Preserving atomicity and isolation for multi-row transactions in column-oriented heterogeneous distributed databases

Dharavath Ramesh*, Chiranjeev Kumar and Amit Kumar Jain

Department of Computer Science and Engineering,
Indian School of Mines,
Dhanbad – 826004, Jharkhand, India
Email: ramesh.d.in@ieee.org
Email: k_chiranjeev@yahoo.co.uk
Email: amit.jain.tech@gmail.com
*Corresponding author

Abstract: Traditional databases have limitation of scalability with respect to data as well as number of clients. Column-oriented databases have overcome this feature by minimising the cost. Column-oriented databases only ensure single row atomic transaction and does not support snapshot isolation. This paper presents about strong snapshot isolation (SI) and atomicity for multi-row distributed transactions in HBase. This HBase snapshot isolation uses a novel approach and handles distributed transactions at the end of individual clients. This is also designed to be scalable across large distributed databases in terms of data distribution. Some experiments have been performed extensively to preserve atomicity for distributed transactions in various environments. Experimental results show that the proposed methodology can serve better to preserve atomicity and snapshot isolation in column-oriented HDDBs for multi-row transactions.

Keywords: column-oriented database; multi-row transaction; heterogeneous distributed database; RDBMS; HBase; snapshot isolation.


Biographical notes: Dharavath Ramesh received his BTech from Kakatiya University, Warangal, India in 2004 and MTech from the Jawaharlal Nehru Technological University, Hyderabad, India in 2009. He is currently an Assistant Professor at the Department of Computer Science and Engineering, Indian School of Mines, Dhanbad, India. He works and publishes widely in the areas of distributed databases, database management systems, and cloud computing. He has organised workshops, including tutorials, and actively served in the programming committees of international conferences. He is a member of the ACM and IEEE.
Preserving atomicity and isolation for multi-row transactions

Chiranjeev Kumar received his ME and PhD from Motilal Nehru National Institute of Technology (MNNIT) and Allahabad University, Allahabad, UP, India. He is currently an Associate Professor at the Department of Computer Science and Engineering, Indian School of Mines, Dhanbad, India. He was a Gold Medallist of his ME batch at MNNIT, Allahabad 2001. His main research interests include mobility management in wireless networks, ad hoc networks, distributed databases, and software engineering. He has contributed many research papers in several refereed journals and conference proceedings of national and international repute. He is also the Guest Editor of many international journals. He is a member of the IEEE and ACM.

Amit Kumar Jain received his MTech from Indian School of Mines (ISM), Dhanbad in 2013. He is currently an Assistant Professor at the Department of Computer Science and Engineering, Gyan Sagar College of Engineering, Bhopal, Madhya Pradesh, India. His main research interests include distributed databases and mobile cloud computing.

This paper is a revised and expanded version of a paper entitled ‘Implementation of atomicity and snapshot isolation for multi-row transactions on column oriented distributed databases using RDBMS’ presented at the IEEE Conference on Communications, Devices and Intelligent Systems (CODIS-2012), Jadavpur University, Jadavpur, Kolkata, India, 28–29 December 2012.

1 Introduction

Consistency and isolation become increasingly important as soon as multiple applications and workloads with different needs interact with each other in the form of transaction execution scenario. Providing semantics and fully-fledged transactions often involves a significant penalty on the performance of the system since it is orthogonal to its goals. In distributed database system, there are four properties (atomicity, consistency, isolation, durability), to guarantee the data integrity of database. Atomicity property guarantees that each transaction is handled entirely instead of partially. Consistency property ensures that any transaction will bring the database from one valid state to another. Isolation property is defined as invisibility of operations of a transaction to the other transactions. Durability property guarantees that transactions that have committed will survive permanently. This paper focuses on the atomicity and isolation property. Since the isolation property critically affects the throughput of transactions in the database system, we should take account of the trade-off between isolation level and system performance. When attempting to maintain the highest level of isolation, a DBMS usually acquires locks on data which may result in a loss of concurrency. The most basic implementation of the isolation is based on a lock of data when it is read or written. However, it is known that the lock-based isolation adversely affects the performance of the database system. Recent DBMSs adopt a more effective method using snapshot, the so-called snapshot isolation (SI). In the SI, each transaction makes a copy (snapshot) of the data at the beginning of a transaction and updates the data on the snapshot, instead of the original data. After the update operation, each transaction sends a request for updating the original data, and the DBMS processes the requests according to the first-committer-wins rule. It has an advantage of the abort probability of a transaction to the lock-based isolation. As a novel isolation scheme, SI has received much attention in various fields.
Column-oriented NoSQL databases such as HBase (Vora, 2011), Big Table (Chang et al., 2006), AmazonS3 (Garfinkel, 2007) offers great scalability and availability for the applications. But this scalability and availability must be measured in terms of cost. First, these databases support very limited transaction features. Second, weak consistency such as SI (Berenson et al., 1995) is not supported by these databases. Third, they have very limited query join support. Moreover, indexing has been done on primary key only. On the other hand NoSQL databases have number of features other than scalability and availability. Some of these are

1. having random access nature for each data item
2. read relevant data (not entire row) only
3. no specified structure in advance like add or delete column instances.

Because of these features NoSQL become much more popular in this era.

On the other hand RDBMS is not much scalable, but has its own positive sides,

1. tremendous support of transactions for different applications
2. easy to add/modify a record
3. high throughput for limited data
4. support of ACID property scenario by transaction manager (TM)
5. supporting different consistency levels for different applications.

From above, we conclude that NoSQL is better for large data applications where data increases rapidly and RDBMS is useful for those applications. But in the current trend applications like corporate world, e-business and e-commerce an instance of database is needed for which is not only scalable but also consistent too. In this paper, we show how consistency and atomicity isolation is achieved for multi-row transaction in NoSQL of HBase using RDBMS.

The catalogue of this paper is as follows: Section 2 describes about literatures. Section 3 describes about HBase in brief. Section 4 tells about SI and also describes about proposed methodology. Section 5 describes about performance evaluation of our algorithm with a suitable example. Section 6 describes about technical results related to our proposed approach.

2 Related literatures

As internet use is growing rapidly, distributed database requirement is also increases in corporate world, e-commerce and e-business. These areas require scalability, consistency and transactional support. Even NoSQL database (such as HBase) is good choice for these applications in view of scalability. Zhang and De Sterck (2010) and Padhye and Tripathi (2012) suggested atomic support and consistent multi row distributed transactions. The idea of supporting atomicity and consistency for multi row distributed transaction is not new. Our work is mainly inspired by Zhang and De Sterck (2010), Padhye and Tripathi (2012), Berenson et al. (1995), Lars (2011), Cahill et al. (2008), Fekete et al. (2004, 2005) and Revilak et al. (2011).
Zhang and De Sterck (2010) proposed a technique to achieve SI by creating number of global tables for storing transactional metadata in HBase itself and using them for each transaction. As the metadata increases to the large extent with the time, the transactional history is also contains the transaction information in logs. That means as time is passed for the older entry, the metadata are not useful in any manner. With this, we can say that removal of old entry from metadata will not harm the system and moreover, improves the system performance. If metadata size is not growing to the large extent, then HBase will not be a good idea for the storage. So, RDBMS is good choice for storing the metadata with high query support and high throughput with limited data.

Several concepts of atomicity and isolation associated detection tools have been presented, including Wang and Stoller (2006) and Flanagan and Freund (2004). Some interleaving scenarios are defined in terms of utmost two locations from an atomic set (Hammer et al., 2008). Isolation and transactional support for distributed data stores is also a widely studied topic, and there has been some related work done, including support for lock-free transactions (Junqueira et al., 2011) and SI (Ramesh et al., 2013) for distributed databases. Zhang and De Sterck (2010, 2011) also implement SI for HBase, allowing multi-row distributed transactions for this column-oriented database. While the former approach uses additional meta-data on top of standard HBase, the latter introduces a more advanced client to support SI transactions. A methodology called ordering block writes in file system operations (except for possible space leaking on temporarily written blocks) was described in Ganger et al. (2000). Another scenario related to transactional databases which supports data durability and consistency through relational SQL or object-oriented interfaces is proposed by Garza and Kim (1988). Application level data integrity semantics are not visible at the storage firmware and therefore storage-level transactions (Seltzer et al., 2000). Prabhakaran et al. (2008) mentioned about synchrony to failure-atomic I/O may incur substantial overhead which are not suitable for protecting application data integrity. An instance called SQLite is a software library which supports conventional relational database manipulation via SQL (SQLite Atomic Commit, October 2012) is incurred in the literature. We further extend our architecture (Ramesh et al., 2012) to support atomicity and isolation for multi row transactions in column-oriented distributed databases. This is the extended approach to our previous work (Ramesh et al., 2012).

3 HBase

In HBase, multiple columns are associated with each row. Columns are grouped together to form a column family and related with each other, i.e., if a table have columns ‘A’, ‘B’, ‘C’ and ‘D’ and when column ‘A’ is referred for any operation (read or update) also accesses ‘B’ and ‘C’ then create a column family named ‘X’ which contains column ‘A’, ‘B’ and ‘C’. The data items for a column family are stored together. Each row has a unique row key. The data items are stored in sorted row key values. The rows are stored in sparse column fashion, i.e., there is no need to store columns those are null because of the sorted map concept.

Table 1 shows the logical view of table in HBase site (source: Apache HBase). There are two rows, row keys are ‘john’ and ‘martin’. Each row contains two column families ‘info’ and ‘roles’, but it might be possible that a row contains only one column family.
For the row key ‘john’ column family ‘info’ contains two columns ‘height’ and ‘state’, where the cell values are ‘9ft’ and ‘Texas’. Here column family roles for row key ‘john’ contains two column named ‘ASF’ and ‘B.Tech’ but for row key ‘martin’ contains three columns ‘ASF’, ‘B.Tech’ and ‘M.Tech’, that shows spare column is stored for row.

Table 1  Logical view in HBase

<table>
<thead>
<tr>
<th>Row key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>{'info': {'height': '9ft', 'state': 'Texas'}}</td>
</tr>
<tr>
<td></td>
<td>roles: {'ASF': 'Student', 'B.Tech': 'Qualifying'}</td>
</tr>
<tr>
<td>Martin</td>
<td>{'info': {'height': '6ft', 'state': 'Arizona'}}</td>
</tr>
<tr>
<td></td>
<td>roles: {'ASF': 'student', 'B.Tech': '2010', 'M.Tech': 'Qualifying'}</td>
</tr>
</tbody>
</table>

Tables 2 and 3 show physical view of column family info and roles. In this, it depicts about how exactly data are stored in HBase. For each row key entry, column key timestamp and cell value is stored. Row key identifies the corresponding row from which column it belongs. Column key is used to identify particular column of column family. Timestamp is used to identify time at which the record is written in HBase. Cell value contains the exact value of column key.

Table 2  Physical view info of column family

<table>
<thead>
<tr>
<th>Row key</th>
<th>Column key</th>
<th>Timestamp</th>
<th>Cell value</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Info: height</td>
<td>1271432980182</td>
<td>9ft</td>
</tr>
<tr>
<td>John</td>
<td>info: state</td>
<td>1271432980091</td>
<td>Texas</td>
</tr>
<tr>
<td>Martin</td>
<td>Info: height</td>
<td>1271432979999</td>
<td>6ft</td>
</tr>
<tr>
<td>Martin</td>
<td>Info: state</td>
<td>1271432987862</td>
<td>Arizona</td>
</tr>
</tbody>
</table>

Table 3  Physical view roles of column family

<table>
<thead>
<tr>
<th>Row key</th>
<th>Column key</th>
<th>Timestamp</th>
<th>Cell value</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>roles: ASF</td>
<td>1271432980181</td>
<td>Student</td>
</tr>
<tr>
<td>John</td>
<td>Roles: B.Tech</td>
<td>1271432980097</td>
<td>Qualifying</td>
</tr>
<tr>
<td>Martin</td>
<td>roles: ASF</td>
<td>1271432978999</td>
<td>Student</td>
</tr>
<tr>
<td>Martin</td>
<td>Roles: B.Tech</td>
<td>1271432988762</td>
<td>2010</td>
</tr>
<tr>
<td>Martin</td>
<td>Roles: M.Tech</td>
<td>1271432987762</td>
<td>Qualifying</td>
</tr>
</tbody>
</table>

3.1 HBase CRUD operations

The basic HBase operations related to the transactions are called CRUD, which stands for create, read, update and delete. The CRUD is given in detail in Lars (2011). In this section, we present the summary of these operations with the support of ACID properties.

- **Put method**: is used to insert or add a row in given table of HBase using row key. Put method is categorised into two parts.
  1. Single put: is atomic in nature and works on single row
  2. List of puts: works on multiple rows and does not give guarantee that, all rows given for that is written or none. But it guarantees that a row is entirely written on nothing. This returns the success of the rows which are written successfully.
Preserving atomicity and isolation for multi-row transactions

- **Get method:** is used to fetch an entire row or selected columns from database. Get method is also categorised into two parts.
  1. Single get: fetches single row (or single row’s selected contents) using row key. A row which is returned via Get will be a complete row which exists in table’s history.
  2. List of gets: fetches multiple rows.

- **Delete method:** is used to delete entire row or selected columns from database. Same as Put and Get, delete is also classified into two parts.
  1. Single delete: deletes row or some columns which are passed to the method if exist, otherwise do nothing.
  2. List of delete: deletes number of rows or number of columns which are specified. This method does not guarantee any specific order in which the data are deleted.

- **Scan method:** is very similar to Get but it takes a range of row key as input and returns all the rows which belong to the range. As HBase does not provide SI, so scan does not provide consistent view of table. If a transaction $X$ scans some rows in a given range, after that a transaction $Y$ put some rows which belong to same range, then transaction $X$, again scans in same range and gets additional rows.

3.2 **Snapshot isolation**

In SI, each transaction operation describes a consistent view. To implement isolation, each transaction must work on its private snapshot which has taken before the transaction starts. SI provides a consistent view of database for the transaction. In order to implement SI, database maintains multiple versions (in terms of database) of each row. A row version is allocated to the transaction when it starts. There is no kind of lock instance will be used when data are read. So write operation does not block even it is performed at the same row. In the same fashion, write operation does not block read operation. Concurrency of SI is higher than serialisable transaction because no locking instance has been appended on it.

4 **Proposed methodology**

We assume a configuration methodology before proceed to the solution. First, beyond the HBase configuration, there exists a proxy server. This proxy server is responsible for handling every transaction request and also supports TM. A RDBMS (MySQL server) process is also running along with proxy server. Proxy sever redirects the request to master or dedicated region server, if the location of row is already known. Beside this, there is a process called cleaner which is running to remove old records from MySQL which are not useful anymore. The responsibility of cleaner process is to recover all failed and incomplete transactions. Cleaner process always runs on active manager node and keeps the track of abort/incomplete transactions. If it finds any abort/incomplete transaction, then it creates recovery process on one of any node and submits this
transaction for recovery. A setup related to this methodology is depicted in Figure 1 in the form of system architecture.

**Figure 1** System architecture (see online version for colours)

Figure 1 depicts about the workflow of proposed architecture. First, all client requests to HBase master server or region server to execute transactions. But, these requests are handled by proxy server, where the TM resides. TM communicates with MySQL database in order to provide atomicity and SI for multi row transactions. TM is also responsible to redirect the requests back and responds to client with transaction status such as *commit* or *abort*.

We now describe a transaction management protocol for implementing atomicity and basic SI model for multi row transactions. To implement this, we propose a configuration model that is prerequisite. Beside the HBase this configuration includes

1. global TM (TM in proxy server) which handles each and every client request
2. MySQL server contains all meta-data which are required by TM
3. cleaner is useful for deleting unnecessary records from MySQL server.

### 4.1 Metadata structure

To implement a distributed atomic transaction with multiple rows (to provide isolation), we use RDBMS to store metadata. We design four global tables according to the structure of RDBMS

1. transaction table
2. snapshot table
3. commit table
4. row table, which are depicted in Figure 2.
Preserving atomicity and isolation for multi-row transactions

Figure 2  Relation among metadata tables (see online version for colours)

*Transaction table* (Table 4): contains transaction id and timestamp columns. First column (transaction_id) is auto incremented primary key of the table and used to identify each transaction. Second column (timestamp) is the time generated by the server and used to identify the starting time of the transaction. Here, we omit RDBMS server’s clock synchronisation with HBase distributed nodes because every transaction relies on global tables.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction_id</td>
<td>int (auto increment)</td>
<td>Primary key</td>
</tr>
<tr>
<td>timestamp</td>
<td>Time (in milli sec)</td>
<td>No</td>
</tr>
</tbody>
</table>

*Row table* (Table 5): contains four columns

1. row_id
2. row_key
3. timestamp
4. region_server_id.

First, row_id is an auto incremented primary key of table. Second, row_key is a key value of the row with table name in HBase. Third, timestamp contains actual time at which row is added. Using timestamp and row_key, a row is uniquely identified in HBase because multiple versions of same row is supported. Four, region_server_id is used to identify the server at which the row resides. Using region server_id, the TM directly fetches the row from region server without using the master server.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>row_id</td>
<td>int (auto increment)</td>
<td>Primary key</td>
</tr>
<tr>
<td>Row_key</td>
<td>varchar</td>
<td>No</td>
</tr>
<tr>
<td>timestamp</td>
<td>Time (in milli sec)</td>
<td>No</td>
</tr>
<tr>
<td>Region_server_id</td>
<td>varchar</td>
<td>No</td>
</tr>
</tbody>
</table>
Commit table (Table 6): contains three columns:

1. transaction_id
2. row_id
3. commit_timestamp.

First column, transaction_id is a foreign key which refers transaction_id column of transaction table. Second, row_id is also a foreign key which refers row_id column of the row table. Third, commit_timestamp is used to find the time at which this transaction is committed, i.e., the time when TM has made entry in the commit table. This table also contains entry for every version of each row key with timestamp at which it is committed.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction_id</td>
<td>int (auto increm ent)</td>
<td>Foreign key (Transac table)</td>
</tr>
<tr>
<td>row_id</td>
<td>varchar</td>
<td>Foreign key (Row table)</td>
</tr>
<tr>
<td>Commit_timestamp</td>
<td>Time (in milli sec)</td>
<td>No</td>
</tr>
<tr>
<td>Region_server_id</td>
<td>varchar</td>
<td>No</td>
</tr>
</tbody>
</table>

Snapshot table (Table 7): contains two columns

1. transaction_id
2. row_id.

First column, transaction_id is a foreign key which refers transaction_id column of transaction table. Second, row_id is also a foreign key which refers row_id column of row table. This snapshot table is used to provide consistent view for each transaction. At first, when a transaction access a row it will make an entry in the snapshot table for future reference.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction_id</td>
<td>int (auto increment)</td>
<td>Foreign key (Transac table)</td>
</tr>
<tr>
<td>row_id</td>
<td>varchar</td>
<td>Foreign key (Row table)</td>
</tr>
</tbody>
</table>

4.2 Programming model

Here, we consider two kinds of multi-row distributed transactions:

1. read transaction: which is related to HBase get () and scan () method
2. update transaction: related to HBase put () and delete () methods.

Algorithms used for transaction operations are:
Algorithm 1  

Read transaction

A transaction $T_i$ starts at a time $t_i$ and wants to read a row with key $r_i$.

Begin
$T_i = \text{INSERT an entity in transaction table with default values.}$

if (result := $\prod_{\text{row id} \in \sigma_{\text{row_key} = r_i \land \text{transaction_id} = T_i} (\text{snapshot} \triangleright\triangleright \text{row})}) \neq \text{NULL}$

return (Get rows for $r_i$ from HBase using row table) exit.

else Result := $\prod_{\text{row id} \in \sigma_{\text{row_key} = r_i} (\text{commit} \triangleright\triangleright \text{row})}$

SELECT $r_i$ FROM result with latest timestamp value

$\text{INSERT an entity in snapshot table with } r_i \text{ and } T_i.$

return (Get from HBase where row_key is $r_i$) exit.

Algorithm 2  

Update transaction

A transaction $T_i$ starts at a time $t_i$ and writes row $r_1$, $r_2$, $r_3$ which are distributed over region server $r_{s1}$, $r_{s2}$, $r_{s3}$.

begin
$(T_i, t_i) = \text{INSERT an entity in transaction table with default values.}$

Result := $\sigma_{\text{row_key} = (r_1, r_2, r_3) \land \text{commit_timestamp} > t_i} (\text{commit} \triangleright\triangleright \text{row})$

If result is not NULL return abort $T_i$ exit;

else put $r_1$, $r_2$, $r_3$ to HBase (wait for response time)

If for all $r_1$, $r_2$, $r_3$ success received

If ($\sigma_{\text{row_key} = (r_1, r_2, r_3) \land \text{commit_timestamp} > t_i} (\text{commit} \triangleright\triangleright \text{row})$) is NULL

(Take lock on commit table and Insert entities in row table for $r_1$, $r_2$ and $r_3$ and Insert entities in commit table for $r_1$, $r_2$, $r_3$ with $T_i$ then release lock )* return success and exit

DELETE from HBase (all successfully received rows)

return abort $T_i$ and exit.

Note: *Denotes all operations belong to same RDBMS transaction.

5 Algorithmic evaluations

To evaluate the proposed algorithms, we consider examples related to bank credit data.

Example 1: This example is related to updating algorithm and shows how atomicity is achieved for multi_row transactions. In this, there is a table named account which contains two columns account_num and amount. A transaction $T_1$ transfers the amount from one account to another account. Assume $A_i$ is the credit account and $A_k$ is the debit account (to transfer $10,000$). Another transaction $T_2$ wants to transfer $5,000$ from $A_i$ to $A_k$ ($T_1$ and $T_2$ are said to be concurrent). Transfer between two transactions is depicted in Figure 3 and the physical locations of the records are depicted in Figure 4.
Now we execute this transaction scenario using updating algorithm and evaluate the performance. Assume $T_1$ and $T_2$ starts at the same time and inserts into transaction table with same timestamp called ‘0005’. Tables 8 to 10 shows transaction entities, commit entities, and row entities at the ‘0005’ timestamp.

**Table 8**  Transaction entries

<table>
<thead>
<tr>
<th>Transaction_id</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_x$</td>
<td>0002</td>
</tr>
<tr>
<td>$T_y$</td>
<td>0003</td>
</tr>
<tr>
<td>$T_1$</td>
<td>0005</td>
</tr>
<tr>
<td>$T_2$</td>
<td>0005</td>
</tr>
</tbody>
</table>
Preserving atomicity and isolation for multi-row transactions

Table 9  Transactions commit entries

<table>
<thead>
<tr>
<th>Transaction_id</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_x$</td>
<td>1</td>
</tr>
<tr>
<td>$T_y$</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 10  Row table entries

<table>
<thead>
<tr>
<th>Row_id</th>
<th>Row_key</th>
<th>Commit_timestamp</th>
<th>Region_server_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>account: $A_k$</td>
<td>126795</td>
<td>165.23.7.11: 3232</td>
</tr>
<tr>
<td>2</td>
<td>account: $A_i$</td>
<td>126734</td>
<td>165.23.7.17: 3232</td>
</tr>
</tbody>
</table>

Now $T_1$ reads from HBase $A_k = 20,000$ with the help of metadata and using our algorithm. At the same time $T_1$ and $T_2$ read from HBase $A_i = 5,000$. Then transaction $T_2$ also read $A_k = 20,000$. Both transactions will update the copy of data such that $T_1$ do $A_k = 10,000$, $A_i = 15,000$ and $T_2$ update $A_k = 25000$, $A_i = 0$. As mentioned in Figure 3, $T_2$ flushes the data first. So it will make put request to HBase for row $A_k$ and $A_i$ and waiting for response. At the waiting time of $T_2$, $T_1$ gets flushed. And also make put request to HBase using row $A_k$ and $A_i$ and waits for the response. Now assume that $T_1$ gets committed first for both rows and make entities in row as well as in commit tables. At the same time $T_2$ also gets committed for both the rows and make entities in row as well as in commit tables for $A_k$ and $A_i$. Then again checks commit table for new entries of $A_k$ and $A_i$. After finding new entries for transaction $T_1$, $T_2$ undo all rows from RDBMS. It has been done after the deletion of all rows from HBase related to this transaction. Finally, $T_1$ is committed and $T_2$ aborts. So, the database is in consistent state.

Example 2: This example is related to read algorithm and shows how SI is preserved using our algorithm. Assume a transaction $T_1$ starts and make its entry in transaction table. It reads amount from the account_num $A_i$ that is $10000$. Because it reads first time and makes an entity in snapshot table with $T_1$ and $A_i$. After that transaction $T_2$ is commenced and updates $A_i = 20,000$ and creates new row in HBase. Transaction $T_2$ also makes a new entity in metadata using updating algorithm. Again the transaction $T_1$ reads $A_i$ through snapshot table which is not updated yet.

We further analyse the performance of a transaction according to its sub transaction. We measure the performance in terms of execution rate of both transaction and sub transaction. Section 5.1 describes about performance analysis according to SI and atomicity of transactions.

5.1 Performance analysis

- at site X (database):
  - $M_1$: $\text{Read (m)}$: $\text{Write (m)}$: $\text{Prepare}$: $\text{Commit}$;
  - $C_1$: $\text{Read (m)}$: $\text{Read (n)}$: $\text{Write (n)}$: $\text{Prepare}$: $\text{Commit}$;
- at site Y (database):
  - $M_2$: $\text{Read (o)}$: $\text{Write (o)}$: $\text{Prepare}$: $\text{Commit}$;
  - $C_3$: $\text{Read (o)}$: $\text{Read (p)}$: $\text{Write (p)}$: $\text{Prepare}$: $\text{Commit}$;
Here the HBase manages ‘n’ messages (transactions) at a time. Site X is the initiator which can tolerate Xn messages at a time. The query instance is not limited, but we consider up to three or four for performance evaluation. Assume that a transaction is included in Xn and a sub transaction has shared its data item ‘m’ like C1 with transaction M1. Then the sub transaction has to commit within the stipulated time. Here, the sub transaction is the recurrence of the main transaction of Xn with the timestamp t. This is evaluated according to differential second order linear formulae (Ramesh and Kumar, 2014); at site X:

\[ X^n - tx = 0 \]  

The generic form according to n queries is:

\[ X(t) = \sum_{n=0}^{\infty} q_n t^n \]  

Determining the coefficients (q_n):

Plugging equation (2) in to the differential equation, we get (3)

\[ X^n(t) = \sum_{n=2}^{\infty} n(n-1)q_n t^{n-2} \]  

Plug equation (3) in to equation (1), here we get

\[ \sum_{n=2}^{\infty} n(n-1)q_n t^{n-2} - t \sum_{n=0}^{\infty} q_n t^n = 0 \]  

Or equivalently

\[ \sum_{n=2}^{\infty} n(n-1)q_n t^{n-2} - t \sum_{n=0}^{\infty} q_n t^{n+1} = 0 \]  

Simplifying equation (4) sum for the summation ‘Σ’ with others, and the problem we face that the powers at both the sums \( t^{n-2} \) and \( t^{n+1} \) are different. Simplifying the index of the first sum up by two units and the index of the second sum down by one unit, we obtain

\[ \sum_{n=0}^{\infty} (n+2)(n+1)q_{n+2} t^n - \sum_{n=4}^{\infty} q_{n-1} t^n = 0 \]  

Split of the 0th term of the first sum:

\[ \sum_{n=0}^{\infty} (n+2)(n+1)q_{n+2} t^n = 2.1.q_2 + \sum_{n=1}^{\infty} (n+2)(n+1)q_{n+2} t^n \]  

After combining the two sums, we get

\[ 2q_2 + \sum_{n=1}^{\infty} ((n+2)(n+1)q_{n+2} t^n - q_{n-1} t^n) = 0 \]  

Factor out \( t^n \):
Preserving atomicity and isolation for multi-row transactions

\[ 2q_2 + \sum_{n=1}^{\infty} \left( (n+2)(n+1)q_n + t^n - q_{n-1} \right) r^n = 0 \]  

The series on the left side is identically to zero; consequently all of its coefficients are equal to zero.

\[ 2q_2 = 0 \]
\[ (n+2)(n+1)q_n + t^n = q_{n-1} \quad \forall n = 1, 2, 3, \ldots \]  

We rewrite this as:

\[ q_2 = 0 \]
\[ q_{n+2} = -\frac{q_{n-1}}{(n+1)(n+2)} \quad \forall n = 1, 2, 3, \ldots \]

5.2 Analytical performance

From equation (9):

\[ q_2 = 0 \]
\[ q_3 = \frac{q_0}{2.3} \]
\[ q_4 = \frac{q_1}{3.4} \]
\[ q_5 = \frac{q_2}{4.5} = 0 \]
\[ q_6 = \frac{q_3}{5.6} = \frac{q_0}{(2.3)(5.6)} \]
\[ q_7 = q_4 = \frac{q_1}{(3.4)(6.7)} \]
\[ q_8 = \frac{q_5}{7.8} = 0 \]
\[ q_9 = \frac{q_6}{8.9} = \frac{q_0}{(2.3)(5.6)(8.9)} \]

Then the general form of the solution is:

\[ X(t) = q_n \left[ 1 + \sum_{n=1}^{\infty} \frac{t^{3n}}{(2.3)(5.6)\ldots(3n-1)(3n)} \right] \]
\[ + q_n \left[ 1 + \sum_{n=0}^{\infty} \frac{t^{3n+1}}{(3.4)(6.7)\ldots(3n)(3n+1)} \right] \]  

This implies:

\[ q_n = \frac{3^n + 1}{2n} \]

For all \( n = 0, 1, 2, 3, \ldots \).
\[ X(t) = \sum_{n=0}^{\infty} \frac{3^n + 1}{2.n!} t^n \] (11)

Retrieving the solution in more familiar form

\[ \begin{align*}
X(t) &= \sum_{n=0}^{\infty} \frac{3^n + 1}{2.n!} t^n \\
&= \sum_{n=0}^{\infty} \frac{3^n t^n}{2.n!} + \sum_{n=0}^{\infty} \frac{1}{2.n!} t^n \\
&= \frac{1}{2} \sum_{n=0}^{\infty} \frac{(3t)^n}{n!} + \sum_{n=0}^{\infty} \frac{t^n}{n!} \\
&= \frac{1}{2} e^{3t} + \frac{1}{2} e^t
\] (12)

where the first term or sum of equation (12) \((\frac{1}{2} e^{3t})\) is the execution time rate of the main transaction and the second term or sum \((\frac{1}{2} e^t)\) is the execution rate of its sub transaction. This evaluation is happened as main transaction and sub transaction where it depends on rate transfer of the messages to its participants.

6 Result analyses

In this, we perform atomicity related test according to transactions.

6.1 Atomicity test

Configuration: Prototype configuration for atomicity test is depicted in Figure 5.

**Figure 5** Prototype configurations for atomicity test

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>172.16.4.222</td>
<td>172.16.5.198</td>
<td>172.16.7.30</td>
</tr>
<tr>
<td>HOST1</td>
<td>HOST2</td>
<td>HOST3</td>
</tr>
</tbody>
</table>
```

```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE MANAGER1</td>
<td>RESOURCE MANAGER2</td>
<td></td>
</tr>
<tr>
<td>TM1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MANAGER</td>
<td>DB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Input: In this, a client sends a request (according to transaction) to TM and waits for the response. Transaction contains two database update operations ($T_1$ and $T_2$) which are distributed among resource Manager$_1$ and resource Manager$_2$. Transaction operations are contrasted in Figure 6.

Figure 6 Concurrent sub-transaction scenarios

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read balance = 5,000 for ano = 1</td>
<td>Read balance = 10,000 for ano = 2</td>
</tr>
<tr>
<td>Lock (ano = 1)</td>
<td>Lock (ano = 2)</td>
</tr>
<tr>
<td>balance = balance – 10</td>
<td>balance = balance + 10</td>
</tr>
<tr>
<td>Commit($T_1$)</td>
<td>Commit($T_2$)</td>
</tr>
<tr>
<td>Release($T_1$)</td>
<td>Release($T_2$)</td>
</tr>
</tbody>
</table>

Case 1: According to the prototype configuration of Figure 5, this environment runs with two resource managers and one TM. Resource Manager$_1$ gets failed after sending ‘ok’ (yes) vote to TM. The entire transaction scenario is shown in Figure 7 in the form of event driven sub-transaction.

Figure 7 Event driven sub-transaction scenario for case1
Result: Transaction fails due to the down instance of resource Manager2 (RM2) after sending positive vote to TM. Then the distributed database goes into inconsistent state. But global logs remain track this situation and recovery can be done by recovery manager as early as possible. Figures 8 and 9 shows success and fail scenario snapshots.

Figure 8  Success snapshot scenario for atomicity test (case1) (see online version for colours)
Case 2: After committing the transaction and before sending ‘ack’ message to TM, resource Manager₁ goes down. Entire transaction scenario is depicted in Figure 10.

Result: Transaction fails due to the down state of resource Manager₂ (RM₂) after committing the transaction but before sending ‘ack’ message to TM. Distributed database goes into inconsistent state. But global logs keep track this situation and recovery can be done by recovery manager as early. Figure 11 depicts about failure scenario snapshots.

Snapshots description: Figures 8, 9 and 11 are the results of snapshot scenarios generated at the time of performing atomicity test. To understand this, we explore one snapshot shown in Figure 11, which is the output of atomicity test in case₂. This snapshot shows that, both participating resource managers are inconsistent (resource Manager₁ and resource Manager₂ windows in Figure 11), but global logs keeps the track of inconsistency in logs (meta-data window in Figure 11). With the help of these logs, recovery manager will recover both the resource managers when it is scheduled. We exemplify this set up with a latency test towards transaction execution ratio. This test measures how long a transaction takes to complete its execution by maintaining atomicity property. Here, we measure latencies in milliseconds (ms) per transaction. In this section, we apply latency test in a distributed environment to find related results.
Figure 10  Event driven sub-transaction scenario for case$_2$

Figure 11  A failure snapshot scenario for atomicity test (case$_3$) (see online version for colours)
Input: At first, we send client transaction requests and find the completion time of these transactions. Each transaction affects two database rows. All consecutive transactions are concern with distinct row sets. But here, both rows are distributed among distinct databases. We increase client requests and check latency accordingly. We ran the number of transactions by increasing the count from 1 to 800 and recorded the completion time in both best and worst cases as performed in distributed environment. The latency output for 1 to 50 transactions is depicted in Figure 12. There is no occurrence of unsuccessful transaction instance while recording the latency in best case. But, there are some unsuccessful transactions occurred while recording the latency in worst case. This is due to the channel error (i.e., network delay or packet loss) in the distributed environment.

Figure 12  Latency output in distributed environment (for 50 transactions) (see online version for colours)

Figure 13  Latency output in distributed environment (for 800 transactions) (see online version for colours)
The completion time of both the cases is depicted in Table 11 as output. The completion time in worst case represents the execution of all 800 transactions when the network speed is medium. On the other hand, completion time in best case represents the execution of all 800 transactions when the network speed is high. All parameters which include completion time and latency outputs are depicted in Table 11. The execution scenario of all transactions from 1 to 800 in distributed environment is depicted in Figure 13.

Table 11  Output of latency test in distributed environment

<table>
<thead>
<tr>
<th>No. of trans</th>
<th>Latency in best case</th>
<th>Latency in worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compl. time in m.sec</td>
<td>Unsuccessful trans</td>
</tr>
<tr>
<td>1</td>
<td>266.22041</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>4,532.48824</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>7,951.83200</td>
<td>0</td>
</tr>
<tr>
<td>300</td>
<td>10,154.46184</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>13,284.96640</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>16,520.84625</td>
<td>0</td>
</tr>
<tr>
<td>600</td>
<td>19,986.12341</td>
<td>0</td>
</tr>
<tr>
<td>700</td>
<td>22,499.99912</td>
<td>0</td>
</tr>
<tr>
<td>800</td>
<td>25,200.22200</td>
<td>0</td>
</tr>
</tbody>
</table>

7 Conclusions and future scope

In this paper, we have presented a methodology to provide atomicity and SI support for multi-row transactions in column-oriented heterogeneous distributed databases such as HBase. This methodology enables multi-row distributed transactions with global SI on HBase tables with read and update transaction instances. There exists no kind of system provides same level of isolation on HBase with these scenarios. This methodology also tries to achieve several design aspects like supporting highly responsive transactions with no blocking read instances and employs effective handling mechanism.

The performance overhead of HBase SI over generic HBase is diffident; especially for longer transactions involving large read and writes operational instances per transaction. This mechanism can be further extended to implement suitable tools to optimise and possibly extending to increase its scalability at the time of distributing the transactional metadata tables.

Acknowledgements

The authors wish to express their appreciation to the editor and anonymous referees for many helpful suggestions that significantly improve this paper. This research work is supported by Indian School of Mines (ISM), Dhanbad, and Government of India. The
Preserving atomicity and isolation for multi-row transactions

authors would also like to express their gratitude and heartiest thanks to the Department of Computer Science and Engineering, Indian School of Mines (ISM), Dhanbad, India for providing their research support.

References


