Aalto University
School of Science
Degree Programme of Computer Science and Engineering

André Medeiros

HAcid: A lightweight transaction system for HBase

Master's Thesis
Espoo, September 24, 2012

Supervisor: Professor Keijo Heljanko
Instructor: D.Sc. (Tech.) André Schumacher

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	André Medeiros		
Title:	HAcid: A lightweight transaction system for HBase		
Date:	September 24, 2012	**Pages:**	77
Professorship:	Theoretical Computer Science	**Code:**	T-79
Supervisor:	Professor Keijo Heljanko		
Instructor:	D.Sc. (Tech.) André Schumacher		

The scalability of a database is an important issue for applications that deal with large amounts of data, such as web services. The presence of rapidly increasing high-volume data sets is a phenomenon commonly known as Big Data. As an alternative to traditional relational databases, the so-called NoSQL distributed databases have proved to be robust in Big Data applications.

Most NoSQL databases, such as Bigtable and HBase, do not depend on high-end hardware, but are designed to easily scale by distributing the workload to a set of servers with conventional hardware. Cloud Computing infrastructures are suitable for these databases. HBase focuses on offering scalability and thus does not provide transactions with ACID (atomicity, consistency, isolation, durability) properties.

Recently, however, there have been many attempts towards supporting ACID transactions in these databases. One important application of this feature is the support for incremental updates to a data repository, such as a web search index. Most of the existing transactional systems for HBase are built on top of HBase itself, with transactional metadata in the database and algorithms in the client-side.

We have built HAcid, a new open-source transactional system for HBase. As most similar existing systems, it is a client library that keeps transactional metadata in HBase to avoid introducing new server-side software. The novelty of HAcid is its lightweight characteristics: it uses minimal bookkeeping resources and is straightforward to install. The purpose is not to minimize transaction latency, but to provide an easy approach to ACID transactions in HBase.

Keywords:	Distributed database, Transaction, Cloud Computing, HBase
Language:	English

Acknowledgements

I am grateful to my supervisor Keijo Heljanko and to my instructor André Schumacher for helpful comments and guidance. I thank Aalto University for the opportunity of using the Triton cluster for this Thesis. This work was supported by the Data to Intelligence and Cloud Software programs of Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT (TIVIT), Tekes, and Academy of Finland (#139402).

This Thesis is dedicated to my two fathers, the Lord God and Elyas Medeiros, to whom I owe everything.

Espoo, September 24, 2012

André Medeiros

Abbreviations and Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application programming interface
CAP Theorem	Consistency, availability, and partition tolerance theorem
DBMS	Database management system
FCW	First-Committer-Wins rule
GFS	Google File System
HDFS	Hadoop Distributed File System
MVCC	Multiversion Concurrency Control
NIST	National Institute of Standards and Technology
OCC	Optimistic Concurrency Control
RDBMS	Relational database management system
SI	Snapshot Isolation
SQL	Structured Query Language
WAL	Write-Ahead Log
WSI	Write-Snapshot Isolation

Contents

Chapter 1

Introduction

Cloud Computing has changed the business of web applications. Companies building applications delivered as services over the Internet no longer need to setup their own expensive datacenters in order to deploy their service. The obstacle of initiating and maintaining new services has been drastically reduced with the help of Cloud Providers, companies that offer scalable computing resources on demand, with a pay-as-you-go billing system.

In the IT industry, the hardware layer of large-scale Internet applications incurs severe costs of hardware acquisition, maintenance, and scalability improvements. In particular, the overhead of establishing a new datacenter is enormous, and was often a barrier for Internet service companies to start a new service. Cloud Providers are companies that encapsulate the computing power of datacenters and sell this as an on-demand service. A *Cloud* is a networked pool of datacenter hardware and software that is shared among many users [1]. Cloud Providers sell computing resources (storage, servers, processing, applications) of their Clouds to Cloud Users, typically companies building web applications. Cloud Computing is defined by NIST [42] as a model of services that enable access to a shared pool of computing resources, composed of five essential characteristics: (i) On-demand self-service, (ii) Broad network access, (iii) Resource pooling, (iv) Rapid elasticity, (v) Measured service; three service models: (i) Software as a Service (SaaS), (ii) Platform as a Service (PaaS), (iii) Infrastructure as a Service (IaaS); and four deployment models: (i) Private cloud, (ii) Community cloud, (iii) Public cloud, (iv) Hybrid cloud. We explain most of these terms, but some of them (e.g. deployment models) are unnecessary for this Thesis.

Typically, traditional datacenters are underutilized (average server utilization from 5% to 20% [45]) since they have to cope with peak loads. Hence, web applications companies that manage their own datacenters have to pay beyond what is actually used. If these companies choose Cloud services, they

pay precisely for those resources that are used, and not for underutilized resources. Coping with peak demands is done by requesting more resources from the Cloud Provider. This type of rapid scalability from small scale to large scale and back is called *elasticity*, and is a key feature in Cloud Computing. Elasticity is made possible by the availability of a massive amount of Cloud resources at the Cloud Provider.

The resources being serviced by the Cloud can be of different types. In IaaS, the Provider offers fundamental computing resources where the consumer can deploy and run arbitrary software, such as operating systems and applications. In PaaS, the service is a platform of programming languages, libraries, services, and tools built on datacenter resources where the Cloud User is able to run an application, even though the underlying infrastructure cannot be managed by the user. In SaaS, the Cloud User can use Cloud software running on the Provider's infrastructure.

Two major players of Cloud Computing since its beginning have been Amazon and Google. Amazon has since 2002 been providing its Amazon Web Services, which includes a number of different Cloud resources available on-demand. Google has been building its software infrastructure to work on datacenters of commodity hardware, as opposed to high-end hardware. The strategy behind this is an economy of scale where costs are reduced and overall computing power is maximized. Instead of buying expensive high-end servers, Google has been building their private Clouds on cheaper hardware, cheaper electricity, cheaper computing, and a smaller number of administrators.

The challenge for Google was to develop fault-tolerant elastically scalable software systems to work on commodity hardware in several datacenters, since at large scale hardware failures are inevitable. Their objective was providing Cloud services for their own Internet SaaS products, and Google entered the commercial Cloud Computing business only in 2008 with Google App Engine. Even though Cloud provisioning is not their core business, Google has given major contributions in the field of software for Cloud Computing.

Google has tackled the Big Data problem, which is to handle an ever-increasing large amount of data. In order to process huge amounts of data for creating an index for Web searches, they have created their own architecture for scalable distributed batch processing, namely MapReduce [19] and the underlying Google File System (GFS) [28]. These are software systems designed to work at the scale of thousands of computers in dozens of datacenters around the world. The architecture for these system was published by Google in 2004, even though the system itself remained private to Google.

Not long after the publication of MapReduce and GFS, Hadoop [25] and its distributed file system (HDFS) [49, 55] appeared as an open-source alternative inspired by Google's architecture. Hadoop is an Apache project that was born from improvements to Apache Nutch, a web search engine. Since then, it has become a solid software framework for distributed processing of large data sets, and has been adopted by major companies like Yahoo! and Facebook [8].

The field of distributed systems for Cloud Computing continued to grow, and in 2006 Google published another influential paper, this time introducing Bigtable [14]. This system was presented as a distributed storage system built on GFS to be an alternative to traditional relational databases management systems (RDBMS), since these were not feasible in the scale required for Big Data. For achieving high scalability, the data model in Bigtable is simpler than in relational databases, hence it does not support a query language, join operations, ACID transactions, and other useful features. Nevertheless, it has been proved to work in several Google products such as Google Analytics, Google Finance, Google Docs, and Google Earth.

Apache *HBase* [24, 27] was created in 2007 as an implementation of Bigtable's design [14], and is tightly associated with HDFS. Being an open-source project has contributed for its wide adoption, and today it stands as one of the main databases for in Cloud Computing systems.

HBase, Bigtable, and several other Cloud data stores are included in the class of the so-called *NoSQL* databases [51]. These are distributed storage systems that were originally designed specifically for highly-scalable operation. SQL and relational models often suffered from poor performance issues when dealing with Big Data, hence NoSQL data stores[1] avoid these.

However, NoSQL data stores have been gradually regaining some of those features [35, 44, 48, 52, 53, 54, 59, 60]. One of these features is the support for transactions. The objective of transactions is to protect data from damage made by uncoordinated or unfinished data operations. They are mechanisms in databases that encapsulate a group of data operations (such as read and write) and ensure that these will not be left incomplete, but will rather behave as a single operation. The precise properties that transactions are required to satisfy are known as ACID, and are described in Chapter 2.

There are a few transactional systems for HBase, such as the ones we discuss in Section 2.4. The contribution of this Thesis is a new transactional system called HAcid, created out of the need for an open-source lightweight transactional system.

[1] In general, we shall use "data store" to refer to non-relational databases.

This Thesis is structured as follows. In the next chapter, we present background knowledge of transactions, distributed databases, and data stores such as HBase. In Chapter 3 we describe HAcid thoroughly. Chapter 4 discusses the performance of the proposed system, and Chapter 5 concludes and summarizes the Thesis.

The reader interested in the use of HAcid can skip to Section 3.5, while academic readers might be interested in earlier sections of Chapter 3.

Chapter 2

Background

A *database* is an organized collection of *data items*, which are records of some real world information [31]. This is a broad definition that includes even non-digital databases, such as ancient tablets for recording measurements of resources. Databases have always been important structures for life and society. Nowadays they are mainly digital and can be found in many organizations.

Databases follow some *data model*, which is an abstract structure for describing the arrangement of data items in data structures. For example, the relational model is a popular data model, central to SQL-based databases. Among other things, data models describe how data items should be grouped and associated with each other.

Data items are any data that has a value that can be atomically read and written. An atomic operation cannot be partially executed. Data item values can be numbers, strings of characters, bytes, or even a whole row in a table. Data items possess a key or location to uniquely identify them. Thus, a data item is described by a key-value pair $\langle key, value \rangle$. The values of all data items in the database at any time comprise the *database state* at that moment.

In order to maintain the data model of a database and allow modifications, databases rely on *database management systems* (DBMS). These are software systems that provide an interface that allows clients to build, maintain, and modify a database stored in hardware. Clients are usually applications interfacing with the DBMS.

DBMSs can have many subsystems, each one for managing a different aspect of the database. An important requirement for all DBMSs is that they support *read* and *write* operations. We use the notation "$r[x = v]$" to represent a read operation that reads value v from data item x. Similarly, "$w[x = v]$" represents a write of value v to x. In many cases the value can

be omitted if it is not relevant in the context: $r[x]$, $w[x]$.

One of the main components that many DBMSs support is a transaction manager. Transactions are thoroughly described in the next section.

2.1 Transactions

A *transaction* is a sequence of read and write operations that are conceptually related. The idea is to have a group of operations execute as if it was one single operation. Transactions are specified by the application programmer, who sorts and groups some read and write operations together, and submits them to the transaction manager in the DBMS, for executing the operations.

Typically, the application that issued the transaction first submits to the DBMS its operations to be executed. After that, the application requests a *commit* of the transaction. Through the commit request, the application is informing that the modifications done by the transaction should be made visible to other transactions and be written to persistent storage in the database. The DBMS, then, can give an outcome for the transaction: either *committed* or *aborted*. In the abort case, some problem occurred with the transactions management or in the execution phase, and the modifications were undone. This undoing command is called *rollback* and is performed by the DBMS.

The execution of a transaction can be formally modeled as a totally ordered set T_i, i.e. a sequence, where elements are operations such as read ($r_i[x]$) and write ($w_i[x]$). We assume that no transaction reads or writes the same data item more than once. The following definitions are inspired by Bernstein et al. [6].

Definition 1 (Transaction)**.** A *transaction* is a totally ordered set T_i with ordering relation $<_i$ such that $T_i \subseteq \{r_i[x], w_i[x] : x \text{ is a data item}\}$.

That is, transactions have their read and write operations totally ordered. For example, $r_1[x] <_1 w_1[x]$ means that transaction T_1 read data item x and then wrote to x. We use the notation $T_1 : r_1[x]\ w_1[x]$, commonly found in the literature [3, 57], to completely describe the operations of a transaction and their ordering. The data items of read operations in transaction T_i comprise the *readset* T_i^R and the data items of write operations in T_i are collected in the *writeset* T_i^W.

We extend the definition of a transaction to allow the final operation to be either a commit (c_i) or an abort (a_i).

Definition 2 (Completed transaction)**.** A *completed transaction* is a totally ordered set T_i with ordering relation $<_i$ such that:

1. $T_i \subseteq \{r_i[x], w_i[x] : x \text{ is a data item}\} \cup \{c_i, a_i\}$;
2. $c_i \in T_i \Leftrightarrow a_i \notin T_i$;
3. If $c_i \in T_i$ (or $a_i \in T_i$), then c_i (respectively a_i) is the greatest element in T_i.

The purpose of transactions is to allow a group of operations to have the same (or similar) safety properties as the basic read and write operations are expected to have. That is, there are essential properties that all transactions should guarantee upon execution.

As an example, consider a database for bank accounts and the transfer of an amount of money from one account to another. In order to transfer, for instance, \$100 from Alice to Bob, it is necessary to subtract \$100 from Alice's balance (debiting) and add \$100 to Bob's balance (crediting). The whole process involves four operations: a read and a write in Alice's balance, and a read and a write in Bob's balance. This sequence of four operations should be run in such a way that they appear to be actually a single operation. The bank cannot tolerate a partial execution of this sequence, where some operations fail.

The so-called *ACID* is a traditional set of properties that database transactions are expected to satisfy in order to provide data reliability. These properties define transactions that are protected from concurrency problems and hardware failures. The acronym ACID stands for four properties, which are explained below.

- **Atomicity:** either all operations of a transaction are performed, or none are. If some operation fails, the entire transaction fails and the database state is left unchanged by the transaction.
- **Consistency:** after the transaction is committed, it has brought the database from one valid state to another. A valid state is a database state that satisfies some rules, e.g., data type constraints.
- **Isolation:** no transaction interferes with another concurrent transaction. That is, the intermediate steps of a transaction are invisible to other transactions.
- **Durability:** when a transaction is committed, the modifications to data items are persisted, even in the event of a system failure.

These terms are traditionally defined informally. A technical description of the ACID properties requires more details concerning the database and its data model. There can also be many levels of rigor in these properties. In particular, Isolation normally incorporates many different levels. Some of these levels, such as Serializability, Snapshot Isolation, and Read Committed are relevant to HBase and its transactional systems. These are discussed in Sections 2.1.3 and 2.1.4.

ACID properties can be achieved through different techniques. A traditional technique is the use of locks, giving to a transaction exclusive access to its related data items. Alternatives techniques are Optimistic Concurrency Control and Multi-versioning, where the latter one takes copies of data items to avoid the need for managing locks. These techniques are the subject of the following section.

2.1.1 Concurrency control techniques

Transactions are concurrent if they are being executed simultaneously. In practice, concurrency is represented as the interleaving of the operations of transactions and controlled through a component called *scheduler* in the DBMS. The interleaving is particularly important when defining the order of *conflicting operations*.

Definition 3. Two operations are said to be *conflicting* if and only if the following conditions hold: (i) they are read or write operations; (ii) both concern the same data item; (iii) at least one of them is a write operation.

We assume that two conflicting operations are never executed concurrently. As an example, operations $r_2[x]$ and $w_1[x]$ are conflicting, and their ordering is intricate, in the sense that transaction T_2 reading x might get different results depending on when transaction T_1 wrote to x.

Thus, after a collection of transactions is executed, the resulting database state might depend on how the transactions were interleaved. A *history* is an interleaving of transactions, and an *outcome* is the resulting database state, hence the outcome depends on the history. Histories are formally defined as follows.

Definition 4 (History)**.** [6] Given a collection of completed transactions $T = \{T_1, \ldots, T_k\}$, a *history* is a partially ordered set H with ordering relation $<_H$ such that

1. $H = \bigcup_{i=1}^{k} T_i$;
2. $<_H \supseteq \bigcup_{i=1}^{k} <_i$;

3. For every pair of conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.

That is, the ordering relations of the transactions are inherited in the history (condition 2), and all conflicting operations of the involved transactions are ordered (condition 3).

If the history is a totally ordered set, we use the following notation, common in the literature [3, 57]. Let $T = \{T_1, T_2\}$, where $T_1 : r_1[x]\ w_1[y]\ c_1$ and $T_2 : w_2[x]\ w_2[y]\ a_2$, and H be defined as $r_1[x] <_H w_2[x] <_H w_1[y] <_H c_1 <_H w_2[y] <_H a_2$. History H can also be represented as

$$H : r_1[x]\ w_2[x]\ w_1[y]\ c_1\ w_2[y]\ a_2\,.$$

Histories are determined by the scheduler in the DBMS. The scheduler can be implemented in different ways, of which Two Phase Locking (2PL) [4, 6] is the traditional approach. Locks are mechanisms that a transaction enables to deny other transactions access to data items. The purpose is to allow a transaction to acquire exclusive access to some data items so that other transactions do not interfere with its operations.

In this method, each data item is associated with a *lock*, which can be held by only one transaction at a time. The DBMS maintains a lock table, where each data item x is mapped to some transaction T_i (x is locked by T_i) or to no transaction (x is unlocked). Before a transaction T_i accesses a data item x (through read or write operations), it must attempt to acquire a lock from the scheduler. The lock of x is granted to T_i if it is not held by another transaction. If the lock is already taken, then T_i must wait until it is released.

Two Phase Locking happens in two phases: first in a *growing phase*, then in a *shrinking phase*. In the growing phase, a transaction obtains locks for the data items it wants to access, then performs its operations on those data items. In the shrinking phase, a transaction releases the locks it has obtained. No locks are released in the growing phase, and no locks are acquired in the shrinking phase. 2PL is a *pessimistic concurrency control* method, in the sense that it denies access to other transactions that might not necessarily request access to the locked data items.

Locking is interesting for providing good Isolation, since it enforces that transactions cannot interfere with each other. On the other hand, without additional precautions it is common to encounter problems such as *deadlocks*, which happen when a transaction T_i is waiting for T_j to release locks, while the T_j is waiting for T_i to release locks. There are no general-purpose deadlock-free locking protocols for databases that always provide high concurrency [37]. Typically, strict locking schedulers exhibit poor performance with many in-flight transactions.

Another method is Optimistic Concurrency Control (OCC) [37], which can be considered the opposite approach to pessimistic concurrency control. In OCC, a transaction is never blocked from executing its operations, but might get aborted when the commit request happens, in order to avoid violation of properties such as Isolation. This is an optimistic approach, in the sense that transactions assume that violation of Isolation will not happen while executing its operations. When aborted, normally the transaction is re-executed until no Isolation violation occurs on the commit request. Violation of Isolation is also called transaction conflict.

OCC might operate with better performance than 2PL when transaction conflicts are rare. When transaction conflict is highly likely, OCC suffers from starvation due to repeated executions of transactions [37].

2.1.2 Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) [4, 5] is a class of methods that keep a list of versions of each data item. Externally (to the client), x is a data item, but internally x is a map from version numbers (typically integers) to proper data items.

For instance, $x(1)$ is a data item representing the first version of x. Version $x(k)$ may or may not be defined, i.e., x is a partial function. If $x(k)$ is undefined, we write $x(k) = \bot$. To "create" a new version corresponds to defining $x(k)$ if it was previously undefined. In MVCC, when a transaction writes to x, instead of overwriting a value, it actually creates a new version $x(k)$, which is written only once. The MVCC method determines how read operations select the correct version to read. In MVCC methods, we refer to x as a *data item*, and $x(k)$ as a *version* of x.

The benefits of MVCC are increased concurrency (each transaction writes to a different version) and unobstructed read-only transactions, since versions are not modified after being created. The versions of x are internal to the concurrency control method and are not seen directly by the client. The drawback of this method is the storage cost of maintaining multiple versions.

2.1.3 Elementary isolation levels

Isolation is about concurrency. If no concurrency is involved, the DBMS deals with transactions sequentially without overlapping in time. The result is a *serial history* H: a history with no interleaved operations of different transactions, i.e., no operation of T_j can appear (in the $<_H$ order) between two operations of T_i, for all $j \neq i$. In this situation, Isolation is perfectly guaranteed.

However, concurrency for transactions is a basic property that most DBMS should support, since the lack thereof incurs severe performance drawbacks. For example, recall the bank accounts database system. It would be an obvious drawback to lock the whole database for the execution of every transfer. Two unrelated transfers should be allowed to execute in parallel.

Ideally, a DBMS should provide **Serializability**, the highest isolation level. With this property, the DBMS allows concurrency but the outcome of a history is always equivalent to the outcome of some serial history. Thus, to the application interfacing with the DBMS, transactions appear to have been executed serially, i.e., sequentially without overlapping in time.

To illustrate Serializability, recall the example of money transfer from Alice to Bob. Let transactions $T_1 : r_1[A]\ w_1[A]\ c_1$ and $T_2 : r_2[B]\ w_2[B]\ c_2$, where A and B are bank account balances for Alice and Bob. History H of $\{T_1, T_2\}$, defined as

$$H : r_1[A]\ r_2[B]\ w_1[A]\ c_1\ w_2[B]\ c_2$$

is a serializable history, because its outcome is the same as the outcome of the serial history

$$H' : r_1[A]\ w_1[A]\ c_1\ r_2[B]\ w_2[B]\ c_2\,.$$

In other words, the accounts will have seen a correct transfer, regardless whether the history was H or H'.

Despite its usefulness, Serializability is a strict requirement over the outcomes of histories, hence it often causes performance to be poor. It is expensive notably in typical Two Phase Locking implementations, since transactions are often blocked from proceeding in order to satisfy the requirement.

Other isolation levels have therefore been proposed, in order to increase performance. The disadvantage is that these levels allow a number of anomalies that cause inconsistencies in the database. In general, the more anomalies we allow, the more concurrency we gain. On the other hand, less anomalies allowed means better data consistency of the outcomes. Some important anomalies are described below, and were originally specified by ANSI SQL-92 [56], but properly redefined by Berenson et al. [3]. We interpret the definitions of Berenson et al. [3] using the terminology of Bernstein et al. [6].

Definition 5 (Dirty Write Anomaly)**.** Given a collection of completed transactions T and a history H of T, we say that H exhibits *Dirty Write* if there are two transactions $T_1, T_2 \in T$ such that $w_1[x] <_H w_2[x] <_H c_1$ or $w_1[x] <_H w_2[x] <_H a_1$.

Dirty Writes mean that inconsistent values can be written to the data item when a transaction commits or aborts. For example, in the case $w_1[x] <_H w_2[x] <_H a_1$, when T_1 should rollback, it is unclear whether to recover the value previous to operation $w_1[x]$ or the value written by $w_2[x]$. Preventing Dirty Writes is equivalent to making transactions get write locks for their writesets [3].

Definition 6 (Dirty Read Anomaly)**.** Given a collection T and a history H of T, we say that H exhibits *Dirty Read* if there are two transactions T_1 and T_2 such that $w_1[x] <_H r_2[x] <_H c_1$ or $w_1[x] <_H r_2[x] <_H a_1$.

What Dirty Read means is that transaction T_2 reads a data item value that was not yet committed. Moreover, if T_1 aborts, then the value it wrote could have been undesirably read by some other transaction.

Apart from Serializability, two basic isolation levels are Read Uncommitted and Read Committed.

Definition 7 (Read Uncommitted)**.** A transaction manager exhibits *Read Uncommitted* if the histories produced by it never have Dirty Writes.

Definition 8 (Read Committed)**.** A transaction manager exhibits *Read Committed* if the histories produced by it never have Dirty Writes nor Dirty Reads.

These are isolation levels important to mention for this Thesis. They disallow some anomalies, but many other kinds of anomalies can still occur with them. There are other isolation levels as well, which are not mentioned here. However, not all levels are defined according to anomalies, such as the important Snapshot Isolation, which is the subject of the next section.

2.1.4 Snapshot Isolation

Proper Isolation is normally achieved by giving transactions exclusive access to data items, i.e, locking. However, locking is not the only technique, because exclusive access can be given by means of *snapshots*.

Snapshot Isolation (SI) [3, 12, 57] is a term used for denoting a method of concurrency control and the resultant Isolation properties that it provides. In SI, a transaction starts by taking a snapshot of the current state of the database, then executes its reads and writes on that "private" snapshot, and finally commits by making its snapshot persist in the database. The commit is successful only if the writeset of the transaction has not been modified by other transactions since the snapshot was taken. That situation is called a

write-write conflict [57], and happens when two transactions overlap in time and have an intersection in their writesets.

In practice, we do not copy the whole database state to make the snapshot, since this is clearly expensive. It suffices to take a snapshot of the readset and writeset of the transaction, i.e., the data items involved. To simplify the management of snapshots, typical SI implementations employ *timestamps*, which are unique integers served in increasing order by some source, usually called *timestamp oracle*. Timestamps are suitable for implementing SI in multi-version databases, which are databases that have built-in support for multiple versions of data items. In these settings, SI takes advantage of techniques from MVCC.

In SI, transactions proceed as follows. When a transaction T_i starts, it acquires a *start timestamp*, denoted in this Thesis as T_i^{start}. Only after that can T_i execute its operations. When no more read or write operations will be performed, T_i acquires an *end timestamp* T_i^{end}, which in the literature is also known as *commit timestamp*[1]. Finally, conflict testing happens to determine whether T_i commits or aborts. If T_i committed, its end timestamp is used to represent the time T_i was committed. If a transaction still does not have a start or an end timestamp, we write $T_i^{start} = \perp$ or $T_i^{end} = \perp$.

Notice the connection between the order of operations and the timestamps of a transaction. The start timestamp is acquired before any operation is executed, and the end timestamp is acquired after the last read or write operation and before the commit operation c_i (or a_i). Transactions that have timestamps are called *timestamped transactions* in this Thesis. The start and end timestamps of a transaction define its lifetime.

Definition 9 (Lifetime)**.** The *lifetime* of a timestamped transaction T_i is the interval $[T_i^{start}, T_i^{end})$.

During the lifetime of a transaction, read and write operations are executed. To simplify the theory needed, we assume that all read operations must happen before any write operation in a transaction, as [44] does. In practice this is commonplace, and it simplifies our discussion. Each read operation uses the newest committed data item version that was committed before T_i^{start}. A write operation to data item x is performed by creating a new version $x(T_i^{start})$.

The start timestamp, therefore, represents the snapshot for T_i: data is written on version T_i^{start} and read from versions smaller than T_i^{start}. The *snapshot* of a transaction is the set of all data item versions that can be read

[1]We avoid the term, since even transactions that aborted would have a commit timestamp, which might be misleading given that a transaction is either aborted or committed.

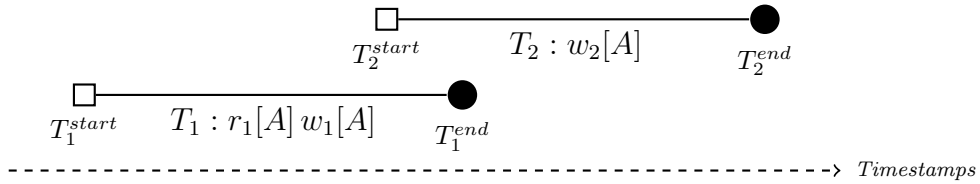

Figure 2.1: A write-write conflict between transactions T_1 and T_2. A line with a square and a filled circle on the ends represents a transaction. A square is a start timestamp and a filled circle is an end timestamp. T_1 conflicts with T_2 because $A \in T_1^W \cap T_2^W$, so T_1 commits and T_2 aborts since $T_1^{end} < T_2^{end}$.

in the transaction. Version numbers are defined by the SI method, and the client application does not need to know what version numbers were used.

When conflict testing happens, the outcome of the transaction is decided: aborted ($a_i \in T_i$) or committed ($c_i \in T_i$). The transaction manager is responsible for checking whether the current transaction conflicts with some other transaction, and deciding which aborts and which commits. In SI, two transactions conflict if they have a write-write conflict, as defined below.

Definition 10 (Concurrent transactions)**.** Two timestamped transactions T_i and T_j are *concurrent* if their respective lifetimes have a non-empty intersection.

Definition 11 (Write-write conflict)**.** Timestamped transactions T_i and T_j have a *write-write conflict* if they are concurrent and $T_i^W \cap T_j^W \neq \emptyset$.

Once two transactions with write-write conflict are detected, one of them must be selected as the "winner" to commit. For resolving this, SI enforces the "First-Committer-Wins" (FCW) rule: the transaction with smallest end timestamp is selected as the winner. The other transaction is aborted. FCW is also useful for preventing an anomaly, namely, Lost Updates, which is explained in details by Berenson et al. [3]. Figure 2.1 is an example of two conflicting transactions, where the FCW rule applies.

A read-only transaction will not interfere with other transactions, it simply needs to find the correct data version that represents its snapshot, and execute the read. In this sense, read-only transactions in timestamp-based SI implementations are never blocked and always committed.

Snapshot Isolation is an interesting isolation level that many databases support. Compared to Serializability in 2PL, it has better performance and avoids many anomalies, including Dirty Reads and Dirty Writes. A key advantage of SI is that read operations are never blocked. On the other hand, there can occur some anomalies in SI histories, the most notorious

being Write Skew. This anomaly happens when a constraint is not satisfied globally (inter-transaction violation), even though SI transactions can satisfy the constraint locally (intra-transaction fulfillment). The following example illustrates an occurrence of Write Skew.

Suppose x and y are integer data items, and let $x+y > 0$ be the constraint we wish to keep satisfied. Let T_1 and T_2 be two completed transactions, $T_1^{start} = 1$, $T_2^{start} = 2$, and H be a history of $\{T_1, T_2\}$ obtained by a Snapshot Isolated transaction manager,

$$H : r_1[x(0) = 5]\ r_1[y(0) = 5]\ r_2[x(0) = 5]\ r_2[y(0) = 5]\ w_1[x(1) = -2]\ w_2[y(2) = -3]\ c_1\ c_2.$$

In words, H has T_1 reading the values of x and y, then writing a negative value to x; T_2 reads the values of x and y, and writes a negative value to y. Transactions T_1 and T_2 do not exhibit a write-write conflict, thus both commit. Locally, the snapshot of T_1 at the timestamp $T_1^{start} = 1$, has the values $x = -2$ and $y = 5$, therefore $x + y > 0$ is satisfied. Similarly, the snapshot of T_2 satisfies the constraint on commit time. However, in a global view of the database state, the constraint is violated. That is, the snapshot of a next transaction T_3 (starting after both T_1 and T_2 committed) will have the values $x = -2$ and $y = -3$, violating $x + y > 0$.

In many applications, Write Skew might not be a problem for data integrity, but for databases such as bank account databases, it is safer to avoid it. Serializability, by definition [3], does not allow any anomaly from a list of eight anomalies, including Write Skew. Therefore, Snapshot Isolation does not guarantee serializable histories [3, 12]. The main reason for Write Skews in SI is because the readsets of transactions are not used during conflict testing in SI, only writesets are used [57].

In transactional systems for Cloud data stores (the focus of this Thesis), Snapshot Isolation is usually the main isolation level supported. For example, the pioneers Percolator [44] and HBaseSI [60] have SI as one of the main properties of their systems. This isolation level is convenient for data stores such as Bigtable and HBase, which are multi-version databases.

2.2 Distributed Databases

In large enterprises, centralized databases were once the traditional approach for storing data. If a database is centralized, all the data is maintained in a single server, such as a specialized database server. The main disadvantages are performance bottlenecks and unavailability during failure recoveries.

In a distributed database, the data is stored in several storage devices across different computers (or "nodes"). The computers are normally inter-

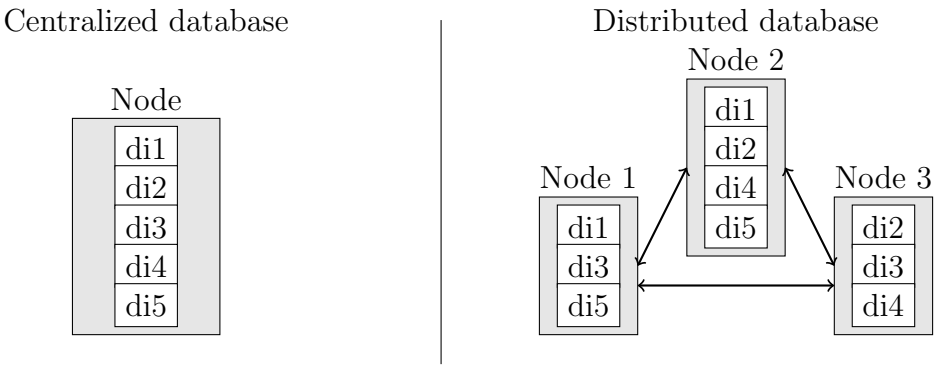

Figure 2.2: An example of centralized and distributed databases. On the right side, five data items are replicated in three different nodes. Edges in the distributed database represent network connection, and "di" stands for "data item".

connected by a network. Data is typically partitioned among the nodes, so that each node holds a portion of the whole database. The DBMS managing the networked nodes provides a global view of the database to the client, even though only local views are given at each node. This property is known as *distribution transparency* [50].

Distributed databases often employ less reliable hardware than centralized databases do. Thus, hardware failures, such as hard disk failures, are common and should be taken into account. *Replication* is an appropriate technique for providing fault-tolerance, by storing copies ("replicas") of every data item in different nodes. Therefore, if one of the replicas is lost through hardware failure of the host computer, there are other replicas in the distributed system. This property is called *redundancy*. The larger the number of replicas employed for each data item, the smaller will be the probability of irreversibly losing that data item. Figure 2.2 compares centralized and distributed databases regarding data item distribution.

Replication can favor availability: the data item is unavailable if a global failure happens, but remains available while at least one replica is accessible. A data item can have replicas in computers spread around the globe, so that clients can access the nearest host node, and latency can be made low in many geographical locations. Availability is a key issue concerning distributed systems.

Another key issue is consistency: the problem of maintaining the same value for all replicas of a data item. When a data item is written, each of its replicas must be updated. Replicas cannot be updated simultaneously, due to the fact that they are distributed. Consequently, the distributed database

will have a temporary inconsistency among the replicas, from the moment the write operation is issued until the last replica is updated.

There are two major categories of methods for updating replicas [50]: *eager* (synchronous) *replication* and *lazy* (asynchronous) *replication*. In synchronous methods, the DBMS makes the data item unavailable to clients while the updates to replicas are being propagated. The data item is available to clients only if all replicas have the same value. Therefore, updates to replicas are grouped in a transaction with ACID properties, which blocks any read operation to the data item.

In asynchronous methods, the data item is kept constantly available, and updates to replicas are propagated independently of each other. Read operations are not blocked, therefore two read operations of the same data item can return different results according to which replica was used for each operation. Eventually, the replicas will converge to the same value.

The distribution of replicas implies a trade-off: making data items temporarily unavailable for guaranteeing consistency, or allowing inconsistency for providing constant data item availability. This is a central consequence of the CAP Theorem, introduced by Brewer [9] and proved by Gilbert and Lynch [29], informally described below.

Consider the three following properties of distributed databases:

- **Consistency**[2]: at any moment all nodes see the same database state.
- **Availability**: every read or write request eventually receives a response whether it was successful or failed.
- **Partition Tolerance**: the system continues to operate despite network partitions, except in case of total network failure. A network is *partitioned* when message losses occur between any two nodes of the system.

The CAP Theorem states that a distributed database can only satisfy at most two of those three conditions. Thus, one of the conditions must be left out. In reality, a system that does not allow partition tolerance is non-distributed (centralized). This is true because the system must prevent partitions from happening, and this is possible only if the database resides in a single node.

Hence, a distributed database must be partition tolerant, so either availability or consistency can be supported, but not both, as stated previously. This result is strong and has affected the design of all distributed databases. Cloud data stores are distributed databases, thus they must also take the

[2]Not to be mistaken with Consistency in ACID.

CAP Theorem into account. The theorem also affects transactional systems for Cloud data stores.

In the following section we focus on the Extensible Record Stores, a class of Cloud data stores pioneered by Google's Bigtable, and the kind of data store we consider for enabling transactions.

2.3 Extensible Record Stores

Bigtable [14] is a proprietary distributed NoSQL database (a Cloud data store) designed to scale to thousands of machines in Google's own data-centers. Google has designed Bigtable out from their need of controlling performance and scalability in data stores. For this reason, the data model is intentionally simple, yet flexible. By choice, transactions were left out of Bigtable's design, since a strict need for them was not found. Nonetheless, many Google products have been successfully developed on top of Bigtable, which indicates its importance.

Bigtable has been the pioneer of a category of data stores named *Extensible Record Stores* [13]. The name indicates that the data model can be easily extended at any moment. This category includes data stores such as Apache HBase [24], Hypertable [34], Apache Accumulo [22], and Apache Cassandra [23].

HBase [27] is likely the most popular Extensible Record Store, and is the data store for which we seek to enable transactional support. We will focus on describing HBase's design, using its terminology, instead of Bigtable's original terminology.

2.3.1 Data model

In HBase, data items are key/value pairs. The keys are multidimensional, with the following dimensions: table, row, column family, column qualifier, and version number[3]. The version numbers are what cause HBase to be a multi-version database. The dimensions "column family" and "column qualifier" are normally grouped together as a pair under the name "column". Typically, the "table" dimension will be clear from the context, hence a key is commonly represented simply as the tuple $(\texttt{row}, \texttt{col}, \texttt{ver})$, where $\texttt{col} =$ (column family, column qualifier) and `ver` stands for the version number.

[3]Version number is more commonly known as timestamp, however we reserve the word "timestamp" for timestamped transactions.

row	personal:name	personal:age	financial:balance	financial:bank
A	123: Adam	123: 34	132: 1630 131: 1690 127: 2120 123: 1911	127: Nordea
B	211: Bob		211: 4124	
C	96: Caroline		125: 851	125: Pohjola 96: OP
S	154: Smith	154: 58		

Figure 2.3: Example HBase table. The notation "100: `data`" means that `data` is stored in version number 100.

An HBase *table* t is a map of data items

$$(\texttt{row}, \texttt{col}, \texttt{ver}) \mapsto \texttt{value}\,,$$

where `col` = `family:qualifier`, `ver` is a 64-bit integer, and `row`, `family`, `qualifier`, `value` are byte strings. Typically, byte strings are decoded as human-readable data, but can be binary data as well.

A *cell* is a set of data items with common `row` and `col` keys, and the *cell key* is precisely (`row`, `col`). Each data item in a cell is called a *version* of that cell. Sometimes we refer to a *row* as the collection of cells with common row key.

A table is organized by grouping cells into rows, and sorting rows lexicographically by their row keys. The sorting allows the table to be partitioned into *regions*, which are different ranges of row keys. The regions of a table are distributed between different nodes, so rows that are lexicographically close will likely be stored spatially near to each other, thus it is wise to choose row keys that provide good locality for data accesses.

As indicated by the example in Figure 2.3, HBase tables are sparse in the sense that null values are not stored. For cells that have at least one version, the latest version corresponds to the data item with highest version number. Thus, the cell (`C`, `financial:bank`) has "125: `Pohjola`" as the latest version, where `financial` is a column family and `bank` is a column qualifier.

HBase's API provides simple operations for reading and writing data. The most important operations are Get and Put, both operating on a single row.

Definition 12 (HBase Get)**.** t.get(`row`, `col`, `ver`) retrieves from HBase the data item in table t whose key is (`row`, `col`, `ver`).

Keys can contain the special symbol $*$ in some entry (except row) to express that the entry is unspecified. So t.get(`row`, `col`, $*$) returns all data items that match `row` and `col` in their keys. On the other hand, t.get(`row`, $*$, `ver`) returns all data items that match `row` and `ver`. t.get(`row`, $*$, $*$) returns all cells with `row` in their keys. t.get($*$, `col`, `ver`) is not allowed, i.e., a Get is a single-row operation.

To get the latest version of a cell, t.get(`row`, `col`) returns the cell whose key matches `row` and `col`, and has the largest version number. For example, the call t.get(`C`, `financial:bank`) to table t in Figure 2.3 returns the key/value pair

$$\langle (\texttt{C}, \texttt{financial:bank}, 125), \texttt{Pohjola} \rangle .$$

Definition 13 (HBase Put)**.** t.put(L) is given a set L of key/value pairs with common row key, and requests the HBase table t to store the given key/values pairs. If a key/value in L has the version number unspecified, HBase takes care of giving that key/value a version number created from the server's wall clock time[4].

Not all dimensions in a key can be arbitrary for a Put. A table has a specified set of possible column families, but column qualifiers can be arbitrary. Column families are normally defined only at the time the table is created, but new families can be added at any time by first disabling [27] the table, which is an expensive operation. Consequently, it is recommended to keep the set of column families fixed, even though it can be modified.

On the other hand, new row keys, column qualifiers, and version numbers can be specified on the fly in Put operations. In this sense HBase is an *Extensible* Record Store, because the schema of a table is allowed to be extended by adding new columns qualifiers, with no overhead compared to using existing column qualifiers. These new column qualifiers do not affect previous existing rows that did not contain such qualifier, since null data items are not stored.

Notice how the Put operation allows a set of data items to be submitted. In fact, the group of write operations being submitted are performed atomically (via locking mechanisms), and hence HBase provides ACID properties for a Put operation. This fact will be discussed more carefully in the following sections. An operation similar to Put is CheckAndPut, which includes one Get operation performed atomically with a Put.

[4]HBase requires its servers to have synchronized clocks in order to make version numbers from clock time. More details on "Implicit Versioning" is given by [27].

Definition 14 (HBase CheckAndPut)**.** t.checkAndPut(`row`, `col`, `ver`, x, L) is an operation that atomically performs: t.get(`row`, `col`, `ver`), and if the result is x, then t.put(L) is executed. All row keys in L must be equal to `row`. CheckAndPut returns **true** if and only if the Get check succeeded.

The CheckAndPut operation is similar to compare-and-set instructions in multithreaded operating systems. It writes to the cell only if the previous value in the cell matches the given value.

Definition 15 (HBase Delete)**.** t.delete(`row`, `col`, `ver`) removes the given data item from table t.

The operation t.delete(`row`, `col`, $*$) removes all versions of the specified cell, while t.delete(`row`, $*$, `ver`) removes all data from the specified row version, and t.delete(`row`, $*$, $*$) removes all data of the given row.

Delete, CheckAndPut, and Put are the only HBase operations that can write/update data. These are single-row operations.

Definition 16 (Row mutation)**.** An HBase row mutation is either a Put, a CheckAndPut, or a Delete.

An HBase Scan is a read operation that spans a range of rows in a table. Two row keys are given to determine the ends of the range, and the scan gradually returns each row whose key is in the range. To return a row means to return all cells with common row key. Contrary to the previously discussed operations, a scan is not atomic: it can be partially executed. That is, a scan might be interrupted by a failure, and the data scanned might be outdated if a write operation occurred during the scan. These issues are discussed in Section 2.3.3.

Definition 17 (HBase Scan)**.** t.scan(`startrowkey`, `endrowkey`) starts a scan operation of the row key range `startrowkey` (inclusive) to `endrowkey` (exclusive) and returns a *scanner* object with two functions: `hasNext`() and `next`(). The `hasNext`() function returns whether or not there is still some row to be scanned from the specified range. In lexicographic order of the row keys, `next`() returns the next row that has not yet been scanned by the scanner.

In the underlying system, an HBase Get is in fact a Scan where the range comprises a single row key. The next section introduces the internal HBase systems that sustain this data model and its operations.

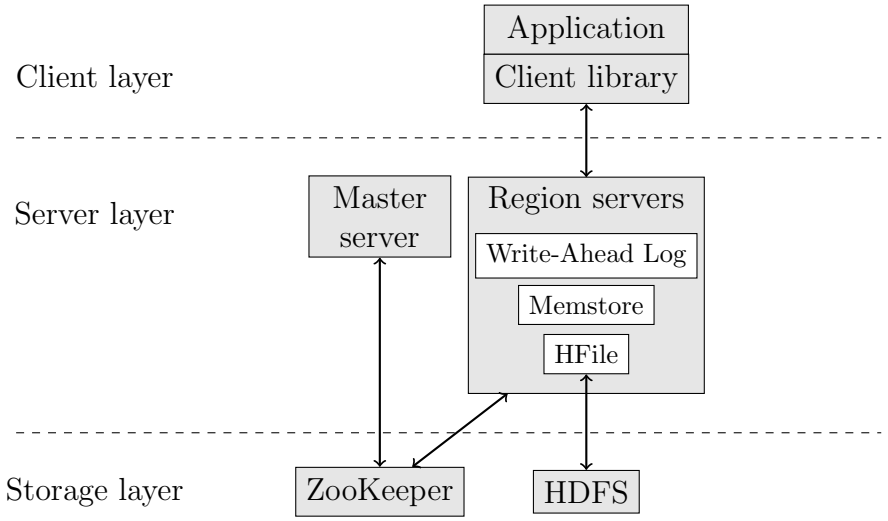

Figure 2.4: The main components in HBase's architecture. Edges indicate important communication between components.

2.3.2 Architecture

We give a quick overview of the internals of HBase, but for an in-depth description, refer to [27]. HBase consists of three layers: the *client*, the *server*, and the *storage* layers. The client layer has the client library for the application; the server layer consists of a *master server* and several *region servers*; the storage layer comprises a distributed file system and a reliable distributed coordination service, typically Hadoop Distributed File System (HDFS) and ZooKeeper, respectively. See Figure 2.4.

For data operations, the client library communicates directly with a region server. As mentioned in the previous section, the rows of a table are partitioned into regions, according to the order of row keys. The region is the basic unit of scalability and load balancing in HBase. Each region is served by exactly one region server.

The region server handles read and write requests to some regions, and initiates a split of regions that have become too large. The region server consists of three main components: *Write-Ahead Log* (WAL), *Memstore*, and *HFiles*. The WAL is a log of all modifications done to data in the region and its purpose is to allow recovery from failures, because any mutation can be redone to bring the data store to the state it should be in before the failure happened. Memstore is an in-memory buffer that contains recently updated data items sorted by key, with the purpose of caching data items to reduce read and write latencies. HFiles are persistent and ordered immutable maps

from keys to values. WAL and HFiles are files in the underlying filesystem, HDFS.

When a Put is processed in the region server, the changes are first written to the WAL, which is stored persistently. If the write to the WAL succeeds, the Put is executed on Memstore. When Memstore exceeds its maximum capacity, it is flushed as an HFile to the filesystem. When the region server receives a Get request, it attempts to find the appropriate data item from Memstore, and if it was not found, the search for the data item continues through HFiles.

HDFS [7, 25] is a distributed filesystem designed with built-in replication, fault tolerance, and scalability. HBase uses HDFS by default, but optionally other filesystems can replace it. Originally, the purpose of HDFS was to support the MapReduce framework, therefore its performance is optimized for batch processing in Big Data. HDFS is typically installed on commodity hardware, so it employs replication to provide fault tolerance.

The assignment of regions to region servers is done automatically by one master server, according to changes in workload and region server failures. The master is also responsible for balancing the load of regions across region servers, performing garbage collection of files, and handling schema changes. The master manages the assignment of regions using ZooKeeper to keep track of regions and region servers.

Apache ZooKeeper [26, 33, 36] is designed to be a smaller data store than HBase, with the purpose of being a highly-reliable distributed coordination service with good read performance. It is typically used as a database-like service for storing configuration and coordination data for distributed systems, and is comparable to Chubby [10] from Google. In HBase, each region server creates a data item in ZooKeeper to represent its existence. Region servers and the master server access ZooKeeper's data store to acquire a reliable global view of the set of current region servers and the regions they manage.

2.3.3 Properties

In this section, we discuss the ACID properties related to HBase data operations mentioned previously.

Because only one region server is serving a row, Serializability of operations on a fixed row can be provided. In fact, Gets and Puts of a row are executed serially by the region server, whereby perfect Isolation can be guaranteed. The region server does this by using row locks that result in atomicity of a row mutation. Durability of a row mutation is ensured by the WAL and HDFS.

Guarantee 1. *Row mutations have ACID properties, where the isolation level is Serializability.*

Notice that ACID properties are guaranteed for single-row operations, but not for transactions with operations on multiple rows. In particular, the CheckAndPut operation is the most similar to a transaction, since it involves a Get and a Put on the same row. It can thus be said that HBase supports simple single-row transactions. This guarantee is extremely important for data integrity, and it is a necessary property for the transactional system discussed in this Thesis. Recalling the definition of data item from Chapter 2, we see that an HBase row can be seen as a data item with a complex data structure for its value.

The Scan is the only operation among the previously mentioned that concerns multiple rows. Internally, scans are allowed to request rows in batches, therefore the data obtained of a row may not be the most up-to-date. If a row is mutated after a scan has requested it in batch, the scan may or may not reflect the mutation. A scan is guaranteed, however, to reflect all data written prior to the construction of the scanner with t.scan(). In other words, the following holds.

Guarantee 2. *The isolation level for HBase Scan operations is Read Committed.*

Both guarantees are important building blocks for transactional systems that are built on top of HBase or Bigtable.

2.4 Transactions in Extensible Record Stores

Recently in 2010 Google published their design of Percolator [44], a transactional management system using Bigtable, tailored for incremental data processing. This meant that ACID multi-row transactions are possible on top of Bigtable. Naturally, given the similarity between Bigtable and HBase, one could ask if multi-row transactions are also feasible with HBase. Table 2.1 compares features supported by (or made possible by) relational databases and Extensible Record Stores.

Independently from Percolator, Zhang and Sterck [59] also in 2010 published a transactional system for HBase, later named HBaseSI [60]. This system is a client library that maintains some special tables in HBase specifically for transaction management. It has no centralized server for managing transactions, so clients concurrently decide to commit or abort transactions based on HBaseSI's metadata[5] tables, which are HBase tables storing transactional

[5]We use the term *metadata* to denote data concerning transaction management.

	SQL-based RDBMS	Extensible Record Stores
Query language	✓	×
Relational operations (join, project, ...)	✓	×
Highly distributed	×	✓
Highly scalable	×	✓
Multi-row transactions	✓	?

Table 2.1: Comparison of features of typical relational databases and Extensible Record Stores. The question mark indicates an ongoing shift of the status of the feature.

management data. Currently, there is no publicly available implementation of HBaseSI.

In 2011, Junqueira et al. [35] presented ReTSO, a lock-free transaction management system, and its open-source implementation, Omid [58], for HBase. ReTSO is a centralized scheme that employs a "transaction status oracle" server to manage transactions. Optimistic Concurrency Control [37] is employed for managing transaction commits.

Recently, in parallel to the development of this work, Padhye and Tripathi [43] have devised a scheme similar to HBaseSI in that it is a client library that maintains transactional metadata tables in HBase. The system, which we will call *Dependency Serialization Graph* (DSG), uses fewer metadata tables than HBaseSI and provides stronger isolation properties in concurrent scenarios.

There are many other transactional systems for Cloud data stores, many of which are not directly related to HBase, such as ElasTras [16], Scalaris [47], G-Store [17], Google's Megastore [2], and Deuteronomy [39]. CloudTPS [53] is worthy to note: it is a decentralized scalable transaction manager suitable for HBase. On the other hand, since it is tailored for web applications, it assumes transactions to be short-lived and access a small number of well-defined set of items.

Although the data consistency that transactions provide is valuable in any situation, it does not come without cost. Supporting transactions leads to performance drawbacks and involves careful design. Many web applications have successfully made use of HBase or Bigtable without transactional support, so there must be a strong reason for adopting transactions. As

pointed out by Percolator authors Peng and Dabek [44], transactional support is essential for concurrent incremental updates to large repositories of data.

For illustrating the problem of incremental data processing, consider the example of a large web index. When a web page is updated with new content, the index entries related to the new content should be recomputed to reflect changes. Typically new content would be indexed only at the next scheduled batch processing, along with the entire web index. Given the Big Data scale of these computations, the frequency of scheduled batch processing can be at least some days. End users, however, desire that the web index is at most some minutes old.

With transactional support, several concurrent content updates can be reflected in the web index without redundant recomputation of the remaining index entries. This drastically reduces the time between content generation and updated index entries. On the other hand, batch processing – for instance with MapReduce – is still advantageous when large percentages of the web index are updated. There is a trade-off between massive recomputations and small incremental updates. For example, Peng and Dabek [44] measured the median latency that MapReduce and Percolator take for reflecting new content in the web index, as a function of the fraction of repository refreshed per hour. When about 10% of the repository is refreshed per hour, Percolator has latency below 10 seconds for updating the index, while MapReduce takes over 2000 seconds for doing the same. On the other hand, Percolator has larger latency (over 2500 seconds) than MapReduce when approximately 40% of the repository is refreshed per hour. This means that Percolator is advantageous when only a small portion of the data repository is being updated.

A transactional system for incremental updates, hence, need not focus on massive throughput of transactions. In a scenario where massive amounts of new content need to be indexed, batch processing methods like MapReduce may have better performance. Latency needs to be low enough to allow an advantage over MapReduce, but extremely low latency is not necessary, even though beneficial.

An important requirement for transactional systems for incremental updates is the support for transactions that span an arbitrary number of data items, and with no time constraints for executing. The system should be able to handle huge indexes, with sizes typically larger than the RAM memory capability of a commodity computer. Put differently, the system should be designed for Big Data.

Given the faulty nature of commodity hardware that Cloud infrastructures are normally built on, the transactional system should also perform

transaction recoveries. That is, the system should recover and carry on transactions that should have committed their changes but were interrupted by some failure.

The decentralized design of Percolator is an intelligent use of Bigtable that satisfies the aforementioned requirements. As HBase is an implementation of Bigtable, it would suffice to have an open-source implementation of Percolator for the purpose of supporting incremental updates. However, there is space for improvement where Percolator is lacking. With Percolator, two concurrent conflicting transactions can unnecessarily abort [43, 60], and the use of locks hinders the progress of concurrent transactions, e.g., a read-only transaction must wait until certain locks are released [43]. Moreover, the ReTSO authors have shown [57] how Serializability – a valuable property not supported by Percolator – can be easily achieved in lock-free transactional systems like ReTSO.

Alternatives to Percolator are HBaseSI, ReTSO, and DSG. Among these, only ReTSO has a publicly available implementation. Compared to other systems, it provides transactional support at the best performance, with very high throughput and low latency. Serializability is also supported, so ReTSO proves to be a solid transaction manager for HBase. However, the centralized nature of ReTSO diverges from the highly-distributed and highly-scalable design characteristics of typical Cloud software like Hadoop and HBase. Although ReTSO is currently capable of handling large amounts of transactional data, its memory limitations will eventually restrict the scalability of the system. In practice this is because ReTSO starts to pessimistically abort transactions when the server's memory is full. Other problems typical to centralized systems can occur, such as downtimes when the ReTSO server fails.

HBaseSI, DSG, and Percolator share much in common. These are all client libraries that simply use the data store in an intelligent way in order to provide transactional support. By doing this, features from HBase such as fault-tolerance and data consistency are immediately inherited. The decentralized client-based design also seems not to limit scalability, which is desirable for Cloud infrastructures.

Transactional metadata tables are employed in both HBaseSI and DSG. Six tables comprise the former and at least two tables are employed in the latter. An interesting design question is: *what is the minimum number of metadata tables required for providing transactional support on top of HBase*? Can such minimum be achieved while maintaining ACID properties? Reducing the number of metadata tables has performance advantages. For instance, in HBaseSI, each transaction has to access six different tables for bookkeeping purposes. This is expected to increase the overhead for transactions.

We have seen reasons to justify the creation of a new transactional system for HBase. The contribution of this Thesis is the design of a new lightweight non-centralized transaction manager for HBase. The objective is to provide a transactional system suitable for general purposes but tailored in performance for incremental data processing. The system is named HAcid, and is presented in the chapter.

Chapter 3

HAcid

HAcid[1] is a client library that applications can use for operating multi-row SI transactions in HBase. No server-side modifications are necessary. Many concurrent HAcid-enabled clients can be active without losing correctness properties such as ACID. This transactional system is intended for general purpose multi-row transactions, and allows running ACID transactions with an arbitrary number of operations across an arbitrary number of different rows.

The system is in many ways similar to HBaseSI, Percolator, and DSG, but innovates by using a minimal number of metadata tables. In particular, in HAcid the timestamp oracle is a table in HBase that also contains transactional data, and timestamps are served by appending the table. This approach is new compared to other transactional systems for HBase, and is beneficial in many ways.

HBase offers single-row transactions, but not multi-row transactions. However, HBase is not far from supporting that feature. The design for HAcid is motivated by the pursuit of the minimal amount of changes to an HBase system to achieve multi-row transactions. HAcid aims to be lightweight, which is a property achieved when the system answers the following questions.

The key questions for lightweight multi-row transactional support are: (a) what is the simplest and minimal data structure for transactional metadata? (b) What is the least amount of changes in the server layer? (c) What is the smallest amount of changes in the client layer? (d) What is the simplest and most familiar API for using the transactional system? (e) What is the minimal performance overhead for transactional support?

These questions define the goals of HAcid's design, discussed in the next section.

[1]The name HAcid refers to HBase and ACID, two important concepts for this transaction manager. Acids and bases in Chemistry were also an inspiration for the name.

3.1 Design

HAcid's design consists of transaction management algorithms in the client side, and transactional data in the server side, i.e., in HBase tables. These two parts of the system enable transactions over *user tables*, which are conventional HBase tables that transactions will modify.

For concurrency control, HAcid employs Optimistic Concurrency Control and Multi-version Concurrency Control. No locks are used, and Snapshot Isolation (SI) is supported.

Because HAcid uses MVCC for Snapshot Isolation, transactions are timestamped and write operations use the start timestamp as the version number for data items, as previously described in Section 2.1.4. For this reason, the user application should not attempt to handle version numbers, which are managed by HAcid algorithms.

3.1.1 HAcid as a transactions certifier

The concurrency control method in HAcid is a *certifier* [4]. Certifiers are methods that execute transaction operations as soon as they are requested without blocking them. When a transaction ends, it undergoes a certification: the process of checking for conflicts to certify that the transaction can commit. If the certification fails, the transaction aborts and optionally restarts. The HAcid client library processes a transaction in three phases, summarized as follows.

1. **Execution phase:**
    - the client library executes read and write operations of a transaction only in this phase;
    - any modifications done by the transaction are invisible to other transactions; and
    - the client does not track nor predict conflicts of transactions in this phase.
2. **Certification phase:**
    - the client searches for conflicts between the current transaction and other transactions, by inspecting a repository of transactional data; and
    - the fate of the transaction is decided: *committed* if there are no conflicts, *aborted* if some conflict was found.

3. **Update phase:**

    - if the transaction committed, the modifications of the transaction are *rollforwarded*, i.e., they are made visible to other transactions; and
    - if the transaction aborted, the modifications are *rollbacked*, i.e, the version is deleted.

Certification is a form of OCC because conflicts are expected to be rare. In other words, concurrency control is done by aborting and restarting transactions, since we optimistically assume concurrent transactions to be isolated.

In HAcid, SI with MVCC allows the operations of a transaction T_i to execute freely without influencing other transactions, since each transaction writes to its own version. The certification at the end of T_i then affects other transactions by determining whether the version of T_i should become visible.

3.1.2 Architecture

HAcid consists of a few components: a client library, an HBase table for transactional metadata called "Timestamp List", and an additional column in all user tables. The table and the column are in the server-side, that is, in an HBase data store. The client library holds transaction management algorithms. Figure 3.1 illustrates the architecture.

The HAcid client library is built on top of HBase's conventional client library. For performing transactions in HBase, the client application uses the API offered by the HAcid library instead of the API from the HBase library. There are no other components in the system, such as a dedicated timestamp oracle or any kind of server process. The only kind of process in HAcid is a client application augmented with the library.

Multiple HAcid clients can be active at the same time, executing transactions in HBase. Concurrency issues are handled by using the metadata table and the special user table column as the only kind of shared data between clients. There is no global knowledge of currently active clients, and clients cannot directly communicate with each other.

The metadata table, Timestamp List, has a central role in HAcid.

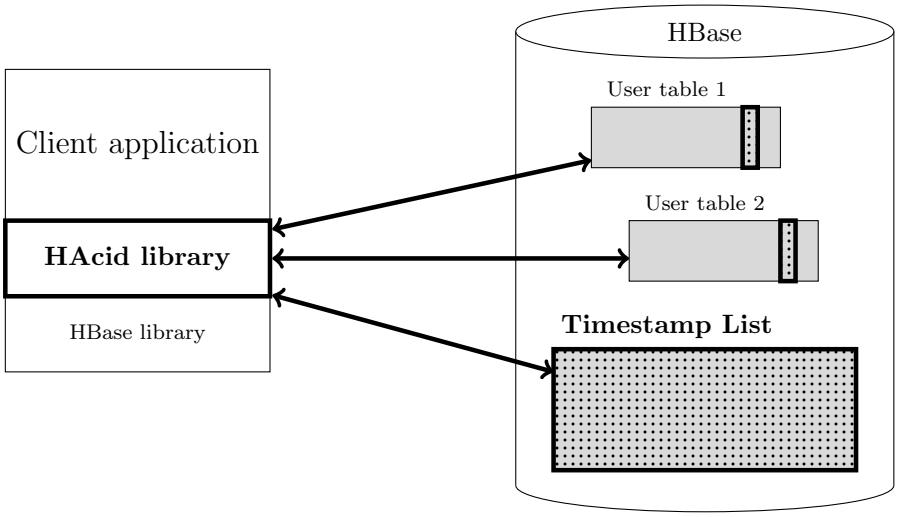

Figure 3.1: Components in HAcid's architecture. The edges leaving the HAcid library indicate which components in the HBase side are modified by transaction management algorithms in the client side.

3.2 Transactional metadata repositories

3.2.1 The Timestamp List

At the core of HAcid's design lies the *Timestamp List*, a table in HBase managed by HAcid algorithms and with two purposes. It acts as a timestamp oracle, and stores all transactional metadata. Its schema is carefully designed to take advantage of HBase properties. This is an innovative approach among transactional systems for Extensible Record Stores. The Timestamp List is illustrated by the example in Figure 3.2, and explained in the following paragraphs.

Each row in the Timestamp List represents a timestamp of some transaction T_i: either a start timestamp T_i^{start} or an end timestamp T_i^{end}. The row key is precisely the timestamp (an integer) represented by that row. Since HBase tables are sorted by row key, the timestamps in the Timestamp List are sorted in increasing order, as rows in the table. Hence the first row represents the first timestamp, and consecutive rows represent consecutive timestamps. The type of a timestamp ("start" for T_i^{start} or "end" for T_i^{end}) is indicated by the column qualifier `type` in every row. There is only one column family in the Timestamp List, so it is not necessary to name it in this Thesis. Hence in this section we refer to column qualifiers simply as *columns*.

row	type	status	writeset	start-ts	end-ts
⋮		⋮	⋮		⋮
8	start	committed			9
9	end		(t2,E),(t2,F)	8	
10	start	active			
11	start	committed			13
12	start	aborted			14
13	end		(t1,A),(t1,B)	11	
14	end		(t1,B),(t1,C)	12	

Figure 3.2: Example Timestamp List. Version numbers are omitted because each cell has only one version. In writeset cell values, `(t1,A)` refers to row `A` in table `t1`.

To serve timestamps, the Timestamp List is first analyzed by clients to determine what is the last existing timestamp, then a new successive timestamp is created. Since a timestamp oracle must serve timestamps in increasing order, such that the latest served timestamp is the largest timestamp, the Timestamp List must be appended with a new row. Fortunately, this can be done rather efficiently in HAcid due to HBase single-row transactions. The append operation for the Timestamp List is explained in Section 3.3.4.

Besides storing timestamp data, the Timestamp List contains columns for transactional metadata. These are: `status`, `writeset`, `start-ts`, and `end-ts`. Timestamped transactions in HAcid are uniquely identified by their start or end timestamps. Therefore, the rows in the Timestamp List corresponding to the start and end timestamps of a transaction can store its transactional metadata. Since there are two rows for each transaction, transactional metadata is partitioned between the start and end timestamp rows. Thus, some of those columns will have non-null values only in the start timestamp row, and others only in the end timestamp row.

The `status` cell in a row can assume one of the three following values for indicating the transaction's status: "active", "committed", or "aborted". A transaction is active if it is not yet committed or aborted. The `status` cells are stored only in start timestamp rows. This is justified from the fact that all transactions have some status, even those that do not yet have an end timestamp.

The `writeset` cell represents T_i^W of a transaction T_i. This column has

non-null values only in end timestamp rows, for the sole fact that the writeset is only known when the transaction ends and no more write operations can be submitted. Since formally T_i^W is a set of data items, the cell value of `writeset` is a list of data item keys. The keys are separated by commas, and each key has the format "`(t1,A)`", where `t1` is a table name, and `A` is a row key. That is, data items in HAcid transactions are rows in user tables.

The cells `start-ts` and `end-ts` are simply pointers for cross-referencing the start and end timestamp rows, linking them together. That is, the cell `start-ts` is non-null only in end timestamp rows, and the cell `end-ts` is non-null only in start timestamp rows. This allows a client reading an end timestamp row in the Timestamp List to discover the corresponding start timestamp, and vice-versa. These are important procedures for HAcid transaction management algorithms.

The Timestamp List is a *tall-narrow* [27] HBase table, i.e., it has a small number of columns and an arbitrarily large number of rows. The version number dimension is not necessary for its schema, since each cell in the Timestamp List has only one version. The table is initialized with one dummy row with row key 1. This timestamp row has `type` = `end`, because for a row to exist, one of its cells must be non-null.

3.2.2 Metadata column in user tables

In theory, the Timestamp List alone is enough for indicating what versions of data are committed, but in practice retrieving this information is expensive. For instance, to perform a read operation of a row `A` in a transaction, the transaction manager can read all rows from the Timestamp List to discover the latest committed transaction with `A` in its writeset. This approach is clearly problematic: expensive scans would be necessary for each read operation. The reason why the Timestamp List is not usable for this purpose is the lack of indexing of rows according to `writeset` entries. That is, given `(t1,A)` (table `t1`, row `A`), we cannot efficiently get from the Timestamp List the set of timestamps with `(t1,A)` in their `writeset`.

Therefore, for good performance of read operations, we employ a metadata column in each user table for storing transactional metadata. All tables that are to be modified by multi-row transactions must include this column. To include the column, a new HAcid-specific column family is introduced, so each user table must be first disabled (a possibly expensive operation) before including the HAcid column family. Fortunately, this installation procedure is necessary only once per user table. More information on HAcid installation is found in Section 3.5.2.

The column serves as a *cache* of some metadata from the Timestamp List.

row	personal:name	financial:balance	HAcid:commit-time
A	25: Adam	45: 1630 35: 1690 31: 2120 25: 1911	45: 0 35: 38 31: 0 25: 27
B	19: Bob	19: 4124	19: 23
C	28: Caroline	42: 851	42: 0 28: 29
S	15: Smith	15: 2380	15: 16

Figure 3.3: A user table augmented with the `commit-time` metadata column. Version 28 of row `C` has been committed at the end timestamp 29. HAcid is a Snapshot Isolation method (recall Section 2.1.4), therefore all transactions that have start timestamp greater than 29 can see the version 28 of row `C`.

The purpose of the metadata column is to indicate which versions of data in user tables are visible to other transactions' reads. That is, this column is responsible for indicating if data is committed.

The metadata column is called `commit-time`[2], and its value holds the end timestamp of the transaction that committed that particular user data version. In other words, in row R the value at $(R, \texttt{commit-time}, v)$ is the end timestamp of the transaction that committed some data to R and version number v. See the example in Figure 3.3.

The value zero in the `commit-time` column has a special connotation: it indicates that the data version is in an unstable state of visibility. If `commit-time` has a positive value (*cache hit*), then the data version is certainly committed, but if `commit-time` is zero (*cache miss*), the data version can either be committed **or** not yet committed.

A cache miss is useful for indicating a situation for recovery. Suppose that a client using HAcid library commits a transaction (in the Certification Phase) but fails before updating the `commit-time` value (in the Update Phase), which remains zero. Another client that reads the zero value can check from the Timestamp List if that particular data should be visible, and proceeds with the Update Phase where the previous client failed. These recovery issues are discussed in detail in Section 3.3.6.

We could have used a separate table for building an index of data items and their corresponding transactions. The table could have data item keys as

[2]The column name is actually `HAcid:commit-time`, with `HAcid` as a column family name, but we omit the column family for convenience, since this family will be obvious from the context.

row keys, and cells for start timestamps of transactions that modified the data item. However, it is beneficial for performance to collocate this information in user tables. A read operation can efficiently retrieve multiple columns from the same row, so we can read both user data and `commit-time` metadata in one operation. Using a separate table for `commit-time` would require two read operations: one at the user data, and another at the `commit-time` table.

The main advantage for using a separate table for this purpose would be no intrusion of HAcid into user tables. However, the `commit-time` column is a minor intrusion, since all other column families in the user tables are preserved.

3.3 Transaction management algorithms

A transaction is originated at the client application using the HAcid library. The client library follows three phases for each transaction: Execution, Certification, and Update. Each transaction has its own phases, i.e., transaction T_1 might be in Execution Phase while T_2 is in its Update Phase.

When the application creates a new HAcid transaction, the HAcid library will first get a start timestamp for it at the Timestamp List. Then, operations are added to the transaction. The whole set of operations need not be specified before executing the operations, but are executed on the fly, as soon as they are added to the transaction. This Execution Phase terminates when the client application requests a commit of the transaction. From that point onward, the HAcid library at the client performs Phases Certification and Update automatically, with no need for input from the client application.

3.3.1 Example run of a transaction

The steps of transaction management in HAcid are described first by an example transaction, then by pseudocode. Consider table `t` to be the user table given in Figure 3.3, a portion of which is repeated in Figure 3.4 for convenience, and the transaction

$$T_1 : r_1[B = x]\, w_1[C = x]$$

that reads column `financial:balance` from row `B` in table `t`, and writes the previously read value x to column `financial:balance` on row `C` in table `t`.

Once T_1 is created by the client application, the attached HAcid library requests T_1^{start} by accessing the Timestamp List and appending it to get a new timestamp row, setting its type to "start" and status to "active". Supposing that 50 is the last timestamp in the Timestamp List, the new last timestamp

row	personal:name	financial:balance	HAcid:commit-time
B	19: Bob	19: 4124	19: 23
C	28: Caroline	42: 851	42: 0 28: 29

Figure 3.4: An example portion of a user table t before being modified by a transaction.

after an append operation will be 51. In our example, $T_1^{start} = 51$, as shown by Figure 3.5 below. The appending operation is described in Section 3.3.4.

row	type	status	writeset	start-ts	end-ts
51	start	active			

Figure 3.5: $T_1^{start} = 51$ in Timestamp List, at the beginning of the processing of transaction T_1.

After T_1 has received its start timestamp, the client application can begin to submit operations to T_1 through the HAcid library. Transaction T_1 is in its Execution Phase. First a read of row B is added to T_1, and HAcid promptly executes an HBase Get operation on table t and row B. Given the start timestamp T_1^{start}, HAcid knows exactly what version of row B to read so returns the value 4124 (at version number 19) to the client application. More information regarding the read version is given in Section 3.3.5.

Next, a write operation is submitted to T_1, which in turn executes an HBase Put operation in table t and key $(\texttt{C}, \texttt{financial:bank}, T_1^{start})$. The Put also simultaneously writes value 0 to $(\texttt{C}, \texttt{HAcid:commit-time}, T_1^{start})$, to indicate that this version is not confirmed to be committed. See Figure 3.6 for the state of table t and row C after these operations are executed.

row	personal:name	financial:balance	HAcid:commit-time
C	28: Caroline	51: 4124 42: 851	51: 0 42: 0 28: 29

Figure 3.6: The modified row C in table t after T_1 executed its operations in the Execution Phase.

When the client application has no more operations for T_1, it requests a commit of T_1 to HAcid. This marks the end of the Execution Phase and the beginning of the Certification Phase. At the beginning of this phase, HAcid

gets an end timestamp for T_1 by appending the Timestamp List in the same fashion as start timestamps are created. The end timestamp row includes also information about T_1, such as `writeset` and `start-ts`. See Figure 3.7.

row	type	status	writeset	start-ts	end-ts
52	end		(t,C)	51	

Figure 3.7: Row $T_1^{end} = 52$ in Timestamp List.

After the end timestamp is created, HAcid starts the certification of T_1 to check for write-write conflicts. For searching for a write-write conflicting transaction T_j with T_1, HAcid scans the Timestamp List (using an HBase Scan operation) from T_1^{start} to T_1^{end}. When a timestamp of type `end` (representing T_j^{end}) is found, its `writeset` is compared with the `writeset` at T_1^{end}: if there is an intersection and the status of T_j (located in the row T_j^{start}) is not aborted, then a conflict has been found. In our example, no conflict is found because there are no timestamps between 51 and 52.

When HAcid has concluded whether or not conflicts exist with T_1, it ends the Certification Phase and starts the Update Phase. This change of phase is marked by a commit decision: the status of T_1 is set to either "committed" (if no conflicts found) or "aborted" (if any conflict was found). In our example, T_1 committed. The decision is done by atomically writing to the start timestamp row of T_1: its `status` cell is set to "committed", and `end-ts` points to T_1^{end}. See Figure 3.8 for the start timestamp row of T_1 at the end of Certification Phase.

row	type	status	writeset	start-ts	end-ts
51	start	committed			52

Figure 3.8: Row $T_1^{start} = 51$ in Timestamp List, at the beginning of the Update Phase.

In the subsequent Update Phase, we update the user tables rows written by T_1. If T_1 committed (which is the case), we update the `commit-time` column to point to T_1^{end}. If T_1 aborted, we delete versions of data that T_1 wrote. See Figure 3.9 for the outcome after the Update Phase.

After the last update has happened in the Update Phase, transaction T_1 has completed its processing.

row	personal:name	financial:balance	HAcid:commit-time
C		51: 4124	51: 52
		42: 851	42: 0
	28: Caroline		28: 29

Figure 3.9: The modified row C in table t after the Update Phase of T_1. Notice commit-time at version 51.

3.3.2 Summary of transaction processing

Note that the three phases in HAcid's processing of a transaction are separated by events such as the creation of an end timestamp when the client application requests a commit for the transaction. The events and phases of a transaction's processing are summarized in Figure 3.10.

There are three events and three phases. The events are: (i) creation of a start timestamp, (ii) creation of an end timestamp, (iii) update of the start timestamp to set the status of the transaction (Commit Decision). All these three events happen in the start and end rows in the Timestamp List corresponding to the current transaction. Each event is an HBase row mutation, hence atomic.

The atomicity of events in HAcid is important for properties such as correctness and transaction atomicity. Phases, in contrast, are not atomic. Consequently, a phase can be partially executed and interrupted by a failure, while an event cannot. This observation is relevant for recoveries in HAcid, an issue that is discussed in Section 3.3.6.

During the Execution Phase of transaction T_i, the writeset of T_i is updated as write operations arrive. This is maintained in the local variable T_i^W. When the client requests the commit of T_i, the variable T_i^W is exported to the cell writeset in the newly created end timestamp row.

In the Certification Phase, an end timestamp row T_j^{end} of a *possibly conflicting* transaction is searched for in the range ($T_i^{start} < T_j^{end} < T_i^{end}$) of

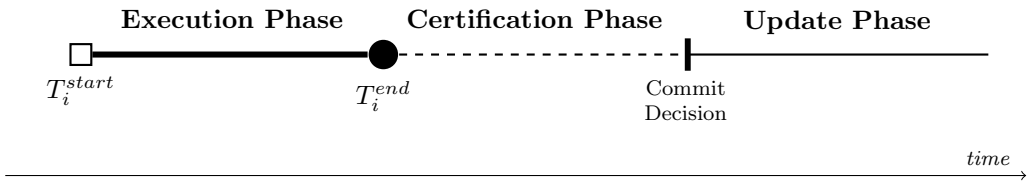

Figure 3.10: Important events and phases in the processing of a transaction in HAcid. Events are represented as nodes and phases are represented as lines.

the Timestamp List. Transaction T_j is possibly conflicting if its end timestamp row T_j^{end} has its `writeset` intersecting T_i^W. When a possibly conflicting transaction is found, we check for its `status` in the start timestamp row T_j^{start}. If its `status` is "committed", then T_j is conflicting, but if its status is "aborted", T_j is not conflicting. If, however, the `status` for T_j is "active", then we pessimistically assume that T_j is conflicting.

The client application has control over the first two events and the Execution Phase, but the following Phases are coordinated by the HAcid library. The commit request from the client application is a blocking request and the response comes from the commit decision event. That is, the commit request lasts as long as the Certification Phase.

3.3.3 Pseudocodes

The transaction management methods illustrated in the previous sections are described in the following pseudocodes. Some subprocedures are sophisticated, and described in details in the next sections. All pseudocodes refer to the management of a transaction T_i.

- Upon the request for initializing transaction T_i:
  1. Create a new timestamp in the Timestamp List, setting its `type` to "start" and its `status` to "active". To create a new timestamp, one must append the Timestamp List. T_i^{start} is the row key of the row created in the Timestamp List. This operation is explained in Section 3.3.4.
  2. Start Execution Phase only if the previous step was successful.
- In the **Execution Phase**:
  - Upon a request for a read operation of table t, row R, column C:
    1. Get the version number $\sigma :=$ ReadVersion(t, R, T_i^{start}), which is a procedure explained in detail in Section 3.3.5.
    2. Return t.get(R, C, σ).
  - Upon a request for a write operation on table t, row R, column C, value x:
    1. Insert (t, R) into T_i^W (local variable).
    2. Execute t.put($\langle (R, C, T_i^{start}), x \rangle$, $\langle (R, \texttt{commit-time}, T_i^{start}), \texttt{0} \rangle$).

  - Upon a commit request from the client application:
    1. If (local variable) $T_i^{end} = \perp$, create a new timestamp T_i^{end} in the Timestamp List, setting its `type` to "end", `writeset` to T_i^W, and `start-ts` to T_i^{start}.
    2. Start Certification Phase only if the previous step was successful.

- In the **Certification Phase**:

  1. Scan through each timestamp τ in the Timestamp List between T_i^{start} and T_i^{end}. If τ is an end timestamp ($\tau = T_j^{end}$ for some $j \neq i$) and its `writeset` cell intersects the `writeset` of T_i^{end}, then:
     1.1. Assign variable T_j^{start} := TimestampList.get(T_j^{end}, `start-ts`), then assign T_j^{status} := TimestampList.get(T_j^{start}, `status`).
     1.2. If T_j^{status} is "active" **or** "committed", then execute CommitDecision("aborted").
     1.3. If T_j^{status} is "aborted", continue the scan operation of Step 1.
  2. Execute CommitDecision("committed").

- CommitDecision(d):

  1. Execute a CheckAndPut operation on row T_i^{start} in the Timestamp List to atomically check if `status` was "active", and set the `status` to d and its `end-ts` to T_i^{end}. (The value of d is either "committed" or "aborted".)
  2. If the CheckAndPut operation succeeded, use d as the *outcome* of this commit decision.
  3. Else, read the `status` on row T_i^{start}, to make it the *outcome* of this commit decision.
  4. Start Update Phase.

- In the **Update Phase**:

  1. If the *outcome* of the commit decision was committed, then for each $(t, R) \in T_i^W$ execute t.put($\langle (R, \texttt{commit-time}, T_i^{start}), T_i^{end} \rangle$).
  2. If the *outcome* was aborted, then for each key $(t, R) \in T_i^W$ execute t.delete($R, *, T_i^{start}$).

Notice how a possibly conflicting transaction T_j with status active makes transaction T_i abort in the Certification Phase. The transaction T_j could

change its status to either aborted or committed, but HAcid takes a pessimistic approach and decides for aborting T_i. This is to satisfy the First-Committer-Wins rule, because if T_j happened to get status committed, then T_i should not be committed.

On the other hand, in face of active possibly conflicting transactions, one can also postpone the commit decision of T_i. In this alternative, the Certification Phase waits for some time and later rechecks if T_i still has status active. The pessimistic abort of T_i is taken only if T_j is still active.

Other approaches could fit as well. The Certification Phase of T_i could pause for executing the Certification and Update Phases of T_j until it gets a commit decision. This would be useful for recovery of a client that failed before finishing the Certification Phase of T_j. However, the client managing T_i cannot know whether the client managing T_j has failed or is too slow. Assuming that slow clients are more common than failed clients, this approach is expensive because it allows multiple clients to perform the Certification Phase of the same transaction.

3.3.4 Appending the Timestamp List

When a timestamp is created in the Timestamp List, it should be the last row in the table. This is to satisfy the requirement for being a timestamp oracle: the latest timestamp is the largest among the previous. Since rows in the Timestamp List are sorted in increasing order by row keys, which are timestamps, the newest timestamp must be the last row.

Hence, creating a new timestamp requires an append operation. There is currently no such operation provided natively by HBase, so we need a custom implementation for appending. For the Timestamp List, our append operation has the following roles: to discover the last row key, to create a new row after the current last row, and to write some transactional data at the same time the new row is created.

The HBase API has no method for retrieving the row key of the last row in a table, i.e., the largest row key in terms of the lexicographic sorting of rows. However, we can easily discover the last row by scanning the table starting at an arbitrary row, with no end row specified for the scan. When the scan ends, the last scanned row is the last row in the table, if no changes have been submitted since the start of the scan. The closer the start scanning row is to the last row in the table, the more efficient will be the scan.

Besides creating the row, the CheckAndPut operation allows data to be written to cells of the row. For the Timestamp List, it is useful to write transactional data at the time we append a new row. The append operation takes as input some key/value pairs (where the key is only the column key)

for the Put part of the CheckAndPut, while the output is the row key of the appended row. The key/value pairs we write when appending a start or an end timestamp to the Timestamp List were already mentioned in the pseudocodes of Section 3.3.3.

To append a new row to the Timestamp List, we use the known last row key R to attempt to create a row with key $R + 1$. Since other concurrent clients might be appending the Timestamp List, the row $R + 1$ could be already existent. To make sure that row $R + 1$ does not yet exist, we use a CheckAndPut operation to atomically check if $R + 1$ does not exist and simultaneously create it if the check passed. If the CheckAndPut fails, it means that row $R + 1$ exists, so we try the CheckAndPut on row $R + 2$, and so forth, until it succeeds.

CheckAndPut operations in HBase do not allow one to explicitly check if a row is inexistent, so we check if cell `type` is null in the CheckAndPut. Since every timestamp (a row in the Timestamp List) must have `type` either "start" or "end", an empty `type` cell means that the row of that cell is inexistent.

A pseudocode of the append operation is given below.

Append operation on the Timestamp List. Input is a set L of column-key/value pairs.

1. Scan the Timestamp List starting from an arbitrary row, with no end row specified.
2. When the scan stops, record R as the row key of the last scanned row, and $K := R + 1$.
3. Execute a CheckAndPut operation on row K in the Timestamp List to atomically check if column `type` was null, and write L to row K if the check passed.
4. If the CheckAndPut failed, do $K := K + 1$ and go to Step 3.
5. Return K as the row key of the newly appended row.

In Section 3.5.3, we explain how the Timestamp List and this algorithm are modified in the implementation for optimizations.

3.3.5 Searching read versions

Recall, from Section 2.1.4, that each read operation in a transaction uses a data item version corresponding to the transaction's snapshot of the database. In HAcid, the version number used in a read operation is called *read version* σ, defined as follows.

row	personal:name	financial:balance	HAcid:commit-time
S		76: 2148	76: 87
		70: 2368	70: 72
	15: Smith	15: 2380	15: 16

Figure 3.11: Row S in table t, where a ReadVersion procedure operates on.

Definition 18 (Read version)**.** Given data item x and transaction T_i, the *read version* σ is defined as

$$\sigma(x, T_i) = \max\{T_j^{start} : x \in T_j^W \wedge c_j \in T_j \wedge T_j^{end} < T_i^{start}\}.$$

In other words, $\sigma(x, T_i)$ is the largest start timestamp of a transaction that wrote to data item x and committed before T_i started. This is equivalent to finding the latest committed version of x that was committed before T_i started. The equivalence is proved in Section 3.4.

To discover $\sigma(x, T_i)$ in HAcid, we use the `commit-time` column in the user table of x, since all committed transactions T_j with $x \in T_j^W$ are registered in the `commit-time` column. That is, the `commit-time` value is the end timestamp and its version number is the start timestamp of a committed transaction with x in its writeset.

When a read operation is submitted in the Execution Phase, it first searches for its read version through the procedure ReadVersion(t, R, T_i^{start}), as previously mentioned in the pseudocode of the Execution Phase.

The procedure works by reading the versions in the cell `commit-time` of row R in user table t, searching the version number that corresponds to $\sigma((t, R), T_i)$. The procedure searches for the largest version number whose `commit-time` value is smaller than T_i^{start}.

Consider an example, where transaction T_1 with $T_1^{start} = 85$ is searching for the read version of row S in table t from Figure 3.11. Procedure ReadVersion(t, S, T_1^{start}) gets all versions of the cell (S, `commit-time`). The largest version number is 76, but its commit-time value is $87 > T_1^{start}$. The second largest version number is 70, whose `commit-time` is 72 and satisfies $72 < T_1^{start}$. Therefore 72 is the version number returned by the procedure.

In this example, there are no zero values in `commit-time`. In case zero values are found, which represent cache misses, the searching procedure visits the Timestamp List to get fresh data. The ReadVersion procedure is explained in detail through the pseudocode below.

ReadVersion(t, R, T_i^{start}):

1. `Versions` := t.get(R, `commit-time`, $*$).
2. For each key/value pair $\langle(R, \texttt{commit-time}, v), z\rangle$ in `Versions`, in decreasing order of version numbers v, do:
   2.1. If $v < T_i^{start}$ and $z = \texttt{0}$, then: *//This is a Cache miss situation*
      2.1.1. Recovery(v, T_i^{start}). *//Explained in Section 3.3.6*
      2.1.2. If TimestampList.$get(v, \texttt{status})$ = "committed", then set z := TimestampList.$get(v, \texttt{end-ts})$.
   2.2. If $v < T_i^{start}$ and $z \neq \texttt{0}$ and $z < T_i^{start}$, then **return** v.
3. **Return** 1.

In Step 3, version number 1 is returned to refer to the initial dummy row in the Timestamp List and to signal that null should be read as the value of the data item. Notice that v and z correspond to the start and end timestamps of some transaction T_j, i.e., $v = T_j^{start}$ and $z = T_j^{end}$ (if $z \neq 0$). Note how recovery of previously failed transactions is performed during the ReadVersion procedure. In the next section we discuss recovery in HAcid.

3.3.6 Recovery

HAcid clients are susceptible to failures, such as crash failures. The HAcid system must be able to complete the processing of transactions even in the face of crash failures. Transaction recovery in HAcid is a procedure for completing the processing of interrupted transactions. In high-level, the procedure recovers a transaction by detecting what phase it was in before the crash failure, and performing the remaining phases and events.

Crash failures can happen at any time during the processing of a transaction: in events or in phases. If a crash affects an event, the event will not be executed, since it is atomic. Therefore, we focus our attention on crashes during phases.

If a crash happens during the Execution Phase, the transaction might have not executed all of its read and write operations. Since operations are originated at the client processing the transaction, it is impossible for other clients to identify what remaining operations need to be executed. Thus, if transaction T_i was interrupted in the Execution Phase due to a failure of its client, other clients can ignore T_i entirely. This does not violate any ACID property.

If transaction T_i was interrupted by a crash during the Certification Phase, then T_i has a start and an end timestamp, its writeset is known

to other transactions, but a Commit Decision has not happened yet. Therefore, clients have enough information for recovering T_i. Any client is allowed to perform the remaining processing for T_i, which are Certification Phase, Commit Decision, and Update Phase.

Finally, the Update Phase concerns only the `commit-time` column in user tables, updating cached values. Hence, recovery of the Update Phase of T_i is done by allowing other clients to update the cache.

Since HAcid has no centralized process for transaction management, concurrent clients are responsible for all transaction processing, including recovery. Moreover, there is no centralized role for the detection of recovery situations and immediate action. Instead, a lazy approach is applied: a transaction is recovered by a client only when it is considered necessary for a read operation in a transaction managed by that client.

Notice how any client can perform recovery of the failed transaction, and recall that clients have no knowledge of concurrent clients. Consequently, two (or more) clients might be simultaneously recovering the same failed transaction. This is not a problem in HAcid, because recovery is designed to be idempotent [20]. In layman's terms, idempotence is the property of having the same results regardless of how many times an operation is applied, as long as it is applied at least once.

In HAcid recovery methods, idempotence is guaranteed by a mechanism that ensures that multiple clients will not disagree on the Commit Decision of the failed transaction. For recovery, only the Certification Phase, the Commit Decision, and the Update Phase can be executed. The Certification Phase is merely about reading (scanning) the Timestamp List, so no changes are made. The Update Phase, however, changes the `commit-time` cache depending on the outcome of the Commit Decision. Since the Update Phase simply consists of cache updates, multiple clients can be performing this phase for the same failed transaction, as long as the clients agree on the outcome of the Commit Decision. Each transaction gets only one Commit Decision, because the Commit Decision is atomic and it tests if a previous Commit Decision has not yet happened (by checking that the transaction's status is `active` before changing it). Therefore, recovery is idempotent.

The pseudocode for recovery is given below, and is a subprocedure of ReadVersion from the previous section. Recall from that pseudocode that v is the start timestamp of some transaction T_j, which is the target of the recovery. Transaction T_i is the transaction performing the read operation that called ReadVersion.

Recovery($v = T_j^{start}$, T_i^{start}):

1. T_j^{status} := TimestampList.get(T_j^{start}, `status`).
2. If T_j^{status} = "active":
   - 2.1 Scan through each timestamp τ in the Timestamp List between T_j^{start} and T_i^{start}. If τ is an end timestamp such that its `start-ts` cell has value T_j^{start} (hence τ is T_j^{end}), then:
     - 2.1.1 T_j^W := TimestampList.get(T_j^{end}, `writeset`).
     - 2.1.2 Perform Certification Phase, Commit Decision, and Update Phase for T_j.
3. Else:
   - 3.1 T_j^{end} := TimestampList.get(T_j^{start}, `end-ts`).
   - 3.2 T_j^W := TimestampList.get(T_j^{end}, `writeset`).
   - 3.3 Perform Update Phase for T_j.

Note that recovery of T_j might not happen in the procedure. The scan of Step 2.1 might not find T_j^{end} in the interval $[T_j^{start}, T_i^{start})$ and hence will ignore the Steps 2.1.1 – 2.1.2. This means that T_j might: (a) have failed during Execution Phase, (b) be still in Execution Phase, (c) be not necessary for the read operation of T_i because $T_i^{start} < T_j^{end}$, so the recovery is postponed until it is necessary for another transaction.

3.4 Analysis of properties

In this section, we discuss correctness properties of HAcid. We present invariants and claims. Invariants are facts that always hold and can be easily verified in the pseudocodes. Claims are facts for which we present arguments.

We focus on analyzing the correctness of only nontrivial parts of HAcid algorithms, which are: (a) append operation on the Timestamp List, (b) correct conflict detection, (c) read versions and relation to Snapshot Isolation, (d) transaction atomicity. Next, we analyze each of these aspects.

3.4.1 Append operation

We start with some invariants, which are confirmed either from Section 3.2.1 or from Section 3.3.4.

Invariant 1. The Timestamp List is initialized with a single timestamp row: the row key is 1.

Invariant 2. The row key of any row in the Timestamp List is a positive integer.

Invariant 3. No row gets deleted from the Timestamp List.

Invariant 4. All rows in the Timestamp List have a non-null value on the column `type`.

Claim 1. *One instance of the append operation increases the number of rows by one.*

Proof. The CheckAndPut (Step 3 of the append operation) writes data only if the column `type` is null. By Invariant 4, if `type` is null on a row K, then K does not exist. So if the CheckAndPut passes, then a row that was inexistent is created. By Step 4 in the procedure, only one successful CheckAndPut can happen during the procedure, so only one row is created. □

Definition 19 (Tightness)**.** A table is *tight* if every two consecutive rows have row keys that are consecutive integers. (In HBase, two rows are consecutive if one is returned after the other in a standard Scan operation.)

Claim 2. *The Timestamp List is tight before and after an append is executed.*

Proof. By induction. Initially, the table has one row (Invariant 1), therefore it is tight since there are no consecutive rows. Suppose, then, that the table has an arbitrary number of rows and is tight from rows 1 until R (the last row) before the append. The procedure identifies row key R as a candidate for the key of the last row in the table. Next, CheckAndPut is attempted at $R+1$, $R+2$, ... , $R+i$ successively. If it succeeded at $R+i$, then it failed at all $R+j$, where $1 \leq j < i$, therefore all rows $R+j$ existed (and remained existent due to Invariant 3). CheckAndPut at $R+i$ succeeded, so row $R+i$ was created and the procedure was terminated (CheckAndPut is an atomic operation, given Guarantee 1 from Section 2.3.3). The table is tight from 1 to R, and also from $R+1$ to $R+i$ because $R+1$, ... , $R+i$ are consecutive rows due to Invariant 2 (and there is no integer between two consecutive integers). □

Corollary 1. *At the moment the CheckAndPut of the append operation creates a row in the Timestamp List, that row is the last one in the table.*

Proof. Given Claim 1, Claim 2, and Invariant 2, the newly created row cannot be elsewhere than immediately after the previous last row, hence it is the new last row. □

The Corollary above demonstrates how the append operation has the correct behavior.

3.4.2 Transaction processing correctness

The Certification Phase in the processing of a transaction must correctly identify write-write conflicts. For this purpose, a Scan of the Timestamp List is performed. Given Guarantee 2 from Section 2.3.3, that Scans have the Read Committed isolation level, data read during a Scan could be stale. Hence conflict searching should be correct despite the presence of stale data. In this section, we show what data can be stale in the Timestamp List, and demonstrate how this does not negatively affect conflict checks.

Invariant 5. The only cell that is overwritten in a row in the Timestamp List is the `status` cell. Moreover, this cell can be changed from "active" to "committed" or "aborted". If `status` is not "active", it will not be overwritten.

A consequence of the Invariant above is that the only column with possibly stale data is the `status` column.

Claim 3. *Given a transaction T_i with non-null start and end timestamps, if there is a write-write conflicting transaction T_j with T_j^{end} in the lifetime of T_i, then it is impossible that both T_i and T_j commit.*

Proof. In the Certification Phase of T_i, the Scan at Step 1 will find the timestamp T_j^{end} and execute Steps 1.1–1.3, because before the Scan begins the Timestamp List is tight from T_i^{start} to T_i^{end} (with T_j^{end} in between), hence this will be seen by the Scan. That is, this fact is not affected by the Read Committed isolation level of Scans.

The row for T_j^{end} in the Timestamp List includes T_j^W in the `writeset` column. Because this data is never overwritten (Invariant 5), it is never stale. The `status` at row T_j^{start}, though, can be stale only if its value is "active".

We show that if T_j commits, then T_i cannot commit. Let the `status` of T_j^{start} be "committed". This information might not be seen by the Scan of Step 1. That is, the `status` cell of T_j given by the Scan could be stale: the value "active" could be seen instead of the actual "committed". However, Step 1.2 in the Certification Phase aborts transaction T_i in both cases of T_j "active" or "committed", hence T_i does not commit. □

3.4.3 Read versions and Snapshot Isolation

For correct Snapshot Isolation, we must guarantee that a transaction reads from its snapshot. The snapshot of a transaction T_i is the database state at the moment T_i^{start} is created. The snapshot of T_i can be represented by the

set of all data item versions that were committed before T_i^{start}. We discuss how read operations in HAcid are equivalent to reading from snapshot.

First, we show how the procedure ReadVersion from Section 3.3.5 correctly determines σ. Then, we demonstrate that reading σ versions is equivalent to reading from snapshot.

Invariant 6. Every write operation in any transaction T_j in HAcid uses T_j^{start} as the version number for the data item version.

Definition 20. Let $x(k)$ be a data item version, where $k = T_j^{start}$ for some transaction T_j (guaranteed from Invariant 6). The commit time CT is defined as

$$CT(x(k)) = \begin{cases} T_j^{end}, & \text{if } T_j \text{ committed,} \\ \infty, & \text{otherwise.} \end{cases}$$

That is, the commit time of a data item version is the end timestamp of the transaction that wrote it, if that transaction committed. Notice that this definition is time-dependent, because every data item version written by a transaction T_j starts with its $CT = \infty$, which later can become T_j^{end} once T_j commits. Since a committed transaction stays committed, if $CT(x(k))$ is a finite value, then $CT(x(k))$ will remain forever unchanged. The definition above allows us to suitably redefine σ as

$$\sigma(x, T_i) = \max\{T_j^{start} : x \in T_j^W \wedge CT(x(T_j^{start})) < T_i^{start}\}.$$

Claim 4. *Given the key/value pair* $\langle (R, \texttt{commit-time}, v), z\rangle$ *from a user table* t*, if* $z \neq 0$*, then* $CT(R(v)) = z$*.*

Proof. The Update Phase implies that $v = T_j^{start}$ and $z = T_j^{end}$ for some transaction T_j, since no other procedure writes nonzero values to `commit-time`. In that case, T_j committed. Therefore $c_j \in T_j$, so $CT(R(T_j^{start})) = T_j^{end}$. □

Claim 5. *The procedure ReadVersion(*t*,* R*,* T_i^{start}*) returns* $\sigma((t, R), T_i)$*.*

Proof. The procedure inspects key/value pairs $\langle (R, \texttt{commit-time}, v), z\rangle$ of the cell $(R, \texttt{commit-time})$ in decreasing order of version numbers v. Since $v = T_j^{start}$ for some transaction T_j, the first v such that the constraints $(t, R) \in T_j^W$ and $CT(R(T_j^{start})) < T_i^{start}$ are satisfied is precisely $\sigma((t, R), T_i)$. The constraint $(t, R) \in T_j^W$ is obviously satisfied because all key/value pairs are from row R. So only the second constraint needs to be checked by the procedure.

If $z = 0$ in Step 2.1 in the procedure, then either $T_j^{end} < T_i^{start}$ or $T_j^{end} > T_i^{start}$ or even $T_j^{end} = \perp$. If $T_j^{end} = \perp$, when this end timestamp is created it will be greater than T_i^{start} since the Timestamp List is

tight. Furthermore $T_j^{end} > T_i^{start}$ implies $CT(R(T_j^{start})) > T_i^{start}$, hence the constraint is not satisfied. Thus only case $T_j^{end} < T_i^{start}$ is relevant, and Recovery in Step 2.1.1 makes sure that T_j aborts or commits. If T_j committed, Step 2.1.2 sets $z := CT(R(T_j^{start}))$.

If $z \neq 0$ in Step 2.2, then by Claim 4 we have $CT(R(T_j^{start})) = z$, so it suffices that $z < T_i^{start}$ to determine that $T_j^{start} = \sigma((t, R), T_i)$. This is done in Step 2.2. □

Definition 21. A data item version $x(k)$ is in the snapshot of a transaction T_i if and only if $CT(x(k)) = \max\{CT(x(l)) : x(l) \neq \perp \wedge CT(x(l)) < T_i^{start}\}$.

The previous definition means that $x(k)$ is in the snapshot of T_i only if $x(k)$ was the latest committed version of x which committed before T_i started. Notice that the set $\{CT(x(l)) : x(l) \neq \perp \wedge CT(x(l)) < T_i^{start}\}$ on the right-hand side is not time-dependent, since finite $CT(x(l))$ are immutable.

Claim 6. *Let $\sigma(x, T_i)$ be obtained from the procedure ReadVersion. The data item version $x(\sigma(x, T_i))$ is in the snapshot of T_i.*

Proof. By definition, $\sigma(x, T_i) = \max\{T_j^{start} : x \in T_j^W \wedge CT(x(T_j^{start})) < T_i^{start}\}$. In other words, $\sigma(x, T_i)$ is equal to the start timestamp T_M^{start} of some transaction T_M that committed at $CT(x(T_M^{start})) = T_M^{end} < T_i^{start}$. We affirm that $CT(x(T_M^{start}))$ is the maximum in $\{CT(x(l)) : x(l) \neq \perp \wedge CT(x(l)) < T_i^{start}\}$, as required for Definition 21.

The proof is by contradiction. Suppose there is $CT(x(T_z^{start}))$ greater than $CT(x(T_M^{start}))$ where $x(T_z^{start}) \neq \perp$ and $CT(x(T_z^{start})) < T_i^{start}$. So $T_M^{end} < T_z^{end}$ by the definition of CT. From the definition of σ, it holds that $T_z^{start} < T_M^{start}$, because T_M^{start} is the largest such start timestamp. Hence $T_z^{start} < T_M^{start} < T_M^{end} < T_z^{end}$, therefore the two transactions T_z and T_M are concurrent. Moreover, $x \in T_M^W \cap T_z^W$, so these two transactions have a write-write conflict. By the First-Committer-Wins rule, T_M commits and T_z aborts. Hence $CT(x(T_z^{start})) = \infty$, contradicting our assumption.

This means that $CT(x(T_M^{start}))$ is the maximum for Definition 21, so $x(T_M^{start})$ is in the snapshot of T_i, where $T_M^{start} = \sigma(x, T_i)$. □

The consequence of the Claim above is that read operations in HAcid are snapshot isolated.

3.4.4 Transaction atomicity

Atomicity is perhaps the most important ACID property. If one write operation in a transaction gets committed, then also all other write operations in that transaction should commit. Here we demonstrate how HAcid transactions satisfy atomicity.

Definition 22. A data item version $x(k)$ is *visible* if $k = \sigma(x, T_i)$ for some (hypothetical) transaction T_i.

Claim 7. *Let T_i be a HAcid transaction, and $x(T_i^{start})$ a data item version that is visible, written by T_i. Then all data item versions written by T_i are visible.*

Proof. Because $x(T_i^{start})$ is visible, $T_i^{start} = \sigma(x, T_l)$ for some hypothetical transaction T_l. The ReadVersion procedure returns $\sigma(x, T_l) = T_i^{start}$ only if the value of $(x, \texttt{commit-time}, T_i^{start})$ is non-null (see Step 2.2). This means that the Update Phase of T_i has been executed, and consequently T_i committed writing "committed" to `status` of row T_i^{start} in the Timestamp List.

Therefore, if T_l were to process a read operation of any data item y that was written by T_i, we would have $\sigma(y, T_l) = T_i^{start}$. This is true because $T_i^{start} = \sigma(x, T_l)$ implies that T_i^{start} is the largest start timestamp such that $T_i^{end} < T_l^{start}$. Hence y is also visible. □

3.5 Implementation and usage guide

HAcid is an open-source library for HBase client applications, enabling Snapshot Isolation multi-row transactions. The source code and library download can be found at [41], where its documentation is also available. The API is minimalistic and designed to be easy to understand and use.

Here we explain how HAcid can be used in practice for an HBase data store, and give some details about its implementation. The HAcid API is a Java library (jar file) in many ways similar to the standard HBase client Java library. Section 3.1.2 is a good representation of the connections between HAcid and an existing HBase installation.

3.5.1 HAcid API

The HAcid API resembles the HBase API. In Java applications that access HBase, it is typical to encounter the classes `HTable`, `Put`, and `Get`. `HTable` represents a table in HBase, `Put` represents a write operation, and `Get` represents a read operation. These are important and common classes. The example below shows the use of these classes.

```
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "mytable");

Put p = new Put(Bytes.toBytes("row1"));
```

```
p.add(Bytes.toBytes("fam1"),
      Bytes.toBytes("qual1"),
      Bytes.toBytes("val"));
table.put(p);

Get g = new Get(Bytes.toBytes("row1"));
g.addColumn(Bytes.toBytes("fam1"),Bytes.toBytes("qual1");
Result r = table.get(g);

table.close();
```

The HAcid API follows a similar style. The most important classes are `HAcidClient`, `HAcidTable`, `HAcidTxn`, `HAcidGet`, and `HAcidPut`. The class `HAcidClient` encapsulates the majority of management algorithms, such as conflict testing, Timestamp List append, ReadVersion, Recovery, etc. `HAcidTable` represents an HBase user table enabled for transactions, i.e., with the `commit-time` column. `HAcidTxn` represents a transaction. `HAcidGet` and `HAcidPut` represent, respectively, read and write operations to be included into transactions.

For instance, a simple use case where a transaction simply reads and writes to a row works as follows.

```
Configuration conf = HBaseConfiguration.create();
HAcidClient client = new HAcidClient(conf);
HAcidTable table = new HAcidTable(conf, "mytable");

HAcidTxn txn = new HAcidTxn(client);

HAcidGet g = new HAcidGet(table, Bytes.toBytes("row1"));
g.addColumn(Bytes.toBytes("fam1"), Bytes.toBytes("qual1"));
Result r = txn.get(g);

HAcidPut p = new HAcidPut(table, Bytes.toBytes("row1"));
p.add(Bytes.toBytes("fam1"),
      Bytes.toBytes("qual1"),
      Bytes.toBytes("val"));
txn.put(p);

txn.commit();

table.close();
client.close();
```

The classes `HAcidPut` and `HAcidGet` mimic the interface of HBase classes `Put` and `Get`, so they are easy to use. The methods `HAcidTxn.get()` and `HAcidTxn.put()` execute read and write operations in the transaction. The modifications made by transaction `txn` are invisible to other transactions until one calls `txn.commit()`. Classes `HAcidTxn` and `HAcidClient` coordinate the three phases and three events of transaction processing.

Note that version numbers cannot be specified in `HAcidGet` and `HAcidPut`. HAcid manages version numbers to ensure correct Snapshot Isolation, therefore its API does not give to the user access to version numbers. We also discourage the access to version numbers in a HAcid user table through HBase's conventional API, except in read-only use cases.

3.5.2 Installation

From the perspective of the API, the user does not need to manage the installation of HAcid when it is used for the first time. It is sufficient to import the HAcid library into the application, and initialize instances of `HAcidClient` and `HAcidTable`. The constructor of `HAcidClient` verifies if the provided HBase data store already has a Timestamp List table present, and creates it if necessary. We refer to the creation of the Timestamp List as HAcid's *initialization*. The constructor of `HAcidTable` verifies if the provided HBase table already has the `HAcid` column family (to include the qualifier `commit-time`), and creates such family if necessary. This method is called *preparation* of a user table.

3.5.2.1 Initialization

Initialization is a simple method, creating the Timestamp List table with one initial row. The purpose of this dummy row is to simplify the append operation, which assumes that the Timestamp List has at least one row. The dummy row is irrelevant to transaction processing. The only non-null cell in this row is `type`, with value "end", so that the append operation can detect its existence. The row key is 1, hence timestamp serving starts from 2 onwards.

3.5.2.2 Preparation

User tables need to be prepared before being targeted by HAcid transactions. Every user table must include the `commit-time` column in the `HAcid` column family, and every version number must correspond to some transaction.

If the user table does not exist before HAcid is installed, then it is prepared at the time it is created, simply by including the `HAcid` column family. However, if the user table already had data, we need to change the version numbers of existing data items. In this case, for every cell (R, C) in the table, we copy the latest version (R, C, v) to $(R, C, 1)$ and remove all versions except 1. That is, the version number 1 is used for all initial data items in the user table.

In HAcid, version numbers are important to indicate the age of data for Snapshot Isolation, hence 1 (the smallest possible version number) is used for initial data. Preparation of a large existing user table might be an expensive procedure and is not safe against interruptions, for instance due to crashes. Hence it is recommended to use empty or small user tables when HAcid is installed.

We also discourage the use of the HBase API on HAcid-prepared user tables, since doing so could violate ACID properties of HAcid transactions. If a user table is prepared for HAcid, it is safest to use only the HAcid API, even for transactions with a single operation. However, read operations are an exception: HBase Gets can be performed on HAcid user tables with no side effects.

3.5.3 Optimizations

There are two optimizations in HAcid that are important to mention. One is the use of σ versions in read operations, and the other is the reversed row key schema for the implementation of the Timestamp List.

Recall that the read version $\sigma(x, T_i)$ is the largest start timestamp of the transaction that wrote to x and committed before T_i^{start}. In Section 3.4.3 we proved that this is equivalent to the start timestamp of the newest committed transaction that wrote to x and committed before T_i^{start}.

Thus, for a read operation, we could simply search in row x for the largest `commit-time` value smaller than T_i^{start}. This would require inspecting all versions of the column `commit-time` in row x. However, to find $\sigma(x, T_i)$, searching for the largest appropriate version number is advantageous given how HBase organizes data.

The versions of a cell in HBase are ordered in decreasing order of version numbers. Consequently, the latest version has the largest version number, so it is the first one to be returned when the cell is read. Hence searching σ does not require one to inspect all versions in the cell.

The other optimization is to use a reversed row key schema for the Timestamp List. Recall that for the append operation we need to find the end of the table, and that HBase does not provide a pointer to the last row. How-

ever, HBase allows one to directly access the first row in a table. This is done by executing a Scan with undetermined start row key. The first row returned by the Scan is the first row of the table at the time the Scan was created.

Hence, in the implementation of the Timestamp List, a timestamp t is represented by row key $M - t$, where M is a large enough constant. The table is prepended instead of appended. To prepend, a candidate for the first row is obtained, then through CheckAndPut we try to insert a row immediately before the first row. This process repeats if the CheckAndPut failed, analogous to the append operation.

Prepending is more efficient than appending, and the reversed row key schema does not disturb HAcid's functionality, as long as row keys $M - t$ are converted to timestamps t.

Chapter 4

Performance evaluation

Good performance is an important aspect for HAcid's lightweight objectives. In particular, we are interested in minimizing the overhead of HAcid in relation to HBase, because HAcid is built on HBase.

In this chapter we present some benchmarks of transaction processing in HAcid. First we study specific aspects of transaction processing, then we examine the overall performance of the system.

We use five identical compute nodes from a high performance cluster, the Triton cluster at Aalto University School of Science. The hardware of these compute nodes are described in Table 4.1. One node hosts the HBase master, while the remaining nodes host HBase region servers. The master node is also where the benchmark applications are executed. On the software side, we used the Cloudera [15] CDH3 Update 3 distribution, which includes HBase 0.90.4 and Hadoop 0.20.2.

4.1 Microbenchmarks

In this experiment we are interested in measuring the overhead of using HAcid. These are called microbenchmarks and are inspired by Percolator's microbenchmarks [44]. We are interested in the performance of a transaction with a single (read or write) operation compared to the performance of the

Processor	2x Intel Xeon X5650 2.67GHz
RAM	48 GB of DDR3-1066 memory
Storage	About 830 GB of local diskspace (software RAID 0)
Chassis/Mobo	HP SL390s G7

Table 4.1: Hardware description of the machines used for the experiments.

	HBase	HAcid	Relative
Read Latency (ms)	0.34	3.36	x9.8
Write Latency (ms)	1.12	7.79	x6.9

Table 4.2: Average latency of HBase operations and HAcid transactions. The column "Relative" indicates the overhead of using HAcid. For instance, a HAcid transaction with a single write operation is about 7 times slower than its corresponding HBase write operation.

corresponding HBase operation alone. As expected, the latency of executing an operation through a HAcid transaction will always be higher than performing it purely in HBase. Ideally, this gap should be as small as possible.

We measure the execution time of a set of random HBase write operations (Puts), and compare with the execution time of the corresponding set of HAcid transactions each with a single write operation. The same is done for read operations (Gets). All operations are run serially, that is, there is no concurrency involved. Each operation targets a random row of a table built for this experiment.

The HBase table contains 10000 rows and three columns in one column family. The values in cells are random integers. We executed 100000 operations of each kind (HBase Put, HBase Get, HAcid transaction with a single write, HAcid transaction with a single read). The results are given in Table 4.2.

HAcid write-only transactions were 6.9 times slower than their HBase counterparts, while HAcid read-only transactions were 9.8 times slower. The results are predictable given the fact that some HBase read and write operations are involved during the processing of a HAcid transaction.

For instance, consider the processing costs of a write-only HAcid transaction. Retrieving a start timestamp requires at least one Put operation, executing the user's write operation is one Put, and retrieving an end timestamp requires at least one Put. Then, for conflict checking, at least the cost of one Get (within the Scan) is needed, while the commit decision is one Put. Finally, to update the user table row, one Put is executed. In total, one Get and five Puts are necessary to the processing of such transaction. Consequently, using the latencies for HBase operations from Table 4.2, a HAcid write-only transaction takes at least 5.94 ms to complete. The measured 7.79 ms latency is close (just 31% higher) to the expected minimum.

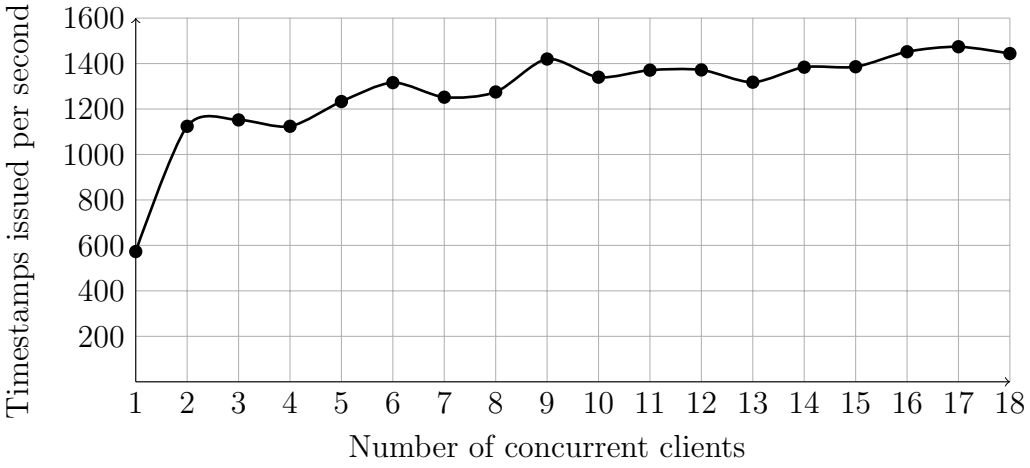

Figure 4.1: Performance of the append operation to issue timestamps to concurrent clients.

4.2 Timestamp throughput

This experiment focuses on measuring the performance of the append operation on the Timestamp List as a timestamp issuing service. We repeatedly request start timestamps from HAcid to measure the throughput of timestamps. We vary the number of concurrent clients requesting timestamps to study how well HAcid handles concurrent scenarios. HAcid should be able to render higher timestamp throughput when the number of concurrent clients increases.

Each client requests 50000 start timestamps, and we execute up to 18 concurrent clients. Figure 4.1 displays a plot of the throughput values obtained in the experiment. Apparently, the append operation on the Timestamp List benefits from concurrency: the throughput for two clients is about twice the throughput for one client. The increase for more than two clients is moderate. In general, one can expect above 1000 timestamps per second being issued.

These results also allows us to calculate the average latency of timestamp issuing. In the case of only one active client in the experiment, the timestamps are requested serially. Therefore the throughput of 556 timestamps per second implies an average latency of 1.8 ms.

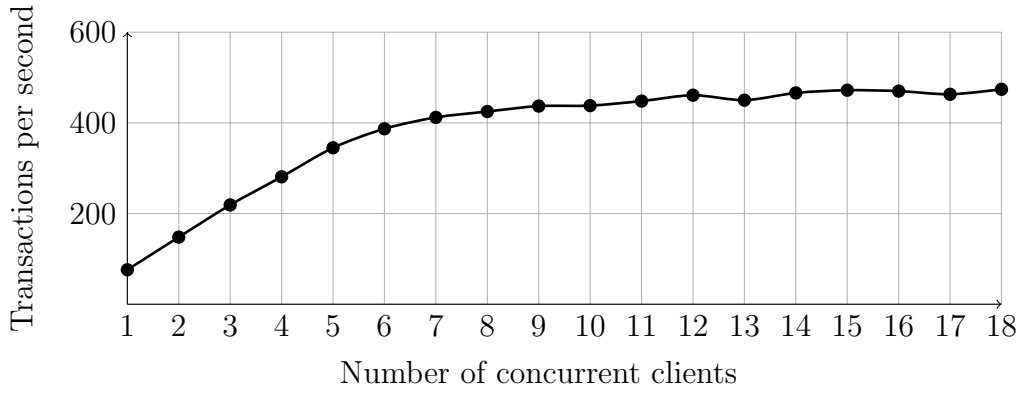

Figure 4.2: HAcid transactions throughput from concurrent clients.

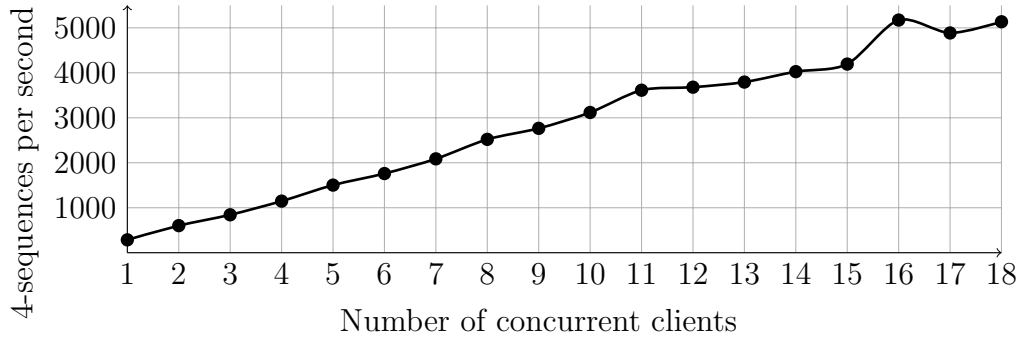

Figure 4.3: Throughput of sequences of four HBase operations. Sequences are simply non-ACID transactions. This figure is comparable to Figure 4.2.

4.3 Transaction throughput

To assess the performance of HAcid in more realistic workloads, we measure the throughput of randomly generated transactions. This experiment allows us to estimate how many transactions per second (TPS) can HAcid provide.

We run 18 concurrent clients that generate random transactions and submit them to HAcid for commitment. Similarly to the microbenchmarks, we use a randomly generated test table. The table contains 200000 rows and three columns, and each client executes 13000 transactions with four operations, which are random reads and writes. Transactions can therefore conflict in this experiment. Figure 4.2 is a plot of the throughput of transactions obtained from concurrent clients. Figure 4.3 is the corresponding plot of throughput when HAcid transactions are not used, but simply sequences of four HBase operations are executed.

The results indicate that throughput is moderate: between 300 and 500 transactions per second. However, HAcid clearly benefits from concurrency: throughput scales almost linearly with the number of concurrent clients, up to seven clients. Throughput does not scale beyond that because timestamp issuing does not scale (see Figure 4.1). Figure 4.3 shows how throughput of simple HBase operations (no ACID properties) scales linearly.

4.4 Discussion

The results imply that HAcid does not offer transactions with high performance. HAcid transactions will necessarily have higher latency than HBase data operations. That said, HAcid has small overhead compared to other transaction systems built on top of HBase. For instance, in similar configurations, HBaseSI [60] provides at most 500 timestamps per second and about 200 transactions per second, while HAcid performs at least 1000 timestamps per second and about 400 transactions per second.

Managing the Timestamp List is definitely the bottleneck in HAcid. All transactions access the Timestamp List, for two purposes: to retrieve timestamps and to manage transactional metadata. In other transaction systems, these different responsibilities are distributed to different servers. In HAcid, when many clients append the Timestamp List, the region server managing the last parts of the table becomes a hotspot in the network. Completed transactions have written at least three times to the Timestamp List.

Multiple clients appending the Timestamp List might cause contention for the last row. If one client is slower to communicate with HBase than a group of other clients, then to append it will likely hang until the remaining clients have appended. Hence, there are no fairness properties among concurrent clients appending to the Timestamp List.

Nevertheless, one must remember that high performance transactions are not a feature that HAcid aims to offer. In the context of incremental updates to a data repository, if many transactions are being submitted, then likely batch processing through a framework like MapReduce would be more suitable. HAcid is advantageous when a light workload of transactions is necessary to update the database.

Chapter 5

Conclusion

In this chapter, we discuss opportunities for improvement in Section 5.1 and review the work done for this Thesis in Section 5.2.

5.1 Future work

HAcid can be easily modified to include new features. For instance, multiple instances of HAcid can coexist within one HBase installation. This can be achieved by having multiple Timestamp Lists, each one related to one instance of HAcid. The instances must be independent, in the sense that a user table is managed by only one instance. In this fashion, we can easily distribute the load of timestamp serving. That is, if we know beforehand the set of users tables that a collection of transactions will target, we can create a HAcid instance for that collection of transactions.

If no user table intrusion is desired, one can substitute the `commit-time` column with a separate table. The table should allow efficient lookup of the commit time of each data item in user tables. The disadvantage of this approach is the decrease in performance compared to the `commit-time` cache column solution.

There are two HAcid features that could be added with reasonable effort but were left out from the version described by this Thesis. These are: the support for Serializability as the isolation level, and garbage collection of versions in user tables.

5.1.1 Serializable Snapshot Isolation

There has been continuous research [11, 12, 21, 40, 43, 46, 57] on how to adapt Snapshot Isolation for guaranteeing Serializability. One of the approaches is

Write-Snapshot Isolation (WSI), by the ReTSO authors Yabandeh and Ferro [57], and convenient to be implemented in HAcid. Their approach consists of minor modifications to the original SI. The main remark is that write-write conflict detection is neither necessary nor sufficient for Serializability. Write-write conflicts concern only the writesets of transactions, but also readsets should be taken into account if Serializability is required.

Their solution consists in replacing write-write conflict detection with read-write conflict detection, that considers readsets as well as writesets, defined as follows.

Definition 23 (Read-Write Conflict)**.** Two timestamped transactions T_i and T_j have a *read-write conflict* if all the following conditions are met:

1. T_i and T_j are concurrent;
2. $T_i^W \cap T_j^R \neq \emptyset$, where $T_i^{end} < T_j^{end}$;
3. $T_i^W \neq \emptyset$;
4. $T_j^W \neq \emptyset$.

That is, the second transaction T_j reads what the first transaction T_i wrote, so by the First-Committer-Wins rule, T_j should abort if T_i committed.

To support WSI in HAcid, we would need search for read-write conflicts instead of write-write conflicts. The readsets of transactions would need to be recorded in Timestamp List rows, as writesets are. It is also necessary to change the definition of a read version, because σ read versions are not necessarily in the snapshot of WSI transactions. This is due to the fact that Claim 6 assumes that write-write conflicts determine abortion. Read version σ should be replaced with the traditional approach: the start timestamp of the newest committed transaction that wrote the same data item.

5.1.2 Garbage collection

After many transactions modify a data item, there will be many versions of that data item. For better performance, it is useful to purge old versions of the data item. Purging frees space in the data store and speeds up read operations, so that less versions need to be inspected.

The challenge with purging is to determine when a version is old enough. We propose one method of detecting old versions, but neither this nor purging is implemented in HAcid. We present a sufficient condition for version oldness.

Definition 24. Let $x(T_i^{start})$ and $x(T_j^{start})$ be data item versions of x. Version $x(T_j^{start})$ *is a successor of* $x(T_i^{start})$ if and only if $T_j^{start} > T_i^{start}$ and $c_j \in T_j$.

Definition 25. Data item version $x(T_i^{start})$ is safe to purge if there exists a successor $x(T_j^{start})$ and for every transaction T_k such that $T_i^{start} < T_k^{start} < T_j^{start}$ it holds that $T_k^{end} \neq \perp$.

Definition 25 is the condition to purge. In other words, a successor of a version $x(T_i^{start})$ is a newer version $x(T_j^{start})$ that is committed, and it is safe to delete $x(T_i^{start})$ if all transactions that started between T_i^{start} and T_j^{start} have already ended. Consequently, these intermediate transactions cannot anymore read the version $x(T_i^{start})$, and no future transaction will have $x(T_i^{start})$ in its snapshot because they will favor the newer version $x(T_j^{start})$.

As future work, the condition can be implemented into HAcid. Purging is specially important if all versions of a data item are inspected to search the (non-σ) read version, for instance when Write-Snapshot Isolation is required.

5.2 Discussion

We have presented a new lightweight client library for HBase that enables multi-row SI transactions. Other transaction systems, such as HBaseSI, have demonstrated how transactions can be provided using solely the HBase client API. The goal in designing HAcid is to be as lightweight as possible inside the class of transaction systems built on top of HBase.

HAcid is focused on achieving minimality in aspects such as data structures, changes in client and server sides, API, and performance. HAcid employs only one metadata table and one cache column. Changes in the client-side are represented by importing a single library to the application. Changes in the server-side consist of installing the metadata table and preparing user tables. The API resembles the standard HBase API and is succinct. Furthermore, performance is apparently better than that of other similar systems.

On the other hand, there are trade-offs when attempting to make these aspects lightweight. A system that focuses on low overhead of transactions likely requires many changes in the server-side of an HBase installation. Little intrusion to user tables potentially requires more metadata tables and data structures, which might lead to performance drawbacks. It might be impractical to achieve minimality in all aspects.

We believe to have achieved a good balance of all the aforementioned issues in HAcid. The result is a solid system that is suitable for different kinds of workloads. Furthermore, HAcid adopts the Cloud philosophy of fault-tolerance: recovery is a central mechanism in HAcid and the fact that the Timestamp List is served by an HBase region server implies that HBase will automatically assign a server to manage it, regardless of failures. In other

transaction systems, the administrator might need to manually manage the timestamp oracle.

Our transaction system is also flexible for being adapted in interesting ways. Not many transaction systems address the problem of garbage collection, and we have seen in this chapter how this feature can be easily included into HAcid. Serializability is also reachable after a few modifications to the system. If better performance is required, one can manually manage a collection of independent Timestamp Lists to increase throughput.

The ideas that comprise HAcid, in particular the Timestamp List, can be extended to other data stores. There are other Extensible Record Stores besides HBase that can implement the transaction processing described in HAcid. However, the structure of the Timestamp List does not depend on many characteristics of HBase. It would be interesting to implement the Timestamp List separately from HBase and assess its performance as a timestamp oracle and a transactional metadata repository.

Software systems for Cloud Computing, such as HBase, are in active evolution. Since the dawn of NoSQL data stores, many have believed that there is "no size that fits all needs" when it comes to databases. On one side are the traditional relational database management systems, equipped with all necessary features. On the other side are lighter data stores that offer less features in order to provide high performance when dealing with large quantities of data. The gap between these two classes of databases is often large. HAcid on HBase is situated between the two extremes, offering to database administrators a new benefit that can be achieved with HBase.

Bibliography

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[2] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[5] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4): 465–483, 1983.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[7] Dhruba Borthakur. HDFS Architecture, June 2012. URL http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.html.

[8] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand S. Aiyer. Apache Hadoop goes realtime at Facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*, pages 1071–1080, 2011.

[9] Eric A. Brewer. Towards robust distributed systems. (Invited Talk). In *Principles of Distributed Computing*. ACM, July 2000.

[10] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[11] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), 2009.

[12] Michael James Cahill and University of Sydney. School of Information Technologies. Serializable isolation for snapshot databases, 2009. URL http://ses.library.usyd.edu.au/handle/2123/5353. Thesis.

[13] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, 2006.

[15] Inc. Cloudera. Cloudera web page, August 2012. URL http://www.cloudera.com/.

[16] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. *CoRR*, abs/1008.3751, 2010.

[17] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010*, pages 163–174, 2010.

[18] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 715–726, 2006.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Operating System Design and Implementation (OSDI 2004)*, pages 137–150, 2004.

[20] Ian Eslick, Andre DeHon, and Thomas Knight. Guaranteeing idempotence for tightly-coupled, fault-tolerant networks. In Kevin Bolding and Lawrence Snyder, editors, *Parallel Computer Routing and Communication*, volume 853 of *Lecture Notes in Computer Science*, pages 215–225. Springer Berlin / Heidelberg, 1994.

[21] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[22] The Apache Software Foundation. Apache Accumulo web page, June 2012. URL http://accumulo.apache.org/.

[23] The Apache Software Foundation. Apache Cassandra web page, June 2012. URL http://cassandra.apache.org/.

[24] The Apache Software Foundation. Apache HBase web page, May 2012. URL http://hbase.apache.org/.

[25] The Apache Software Foundation. Apache Hadoop web page, May 2012. URL http://hadoop.apache.org/.

[26] The Apache Software Foundation. Apache ZooKeeper home page, January 2012. URL http://zookeeper.apache.org/.

[27] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP) 2003*, pages 29–43, 2003.

[29] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[30] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin / Heidelberg, 1978.

[31] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[32] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[33] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11. USENIX Association, 2010.

[34] Hypertable Inc. Hypertable web page, June 2012. URL http://www.hypertable.org/.

[35] Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-free transactional support for large-scale storage systems. pages 176–181, June 2011. URL http://dx.doi.org/10.1109/DSNW.2011.5958809.

[36] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[37] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[38] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[39] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research*, pages 123–133, 2011.

[40] Yang Lu. Serializable Snapshot Isolation in Shared-Nothing, Distributed Database Management Systems. Master's thesis, Brown University, Providence, Rhode Island, 2012.

[41] Andre Medeiros. HAcid source code repository at Bitbucket, August 2012. URL https://bitbucket.org/staltz/hacid.

[42] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[43] Vinit Padhye and Anand Tripathi. Scalable Transaction Management with Serializable Snapshot Isolation on HBase. Technical report, Department of Computer Science, University of Minnesota, Twin Cities, Feb 2012. URL http://ajanta.cs.umn.edu/papers/serializable-hbase-transactions.pdf.

[44] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, (OSDI) 2010*, pages 251–264, 2010.

[45] Kash Rangan. The Cloud Wars: $100+ billion at stake. Tech. rep., Merrill Lynch, May 2008.

[46] Stephen Revilak, Patrick E. O'Neil, and Elizabeth J. O'Neil. Precisely Serializable Snapshot Isolation (PSSI). In *Proceedings of the 27th International Conference on Data Engineering (ICDE), 2011*, pages 482–493, 2011.

[47] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010*, pages 448–454, 2010.

[48] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–10, 2010. doi: http://doi.ieeecomputersociety.org/10.1109/MSST.2010.5496972.

[50] Eljas Soisalon-Soininen and Sami El-Mahgary. T-106.5241 Distributed Databases course at Aalto University, 2011. Lecture notes.

[51] Christof Strauch. NoSQL Databases. *Lecture Notes Stuttgart Media*, pages 1–8, 2010. URL http://nosql-database.org/.

[52] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. *PVLDB*, 3(1):506–517, 2010.

[53] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. Technical Report IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, February 2010. http://www.globule.org/publi/CSTWAC_ircs53.html.

[54] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Consistent Join Queries in Cloud Data Stores. Technical Report IR-CS-068, Vrije Universiteit, Amsterdam, The Netherlands, January 2011. http://www.globule.org/publi/CJQCDS_ircs68.html.

[55] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, original edition, June 2009. ISBN 0596521979. URL http://www.worldcat.org/isbn/0596521979.

[56] ANSI X3.135-1992. American National Standard for Information Systems – Database Language – SQL, November 1992.

[57] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12*, pages 155–168, 2012.

[58] Yahoo! GitHub repository. Omid: Transactional Support for HBase., May 2012. URL http://github.com/yahoo/omid.

[59] Chen Zhang and Hans De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase. In *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing*, pages 177–184, 2010.

[60] Chen Zhang and Hans De Sterck. HBaseSI: Multi-row Distributed Transactions with Global Strong Snapshot Isolation on Clouds. *Scalable Computing: Practice and Experience*, 12(2), 2011.