Supporting Multi-row Distributed Transactions with Global Snapshot Isolation Using Bare-bones HBase

Chen Zhang
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
Email: c15zhang@cs.uwaterloo.ca

Hans De Sterck
Department of Applied Mathematics
University of Waterloo
Waterloo, Canada
Email: hdesterck@math.uwaterloo.ca

***Abstract*—Snapshot isolation (SI) is an important database transactional isolation level adopted by major database management systems (DBMS). Until now, there is no solution for any traditional DBMS to be easily replicated with global SI for distributed transactions in cloud computing environments. HBase is a column-oriented data store for Hadoop that has been proven to scale and perform well on clouds. HBase features random access performance on par with open source DBMS such as MySQL. However, HBase only provides single atomic row writes based on row locks and very limited transactional support. In this paper, we show how multi-row distributed transactions with global SI guarantee can be easily supported by using bare-bones HBase with its default configuration so that the high throughput, scalability, fault tolerance, access transparency and easy deployability properties of HBase can be inherited. Through performance studies, we quantify the cost of adopting our technique. The contribution of this paper is that we provide a novel approach to use HBase as a cloud database solution with global SI at low added cost. Our approach can be easily extended to other column-oriented data stores.**

***Keywords*-Cloud Database; HBase; Snapshot Isolation**

I. INTRODUCTION

Snapshot isolation (SI), proposed by Berenson et al. [4], is an important database transactional isolation level which guarantees that all reads made in a transaction will see a consistent snapshot of the database that remains unaffected by any other concurrent update transactions. The transaction itself will successfully commit only if none of its updates conflict with any concurrent updates committed since that snapshot. For example, consider a transaction T1 that reads data items x and y and then reads x, writes x and commits. Meanwhile, another transaction T2 writes x (when T1 reads y) and commits. In this scenario, with a snapshot taken before T1's first operation, T1 will get the same value for both reads of x even if T2 writes to x in between T1's reads of x. T2 will attempt to commit before T1, and with the "first-committer-wins" rule, T2 succeeds and T1 fails, since T1's update on x conflicts with that of T2 which occurs after the snapshot for T1 was taken, and before T1 commits.

The benefit of using SI is that reads will never be blocked, resulting in increased concurrency and high throughput while still avoiding various kinds of data inconsistency referred to as anomalies [4]. However, SI does not guarantee serializability (recently, in 2008, [6] proposed a novel way to support serializability on existing DBMS that support SI) and will have the "Write Skew" anomaly, in which case separate transactions update different items bounded by a shared constraint and commit successfully, resulting in a global database state that breaks the bounding constraint. This anomaly can be tolerated if its occurrence is minimized due to proper programming and when better system performance is desired, especially in read-heavy loads. Another phenomenon allowed by SI is that the second of two consecutive transactions initiated by the same client or user may not see the updates of the first (successful) transaction, since SI does not require the snapshot that a transaction reads from to be the most recently committed snapshot before the transaction starts. Major DBMS including Microsoft SQL Server, Oracle, MySQL, PostgreSQL, Firebird, H2, Interbase, Sybase IQ, and SQL Anywhere support SI due to its performance benefits.

With the proliferation of cloud computing, it would be good for applications that require SI to take advantage of what cloud computing can provide. Unfortunately, there is currently no solution for traditional DBMS to be easily replicated with global SI for distributed transactions (transactions involving two or more network hosts) in cloud computing environments. For example, the recent commercial release of Microsoft SQL Azure supports a cloud-based relational database service built on SQL Server technologies. However, it is essentially based on database partitioning into a shared-nothing architecture and does not support distributed transactions, let alone global SI, across multiple partitions. Note that SQL Azure does support SI for transactions within a single database. Oracle 11g uses Oracle Streams for rapid database replication. However, it is still required that all tightly coupled branches of a distributed transaction must run on the same instance. Only between transactions and between services can transactions be load-balanced across all of the database instances. This means that global SI for distributed transactions across database instances is not supported either.

HBase is an alternative to supporting basic database functionality on clouds with random access performance on par with open source relational databases such as MySQL. Hadoop is a popular open source cloud computing framework that has proven to scale and perform well on clouds [1]. HBase provides a distributed, column-oriented data store modeled after Google's Bigtable [7]. Although HBase is not a full-fledged DBMS and only provides simple syntax to query sparse tables with no SQL and Join support, it has been demonstrated to successfully support a wide spectrum of useful applications from commercial vendors and web sites that used to employ DBMS in the backend, such as, Adobe, Streamy, etc. However, HBase only provides single atomic writes based on row locks and very limited transactional support (See Section II.C). This is because transactions for HBase and the like (such as HyperTable) are intrinsically distributed transactions involving multiple data store locations, which is expensive to manage. In a more general sense, column-oriented data stores face difficulties in handling transactions because the operations are column-based, meaning that instead of easily locking down and operating on a row as in traditional DBMS, operations on a row are split into sub-operations on individual columns. Row locks are also inefficient in column stores because that is not how the data are stored physically [2]. As a result, no work has been done to achieve SI for multi-row distributed transactions with HBase or other column-oriented stores.

In this paper, the main contribution is to provide a novel mechanism that uses HBase as a cloud database solution for simple database transactions with global SI at low added cost. In our approach, we make use of several HBase features for achieving SI. We adopt a global transaction ordering method and design several HBase tables to manage SI over simple database transactions composed of read and write operations, i.e., select (read), insert (write), update (write), and delete (write) operations, over multiple data rows. We show how these simple database transactions with global SI guarantee over the whole cloud can be easily supported by using bare-bones HBase with its default configuration so that the high throughput, scalability, fault tolerance, access transparency and easy deployability of HBase can be inherited.

The remainder of the paper is structured as follows: in Section II we introduce some background information concerning the operational definition of global SI, the introduction to HBase, and related work. In Section III we describe our technique for achieving global SI with HBase. In Section IV we evaluate the performance of our technique. Section V concludes.

II. Background

A. Global SI

SI was originally proposed for centralized database systems [4]. It is not in the ANSI/ISO SQL standard but is adopted by major DBMS due to its better performance than Serializablity. SI can be formally defined as follows:

Definition 1. SI: *A transaction history conforms to SI if, for every successfully committed transaction T, 1) there exists a snapshot timestamp start(T) such that T sees the database in a state that contains the updates from all the transactions that have committed up to start(T), and 2) T is allowed to commit only if, when T is ready to commit, there are no other transactions with conflicting updates that have committed after start(T).*

In SI, the database system thus assigns a timestamp start(T) to a transaction T which identifies the snapshot of the database that T will read from. This snapshot will not be affected by any other updates after start(T) and will be used throughout T's lifespan by T. start(T) can be chosen to be any time less than or equal to the actual start time of transaction T. When T commits, the system assigns T a timestamp commit(T) more recent than the actual start time of T. If no conflicting committed updates from other transactions happen after start(T), T can successfully commit. Global SI is obtained by applying the definition above to distributed or replicated database systems.

B. HBase

HBase is the database component of Hadoop. Hadoop is an open source implementation of a subset of Google's system for large-scale data processing: MapReduce [8], Google File System [9] and BigTable [7]. In HBase, applications store data into sparse tables, which are tables with rows having varying numbers of columns. Every data row has a unique and sortable row key. Columns are grouped into column families. The data for the same column family are stored physically close on disk for efficient querying. HBase employs a master-slave topology. Tables are split for distributed storage into row-wise "regions". The regions are stored on slave machines called "region servers". Each region server hosts distinct row regions. As a result, a transaction could involve multiple region servers, becoming a distributed transaction. In HBase, both the master and region servers rely on the Hadoop Distributed File System (HDFS) to store data. In the latest release at the time of writing (version 0.20.1), a pool of multiple masters is supported eliminating a single point of failure. When a region server fails, its data can be recovered from HDFS and be hosted by a new replacement region server.

Table I shows an example HBase table taken from the HBase website (slightly modified). The table contains one row with row key "com.cnn" and columns "anchor:cnnsi.com" and "anchor:my.look.ca" grouped by column family "anchor:". Each HBase row-column pair, for example, row "com.cnn" and column "anchor:my.look.ca", is assigned a timestamp (a Java Long type number), either explicitly passed in by the user when it is inserted, or

Table I
AN EXAMPLE HBASE SPARSE TABLE

Row Key	Timestamp	Column	Value
com.cnn	T9	anchor:cnnsi.com	cnn
	T9	anchor:my.look.ca	cnn.com
	T8	anchor:my.look.ca	bbc.com

implicitly assigned by the system. The value for each row-column pair is uniquely determined by its row key, column and timestamp. Currently, only simple queries using row keys and timestamps are supported in HBase, with no SQL or Join queries. An iterator-like interface is available for scanning through a row range. However inadequate the query capability may seem, if the tables are formulated properly, some efficient problem-specific search methods can be developed, especially for data with graph-like structures such as directed acyclic graphs for workflows [15].

HBase has a rudimentary set of primitives for transactions that are built on top of its row lock mechanism (note that HBase only has exclusive locks). For every transaction, writes are applied when committing the transaction. The read and write sets for each transaction are kept and checked at commit time by each region server involved to see if there exist any conflicting write sets from other concurrent transactions committed after the transaction starts. If there are no conflicts, each region server will vote "yes" for this transaction. If all the region servers have voted "yes", the transaction can proceed to commit. However, from this point on, no mechanism exists to guarantee whether the transaction would continue or abort, which may happen in the event of a failure of the client mid-commit, after region servers voted "yes". If the transaction does not crash and continues to commit, then it locks the rows and writes. Nevertheless, lost updates could occur if two concurrent transactions commit at roughly the same time on conflicting rows. Suppose transaction T1's write set contains row x, y and z, transaction T2's write set contains row y and z, and there are no conflicting updates to x, y or z committed from other transactions since T1 and T2 started. T1 and T2 will both proceed to commit and could write to the conflicting rows interleavingly which results in lost updates since future reads from clients are normally made with the most recent timestamp. In short, the transactional support provided by HBase does not satisfy SI and is not reliable for practical transactional processing.

C. Related Work

Currently, there is no solution for traditional DBMS to be easily replicated on a large scale with global SI for distributed transactions in a cloud computing environment. This is mainly because it is expensive to maintain strong data consistency among replicated database replicas with a high degree of scalability [10], [11], [12], [13]. It is also non-trivial to deploy multiple full-fledged traditional DBMS on-demand on clouds. Existing cloud-scale data services such as HBase, Amazon SimpleDB and HyperTable do not support ACID (Atomicity, Consistency, Isolation and Durability) transactions with satisfactory isolation levels and data consistency guarantees. [5] shows an attempt to use Amazon S3 as a storage system for general-purpose database applications but only provides low-level protocols to read and write data from and to the S3 storage service, sacrificing ACID properties for scalability and availability. HadoopDB [3] is a new project aiming at analytical database use cases that rarely involve conflicting update transactions (mainly read-only with a few serialized insertions of new records) and the prototype is also based on data partitioning which does not support SI over distributed transactions. It also requires extra SQL parsing and planning and demands the existence of a traditional DBMS on every processing node which is expensive to deploy and configure. The most relevant research on transactions for HBase is described in [14], which presents a scalable transaction manager for cloud database services to execute serializable ACID transactions of web applications, using HBase. They implemented a Transaction Processing System composed of a number of Local Transaction Managers (LTM). Their system first loads data from HBase, splits the loaded data across LTMs and further replicates the data for fault tolerance, which introduces extra overhead in data management, system deployment and maintenance as well as performance, compared to using bare-bones HBase. Data freshness cannot be guaranteed because of the existence of multiple data copies. Additionally, the list of primary keys accessed by the transaction must be given before executing the transaction whereas in normal transactions, data items are usually not all known at the beginning of a transaction but known only when they are accessed. This further means for example that range queries cannot be supported by their system. In comparison to their system, we support common transactions that don't need a complete list of all the data items accessed at transaction starting time. This requires more complex logic than using a known list of primary keys from the start. More importantly, we support global SI which has significant performance benefits compared to serializability, and our system does not require additional processes to be deployed to provide the transaction support.

III. GLOBAL SI WITH HBASE

In this section, we describe in detail our mechanism for achieving distributed transactions with global SI using HBase. We handle two types of transactions, read-only and update transactions. A read-only transaction consists of read operations only, whereas an update transaction may contain a combination of multiple read, insert, update and delete operations. Our technique uses bare-bones HBase with its default configuration. The general idea is to create several

Table II
IDENTIFICATION LABELS

Name of Label	Type	Globally Unique	Order Significant
start (Si)	timestamp	no	yes
write (Wi)	unique ID	yes	no
precommit (Pi)	unique ID	yes	no
commit (Ci)	timestamp	yes	yes

HBase tables to manage snapshots, update conflicts, and concurrent transaction commits, and to guarantee database ACID properties. We make use of some essential HBase features to achieve global SI in distributed transactions in a convenient way. Firstly, the HBase master maintains a single table-like global view for all clients. This is useful because any data change is instantly visible to all clients. Secondly, HBase supports storing multiple versions of data under the same row and column, differentiated by timestamps, and allows concurrent reads and writes of new data versions. This means that all versions of data may be reconstructed and reads/writes with different versions don't block each other, offering very high throughput. Thirdly, HBase supports atomic row writes so that simultaneous writes on the same data item do not garble data. Finally, HBase provides a simple and efficient search mechanism on sparse columns with non-empty values. For example, it is easy and fast to return all the rows in a very big table containing records related to a person named "Bill Gates", if that name is used as a sparse column name with non-empty values in those rows. This is useful when we search for conflicting updates made by concurrent transactions.

Additionally, we use four different identification labels for four types of transaction operations. They are the start, write, precommit and commit labels, each acquired by a transaction when it starts, writes, precommits and commits. These labels play different types of roles in the system. As is shown in Table II, the start and commit labels are globally well-ordered timestamps. Commit timestamps are globally unique for each transaction, but two transactions can have the same start time. The write and precommit labels are unique IDs, but they do not correspond to a global time and their order is not significant. Read transactions only need to acquire a start timestamp, while each update transaction will have to acquire all four types of labels.

The general approach to handle transactions on individual data columns is as follows. For read-only transactions, first get a start timestamp which identifies the snapshot the transaction will read from. Every subsequent read in the read-only transaction will only see the latest version of data up to the snapshot. Update transaction T also first gets a start timestamp identifying the snapshot, and it gets a write label to be used later for all writes to the data tables. Every

Table III
VERSION TABLE

Data Item Location	Commit Timestamp
L1	C1
L2	C2

Table IV
COMMITTED TABLE

Commit Timestamp	L1	L2
Ci	Wi	Wi
Cj	Wj	

transaction T maintains a data structure to store its read and write sets. When reading a data item, get the newest version of data committed up to the snapshot, unless the data has been read or written by T before, in which case the version of data from the data structure is read instead. When writing to a data item (insert/update/delete, delete is handled as inserting an empty value), write to the HBase user data tables directly, taking as the HBase write timestamp the transaction's unique write label. Record the write operation in the data structure mentioned above. All the writes are performed immediately and do not wait until commit time. This can be done because HBase supports concurrent writes of new data versions differentiated by timestamps. When trying to commit, get a precommit label and go through a precommit process to make sure that there are no concurrent commit requests for transactions trying to update conflicting data items. Finally, proceed to commit atomically. More details of our approach are given now.

A. Data Structures

To achieve global SI, we need to create several simple HBase tables in addition to the user data tables to store transaction management metadata. The tables are: Version Table, Committed Table, Precommit Table and several identification label tables.[1]

The Version Table is used for retrieving the commit timestamp of the transaction that wrote the last-known committed version of a data item. A data item is represented as a row in the Version Table. The row key is the data item location (a concatenation of the table name, row key and column name where the data item is stored in the user data tables). The column value is the commit timestamp of the transaction. For example, the first row in Table III means the last-known committed version of the data item with location "L1" is committed by a transaction with commit timestamp "C1". This commit timestamp is used by read operations to minimize the search scope for finding the latest committed data version according to the snapshot, as explained below.

[1]Column families for tables are omitted for simplicity.

Table V
PRECOMMIT TABLE

Precommit Label	L1	L2	Committed
Pi	Y	Y	Ci
Pj	Y		

The Committed Table keeps records of all the data items each committed transaction writes to. A transaction is deemed as committed only after its corresponding record appears in the Committed Table. The Committed Table is used to check for conflicting update transactions at transaction commit time and to retrieve the latest committed data versions according to a transaction snapshot. Each row in the Committed Table represents a committed update transaction. The transaction commit timestamp is used as the row key so that we can easily check all the committed updates within a timestamp range. Each column in a row is named as a data item location. The value for a column is the write label used for writing to the data item by the transaction. For example in Table IV, the first row stores records for a transaction committed at timestamp "Ci" that writes to data item "L1" and "L2" with HBase write timestamp "Wi". As soon as a transaction record is successfully inserted into the Committed Table, the transaction has been committed atomically and its results are made durable (supported by HDFS). The atomicity is guaranteed because HBase supports atomic row writes and we are inserting a single row to the Committed Table. The Precommit Table is used to detect and avoid concurrent commit requests on potentially conflicting data items. This can be illustrated using the same example scenario given when explaining problems with the HBase's built-in transactional support in Section II.C. In this case, if T1 and T2 try to commit at almost the same time, scanning the Committed Table won't show any conflicting updates and thus they would both proceed to commit at the same time, making lost updates possible. The use of the Precommit Table can avoid such kind of problems from happening. Each row in the Precommit Table represents a tentative commit request from an update transaction. The row key is the precommit label and the columns are the data locations a transaction has written to, with an extra column called "Committed". The values of the data location columns written to are set to "Y" which simply means that the field is non-empty, in order to leverage the efficient column search mechanism provided by HBase mentioned before. If the value of the "Committed" column is set to "Y", it means that the transaction corresponding to that row has already been successfully committed. See Section III.C for more details on how precommit works.

To issue globally unique labels, we use several label tables, namely, the "Write Label" Table, the "Precommit Label" Table and the "Commit Timestamp" Table (Table VI shows the Write Label Table, and the other two label tables are similar). Each of these tables is used as a global sequencer for issuing unique incremental integer labels. Furthermore, we also need another table called "Committed Index" Table to store the most recently assigned snapshot. See Section III.B for more details about the usage of these tables. Finally, we also need a DataSet (DS) for each transaction T: A Java HashMap containing the most recent data values of the data locations read or written by T, indexed by data item location (e.g., (L1, value), (L2, value), etc.).

Table VI
WRITE LABEL TABLE

Write Label	Counter
current label	78

Table VII
COMMITTED INDEX TABLE

Most Recent Snapshot	Snapshot Value
current snapshot	98

B. Identification Labels and Timestamps

As mentioned before, we need four types of identification labels. The uniqueness of write, precommit and commit labels is based on an atomic HBase function named "incrementColumnValue". After setting an initial Java long type number as value for a cell (row R, column C) in a table, if "incrementColumnValue" is called on that cell, the value of the cell will be incremented atomically and the new value will be returned. Therefore, in our system, we define three globally accessible tables for the three types of labels, each named after the label name with only one column and one row, as explained above. Considering write labels for example, a transaction gets its write label by calling the "incrementColumnValue" function on the Write Label Table, and the system will automatically increment the value by one in the table and return the value to the transaction as its write label. In this way, all write labels are globally unique. Moreover, the Commit labels are globally well-ordered in time, and can be used as a global time system to order committed transactions, on which the SI system can be built.

Obtaining start timestamps for transactions, however, is trickier. The purpose of a start timestamp is to identify the snapshot from which a transaction will read, so that the transaction will take into account data from all transactions committed up to the snapshot time. Furthermore, this snapshot has to be consistent to the transaction during the whole lifespan of the transaction. Since all the committed transaction information is stored in the Committed Table, a straightforward way to define a start timestamp of a transaction is to look into the Committed Table and define the start timestamp by the latest commit timestamp. In this way, a transaction will be able to retrieve all committed transactions

before its start timestamp. There is, however, a potential problem with such an approach. Since a transaction obtains its commit timestamp before writing to the committed table, it is possible that a transaction Ti gets its commit timestamp before transaction Tj, yet takes longer to finish the commit process and therefore only appears later in the Committed Table. During this process, if transaction Tk starts, it may see Tj in the Committed Table and Choose Cj as its start timestamp Sk, such that it will read from transaction Tj and all transactions with smaller commit timestamps than Cj in the Committed Table. However, Ti with Ci<Cj may appear in the Committed Table after Tk has already started to read, thus potentially corrupting the snapshot that Tk reads from. Therefore, in order to impose consistent snapshots, we choose, when Tk starts, its start timestamp Sk as the largest Cj in the Committed Table that is part of an increasing sequence starting from the smallest Cj in the table, without a gap. Indeed, if the first Ci in the Committed Table has value 1, then, for any Ci, all integer values from 1 to Ci have to appear in the Committed Table eventually, since commit timestamp values are only given out to transactions that will commit. If there is no gap between Ci and any earlier commit timestamps already present in the Committed Table, one can be sure that no new committed transactions Tj with Tj<Ti will appear in the Committed Table, so Ci can be used as a valid snapshot time. In this way, it can be guaranteed that a transaction will always see the same fixed set of earlier transactions during its whole transaction lifetime. This approach may sacrifice a few recent transactions in order to be consistent. However, according to the SI definition, SI is satisfied. Furthermore, in order to make determining the start timestamp more efficient, each transaction will store its start timestamp in the "Committed Index" table so that all the following transactions can start searching for gaps starting from this point on afterward.

C. Protocol Walkthrough

The detailed approach for doing an update transaction Ti with global SI is as follows (using the data shown in the tables in Section III.A as example): 1) Get start timestamp Si and write timestamp Wi. 2) Read/Write data items. 3) Go through a Precommit phase to determine if there are potential conflicts. 4) Commit. Note that read-only transactions only need to obtain the start timestamp and then read; there is no need for Precommit or Commit.

To Read (select) a data item, for example, from location L1, first check if L1 is in the DS. If found, use that value and return; otherwise, proceed as follows: 1) Get the "Commit Timestamp" for L1 from the Version Table. If the record exists, it will return C1; otherwise, set C1=∞. 2) If C1<=Si: Scan the Committed Table in the range [C1, Si], read the latest version from the Committed Table and use it to read from L1. Update the DS as well as L1's record in the Version Table. 3) If C1>Si: Scan the Committed Table in the range [1, Si]; if found, read the latest version from the Committed Table and use it to read from L1; otherwise, read from L1 and update the DS only.

To write (insert, update and delete), for example, to location L1, first check if L1 is in the DS. If found, update that value; otherwise, add a new entry for L1 to the DS. Then write to HBase with timestamp Wi. Because Wi is unique among transactions, it is safe for transaction Ti to write to HBase without the danger of garbling data.

To Precommit: 1) Get Precommit label Pi and check the Committed Table in the range [Si+1,∞) for rows that contain columns conflicting with Ti's writeset. If there are any, abort; otherwise, add a row Pi to the Precommit Table, put "Ys" in all updated data item columns (L1, L2, for example). 2) Check the Precommit Table in the full range for rows (except itself) that contain conflicting columns. If there is any conflicting record that does not have its commit timestamp in the column "Committed" yet, or if there is any conflicting record with a commit timestamp under the column "Committed" that is larger than Si, abort and delete row Pi. Otherwise, proceed to commit.

Note that the sequence of checking must be first the Committed Table and then the Precommit Table, in order to rule out the case in which some conflicting concurrent commits have not shown up yet during the first check in the Committed table. This works because, in order for those in-progress commit records to show up in the Committed table, they must have already finished the Precommit process and have records in the Precommit table.

To commit: 1) Get a Commit timestamp Ci. 2) Add a row Ci to the Committed Table, with the update data items as columns, and the write timestamp Wi as value for those columns. 3) Set the "Committed" column for row Pi in the Precommit Table to "Ci".

We discuss the correctness of our approach according to the database ACID properties. Atomicity and durability are maintained as described in the commit procedure above because adding a row in the Committed table is guaranteed to be atomic and durable by HBase. We assume that as long as atomicity is maintained and only valid data are inserted to the HBase tables, the consistency property is also maintained. For the isolation property, according to the read procedure above, only the data from committed transactions up to when the snapshot is taken are read except for the data just written by the transaction itself. Additionally, write operations from different transactions won't interfere or overwrite one another because no writes to the same data item with the same timestamp are allowed. Therefore, isolation, in particular, global SI, is maintained.

IV. Performance Evaluation

To quantify the cost of adopting our approach, we are interested to know the performance of our implementation

in the following aspects: (1) The cost of our label and timestamp acquisition mechanism mentioned in Section III.B. We want to investigate whether the mechanism would become a performance bottleneck because every transaction would need to access a small number of globally accessible tables for obtaining labels and timestamps. (2) The cost of scanning a growing Committed Table without using the Version Table for getting the latest data version up to the transaction snapshot for read operations. This would illustrate the rationale of using the Version Table whose purpose is to minimize the range of rows to scan in the Committed Table. (3) The cost of performing sequential reads and writes with transactional SI compared to the cost of performing the same set of reads and writes with pure HBase (without transactions). The benchmarks we use are adapted from Section 7 of the Google BigTable paper [7] because HBase also targets random access performance similar to BigTable.

We implemented a java class called TransactionalSI and several scripts for setting up the initial tables needed. A client program is also implemented to submit transactions to HBase. HBase is deployed on a 3-machine cluster connected with GigE ethernet. The "gridbase1" machine has 2 dual-core 2.6Ghz CPUs and 8GB of memory; "gridbase2" has 2 quad-core 2.0GHz CPUs and 8GB of memory; and "gridbase3" has 4 quad-core 1.6GHz CPUs and 16GB of memory. Both the Hadoop HDFS namenode and the HBase master run on "gridbase1". Each "gridbase" machine also runs an HBase region server. Note that in our tests, we run concurrent clients which are evenly distributed across all three machines. The clients are running on the same machines as the HBase servers, competing for machine resources. Thus, our tests are intended for proof-of-concept purpose only.

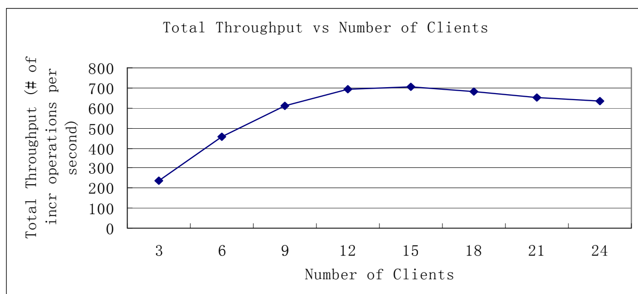

Figure 1. Label mechanism total throughput vs total number of clients.

Figure 1 shows the throughput of using one globally accessible table to issue unique timestamps with varying number of total concurrent clients. In this test, all clients continuously and consecutively acquire timestamps from a single global table location using the "incrementColumnValue" method. The results show that with our 3-machine cluster, the HBase region server serving the single-row global table reaches a highest throughput of issuing about 700 timestamps per second, implying an upper limit of serving 2.5 million transactions per hour. Theoretically, the server should maintain the maximum throughput as the number of clients increases. However a slight decrease of throughput is witnessed in Figure 1, due to the competition of resources from the clients running on the server machines. Overall, the performance is still quite good considering our client deployment and cluster machine capacity. Figure 2

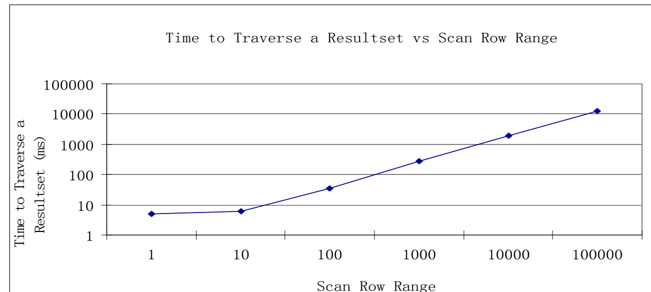

Figure 2. Time for scanning a row range in an HBase table as a function of the number of rows scanned.

shows the time spent in scanning an increasing number of rows of a single table. There is a linear relationship between the time spent and the number of rows scanned, indicating that in our SI protocol the time to scan the Committed Table grows linearly with the number of rows to be scanned, if we don't use the Version Table. When this number is growing large due to the accumulation of committed transaction records through time, the time spent in scanning for reads will become overly costly. As such, the Version Table is necessary, given that it limits the number of rows to be scanned within a reasonably small range, especially for data items read frequently. Figure 3 and Figure 4 compare the

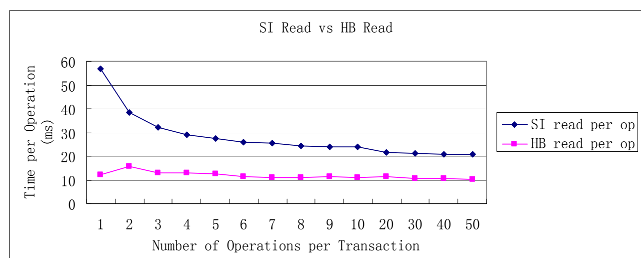

Figure 3. SI Write vs HBase Write with varying transaction length.

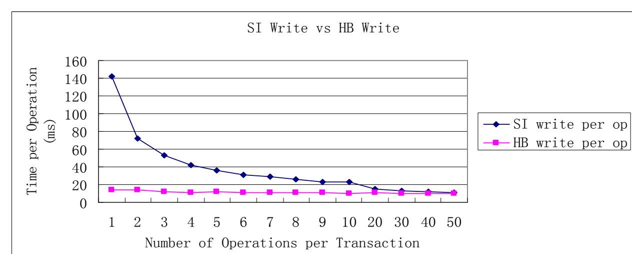

Figure 4. SI Write vs HBase Write with varying transaction length.

overall performance of sequential read/write operations with

transactional SI, and with pure HBase without transactions. In the tests, we measure the total time a single client spends on performing transactions with a varying number of read/write operations per transaction, compared to the cost of performing the same set of reads and writes with pure HBase without transactional concerns. Figure 3 shows that the cost of doing reads in transactions with SI is about twice the cost of using HBase directly, but larger for very short transactions with less than four read operations. The extra cost in using SI is introduced by the need to search for a proper version up to the transaction snapshot which involves a point-read to the Version Table and a short scan on the Committed Table, and by the need to acquire a start timestamp. The test is set up such that locations are read from frequently and thus appear in the Version Table, and only short scans of the Committed Table are required for reading. For infrequently read values, the read time will be longer. Figure 4 shows that the cost of doing writes with SI is much higher for very short transactions with less than four write operations, and becomes about the double of the cost of using pure HBase with relatively short transactions containing five to ten write operations. The cost goes down further as the transactions contain more write operations to become almost the same as the cost for doing writes with pure HBase. The reason for the high performance penalty in short transactions is due to the extra Precommit and Commit processes which require the scanning of the Precommit and Committed tables, and the cost of acquiring the four labels/timestamps. These costs are amortized as the number of operations per transaction grows. In general, for medium-size transactions under light loads and without write conflicts between concurrent transactions, we can expect the cost of adopting our technique to be about double the cost of using pure HBase. We would argue that the added cost is reasonably low considering the benefits of having transactional SI over the whole cloud.

V. Conclusions and Future Work

In this paper, we describe an approach to use HBase as a cloud database solution for simple multi-row distributed transactions with global SI guarantee. We make use of several HBase features for achieving global, cloud-wide SI, and we design a set of HBase tables to support global SI for simple database transactions. We show the validity of our approach in terms of database ACID properties and quantify the cost of our approach with performance evaluations. For future work, we may support more complex database queries, further optimize our approach for better performance, and include appropriate handling of failing processes. Extending our approach to other cloud-scale column-oriented data services is also in our agenda.

References

[1] Terabyte sort. http://sortbenchmark.org/.

[2] D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented Database Systems. In *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An Architectural Hybrid of Mapreduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A Critique of ANSI SQL Isolation Levels. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1–10, 1995.

[5] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 251–264, 2008.

[6] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 729–738, 2008.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 205–218. USENIX Association, 2006.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:107–113, 2008.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, 29–43, 2003.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 173–182, 1996.

[11] K. Salem, K. Daudjee. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the VLDB Endowment*, 715–726, 2006.

[12] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Applying Database Replication to Multi-player Online Games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, Article No.15, 2006.

[13] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, 155–174, 2004.

[14] Z. Wei, G. Pierre, and C.-H. Chi. Scalable Transactions for Web Applications in the Cloud. In *Proceedings of the Euro-Par Conference*, LNCS 5704, 442–453, 2009.

[15] C. Zhang and H. De Sterck. CloudWF: A Computational Workflow System for Clouds Based on Hadoop. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, LNCS 5931, 393–404, 2009.