Scalable Computing: Practice and Experience
Volume 12, Number 2, pp. 209–226. http://www.scpe.org
ISSN 1895-1767

HBASESI: MULTI-ROW DISTRIBUTED TRANSACTIONS WITH GLOBAL STRONG SNAPSHOT ISOLATION ON CLOUDS

CHEN ZHANG* AND HANS DE STERCK†

Abstract. This paper presents the "HBaseSI" client library, which provides global strong snapshot isolation (SI) for multi-row distributed transactions in HBase. This is the first strong SI mechanism developed for HBase. HBaseSI uses novel methods in handling distributed transactional management autonomously by individual clients. These methods greatly simplify the design of HBaseSI and can be generalized to other column-oriented stores with similar architecture as HBase. As a result of the simplicity in design, HBaseSI adds low overhead to HBase performance and directly inherits many desirable properties of HBase. HBaseSI is non-intrusive to existing HBase installations and user data, and is designed to be scalable across a large cloud in terms of data size and distribution.

Key words: distributed transaction, cloud database, HBase, snapshot isolation

AMS subject classifications. 68U01, 68N01, 68M01, 68P01

1. Introduction. Column-oriented data stores (column stores) are gaining attention in both academia and industry because of their architectural support for extensive data scalability as well as data access efficiency and fault tolerance on clouds. Data in typical column stores such as Google's BigTable system [3] are organized internally as nested key-value pairs and presented externally to users as sparse tables. Each row in the sparse tables corresponds to a set of nested key-value pairs indexed by the same top level key (called "row key"). The second level key is called "column family" and the third level key is called "column qualifier". Each column in a row corresponds to the data value (stored as an uninterpreted array of bytes) indexed by the combination of a second and third level key. Scalability is achieved by transparently range-partitioning data based on row keys into partitions of equal total size following a shared-nothing architecture. These data partitions are dispatched to be hosted at distributed servers. As the size of data grows, more data partitions are created. In theory, if the number of hosting servers scales, the data hosting capacity of the column store scales. Concerning data access, at each data hosting server, data are physically stored in units of columns or locality groups formed by a set of co-related columns rather than on a per row basis. Column stores derive their name from this property. This makes scanning a particular set of columns less expensive since the data in other columns need not be scanned. Persistent distributed data storage systems (for example, with file replication on disk) are normally used to store all the data for fault tolerance purposes.

Column stores provide database-like table views, and it would be desirable if distributed transactions can be supported on them so that applications that used to be built around traditional database management systems (DBMS) can make use of cloud column stores for transactional data processing, with improved scalability. Indeed, many applications, such as a large number of collaborative Web 2.0 applications, would benefit from transactional multi-row access to the underlying data stores [1]. In fact, those modern applications pose high requirements on scalability and fault tolerance and there are currently no existing DBMS solutions (even parallel database systems) to fully cater to those requirements due to the overhead of managing distributed transactions and the fact that it is impossible for DBMSs to guarantee transactional properties in the presence of various kinds of failures without limiting system scalability and availability [1, 4, 7]. Unfortunately, no out-of-the-box support for transactions involving multiple data rows exists in column stores. This is mainly because multi-row transactions in column stores are intrinsically distributed transactions [6] and traditional approaches would suffer from similar problems as in existing distributed DBMS solutions.

This paper presents a novel light-weight transaction system with global strong snapshot isolation on top of HBase (which is a representative open source column store modeled after Google's BigTable system), without using traditional methods of handling distributed transactions, such as standard 2-phase commit protocols, consensus-based commits, atomic broadcast, or explicit data locking. A preliminary version of our system, providing weak SI for HBase, was presented in [11]. HBaseSI, described in this paper, recycles some of the design principles of the initial system from [11] but uses a different, more efficient solution for handling distributed synchronization, with added support for global strong (and not weak) SI and an efficient failure handling

*David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada (c15zhang@cs.uwaterloo.ca).
†Department of Applied Mathematics, University of Waterloo, Waterloo, Canada (hdesterck@uwaterloo.ca).

mechanism. Our work in [11] described the first ever SI system for column-stores. Independently and at the same time, the Google Percolator system was presented in [8]. Percolator provides global strong SI for Google's column store system, BigTable. Percolator shares many design principles with our SI system, but there are also many important differences in design goals. The current paper extends and improves the system for SI in HBase that we presented in [11].

The solution presented in this paper is called "HBaseSI". HBaseSI targets the same type of OLTP(Online Transaction Processing) workloads as HBase, taking advantage of HBase's random data access performance comparable to open source database systems such as MySQL. It is implemented as a client library and does not require any extra programs to be deployed or running in addition to existing HBase servers. In addition, HBaseSI is non-intrusive to existing user data that have already been stored in HBase since it does not require modifications to existing user data tables. In HBaseSI, transactional management meta-data are written by each transaction to a separate set of HBase tables. There is no central "commit engine" that decides which of the transactions that are ready to commit can actually commit; instead, the transaction processes decide autonomously, in a distributed fashion, whether they can commit or have to fail, using the information stored in the additional meta-data HBase tables. As a result, little performance overhead pertaining to distributed synchronization is added by the transactional management logic. Many of HBase's desirable properties are directly inherited as well, such as fault tolerance, access transparency and high throughput. Concerning scalability, the design target of HBaseSI is to be fully scalable across a large cloud in terms of data size and distribution. In its current design, HBase does not target scalability in terms of the number of transactions per unit of time.

This paper is structured as follows: in Section 2 we introduce some background information about snapshot isolation and HBase. In Section 3 we describe the design and implementation of HBaseSI in detail. In Section 4 we evaluate the performance of HBaseSI. Section 5 gives comparison to related work, and section 6 concludes.

2. Background.

2.1. Snapshot Isolation. For our purposes, we can describe Snapshot isolation (SI) as follows.

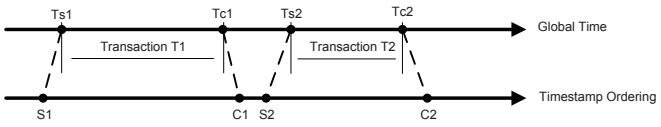

Fig. 2.1: Illustration of SI.

A transaction Ti acquires a start timestamp, S(Ti), at the beginning of its execution (before performing any read or update operations), and acquires a commit timestamp, C(Ti) at the end of its execution (after finishing any read or update operations). We will also use the shorthand notation Si=S(Ti) and Ci=C(Ti) in what follows. The timestamps Si and Ci are ordered: they inherit their ordering from the ordering of real global times Tsi and Tci to which they correspond (Fig. 2.1). This ordering implies in particular that all read and write operations of Ti happen (in real, global time) after the time corresponding to Si, and all write operations of Ti happen (in real, global time) before the time corresponding to Ci. Transactions Ti and Tj are called concurrent if their lifespan intervals (Si,Ci) and (Sj,Cj) overlap. A transaction Ti that commits successfully is called a successful or committed transaction.

Global Strong SI can then be described as follows. A transaction history H satisfies global strong SI, if its (successful) transactions satisfy the following two conditions: 1. Read operations in any transaction Ti see the database in the state after the last commit before Si. In other words, all updates made by the committed transaction Tj which has the last Cj <= Si are visible to Ti. However, read operations in transaction Ti that read data items that have previously been written by transaction Ti itself, see the data values that were last written by Ti; 2. Concurrent transactions have disjoint writesets.

We add the qualifier 'global' when we define strong SI because we want to investigate Snapshot Isolation for a distributed system in this paper, and want to stress that the definition above applies to the global system. Additionally, the above definition does not regulate the behavior when two concurrent transactions with overlapping writesets both try to commit. In many occasions, a rule called "first committer wins" is employed, which will cause the failure of the transaction that is second in attempting to commit. To better illustrate this rule, let's look at an example set of transactions shown in Fig. 2.2. T1 and T2 must have disjoint

writesets in order to both commit successfully. If they have overlapping writesets, only T1 will successfully commit and T2 will abort, because T1 attempts to commit before T2.

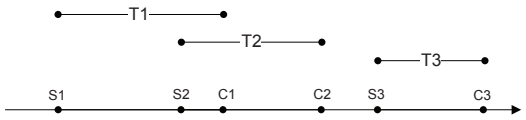

Fig. 2.2: An example SI scenario.

The strong notion of SI as defined above is different from the original definition of SI [2], which allows Si to be chosen corresponding to any time in the past before the first read or update operation in transaction Ti. This relaxed version of SI is also called weak SI in [5]. To illustrate this difference, we assume that T1 and T2 in Fig. 2.2 have disjoint writesets and both commit successfully. According to the definition of strong SI, T3 must see all the committed results as of timestamp S3, which include the commits for both T1 and T2. However according to weak SI, it is allowed that T3 use a snapshot between C1 and C2 that includes only the committed results from T1. Typically, versions of the strong notion of SI are implemented in stand-alone, non-distributed commercial databases. SI is not included in the ANSI/ISO SQL standard but versions of it are adopted by major DBMSs due to its better performance than Serializablity at the cost of having a potential write-skew anomaly [2].

2.2. HBase. HBase is a column-oriented store implemented as the open source version of Google's Big-Table system. Rows in each table are automatically sorted by row keys. The data value for each row-column combination is uniquely determined by the row key, column and timestamp. The timestamp facilitates multiple data versions. Timestamps are either explicitly passed in by the user when the data value is inserted, or implicitly assigned by the system. Each table is horizontally partitioned into row regions and each region is hosted by a distributed "region server" with region data stored in persistent storage (Hadoop HDFS, which stands for Hadoop Distributed File System). Currently, only simple queries using row keys and timestamps are supported in HBase, with no SQL or join queries. It is also possible to scan and iterate through a set of columns row by row within a row range. The scalability of HBase is attributed to the shared-nothing architecture of data regions hosted by distributed region servers. However, there could still be bottlenecks in the system in the case when a single region server gets overloaded by too many requests on the same data region. In fact, at each region server, all the read/write requests to a particular row in a table region are serialized.

Our choice of using HBase as the basis for investigating transactional SI solutions for clouds is not random. HBase enjoys several nice properties that are important for simple and efficient SI implementations. First, HBase offers a single global system view with access transparency, meaning that clients access all the HBase tables as if they are hosted at a centralized server without knowing that they are actually contacting different distributed region servers for fractions of data. This significantly reduces the complexity of transactional protocol implementation. Second, HBase provides multi-version data support distinguished by the timestamp the data item is written with. This feature can directly facilitate the SI protocol implementation. Third, HBase guarantees single atomic row operations (reads/writes) with strong data consistency in the global table view.

3. HBaseSI.

3.1. System Design. A major design principle of HBaseSI is to provide transactional SI for HBase with minimum add-ons to existing HBase installations and administration. It may also be an advantage if it is possible for HBase users with existing data tables to employ HBaseSI with minimum effort. To this end, HBaseSI is implemented as a client library in Java with no extra programs to be deployed. Applications that need to do transactions use the client library to interact with HBase instead of using the standard HBase API. Each transaction writes its own transactional meta-data (e.g. transaction ID, commit timestamp, commit request, etc.) to a set of global system HBase tables (separate from existing user data tables), and in the meantime, queries those tables to obtain information about other transactions. Based on the information obtained, and by accessing this information with atomic read/write operations provided by HBase, a transaction can autonomously decide to commit or abort. From a user's point of view, using HBaseSI requires no modification to any existing data tables.

Table 3.1: W counter table. W stands for "HBase write timestamp".

Row Key	Counter
W	86

Table 3.2: R counter table. R stands for "commit request ID".

Row Key	Counter
R	78

HBaseSI employs several HBase tables in addition to the user's data tables. These additional system tables are three Counter tables (Tables 3.1, 3.2 and 3.3) for providing globally unique counters, a CommitRequestQueue table (Table 3.4) that acts as a queue for transactions that are submitting requests to commit, a CommitQueue table (Table 3.5) that acts as a queue for transactions that are cleared to commit, and a Committed table (Table 3.6) that keeps track of successfully committed transactions and their writesets.

The Counter tables are intended to serve as a set of centralized locations for issuing globally unique IDs that may be used as well-ordered counters. Each of the tables is a single-row-single-column table. The HBase incrementColumnValue function is used on the column "Counter" to dispense globally unique and strictly incremental time labels to transactions atomically. The W Counter table (Table 3.1) issues a unique ID to each transaction at the start of the transaction. W stands for "write timestamp". This ID will be used as the unique ID for the transaction, and as HBase timestamp when writing data to HBase tables (note that in this paper we use two types of timestamps: "HBase write timestamps" are used as write timestamps for HBase to distinguish different data versions, and "transaction timestamps" are timestamps used for transaction ordering purposes). The order of the W counter is not important, as long as each W counter is unique. The R Counter table (Table 3.2) issues unique commit request ordering IDs dispensed to transactions that are attempting to commit, establishing an order among the transactions attempting to commit, which is, among other things, used for enforcing the "first-committer-wins" rule [2]. R stands for "commit request ID". The C Counter table (Table 3.3) issues the final unique commit timestamps, each of which is used as the actual commit timestamp of a transaction. Different from W counter values, the strict global ordering of the R and C counter values is very important to the correctness of the HBaseSI protocol.

The CommitRequestQueue table (Table 3.4) is used as a queue for ordering commit attempts and checking for conflicting updates among concurrent transactions that try to commit at almost the same time. A transaction Ti, when trying to commit, enters this queue table by first inserting a row containing its unique transaction ID Wi (obtained from the "W Counter table") as the row key and its writeset as the columns. (The writeset column names are unique identifiers for the data locations in the user data tables.) After this row is inserted, the transaction requests and obtains a commit request counter value Ri (from the "R counter table") and enters Ri into the "RequestOrderID" column of its row. The sequence of first inserting a row, then getting a Ri counter value, and finally putting it under the "RequestOrderID" column is essential for the queuing mechanism of our SI protocol as we will explain later. The transaction's writeset items are marked as "Y", and this information is used to detect conflicting updates. The "RequestOrderID" column is used to order the commit attempts and enforce the "first-committer-wins" rule.

The CommitQueue table (Table 3.5) is a queue for transactions that are already cleared for committing but just waiting for their turns to be actually committed according to the ordering of their commit timestamps. Each row in this table corresponds to a transaction and is indexed by the unique transaction ID obtained from the "W Counter table" (Table 3.1). The "CommitTimestamp" column stores the timestamp obtained from the "C Counter" table (Table 3.3) which is used as the commit timestamp of the transaction. Note that a transaction Ti first writes a row in this table with row key Wi, then requests and obtains its CommitTimestampCi, and finally adds Ci to its row. This sequence is again essential for the queuing mechanism to work properly, as explained below.

The Committed table (Table 3.6) stores the meta-data records for all the committed transactions. Each row in this table represents a successfully committed transaction indexed by the commit timestamp as the row key with the writeset data items as columns, containing the HBase timestamps used to actually write the data

Table 3.3: C counter table. C stands for "commit timestamp".

Row Key	Counter
C	54

Table 3.4: CommitRequestQueue table.

Row Key	writeset item 1	writeset item 2	writeset item 3	RequestOrderID
W1	Y		Y	R1
W2	Y	Y		

to the user's HBase data tables. In fact, for any transaction, successfully inserting a row into this table means that the transaction is committed atomically and the data becomes durable. Moreover, any row key of the table can identify a consistent snapshot because the rows in the table are strictly ordered and automatically sorted by row keys, and committed transactions are guaranteed to arrive in the Committed table in order due to the queuing mechanism, as explained below. The Committed table is also used by transactions in various functional ways, such as looking for the most recently committed version of data when reading, and checking for writeset conflicts at commit time against previously committed records. Note that HBase's sparse column nature is crucial here for efficiency: the table can contain many columns, but each column typically contains only few elements, and can be scanned efficiently.

In HBaseSI, each transaction sees a consistent snapshot of all the data in HBase user tables, identified by the start timestamp of the transaction. When a transaction Ti starts, it first gets its start timestamp by reading the last row of the Committed table at the time it starts, and uses the row key of that row Cj as the start timestamp. So we have Si=Cj, and Ti will see all data committed by Tj, and any transaction committed before Tj. Transaction Ti also obtains a unique ID Wi from the "W Counter" table as its transaction ID. Then it performs reads/writes based on the snapshot identified by the start timestamp. Data being read/written are first saved in in-memory readset/writeset data structures so that repeated reads can be efficiently served from memory, except for the first read/write of a certain data item. In this way, it is guaranteed that the transaction reads its own writes at all times. Writes are applied to the user data tables immediately (speculatively) using the transaction ID Wi as the unique timestamp to write to HBase (recall from Sect. 2 that a timestamp can be specified when writing data to HBase). At commit time, the transaction puts itself into the CommitRequest table, may wait for its turn if there are any conflicting commit attempts, then checks for conflicts with committed transactions, and finally enters the CommitQueue table if it is cleared to commit. It then waits for all the other concurrent transactions in the CommitQueue table with smaller RequestOrderID to commit, and finally commits by atomically inserting a simple record row into the Committed table to make its writes durable. The pseudocode of the protocol is provided in Listing 1.

```
1  Transaction {
   Writeset {(dataLocation(n),value(n))}; //containing N items
3  Readset {(dataLocation(m),value(m))}; //containing M items
   Long Wi, Si; //Wi is transaction ID, Si is start timestamp
5  Long Ri; //Ri is request order ID
   Long Ci; //Ci is commit timestamp
7
   //method called at the start of transaction
9  Start() { //transaction starts
     Wi = GetTimestamp (W counter);
11     Si = LastLineFromCommittedTable().getRowKey();
   }
13
   //method to read data value
15 Read(dataTable, dataRow, dataColumn) {
       dataLocation = dataTable + dataRow + dataColumn;
17     if (dataLocation in WriteSet) {read from WriteSet; return dataValue;} //read own writes
       if (dataLocation in ReadSet) {read from ReadSet; return dataValue;} //repeated read-only value
19   committedRecord = ScanForMostRecentRow (in Committed table, range [0, Si] containing a column named
          as dataLocation); //Scan in range [0, Si] (row keys are C counter values not less than 0),
        and return the last record in the list
       Wread = committedRecord.valueAtColumnDataLocation(); //find the latest data version in snapshot.
```

Table 3.5: CommitQueue table.

Row Key	CommitTimestamp
W1	C1
W2	

Table 3.6: Committed table.

Row Key	writeset item 1	writeset item2	writeset item3
C1	W1	W1	
C2	W2		W2

```
        If the data item is not in the Committed table, Wread will be set to null
21      dataValue = readData(in dataTable, in dataRow, in dataColumn, with Wread); //read data. If Wread
        is null, no timestamp will be specified in the HBase read (recall that it is optional to
        specify a timestamp in reading from HBase)
        ReadSet.add (dataLocation, dataValue);
23      return dataValue;
    }
25
    //method to write data value
27  Write(dataLocation, dataValue) {
      WriteSet.add(dataLocation, dataValue);
29    writeToDataTable (dataLocation, dataValue, using Wi); //directly write to data tables with HBase
        timestamp Wi
    }
31
    //method for commit attempt
33  boolean Commit() {
        EnqueueForCommitRequest(); //queue up for requesting to commit
35      CheckConflictsInCommittedTable (up to Si, with conflicting WriteSet); //do scan in the Committed
        table for writeset columns in range [Si + 1, +INFINITY) and verify that there are no
        writeset conflicts
        If (clearedToCommit) {
37        EnqueueForCommitting(); //when cleared to commit, queue up to finally commit
        } else {
39       doCleanup(); //abort transaction, remove rows in system tables and data items written to user
          tables
        }
41  }

43  //method to get a counter value
    GetTimestamp(HBaseTimestampTable) {
45    IncrementColumnValue (HBaseTimestampTable) //the mechanism to issue globally unique and well-
        ordered timestamps from a central HBase table
    }
47
    //method to enqueue for commit request
49  EnqueueForCommitRequest() {
        WriteHBaseTableRow (into CommitRequestQueue Table, row Wi, columns WriteSet);
51      Ri = GetTimestamp(R counter);
        WriteHBaseTableRow (into CommitRequestQueue Table, row Wi, column Ri);
53    PendingCommitRequests = GetRowsWithConflictingWriteSet(From CommitRequestQueue Table); //one-time
        scan
      while (PendingCommitRequests.isNotEmpty()) { //there exist requests to update conflicting data
55        select a row from PendingCommitRequests;
        if (row has disappeared) {
57            remove row from PendingCommitRequests; //the other transaction has completed
        } else {
59            wait until Ri appears in the row;
              if (row.Ri is larger than its own Ri) { //the other request is later than self
61                remove row from PendingCommitRequests; //no need to consider
              } else { //the other request is earlier than self
63                wait until row disappears; //wait till the other request is handled
          remove row from PendingCommitRequests;
65            }
          }
67    }
    }
69
    //method to enqueue for committing
71  EnqueueForCommitting() {
```

```
    WriteHBaseTableRow (into CommitQueue Table, row Wi);
73  Ci = GetTimestamp(C counter);
    WriteHBaseTableRow (into CommitQueue Table, row Wi, Ci);
75  PendingCommits = GetAllRows (From CommitQueue Table); //one-time scan
    while (PendingCommits.isNotEmpty()) {
77      select a row from PendingCommits;
      if (row has disappeared) {
79        remove row from PendingCommits; //the other transaction has completed
      } else {
81        wait until Ci appears in the row;
          if (row.Ci is larger than its own Ci) {
83          remove row from PendingCommits; //no need to consider
          } else {
85          wait until row disappears;
            remove row from PendingCommits;
87        }
      }
89 }
   //proceed to commit
91  WriteHBaseTableRow (into Committed Table, row Ci, columns WriteSet each containing value Wi); //
        atomic commit operation
    DeleteOwnRecordIn(CommitQueue table);
93  DeleteOwnRecordIn(CommitRequestQueue table);
   }
95
   Main() {
97  Start();
    ... //do reads and writes
99  Commit();
   }
101
   }
```

Listing 1: Pseudocode for the HBaseSI protocol.

It is important to understand in detail how HBaseSI handles distributed synchronization among concurrent transactions concerning the global ordering of transaction commit requests and commits. HBaseSI makes use of distributed queues to manage transaction commits and to guarantee the "first-committer-wins" rule, instead of using other traditional methods such as data locks or consensus-based protocols. The benefit is simplicity in design and implementation, which may in turn improve performance by avoiding the complexity of handling deadlocks and mandating complicated negotiation protocols for reaching consensus on transaction commit decisions between distributed data hosting servers involved in each transaction. HBaseSI makes use of two queues, implemented as two HBase tables. One is the CommitRequestQueue (Table 3.4); the other is the CommitQueue (Table 3.5). The protocol to ensure a correct sequence of entering and exiting a queue is the same for the two queues and therefore we explain the protocol using one queue, the CommitRequestQueue, as an example. Recall that when a transaction Ti makes a request to start the commit process, it first inserts a row indexed by its unique transaction ID Wi (obtained from the "W Counter table"), then gets a commit request counter value Ri and puts it under the "RequestOrderID" column of its row. The Ri value determines the order of Ti in the queue. This sequence of operations is of essential importance to guarantee that no concurrent transaction will leave the queue out of order, as we explain now. After transaction Ti inserts counter value Ri into its row in the CommitRequestQueue table, it reads all records in the table once. It then waits until all rows of transactions Tj it has read obtain Rj values in the table. This is essential to allow the queue to function based on the order of the R counter values: Ti is guaranteed to see any transaction Tj still in the queue that may have Rj<Ri, even if Rj appears in the table after Ri. This is so because Ti reads the table *after* it has obtained Ri, and any Tj still in the queue that may have Rj<Ri is guaranteed to have its row in the table at that time, because it inserted its row *before* requesting Rj. Ti will not proceed to the commit process until all Tj with Rj<Ri have left the queue, guaranteeing that transactions are processed in order and establishing the "first-committer-wins" rule. Based on the strict sequence of transactions entering the queue table, the protocol to ensure the ordering of exiting the queue is shown in Listing 1: pseudocode line 49 to 69. The pseudocode contains an optimization of the basic queuing protocol: transactions in the queue only need to wait for transactions that have a conflicting writeset. The same queuing protocol, using C counter values Ci, is also used to guarantee that transactions that are cleared to commit arrive in order in the Committed table, see lines 71-89 in the pseudocode. Using this queuing protocol, we can make sure that transactions follow the exact order as specified by their globally unique and well-ordered counter values. With the queuing mechanism, we can easily enforce a strict global ordering of transaction commits.

Table 3.7: Shop table.

Row Key	iPhone4	BlackBerry
Stock	1	3

Table 3.8: Committed table.

Row Key	Shop:Stock:iPhone4	Shop:Stock:BlackBerry
C6	W6	W6

Transactions in HBaseSI satisfy ACID properties as well as strong SI. Atomicity is provided by the underlying HBase atomic row write functionality because the final commit process only requires a single row write to the Committed table (Listing 1: pseudocode line 91). Durability is guaranteed by the underlying persistent data storage mechanism, i.e., Hadoop HDFS, because all the data in HBase are stored in HDFS. Consistency is maintained because only valid data is inserted into the HBase tables through the provided APIs and transactions never leave HBase in a half-finished state. The isolation level provided by HBaseSI is strong snapshot isolation. Strong SI requires that a transaction reads/writes in isolation upon a consistent snapshot of data identified by a start timestamp. Seen from the protocol above, our system guarantees that a transaction can see all the updates committed before it starts (start timestamps are row keys from the Committed table and any row key in the Committed table can identify a consistent snapshot containing all the previous committed updates). Our system also guarantees that transactions can only commit (atomically) if no conflicting updates have been inserted by previously committed concurrent transactions. Therefore strong SI holds.

3.2. Protocol Walkthrough by Example. We now describe the transactional SI protocol along with the system table usage in more detail by walking through the process of handling two concurrent transactions with conflicting updates under a concrete example scenario. In this example scenario, Alice and Bob intend to purchase smart phones from an online shop. They make their purchases by doing transactions involving several data tables of the shop stored in HBase, for example, item inventory, billing, etc. For simplicity, we limit their transactions to updating the same "Shop" table containing information about the number of available smart phones in stock.

Initially, the Shop table shows that the stock is updated with 1 iphone4 and 3 BlackBerrys (Table 3.7) by a transaction with unique ID W6 and commit timestamp C6 (Table 3.8). The Committed table contains a record for this stock update. Bob and Alice start transactions Ta and Tb concurrently, with start timestamps Sa=C6 and Sb=C6 (note that snapshots of different transactions can be the same, such as in this case). Now let's assume that Alice and Bob both read the stock of iPhone4 and BlackBerry, and then Alice decides to buy 1 iPhone4 while Bob would like to buy both an iPhone4 and a BlackBerry. Transactions Ta and Tb query the Committed table using the start timestamps Sa=C6 and Sb=C6 to get the most recently committed version of the stock data of both types of phones. They will both obtain HBase timestamp W6 and use W6 to read the stock from the Shop table and put the results into their readsets. (Listing 1: pseudocode line 15 to 23) After that they perform writes to update the stock and put data into their writesets (Listing 1: pseudocode line 27 to 31). Note that writes are applied to the Shop table immediately using timestamp Wa by Ta and Wb by Tb respectively, which is facilitated by the multi-version support of HBase (Sect. 2.2). We choose to write the data into the data tables speculatively to make the commit process faster. The writes become visible to other transactions only after the transaction has successfully committed.

When they are ready to attempt to commit, Ta and Tb use their transaction ID (Wa for Ta and Wb for Tb) as the row key to add a row to the CommitRequestQueue table with their writeset items as columns respectively (Table 3.9). Both transactions enter the CommitRequestQueue table a row with values for their writesets, and then request their commit request ID from the R Counter table. Then they put the commit request IDs, Ra and Rb, under the RequestOrderID column and perform a scan of the entire CommitRequestQueue table for all other row records with conflicting writeset items. This is to find any conflicting concurrent commit requests that may have Rj<Ra or Rj<Rb in the queue from transactions Tj. In our example, assume that Tb finishes inserting Rb into its row and that the row for Ta has not appeared in the table yet. Tb then scans the CommitRequestQueue

Table 3.9: CommitRequestQueue table.

Row Key	Shop:Stock: iPhone4	Shop:Stock: BlackBerry	RequestOrderID
Wa	Y		Ra
Wb	Y	Y	Rb

Table 3.10: CommitQueue table.

Row Key	CommitTimestamp
Wb	Cb

and finds no conflicts (Ta has not inserted its row yet). Then Tb can proceed to scan the Committed table to check if there are any conflicting committed transactions with commit timestamp larger than its start timestamp (C6). Assume there are none. Tb is now cleared for committing and atomically (line 91) adds a row with its transaction ID Wb as the row key to the CommitQueue table (Table 3.10). After adding the row, it requests and obtains a commit timestamp Cb and then puts it into its row under the CommitTimestamp column. It then waits in the CommitQueue for its turn according the CommitTimestamp to finally commit. This wait in the CommitQueue guarantees that all committed transactions Ti appear in the Committed table in the order of their commit timestamps Ci, and thus that all the records appearing in the Committed table are well ordered. After Tb finishes committing (see the resulting Committed table in Table 3.11), it will delete its row in both the CommitQueue and CommitRequestQueue (Listing 1: pseudocode line 92-93). In the meantime, assume Ta finishes inserting its row a bit later, and after it scans the CommitRequestQueue table for rows with conflicting columns, it sees that Tb has already entered the CommitRequestQueue with a conflicting writeset and RequestOrder ID Rb. Since Rb<Ra, Ta waits until row Tb disappears (meaning that Tb has either been committed or aborted) before proceeding (Listing 1: pseudocode line 54-65).

3.3. Read Optimization. An optimization for performance to the protocol above is necessary because the size of the Committed table grows linearly as transactions commit (each committed transaction creates a corresponding row that persists in the Committed table). Recall that when reading a data item, HBaseSI needs to scan all the rows in the Committed table up to the snapshot timestamp and iterate through the records in the result list of the scan to find the most recent data version. As shown in Fig. 4.4 below, the time it takes for scanning and iterating through the records grows linearly as the number of rows containing the target columns to scan increases. It would be good if only a small range of the committed table needs to be scanned by newly arrived transactions if the most recently known committed data version is kept somewhere globally visible. Following this idea, an extra system table called "Version table" is created (Table 3.12). Each row in the version table corresponds to a data item that has been written to, identified by its table, row and column name combination. Instead of using a centralized system component to constantly update the Version Table records, every transaction is responsible to update the records when new versions of data are read. With the Version table, when a transaction Ti tries to read any data item, it needs to query the version table first to see if there is a data version record. If there is a record and the commit timestamp Cj in the record is before Si, then Ti only scans the Committed table in the range [Cj, Si]. If no previous version is found or the version found is more recent than the snapshot time Si, a full scan of the Committed table up to the snapshot point Si is necessary. If a newer version is detected and read, the reading transaction updates the Version table record after reading the data item.

The adjusted pseudocode for reading with Version table is in Listing 2.

```
1 Read(dataTable, dataRow, dataColumn) {
      dataLocation = dataTable + dataRow + dataColumn;
3     if (dataLocation in WriteSet) {read from WriteSet; return dataValue;}
      if (dataLocation in ReadSet) {read from ReadSet; return dataValue;}
5     Cj = ScanVersionTable (dataLocation); //if the data item doen't exist in the Version table, Cj =
          0
      if (Cj <= Si) {
7     committedRecord = ScanForMostRecentRow (in Committed table, range [Cj, Si] containing a column
          named as dataLocation); //Scan in range [Cj, Si], and return the last record in the list
```

Table 3.11: Committed table.

Row Key	Shop:Stock: iPhone4	Shop:Stock: BlackBerry
C6	W6	W6
Cb	Wb	Wb

Table 3.12: Version table. For example, the most recently read version of the data item stored in user data location DataLocation1 was committed by the transaction with commit timestamp C17.

Row Key	CommittedTimestamp
DataLocation1	C17
DataLocationM	C8

```
    } else {
9     committedRecord = ScanForMostRecentRow (in Committed table, range [0, Si] containing a column
          named as dataLocation); //Scan in range [0, Si] (row keys are C counter values not less than
           0), and return the last record in the list
    }
11    if (committedRecord > Cj) {
        UpdateVersionTable (dataLocation, committedRecord);
13    }
      Wread = committedRecord.valueAtColumnDataLocation(); //find the latest data version in snapshot.
          If the data item is not in the Committed table, Wread will be set to null
15    dataValue = readData(in dataTable, in dataRow, in dataColumn, with Wread); //read data. If Wread
          is null, no timestamp will be specified in the HBase read (recall that it is optional to
          specify a timestamp in reading from HBase)
      ReadSet.add (dataLocation, dataValue);
17    return dataValue;
}
```

Listing 2: Read with Version table.

3.4. Handling Stragglers. In the protocol above, a transaction needs to wait in two queues, the CommitRequestQueue and the CommitQueue. Due to many possible failure conditions, transactions could stay in waiting forever if one or more of the previously submitted transactions get stuck in the commit process and never delete their corresponding rows in the above two queue tables. We call those transactions that do not terminate properly in a timely manner "stragglers". Measures must be taken to not only prevent such stragglers from hampering the other active transactions, but also to avoid any potential data inconsistency issues caused by re-appearing transactions that had been deemed to be dead.

HBaseSI handles stragglers by adding a timeout mechanism to the waiting transactions. More specifically, the waiting transactions can kill and remove straggling/failed transactions from the CommitRequestQueue or CommitQueue based on the clock of the waiting transaction if a preconfigured timeout threshold is reached. A problem associated with this method is that a straggler may come back to life and try to resume the rest of its commit process after its records in either queues are removed, which could cause data inconsistencies and incorrect SI handling. The solution to this problem is to use the HBase atomic CheckAndPut method on two rows at once in the Committed table when doing the final commit rather than only using a simple atomic row write operation on one row. The difference between CheckAndPut and simple row write is that the former method guarantees an atomic chain of two operations involving checking a row and writing to a possibly different row in the same HBase table, whereas the latter method only guarantees atomicity for a single row write operation. To use the CheckAndPut method, we first add an extra row called "timeout" in the Committed table (Table 3.13). When it starts, each transaction first marks the column named after its unique transaction ID Wi (obtained from the W Counter table) in the "timeout" row as "N", meaning that the transaction is not in timeout by default (a non-empty initial value "N" must be set because the CheckAndPut method does not work with empty column values). Later, in the commit process, if a transaction is deemed a straggler, other transactions will put a "Y" under the column named after the unique transaction ID of the straggler in the "timeout" row, and then delete the corresponding records of the straggler in both the CommitRequestQueue and the CommitQueue. (Note that the sequence of first marking the straggler in the Committed table and

Table 3.13: Committed table.

Row Key	writeset item 1	writeset item 2	W6	Wi	Wj
T6	W6	W6			
timeout			N	N	Y

only then deleting rows in the two queues is essential to the correctness of the SI mechanism). When a healthy transaction commits, it performs an atomic CheckAndPut: it checks for "N" in the "timeout" row, and if the check is successful, it puts its row into the Committed table. If the value under its corresponding column is still marked as "N", it can indeed successfully insert its row into the Committed table; otherwise it knows it has been marked as a straggler and should abort by deleting its records in both the CommitRequestQueue and the CommitQueue tables, if those records still exist. In this way, HBaseSI can make sure that no transaction can commit once it is marked as a straggler. There is no problem if after a transaction commits successfully by inserting a row into the Committed table, it fails to delete the corresponding rows in the queues on time; those records will be removed by waiting transactions after the timeout and SI is not compromised. Note that for garbage cleaning purposes, after a transaction successfully commits, it will remove the corresponding column value in the row "timeout".

3.5. Discussion. In this section, some further issues about the HBaseSI design and usage are discussed. First, there is no roll back or roll forward mechanism in HBaseSI and there is no explicit transaction log either. It is interesting to ponder on how HBaseSI supports ACID transactions, even in the face of failures, without those traditional mechanisms used in DBMSs. In fact, this all attributes to two very important HBase properties. The first one is that HBase stores many versions of data and allows reads/writes of data using a specific timestamp. This HBase property makes it possible for every concurrent transaction to write preliminary versions of data but only the successfully committed transactions get to publish the write timestamps they used in the Committed table for future reads. In other words, no roll back is necessary because uncommitted data won't be used in any case. The other property is the atomicity of the HBase row write and CheckAndPut methods. Using these atomic methods, HBase guarantees that once a row is inserted into the Committed table successfully, it becomes durable and is guaranteed to survive failures (media failure is handled by HDFS which stores data replicated across distributed locations).

Second, we discuss some design choices that affect performance such as scalability and disk usage. HBaseSI inherits many of the desirable properties of HBase because it is only a client library and imposes little overhead concerning system deployment. However, users need to be aware that in order to achieve several design goals, HBaseSI made some sacrifices for performance. For example, four important goals HBaseSI tries to achieve are: 1. global strong SI across table boundaries; 2. non-intrusive to user data tables; 3. non-blocking start of transactions with snapshots that are as fresh as possible (strong SI), and non-blocking reads; 4. strict "first-committer-wins" rule without lost transactions (transactions only abort when there is no chance they will be able to commit successfully). In order to achieve goal 2, HBaseSI is designed to use a separate set of system HBase tables for maintaining transactional metadata for all user tables instead of creating extra columns in each separate user table, which inevitably creates potential performance bottlenecks at the small number of global system tables. HBaseSI is therefore not designed to provide scalability in terms of the number of transactions per unit time, but its target is to provide scalability in terms of cloud size and user data size. HBaseSI makes the final commit process as short as possible and allows writes to insert preliminary data into the user data tables as the transaction proceeds rather than waiting till the commit time to apply all the updates (note that when a transaction aborts, it should remove its written items from user tables), avoiding chances for varyingly large waiting latency incurred by transactions with large writesets to be applied at commit time. In essence, HBaseSI trades disk space for high throughput in transaction commits. Additionally, it is important that the number of data versions HBase table locations can hold is set sufficiently high. Furthermore, a dedicated garbage cleaning mechanism should be created for optimized disk usage, with a policy on maximum transaction duration (such a policy is important to guarantee that the data that gets garbage cleaned is not needed by any long-running transactions).

Third, we discuss the efficiency of having transactions wait in queues when committing. Recall that in the

HBaseSI protocol, update transactions first wait in the CommitRequestQueue for the purpose of establishing an order in committing transactions and guaranteeing the "first-committer-wins" rule, and then wait in the CommitQueue after they are cleared for committing for the purpose of guaranteeing a correct global sequence of commits so that each row in the Committed table can identify a consistent snapshot of the data tables. Note that the first wait is only for transactions with conflicting writesets, but the second wait results in sequential processing of all concurrent transactions, no matter whether the writesets are in conflict or not. Although these two waits are essential for the commit queuing mechanism to work so that global strong SI can be achieved, it may sometimes be more efficient to relax the second wait to the extent that a transaction only waits for other transactions that use the same set of user tables. This would require transactions to declare in advance which groups of tables they use. This relaxation is reasonable in real-world applications. HBaseSI can be very easily adapted to such extended usage scenarios to make transactions more efficient in terms of minimizing unnecessary wait times in the CommitQueue. The decision of whether to use the extended scheme would be at the users' discretion.

Finally, we discuss the cost of adopting HBaseSI and the easiness of reverting back to non-SI default HBase. Normally, once one starts to use HBaseSI, all the read/write operations must be performed through the HBaseSI API rather than the default HBase API. Only through the HBaseSI API can a transaction find the correct timestamp used in writing the most up-to-date data, or make its committed updates accessible. However, it is very easy to write a small tool to help restore the user data tables back to a state that users can use their data tables in the default HBase manner. The tool only needs to write the latest version of committed data to all the user data tables once, without specifying timestamps (so that the HBase default timestamps are used). The next time users want to use HBaseSI again, they can directly use it without any required changes.

4. Performance Evaluation on Amazon EC2. The general purpose of this performance evaluation section is to quantify the cost of adopting the HBaseSI protocol in handling concurrent transactions. Therefore tests are performed on each critical step of the HBaseSI protocol, with comparison to the performance of bare-bones HBase when possible. Additionally, because HBaseSI is the first system that achieves global strong SI on HBase, there are no other similar systems to compare with for some of the properties. As a result, for those properties, the tests serve the purpose of showing the users the expected behavior of the system. Furthermore, as mentioned in Sect. 3.3 above, HBaseSI uses a set of global system tables that facilitate non-blocking reads and a strict "first-committer-wins" rule, but may become performance bottlenecks if accessed by many concurrent transactions. In order to make the performance effect of this design decision more apparent, we decided to deploy all systems tables at the same HBase region server by running all the HBase components on a single machine, mimicking the possible extreme real-world condition when concurrent transactions significantly outnumber the HBase servers. The test results are thus expected to show the system performance under heavy loads.

We use 20 Amazon machines in total to perform the tests and we are aware that performance variations may be observed in Amazon instances [9]. The test results may be affected by this to some extent but should be enough for proof-of-concept purposes. A high memory 64-bit linux instance with 15 GB memory, 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each) and high I/O performance is used to host both the Hadoop and HBase server components. Up to 19 other 64-bit linux instances with 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each) and high I/O performance are used to run client transactions. All these machines are in the same Amazon availability zone so that the network conditions for each instance are assumed to be similar.

In the tests, each machine runs a single client program issuing transactions if the total number of clients is less than 19. If the total number of clients is more than 19, an equal number of concurrent clients are run at each machine instance. For example, each machine instance can run 1, 2, or more clients with the total number of transactions being 19, 38, etc. At each client, transactions are issued consecutively one after another. In other words, a new transaction will only be issued when the previous one has finished executing, having either committed or aborted.

The goal of Test 1 is to measure the performance of the timestamp issuing mechanism in terms of throughput. In the test, each client connects to the server and requests a new timestamp directly after being granted one. After a starting flag is marked in an Indicator table, all clients run for a fixed period of time and stop. The throughput is calculated by dividing the total number of timestamps issued by the length of the fixed time period. Figure 4.1 shows the result of this test. Apparently the server gets saturated at a total throughput of about 360 timestamps per second, or about 30 million timestamps per day. Note that the timestamp generating

mechanism currently used by HBaseSI is the most straightforward solution a user can get by using bare-bones HBase functionalities. Other more efficient timestamp generating mechanisms with much higher throughput can also be adopted if the user desires, such as the one used by Google's Percolator system [8] which generates 2 million timestamps per second from a single machine.

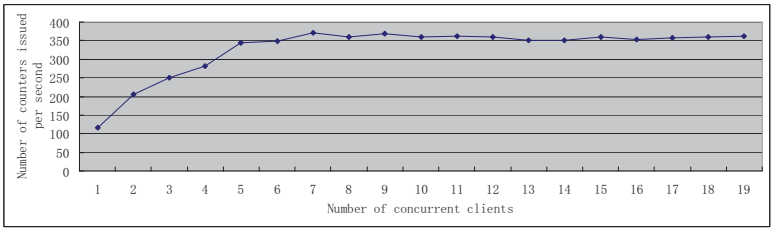

Fig. 4.1: Test 1, performance of the timestamp issuing mechanism through counter tables.

The goal of Test 2 is to measure the performance of the start timestamp issuing mechanism via the Committed table in terms of throughput, i.e., how many transactions can be allowed to start per second (in order for a transaction to start, a start timestamp must be issued first) with an increasing number of concurrent clients. Recall that the mechanism to obtain a start timestamp is different from getting a unique counter value from one of the counter tables. Instead, a transaction needs to read the last row of the Committed table at the time it starts and use the row key as its start timestamp. In this test, the clients all connect to the server first and then wait for a signal in the Indicator table to start at the same time. During the test, a program is run at the EC2 instance running the HBase server inserting a new row to the Committed table continuously, mimicking the real-world scenario where the Committed table keeps growing in size because of newly committed transaction records. The throughput is calculated in the same way as Test 1. Figure 4.2 shows the result for Test 2. The throughput stabilizes at about 420 timestamps per second due to server saturation, slightly higher than the result obtained from Test 1. The higher performance is expected because in Test 1 an atomic function call to increment a common column value is issued each time a counter value is to be obtained by each concurrent client, potentially causing a blocking write conflict at the HBase server, while in Test 2, only scanning the last row of the Committed table is necessary. The performance is thus satisfactory to the extent that the start timestamp mechanism is not the limiting bottleneck for starting new transactions even if the mechanism requires that every transaction should read from the Committed table at starting time.

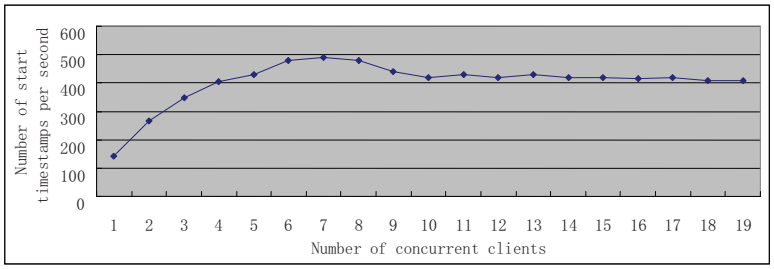

Fig. 4.2: Test 2, performance of the start timestamp issuing mechanism.

The goal of Test 3 is to study the comparative performance of transactions with SI that contain a set of read/write operations, against executions of the same number of read/write operations with bare-bones HBase, for varying numbers of operations per transaction. In the test, we run 1 client only, vary the number of operations per transaction and measure the time spent on each read/write operation. Additionally, in order to control the performance overhead associated with scanning a growing Committed table (recall from Sect. 3 that each SI read needs to scan the Committed table first to get the most up to date data version before actually reading the data), after each client run, the Committed table is manually cleaned. (In this test, no previous data versions exist, because the Committed table is cleaned up after each previous transaction execution and data locations are only written to once, but a quick scan is still executed for every read). The result of the test quantifies the performance overhead of transaction SI over bare-bones HBase. The results in Fig. 4.3 show the

startup/commit overhead of the protocol and how it can be amortized as the number of read/write operations per transaction grows. This indicates that the protocol is more efficient for transactions involving a larger number of operations per transaction or transactions with longer inter-operation intervals (user "think time" during user interactions) to better amortize the transaction startup/commit overhead.

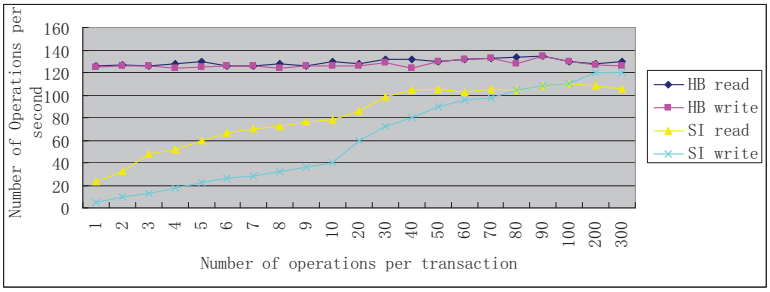

Fig. 4.3: Test 3, comparative performance of executing transactions with SI against bare-bones HBase without SI.

The goal of Test 4 is to measure the time needed to scan a column in a data table over a growing row range (each row contains a data value in the column scanned). The expected result is a linear growth of time corresponding to the number of table rows scanned. The result is used to show the necessity of using the Version table when performing reads in order to avoid costly full scans of the Committed table on every read. In this test, a single client is executed to scan a data table with a continuously growing row range. The test result is shown in Fig. 4.4 and is exactly according to expectation with linear growth in time.

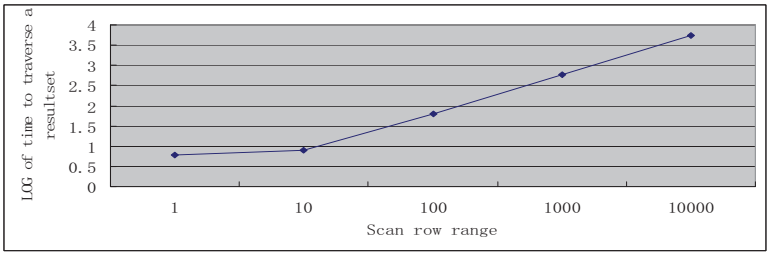

Fig. 4.4: Test 4, time to traverse a resultset against a varying number of rows to scan.

The goal of Test 5 is to measure the comparative performance of transactional SI with the use of the Version table on workloads complying with the TPC-W benchmark [13], which is used widely for evaluating database performance under OLTP loads. The TPC-W benchmark describes several different kinds of workloads with mixed read/write operations corresponding to real-world e-commerce scenarios, such as online shopping. A "browsing mix" is composed of transactions containing 95% read and 5% write operations; a "shopping mix" is composed of 80% reads and 20% writes; and an "updating mix" is composed of 50% reads and 50% writes. In the test, we run clients executing the above three kinds of workloads with a varying number of concurrent clients, each executing a random number of reads/writes according to the above specifications with an average of 15 operations per transaction, upon a table with 10,000 data rows. We measure two things: overall throughput (number of transactions per second) and average commit time for update transactions (the average time spent in the commit process). The overall throughput includes both successful and aborted transactions and shows the general system capacity in handling concurrent transactions. It is also interesting to see how much time is spent in the CommitRequestQueue and the CommitQueue separately because for different types of mixed workloads, the ratio of the number of update transaction requests and the number of actually committed transactions is different. The result for total throughput is shown in Fig. 4.5. An interesting point for this result is the comparative performance between these types of workloads. As we can see, as the number of concurrent clients grows, the shopping mix has the lowest throughput while the browsing and update mix have similar throughput. The reason why the shopping mix has the lowest throughput is because this mix actually has the most number

of successful update transactions processed among the three mix types: the browsing mix doesn't have many costly update transactions, and the updating mix doesn't have many successfully committed update transactions either because of the higher probability of having conflicts (and we count failed transactions in the throughput). The sharp drop in throughput, especially for the updating mix, when there are 95 concurrent clients is because of both the server saturation and the extra wait time in the CommitQueue.

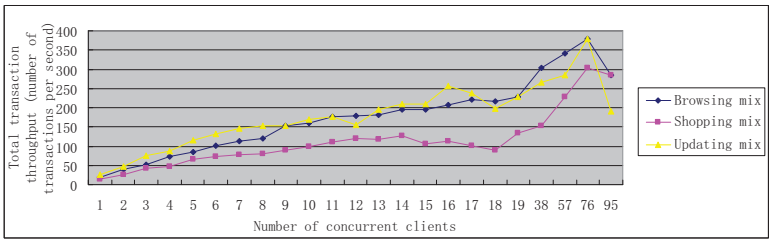

Fig. 4.5: Test 5, general performance of executing transactions with SI under TPC-W workloads.

Results for the average commit time for all three types of mixed workloads are shown in Figs. 4.6, 4.7 and 4.8, respectively. As for the browsing mix (Fig. 4.6), update transactions are relatively rare (5%). Therefore conflict probability is low. Transactions that get queued in the CommitRequestQueue are also likely to be able to commit successfully in the end. Therefore transactions tend to spend almost the same time on average staying in both queues. As for the shopping mix (Fig. 4.7), more update transactions are queued up for committing after passing the commit request checking stage at the CommitRequestQueue. Because there are almost no conflicts (the conflict rate will increase with a large number of concurrent clients, especially with respect to the total number of data items under shared access) and the CommitRequestQueue is basically skipped for most committing transactions, the wait time in the CommitQueue is much higher. As for the updating mix (Fig. 4.8), because there is a much higher conflict probability than for the other two mix workloads, more transactions are aborted at the checking stage in the CommitRequestQueue. Therefore the point at which the wait time in the CommitQueue outruns the wait time in the CommitRequestQueue comes later than for the shopping mix workload. The results in Figs. 4.6, 4.7 and 4.8 also indicate that the timeout threshold used in the straggler handling mechanism should be set according to the type of mix workload and may need to be adjusted according to the number of concurrent requests.

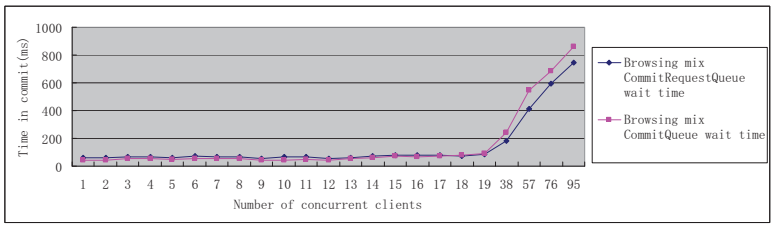

Fig. 4.6: Test 5, browsing mix wait time in both CommitRequestQueue and CommitQueue.

The goal of Test 6 is to test the effectiveness of the straggler handling mechanism. We use the "shopping mix" from Test 5 with 19 concurrent clients and add an abort ratio at the end of each transaction. With an increasing abort ratio, we measure the total throughput in terms of transactions per second. Because the artificially inserted aborts occur at the end of transactions while transactions wait in the CommitRequestQueue after completing all the reads/writes, we still count the aborted transaction into the calculation of the throughput. The failed transactions become stragglers in the CommitRequestQueue table that have to be removed by live transaction processes. The results show how random transaction faults affect the performance of the SI protocol. As seen in Fig. 4.9, the system achieves throughput similar to the case with no artificially inserted faults (because we also count the aborted transactions in the throughput calculation). We can also see from Fig. 4.10 that the duration of successful transactions stays almost constant in the face of failures, indicating that the straggler handling mechanism is effective in bounding healthy transaction duration.

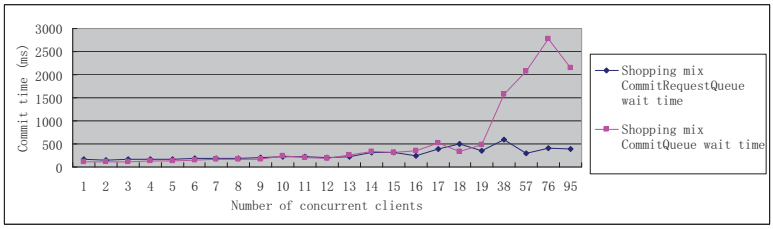

Fig. 4.7: Test 5, shopping mix wait time in both CommitRequestQueue and CommitQueue.

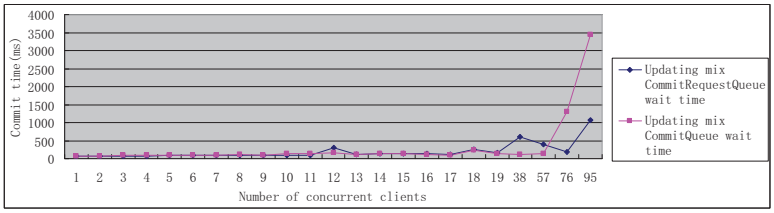

Fig. 4.8: Test 5, updating mix wait time in both CommitRequestQueue and CommitQueue.

5. Related Work. Several transactional systems exist for HBase, but none provide SI. The HBase project itself includes a contributed package for transactional table management, but it does not support recovering transaction states after region server failures. G-store [4] supports groups of correlated transactions over a pre-defined set of data rows (called "Key Group") specified for each group of transactions respectively, but assumes that the number of keys in a Key Group is small enough to be owned by a single node. CloudTPS [10] implements a server-side middleware layer composed of programs called local transaction managers (LTMs), but introduces extra overhead of middleware deployment, data synchronization, and fault handling.

Only recently two relevant papers were published independently at almost the same time about achieving snapshot isolation for distributed transactions, for HBase and for BigTable: we published a paper describing our preliminary system (the predecessor of the system described in this paper) to support transactions with SI on top of HBase [11], and Google published a paper about their system called "Percolator" [8] supporting transactions with SI on top of BigTable. The two systems share many design ideas yet are different in some major design choices.

HBaseSI is an extended and improved version of our preliminary system of [11]. It is similar to the preliminary system and similar to Google's Percolator [8] in that: all three systems are implemented as a client library rather than a set of middleware programs and allow client transactions to decide autonomously when they can commit (there is no central process to decide on commits); they all rely on the multi-version data support from the underlying column store for achieving snapshot isolation, and store transactional management data in column store tables; they all make use of some centralized timestamp issuing mechanism for generating globally well-ordered timestamps; and after starting using either of the systems, users must use the systems for all the subsequent data processing operations in order to guarantee data consistency. HBaseSI is superior to the preliminary system in that: HBaseSI is the first system on HBase to support global strong SI rather than the "gap-less" weak SI in the preliminary system; it uses a completely different mechanism in handling distributed synchronization (HBaseSI uses distributed queues to guarantee a correct sequence of transaction execution, while the preliminary system uses a complicated and inefficient mechanism to obtain snapshots); the preliminary system is inefficient because its PreCommit table grows without bound and has to be searched in its entirety by transactions attempting to commit; it provides a simple mechanism for handling stragglers, whereas handling stragglers for the system proposed in [11] would be overly complicated.

In addition to the similarities listed above, HBaseSI shares with Percolator its support of global strong SI. HBaseSI and Percolator are also very different in several other aspects: HBaseSI focuses on random access performance with low latency whereas Percolator focuses on analytical workloads that tolerate large latency; HBaseSI is non-intrusive to existing user data tables and stores the version information and transaction informa-

Fig. 4.9: Test 6, throughput seen at each client under a varying failure ratio.

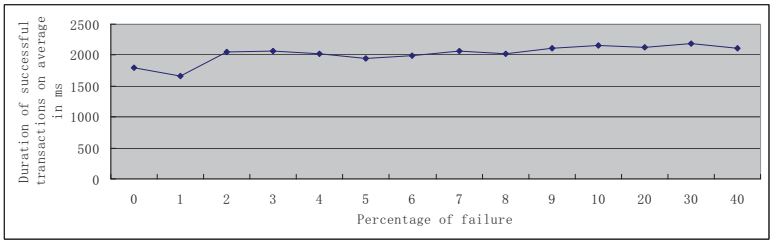

Fig. 4.10: Test 6, average duration of successful transactions under a varying failure ratio.

tion in extra system tables, whereas Percolator is intrusive to existing user data and stores the same information in two extra columns in every user tables (but this design decision of HBaseSI makes it less scalable than Percolator concerning the number of concurrent transactions); HBaseSI supports non-blocking starts of transactions and does not block reads, whereas Percolator may block reads while data is being committed which may harm performance; HBaseSI uses distributed queues in handling synchronization and concurrency rather than using traditional techniques such as data locks as in Percolator; and two concurrently committing transactions could unnecessarily both fail in Percolator but not in HBaseSI. In short, the two systems are designed with different purposes in mind and each may excel at one aspect and not another. Note also that the protocol described in Percolator cannot be trivially ported onto HBase, because HBase does not support BigTable's atomic single-row transactions, allowing multiple read-modify-write operations to be grouped into one atomic transaction as long as they are operating on the same row.

6. Conclusions and Future Work. This paper presents HBaseSI, a light-weight client library for HBase, enabling multi-row distributed transactions with global strong SI on HBase user data tables. There exists no other systems providing the same level of transactional isolation on HBase yet. HBaseSI tries to achieve several design goals: achieving global strong SI across table boundaries; being non-intrusive to existing user data tables; strictly enforcing the "first-committer-wins" rule for SI; supporting highly responsive transactions with no blocking reads; and employing an effective straggler handling mechanism. The performance overhead of HBaseSI over HBase is modest, especially for longer transactions involving a larger number of read and write operations per transaction. Future work includes implementing some helpful tools to optimize disk usage and possibly extending HBaseSI to increase its scalability by distributing the transactional metadata tables.

REFERENCES

[1] D. Agrawal, A. E. Abbadi, S. Antony, and S. Das, *Data management challenges in cloud computing infrastructures*, Proc. DNIS'10 (2010), pp. 1–10.
[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, *A critique of ANSI SQL isolation levels*, Proc. of SIGMOD (1995), pp. 1–10.
[3] 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*, Proc. OSDI, USENIX Association (2010), pp. 205–218.
[4] S. Das, D. Agrawal, and A. El Abbadi, *G-store: a scalable data store for transactional multi key access in the cloud*, Proc. SoCC '10 (2010), pp. 163–174.
[5] K. Daudjee and K. Salem, *Lazy Database replication with snapshot isolation*, Proc. of VLDB (2006), pp. 715–726.

[6] F. Farag, M. Hammad, and R. Alhajj, *Adaptive query processing in data stream management systems under limited memory resources*, Proc. of the 3rd workshop for Ph.D. students in information and knowledge management, PIKM '10 (2010), pp. 9–16.
[7] P. Helland, *Life beyond distributed transactions: an apostate's opinion*, CIDR (2007), pp. 132–141.
[8] D. Peng and F. Dabek, *Large-scale incremental processing using distributed transactions and notifications*, Proc. OSDI, USENIX Association (2010), pp. 1–15.
[9] J. Schad, J. Dittrich, and J. Quiané-Ruiz, *Runtime measurements in the cloud: observing, analyzing, and reducing variance*, Proc. VLDB Endow. (2010), 3, 1-2, pp. 460–471.
[10] Z. Wei, G. Pierre, and C.-H. Chi, *Scalable Transactions for Web Applications in the Cloud*, Proc. of the Euro-Par Conference (2009), pp. 442–453.
[11] C. Zhang and H. De Sterck, *Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase*, Proc. of Grid2010 (2010).
[12] The Apache Software Foundation, *An open-source, distributed, versioned, column-oriented store. http://hbase.apache.org/*, retrieved April 15, 2011.
[13] The Transaction Processing Performance Council, *A transactional web e-Commerce benchmark. http://www.tpc.org/tpcw/default.asp*, retrieved April 15, 2011.

Edited by: Dana Petcu and Marcin Paprzycki
Received: May 1, 2011
Accepted: May 31, 2011