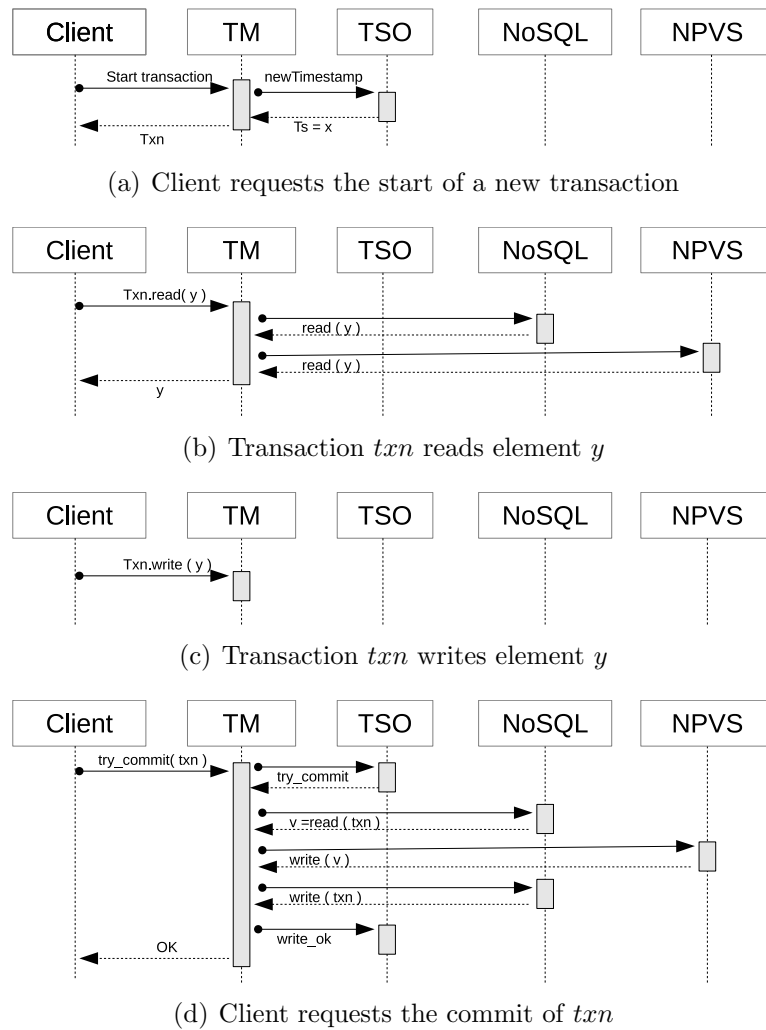
Figure 3.3: pH1 : functionality overview

Each column is built from its name, value and type. This last attribute enables to identify if a given column is to be written or deleted.

Transactional operations

In the context of a transaction, the client will be able to perform the basic CRUD[1] operations. The four operations described below follow algorithms

---

[1]Create, Remove, Update, Delete

Table A	Key x	Column A	Value 1	Write
		Column B	Value 2	Write
		Column B	Value 3	Delete
	Key y	Column A	Value 1	Write
Table B	Key w	Column A	Value 1	Delete
		Column A	Value 2	Write

Figure 3.4: Example of a transaction's write-set

that comply with the isolation level (SI). There is also a fifth operation – the commit operation – that cannot be invoked by the client. This specific operation is only invoked by the transaction manager.

Transactional operations are organized in two groups, the READ group and the WRITE group. The READ group is composed by the Get and Scan Operation. The WRITE group is composed by the Write and Delete operation. While the operations in the first group are immediately executed and their results returned to the client, the operations on the second group will only be persisted if the transaction is successfully committed.

Determination of the latest version of an element

As described earlier, Snapshot Isolation requires that a transaction, when reading an item, reads the most recent version of that element up to the transaction's start timestamp (Ts). For a given transaction, the latest version of an element refers to the most recent version of the element which precedes the transaction's start timestamp (Ts).

Also as described earlier, the different versions of some element may exist on three sources of data: (i) the transaction's write-set, (ii) the NoSQL database and (iii) the NPVS. The order in which structures are accessed is important to determine the latest version of a tuple, according to the isolation

level.

(a)

(b)

Figure 3.5: Selection of the best version for an element to be read

As a result, given the pH1's architecture, there is a specific order in which these structures must be accessed in order to find the latest version of a specific element. Therefore, the latest version will be found in the:

1. The **transaction's write-set**: If it was inserted by the current transaction;

2. The **NoSQL database**: If it was inserted by a transaction that modified the element, and successfully committed before the beginning of the current transaction (i.e the Tc of the latest transaction is lower than the current Ts);

3. The **NPVS**: If it was inserted by a transaction that modified the element in a more distant past.

In order to better understand this process, we will go over an example depicted in Figure 3.5. This Figure shows three different transactions, each

one with its start timestamp (Ts) and also, a view of the data sources (Figure 3.5(b)) (in this example we do not consider the existence of any other concurrent transactions).

Lets consider that transaction named txn 1 performed a write operation for the key composed by [$Table\ A, Key\ x, Column\ A$], that has not yet been committed. Any read operation performed by this transaction over this key will return the value present in its write-set, since it is the latest version available.

Secondly, a transaction txn 2 will perform a read operation for the key [$Table\ B, Key\ y, Column\ B$]. Since the transaction does not hold modifications in its transaction write-set (empty in the Figure 3.5(a)), according to the order defined, the NoSQL database is then checked. The version found in the NoSQL database is approved, because the transaction's start timestamp (Ts=126) is higher than the commit timestamp for the stored value (Tc=123).

Finally, consider that transaction txn 3 wants to read the element composed by the key: [$Table\ A, Key\ x, Column\ A$]. Supposing that no version of the element is present in the transaction's write-set, since no write operation for this table, key and column was performed, the NoSQL database is then checked. The NoSQL holds the desired table, key and column; however, its commit timestamp (Tc=150) is higher than the start timestamp for the current transaction (Ts=143), and thus is not the latest version. Therefore, the NPVS is searched and the version chosen is the one that has the highest timestamp, strictly lower than the start timestamp for this transaction; in this case, the version with commit timestamp Tc=140.

Get operation.

The get operation is invoked by the client in order to read a specific tuple. As described earlier, the implementation of this operation in pH1 has to, accordingly to the NoSQL interface, return the latest version of the element that matches the search criteria, which is built from the table, key and column to be searched.

The search criteria creates a search key that is used to find the element in the different data sources. As Algorithm 1 shows and as described in the previous process, we first check if the search key exists in the transaction's write-set, since if it does, it is the latest version and it is returned to the client with no further search.

However, the read algorithm was devised to handle the faults that may occur in the NoSQL database, as explained in section 2.1.1. If the element to be searched is not present in the transaction's write-set, the algorithm assesses whether there has been a fault in the NoSQL database when trying to perform a write operation regarding a previous transaction (fault A). Briefly, fault A occurs when the NoSQL database detects an error while a write operation is being performed. If there was, both the NoSQL database and the NPVS must be checked.

Next, we have to check if the second possible fault (fault B) occurred. As described in Section 2.1.1, this fault happens if the tuple to be searched was deleted from the NoSQL database. This fault is manifested by the absence of the search key in the NoSQL database (please note that the NoSQL database should hold the most recent version for all elements). If this is the case, the version held by the NPVS is returned if it matches the latest version criteria. Please note that if the algorithm reaches this point, it means that both possible faults occurred.

Algorithm 1: TM: *Get* operation

Data: table, key, column, ts

1 Procedure `DetectFault()`

3   **if** *fault occurred* **then**

4 $FaultDetected = true$

1 $searchKey \leftarrow table + key + column$

2 if $searchKey \; \exists \; WriteSet(write)$ **then**

3   **return** $WriteSet.read(searchKey)$

4 if $FaultDetected = true$ **then**

5 $NoSQLcontent \leftarrow NoSQL.read(table, key, column)$

6 $NPVScontent \leftarrow NPVS.read(table, key, column)$

7   **if** $NoSQLcontent = empty$ **then**

8   **if** $ts < NPVScontent.ts$ **then**

9   **return** $NPVScontent$

10   **if** $ts > NoSQLcontent.tc$ **then**

11   **if** $NoSQLcontent.tc > NPVScontent.tc$ **then**

12   **return** $NoSQLcontent$

13   **else**

14   **return** $NPVScontent$

15   **else**

16   **return** $NPVScontent$

17 else

18 $NoSQLcontent \leftarrow NoSQL.read(table, key, column)$

19   **if** $NoSQLcontent = empty$ **then**

20 $NPVScontent \leftarrow NPVS.read(table, key, column)$

21   **if** $ts < NPVScontent.ts$ **then**

22   **return** $NPVScontent$

23   **if** $ts > NoSQLcontent.tc$ **then**

24   **return** $NoSQLcontent$

25   **else**

26 $NPVScontent \leftarrow NPVS.read(table, key, column)$

27   **return** $NPVScontent$

If fault B did not occur, we still have to ensure that when the compared with the NPVS, the version held by the NoSQL database is the latest. More-

over, if fault A was not detected, that does not exempt the existence of fault B. Thus, the NPVS has to be checked to assess if it has the best version for the tuple.

In the absence of both faults, the NPVS is only searched if the version held by the NoSQL database does not comply with the latest version criteria.

Scan operation.

The scan operation is very similar to the Get operation, and was also devised to cope with the faults explained. The main difference lies on the fact that the search is made for a range of keys/elements, instead of only searching for a single one.

Write operation.

As intended by the client, the write operation inserts an element in the storage layer. In pH1, this operation will have to modify the three data sources. Before the transaction tries to commit its changes, the elements to be written will be placed in the transaction's write-set.

The element is a composition of a column plus its corresponding value that will exist in the context of a given table and key. A single key may hold several columns and their corresponding values.

Algorithm 2: TM: *Write* operation

Data: table, key, column, value
1 **if** $table, key \notin WriteSet$ **then**
2 $\quad\lfloor\ WriteSet.insert(table, key)$

3 $searchKey \leftarrow WriteSet.read(table, key)$
$searchKey.insert(column, value)$

As Algorithm 2 shows, upon the start of the write operation, the trans-

action's write-set is searched to check the existence of the searched table and key. If both exist, the key is retrieved and a new column with matching column, name and value is added. If both the table or key do not exist prior to this operation, they will be initiated and only after the new column is added.

Delete operation.

When the client tries to delete a tuple from the database, the delete operation is invoked. Similarly to the write operation, the delete operation will be found in the transaction's write-set until the commit of the transaction.

Algorithm 3: TM: *Delete* operation

Data: table, key, column
1 **if** $table, key \notin WriteSet$ **then**
2     $WriteSet.insert(table, key)$
3 $searchKey \leftarrow WriteSet.read(table, key)$
4 $searchKey.remove(column)$

As described in Algorithm 3, when this operation begins, the transaction's write-set is checked in order to find a matching table and key, creating both if they are not present. For the given table and key, the desired column is added to the transaction's write-set to be deleted.

Commit operation.

After the certifier module decides that a given transaction can commit its changes, the commit operation is executed for that specific transaction. This operation will be responsible to modify the data sources according to the changes present in the transaction's write-set.

However, before persisting its changes, the elements existing in the transaction's write-set will be put under a conciliation process. This process ensures that one operation is not invalidated by another. As an example, lets again consider Figure 3.4. Notice that, $[Table\ A, Key\ x, Column\ B]$ has a write operation, followed by a delete operation that supersedes the write operation.

The conciliation process will go through all the operations that exist in a transaction's write-set, resolving similar situations. This process ensures that only the needed operations are actually executed in the data sources.

Algorithm 4: TM: *Commit* operation

  Data: $WriteSet, CommitTimestamp(Tc)$

1 Procedure ConciliationProcess()

1 foreach $table \in WriteSet$ **do**

2 $ColumnsToWrite \leftarrow WriteSet(ColumnsToWrite)$

3 $ColumnsToRemove \leftarrow WriteSet(ColumnsToRemove)$

4 $Data \leftarrow NoSQL.get(table, ColunmnsToWrite)$

5   **foreach** $element \in ColumnsToWrite$ **do**

6   **if** $element \in Data$ **then**

7 $NPVS.write(element, Tc)$

8 $NoSQL.batch(ColumnsToWrite, ColumnsToRemove, Tc)$

9 $notifyTSO()$

When the conciliation process ends, as Algorithm 4 shows, the NoSQL database is checked to verify if the keys to be written exist. If they do, the retrieved keys are written into the NPVS (i.e. previous versions of the elements). The details of this process will be described later in Section 3.2.2.

Following, the tuples to be written and/or deleted are sent to the NoSQL database in a batch call (all at once). If this procedure succeeds, the TSO is notified in order to allow it to generate new timestamps, since during the time that a transaction is being certified, the TSO is unable of generating

new timestamps.

3.2.2 NPVS : Non-Persistent Version Store

As detailed in section 2.2.2, the *Snapshot Isolation* level requires the maintenance of versioned tuples (MVCC). In order to support a wider range of NoSQL databases, specially those that do not natively support multi-version of tuples, we introduced the Non-Persistent Version Store in pH1 to manage and store tuple versions.

Distributed cache services.

First of all, there are already a great number of cache systems available that could have been used in this project. Among others, we looked at EhCache [21], provided by Terracotta. EhCache builds a distributed cache system over a cluster formed by several servers, denominated *Server Array*. Within this array it is possible to build strong consistency guarantees when inserting data across servers. This system uses an abstraction called *cache manager* that would allow a client to read and insert elements; placing them across three different tiers of data: memory, off-heap and disk. After an insertion, the system would then move elements across these tiers based on a element usage metric, keeping frequently used elements in memory and moving the least used to the off-heap and disk.

In order to assess the usability of this system, we deployed a small test environment to try to understand if the EhCache offered the performance needed. This test was performed by measuring the amount of insert operations per second in a strong consistency setup.

This preliminary test led us to conclude that the approximate four hundred operations per second was too short. This along with the disk per-

sistence scheme and the need to employ a commercial license in order to eliminate a limitation on cache size kept us away from using this system. Therefore, we developed the NPVS.

Architecture.

The architecture of the *Non Persistent Version Store* spreads along a group of individual and equal nodes, building a hommogeneous distributed repository system. Each *NPVS* node is able to receive and process client requests.

The individual nodes connect through a channel established by a group communication toolkit called *JGroups* [7]. 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.

As data is inserted, it is not partitioned across the cache nodes. Instead, data received in one node is replicated to every other nodes, allowing each of them to be able to answer a read request by a client, relying only in its local data. The client of a NPVS node will be one *transaction manager* instance.

To enable the communication between nodes, the *NPVS* system defines three types of messages that will be sent through the established channel. A description of such messages follows:

- ***Write* message**: This message will hold the elements to be replicated across all nodes.

- ***Ack* message**: This message will acknowledge the reception of a write message to the sender node.

- ***Eviction* message**: This message is used whenever a node received an eviction call from its client and wants to diffuse that information to the remainder nodes.

Data model.

The data model is very alike to the one used in the transaction's write-set, described in section 3.2.1, except for the addition of a field for a second timestamp.

Table A	Key x	Column A	Value: 1	Tc = 134	Ts = 129
		Column A	Value: 2	Tc = 119	Ts = 110
		Column A	Value: 0	Tc = 101	Ts = 099
Table B	Key y	Column B	Value: 'a'	Tc = 109	Ts = 100
		Column B	Value: 'a'	Tc = 100	Ts = 095

Figure 3.6: The view of the NPVS data model

As we can see in Figure 3.6, the data model will hold several tuple versions for a given key. Like in section 3.2.1, each column is sorted by the respective table and key.

The first timestamp (Tc) will keep the time at which the transaction committed. The second timestamp (Ts) will store the start timestamp for the transaction that wrote this element. The need for this second timestamp comes from the Get operation in order to deal with the existence of fault B, as explained in Section 3.2.1. Recall that, fault B occurs when a tuple is removed from the NoSQL database by a concurrent transaction with a commit timestamp (Tc) higher than the current transaction's start timestamp (Ts). When a transaction tries to get the key that was concurrently deleted, the

get operation will not find valid version in the NoSQL database and will try to verify if the NPVS holds a version for that key. Therefore, if a version for key actually exists in the NPVS, it can only be used if it results from a transaction that concurrently was able to successfully commit during the lifetime of the first transaction. Thus, the NPVS needs to store a second timestamp (Ts) to allow the transaction performing the Get operation to ensure that, in the presence of fault B, the version in the NPVS relates to a concurrent transaction, verifying if the version in the NPVS (Ts) is higher than the transaction's start timestamp ($NPVS.Ts > Txn.Ts$).

Write operation.

A write request may arrive at any node that builds the NPVS system. The node that receives the request will store it locally and will forward it to the other nodes through the channel established by the group communication toolkit. Algorithm 5 shows this operations in detail.

Algorithm 5: NPVS: *Write* operation

Data:
table, key, column, value, CommitTimestamp (Tc), StartTimestamp (Ts), channel

1 **if** $table, key, column \notin npvs$ **then**
2 $npvs.insert(table, key, column)$

3 $npvs.get(table, key, column).InsertNewVersion(value, Tc, Ts)$
    $msg \leftarrow WriteMessage(table, key, column, value, Tc, Ts)$
4 $channel.send(msg)$

If the key to be inserted was not previously present in the NPVS, a new key is created and the value is inserted along with the respective timestamps associated with the element. If there are already older versions, the new version is added to the specific key.

After the element is stored locally in the node, a write message is multi-casted through the channel, containing the element to be replicated.

Algorithm 6: NPVS: Handle the reception of a *Write* message

Data: channel

1   $msg \leftarrow channel.GetWriteMessage()$
2   $obj \leftarrow msg.GetPayload()$
3   **if** $obj \notin NPVS$ **then**
4     $\lfloor$ $NPVS.insert(obj)$
5   $NPVS.get(obj).InsertNewVersion(obj.value, obj.Tc, obj.Ts)$
6   $msg \leftarrow AckMessage(obj)$
7   $channel.send(msg)$

The nodes that received this write message will store the containing element and, they will acknowledge the reception by returning a message, as in Algorithm 6.

Get operation.

The replication strategy used allows every node to be able to serve any read request. The read request, as Algorithm 7 shows, will verify if the corresponding table, key and column exists.

Algorithm 7: NPVS: *Get* operation

Data: table, key, column, StartTimestamp (Ts)

1   **if** $table, key, column \in NPVS$ **then**
2     $\lfloor$ **return** $NPVS.get(table, key, column, Ts)$

If it does, the version to be returned follows the criteria for choosing the best version, as explained in 3.2.1, returning the version strictly lower than the read timestamp (Ts).

Eviction.

As transactions successfully commit or abort, the need to store previous versions for elements modified by a transaction disappears. This happens because, accordingly to what has been explained in Section 3.2.1, at any given time, the most up-to-date version of any element, for all committed transactions, is stored in the NoSQL database.

Periodically, the *transaction manager* will verify the start timestamp for the older active transaction and will send it to the NPVS. Elements with versions prior to the received timestamp may be removed since their corresponding transactions are already finished. So, when the NPVS receives this information, the receiving node multicasts this information to the remainder nodes, and each node runs through every key, removing all versions prior to the one sent by the *transaction manager*.

3.2.3   Certification authority

After the set of operations for a given transaction is finished, the client will try to commit the transaction in the transaction manager, which in turn will contact the certifier module, which based on the transaction's write-set will decide whether the transaction should commit or abort.

The certifier module used in pH1 was developed by Yahoo! [23] in the context of the OMID project [22]. 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[2] and currently, this project only supports this NoSQL implementation.

The OMID project uses a *Snapshot Isolation* compliant certifier, that does not use mutual exclusion primitives to enable the execution of con-

current 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.

The name of the certifier module the certifier module of OMID can be misleading, *The Status Oracle* (TSO), but it is actually responsible for two main functions: (i) generating new timestamps and (ii) certifying transactions. The implementation of this module acts as a server, replying to start and commit requests.

Client interface.

The *Timestamp Oracle* modules relies on a basic interface that offers two methods:

- **start transaction** *:* This function is executed whenever the client (in the case of pH1 the *transaction manager* module) needs a new start timestamp (Ts) for a new transaction.

- **commit***:* This function is executed whenever the client wants to try commit the transaction.

Timestamp oracle.

As described previously, as new transactions are started or committed, we need to provide them with timestamps. These timestamps will be associated with the begin and end of transactions.

The generation of timestamps must ensure that each timestamp is unique and, thus they are totally ordered. Specifically, the need for total order, as described in Section 2.2.3, is key to the correct behaviour of the system.

Since, this ensures that timestamp B is greater than timestamp A if A preceds B ($A \rightarrow B$).

Certifier.

This module relies only on the operations present in the transaction's write-set that is to be committed.

After receiving the modifications produced by the transaction, the certifier will verify if there are no other concurrent transactions that modify the same elements. The *timestamp oracle* is capable of such decision, once it holds information concerning all previously committed transactions, and can therefore assess if there is a temporal overlap for the elements within a transaction.

Once the certification process is successful, the transaction can be committed. The certifier will then produce a new timestamp that will be used by the *transaction manger* as the commit timestamp for the transaction. Upon issuing a commit timestamp, the certifier will wait to be informed that the modifications succeeded, and only then will it be able to provide new timestamps. On the contrary, if the transaction conflicts with another previously committed and concurrent transactions, the certifier orders the transaction to abort and discard its modifications.

Chapter 4

Experimental evaluation

Along this chapter, we aim to characterize the performance of pH1 by using two benchmark systems regarding two different metrics: overall throughput and latency. Therefore, we set up two scenarios that consisted in (i) testing the NoSQL database without transactional guarantees and (ii) adding the pH1 middleware layer to the underlying NoSQL database. Then we established a comparison among these settings in order to measure the cost of adding transactional guarantees.

We have used two different benchmark systems to assess the system. On the one hand, we used the Yahoo Cloud Service Benchmark (YCSB) [10] popularly used to test NoSQL databases. On the other hand, we used an optimized implementation of the TPC-C benchmark [11] that mimics a real-case scenario and it is widely used to benchmark relational databases.

Along this chapter, Section 4.1 explains the experimental setting and results regarding the YCSB benchmark. Later, Section 4.2 follows the same structure for the TPC-C benchmark.

4.1   YCSB

The Yahoo Cloud Service Benchmark was described in [10] as a new benchmark that would allow the comparison among data stores designed for the cloud computing paradigm. In other words, Yahoo! [23] 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 accordingly to a pre-defined workload. The type of operations available include Read, Write, 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 operation comprised of ten single update operations.

The YCSB benchmark system allows for a certain level of configurability. Among other customizations, the benchmark system allows to choose the desired distribution of data to be used. The assortment of possible distributions goes from uniform and zipfian, to a completely custom distribution. The decision for the right data distribution along with the correct workload configuration allows to simulate a real world scenario.

4.1.1   Experimental setting

We used five identical machines comprised of a Intel i3-2100 3.1GHz 64 bit processor, 4GB of RAM and 7200 rpm SATA 2 drives. The machines were

interconnected by a switched Gigabit local area network. All the machines used Ubuntu 12.04 as its operative system.

As Figure 4.1 shows, we deployed a Cassandra cluster comprised of two nodes. The timestamp oracle and certifier (TSO) was deployed in a different and single machine. Each one of the two remainder machines ran the YCSB client instance co-located with a pH1 instance.
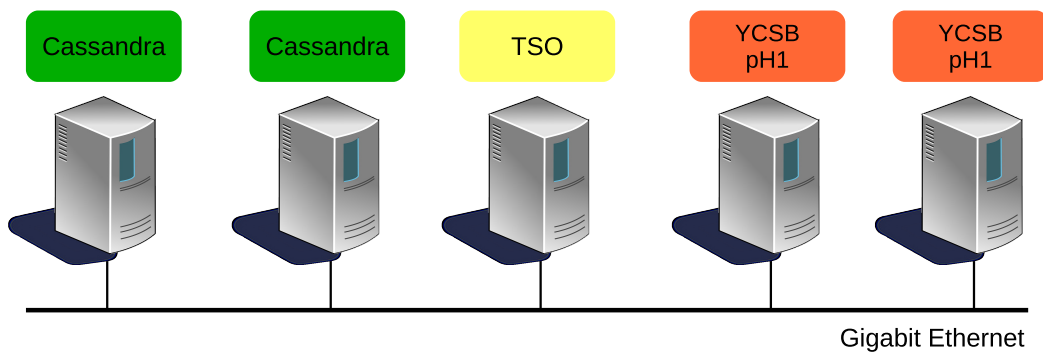

Figure 4.1: YCSB: test environment description

Load phase.

This benchmark is divided in two phases. The first one, the load phase is aimed in pre-loading one million entries into the database. Each entry was composed of ten fields, each one loaded with random content, at about 1KB in size per field.

This insertion populated the Cassandra cluster with about 1GB of data. The Cassandra cluster was configured to replicate the data among the two nodes (replication factor of 2) and was configured to use a random partitioning strategy.

Benchmark execution phase.

The second phase consists in the execution of the benchmark. We have configured the benchmark with 25 concurrent clients in each test machine, for a total of 50 clients. The workload was composed of 45% Reads, 12.5% Updates, 12.5% Multi Updates and 30% Scans.

We have configured the benchmark to use a uniform distribution of requests. In this configuration, keys are chosen at random, but keys are drawn from the keyspace with equal probability.

The benchmark ran until the completion of a total of 450000 operations, in the two different scenarios. For the first one, we ran the benchmark against Cassandra, without the pH1 middleware layer, thus without transactional guarantees. For the second one, we repeated the test, introducing the transactional guarantees offered by the pH1 middleware layer. In this second configuration, each operation corresponds to a transaction. With the absence of transactions composed by several operations (except for the multi update operation), we tried to emulate the same behaviour as the "*auto commit*" function in the ODBC [1] driver.

4.1.2   Throughput analysis

The results presented in this section come from the sum of the two instances running the YCSB benchmark, representing the total of 50 concurrent clients, and are the average of 5 independent runs.

As Figure 4.2 clearly depicts, and as expected, the throughput of the pH1 middleware layer is lower than Cassandra's without any transactional guarantees. After the initial ramp up time of 40 seconds, the throughput remains stable for the rest of time the test lasted (approximately 6 minutes).
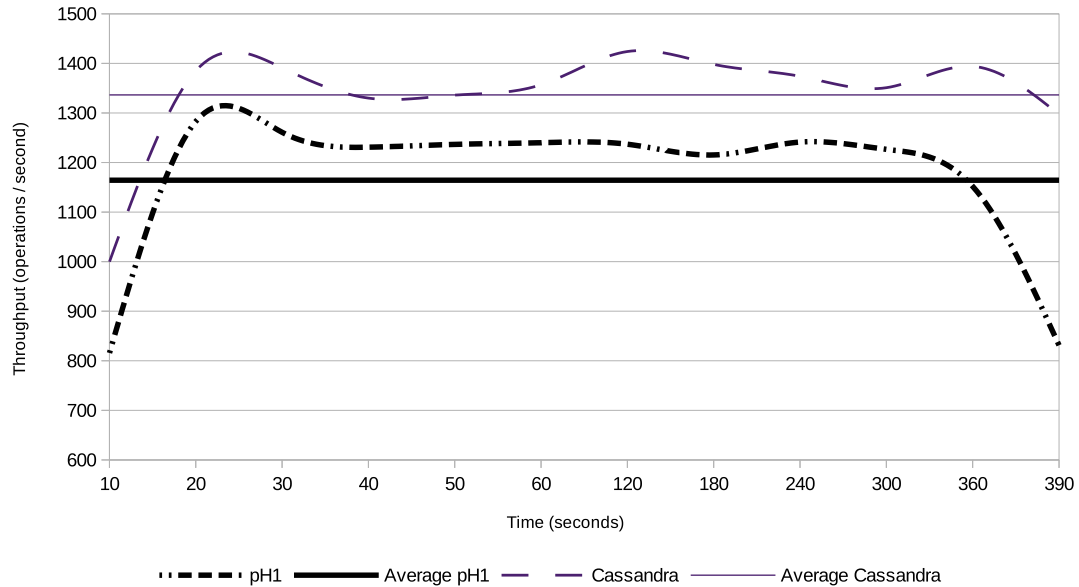
Figure 4.2: Throughput results for YCSB benchmark

It is also possible to establish a relationship between Cassandra's and pH1's throughput evolution, which indicates that the throughput for pH1 is closely tied to the underlying NoSQL database.

The introduction of the pH1 middleware layer only incurred in a 13% loss in throughput, with a performance of 87% of the initial throughput registered by Cassandra.

4.1.3 Latency analysis

To what concerns latency, Figure 4.3 exhibits three different behaviours. Read only operations like the `read` and `scan` operations show larger latencies for the pH1, when compared with Cassandra. Surrounding each of these operations, there is the extra cost of transaction initialization, which justifies the difference. In what regards the `update` and `multi update` operations, as

they will modify the data sources, the addition of the transactional context adds the penalty of transaction certification. A single update transaction in the pH1 will be 3.5 times slower when compared with Cassandra due to that fact. The `multi update` transaction has however better performance. Although executed in a transactional context, the `multi update` transaction is 2 times faster when compared with Cassandra; mainly by the fact that each singular operation in it is executed in batch during the commit phase of the transaction.
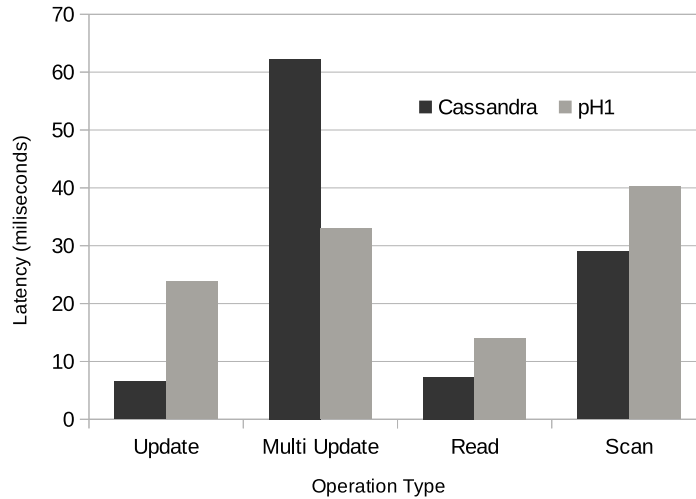

Figure 4.3: Latency results for YCSB benchmark

Figure 4.4 presents the histogram of completion time for the several operations used. From there we can verify that the peak time for scan operations distances from the remainder operations. The off sync observed is in this case justified by the underlying NoSQL database. As described in section 2.1.1, Cassandra may use one of many partitioners. As explained in the load phase description, the random partitioner was used as it better balances data across the Cassandra cluster. The fact that this partitioner does not order

keys by it's lexicographical value, but instead by its hash value, harms ranged operations like scans. Scan operations suffer from this problem since the keys present in a range may not be present in the same node, but scattered along several of them. Therefore, during a scan operation, multiple nodes have to be contacted which introduces an extra hop in communication, affecting latency.

Figure 4.4 also allows us to better understand the distribution of latencies per operation type. From there, we can extract that the average time for a Update, Multi Update, Read and Scan operation is respectively: 13 to 14 milliseconds, 17 to 18 milliseconds, 7 to 8 milliseconds and 27 to 28 milliseconds.
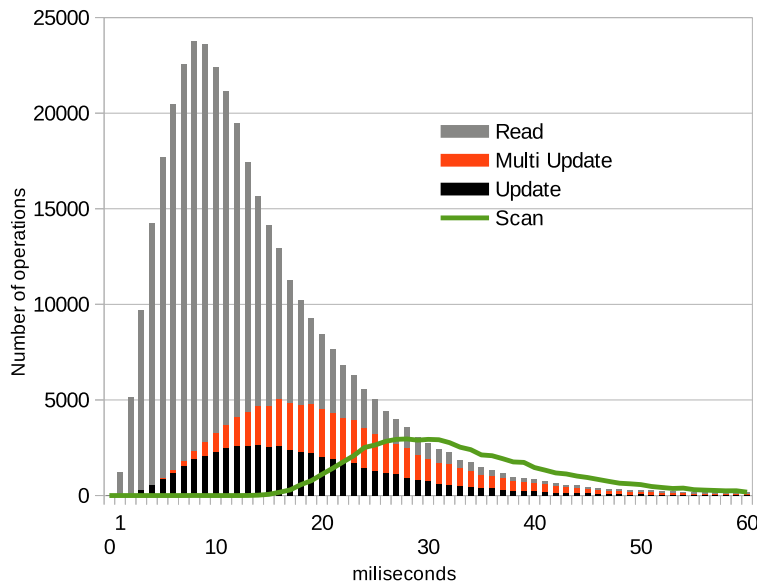

Figure 4.4: Latency histogram for pH1

A more precise analysis of the latency on each operation type, allowed us to verify that during the commit phase of a transaction, the average cost of performing a write operation in the NPVS was about 2 milliseconds per

transaction. This value represented in average 8.4% of spent time in a `Update` transaction.

This along with other results are depicted in Table 4.1, where we can verify the mean time on several stages for a transaction.

Characteristic	Mean Time (ms)
Certification	1.18
Successful commit	8.32
NPVS Write Operation	1.9

Table 4.1: Mean time of several moments in the lifetime of a transaction

The time necessary to wait for a reply on the certification of a transaction was on average 1.18 milliseconds per transaction. If the transaction could successfully commit, then it would take another 8.32 milliseconds to flush the modifications to the data sources as explained for the commit operation in section 3.2.1.

Associated with these results there is a transaction success rate of 91.43%. This means that we registered only 8.57% of aborted transactions.

The main conclusion that we can draw from this last test is that indeed there was an increase in latency for all the tested operations except for the Multi Update operation where there was a reduction of 50% of spent time in this operation.

4.2 TPC-C

In order to verify the versatility of our middleware layer, we used a significantly different benchmark system. For this purpose we used PyTPCC[1], an optimized implementation of the OLTP standard benchmark TPC-C.

This benchmark 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 benchmark 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 benchmark is measured in "tpmCs" or "transactions per minute of New-Order" transactions.

4.2.1 Experimental setting

The experimental setting for this test was deployed in a distributed scenario composed of 10 independent and equal machines with specification as described in 4.1.1. As depicted in Figure 4.5, the Cassandra cluster was deployed in 6 of these 10 machines, while one machine held The Status Oracle (TSO).

The PyTPCC clients ran in the remainder 3 machines in total of 300 clients (100 clients threads per machine and 10 clients per warehouse), each one co-located with a pH1 instance, in test runs that lasted for 45 minutes.

As in the the previous evaluation, we deployed this benchmark using

---

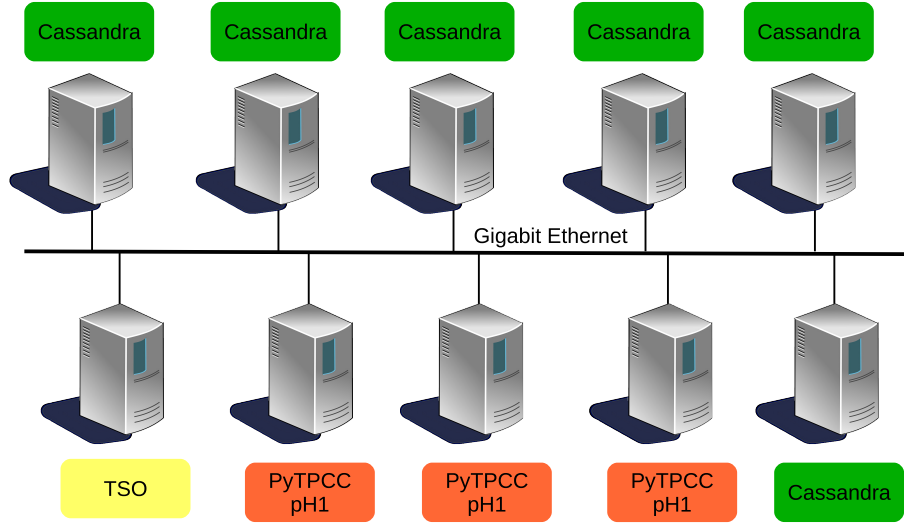[1]https://github.com/apavlo/py-tpcc/tree/master/pytpcc

Figure 4.5: Test setup for the TPC-C evaluation

two different scenarios. Firstly, we ran the benchmark over Cassandra without the pH1 middleware. Secondly, we ran the benchmark using the pH1 middleware layer, thus introducing transactional guarantees.

4.2.2   Throughput analysis

Results regarding the scenarios described in the previous section are depicted in Table 4.2 and they represent the average results registered for the machines that hosted the PyTPCC clients, in the the average of 5 independent tests.

Configuration setup	Throughput (tpmC)
(i) Cassandra	12960
(ii) Cassandra + pH1	7560

Table 4.2: PyTPCC results on average

Since the results for pH1 are bounded to the underlying database, the

first scenario, (i) Cassandra, represents the maximum throughput that the pH1 could possibly achieve for this configuration. The results achieved are consistent with the ones presented for the YCSB benchmark, as they show a throughput decrease when the pH1 middleware layer is introduced. In detail, the results achieved by this second experiment (ii) incur in a 41% throughput loss.

As expected, there is a significant throughput penalty when the pH1 is introduced for the TPC-C benchmark. In comparison with the YCSB benchmark where only 12.5% of all transactions actually had composed operations (Multi Update), in TPC-C every transaction was built from of composed transactions, performing scan and write operations accordingly to the specified workload. Furthermore, the existence in some cases of several range scan operations in almost all of the transactions performed by TPC-C, justifies the bigger throughput penalty observed in comparison with YCSB, where only 30% of all executed transactions where single range scan operations. This second statement is specially relevant as range scan operations topically have smaller performance when executed over datasets scattered along several data partitions, as it happens in Cassandra. Despite of that, as a result, we can state that the pH1 is very versatile and can cope with both read-only and write-heavy workloads with much reasonable results.

Chapter 5

Conclusion

This thesis 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.

The prototype was built based in three different modules. Two of them are part of the presented contribution and the third, the certifier/timestamp oracle was reused from the OMID[22] project.

We tested our prototype on top of Apache Cassandra and ran a group of tests using two different benchmarks in order to evaluate the overall cost of adding transactional guarantees. In a nutshell, when using the YCSB benchmark, a read-intensive workload, throughput decreased by 13%, while with TPC-C, a fully transactional and write-intensive workload the impact went

up to 41%. While we do not have a similar system to compare these results against, we believe they provide an accurate measure of the impact of offering the transactional programming abstract with strict ACID guarantees.

5.1   Future work

One of the main goals for this project was to create a middleware that would be able to be applied to any NoSQL database, even the ones that do not offer support for versioned tuples.

Our prototype was implemented over Cassandra as a prototype, however it is our desire to test the pH1 middleware layer against other NoSQL databases. As a result, we would like to verify that (i) our contribution is able to offer the same set of guarantees and (ii) the relative cost of offering these guarantees stays in the same order as the one achieved for Cassandra.

Finally, in the beginning of this project, there was a goal to include a query engine as the client for our middleware, creating a transactional system with SQL interpretation capabilities. Therefore, since this project strayed from this initial objective, the integration of such component in our contribution remains as a future goal. This will allow us to: (i) get a quantitative comparison between our contribution and relational systems and (ii) infer on the relative cost of adopting our contribution.

Bibliography

[1] The linux/unix odbc. http://www.easysoft.com/developer/interfaces/odbc/linux.html.

[2] Apache. Hbase. http://hbase.apache.org.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[4] Jason Baker, Chris Bondç, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean M. Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *In Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, January 2011.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2), May 1995.

[6] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. 1986.

[7] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In R. Meersman and Z. Tari, editors, *On The Move To Meaningful Internet Systems, International Symposium on Distributed Objects, Middleware, and Applications (DOA)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1485–1502, 2006.

[8] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC'10*, 2010.

[11] Transaction Processing Performance Council. *TPC Benchmark C.* 2010.

[12] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS*

symposium on Operating systems principles, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[14] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[15] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[16] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[17] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43:12–14, 2010.

[18] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[19] Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. Precisely serializable snapshot isolation.

[20] M Tamer èOzsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011.

[21] Terracotta. Distributed and highly scalable cache. http://ehcache.
     org/.

[22] Yahoo! Omid. https://github.com/yahoo/omid/wiki.

[23] Yahoo! Yahoo! http://www.yahoo.com.

[24] Wei Zhou, Guillaume Pierre, and Chi-Hung Chi. Cloudtps: Scalable
     transactions for web applications in the cloud. *IEEE Transactions on
     Services Computing*, 99(PrePrints), 2011.