Transaction support for HBase

Krishnaprasad Shastry
Hewlett Packard
India Software operation
Bangalore-48
+91-8033866316
krishnaprasad.shastry@hp.com

Sandesh Madhyastha
Hewlett Packard
India Software operation
Bangalore-48
+91-8033867304
Sandesh-v.madhyastha@hp.com

Saket Kumar
Hewlett Packard
India Software operation
Bangalore-48
+91-8033868318
Saket.kumar3@hp.com

Kirk M Bresniker
Hewlett Packard
1501 Page Mill Road, Palo Alto
California- 94304-1100, U.S
+1-650 583533
kirk.bresniker@hp.com

Greg Battas
Hewlett Packard
11060 Desert Glen Drive, Fishers
Indiana- 46037,U.S
+1-3178423618
greg.battas@hp.com

ABSTRACT

NoSQL technologies such as HBase, Cassandra, MongoDB are becoming popular due to their ability to scale and handle large volumes of data as opposed to a traditional Relational Database Management System (RDBMS). However they lack two major functionalities provided by traditional RDBMS namely “transactional support” and “SQL interface”. Transactions are designed to maintain database integrity in a known, consistent state, by ensuring that interdependent operations on the system complete successfully or all the operations are canceled. This paper describes a non-intrusive approach to provide transaction support for HBase based on optimistic concurrency model.

1. INTRODUCTION

NoSQL technologies such as HBase[4], Cassandra[5], MongoDB[6] are becoming popular due to their ability to scale and handle large volumes of data at breakthrough levels of cost and query performance. However transaction support is lacking in these NoSQL products. Without multi-row, multi-object transaction support in NoSQL products, the application has to implement transactions as part of its business logic. This makes development and maintenance of applications complex.

The workloads such as online transaction processing (OLTP), event processing, real-time analytics, etc. are characterized as operational workload. These workloads typically have stringent requirements in terms transactional data integrity, sub-second response time, concurrency and availability. With the growing “Internet Of Things (IOT)” there is a significant increase in number of data generation sources, volume of data and the type of the data that needs to be captured as part of transactions. These next generation operational applications need transactional support on multi-structured data types. For example, there are several Web2.0 applications, like “online shopping”, “online gaming”, “online index updates” etc., that require transaction support.

Several of these next generation operational applications will benefit from the flexibility in schema, data types and the scalability of NoSQL products like HBase, Cassandra and MongoDB. But the lack of transaction support is currently preventing from moving to these NoSQL technologies.

There are many attempts in academia as well as the open source community to provide transaction support for NoSQL products. In this paper we describe a non-intrusive approach to provide transaction support for HBase.

2. OUR SOLUTION

The design goal for our solution is to develop a non-intrusive transaction system for HBase. Another aim is to make this solution portable across different NoSQL technologies, thus no functional dependency of HBase.

Our solution provides transaction support to HBase by leveraging the versioning capability in HBase to implement snapshot isolation. The transaction functionality is implemented in a highly available centralized transaction server.

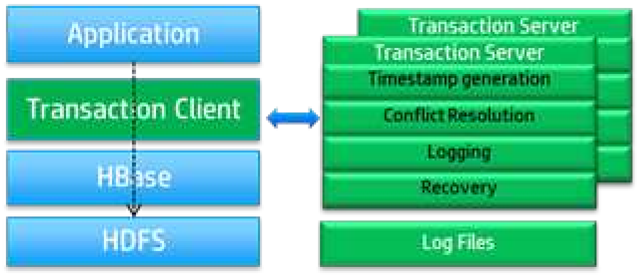


Figure 1: Architecture

Figure 1, illustrates the architecture of the solution. The transaction client provides transaction management APIs like beginTransaction, endTransaction etc. It also extends the generic HBase APIs, like get, put etc. to provide transactional support. The transaction client intercepts the HBase APIs from an application to provide transaction capabilities using the transaction server. It leverages the multi-versioning feature of HBase to write in-process transaction data into HBase tables.

The transaction server generates the transaction IDs, maintains begin and commit timestamp and manages the state of transactions. It implements the logic to resolve the conflicts during the transactions. The transaction server also implements the logging and recovery logic.

The transaction server maintains a table to track the status of transactions. The transaction state can be in (a) begin (b) commit-ready (c) committed (d) aborted. At the beginning of transaction, the transaction server generates globally unique transaction IDs that will have the value less than the epoch value, January 1, 1970. The transaction ID is used as version number for writes (put operation) from in-process transactions. The actual timestamps are used as version number to write the committed records. The committed records will always have the version numbers greater than epoch. We effectively use this data to control the visibility of in-process writes, thus provide snapshot isolation.

The transaction server maintains the timestamp value of the latest committed transaction, which is called as Last Commit Timestamp (LCT). The LCT value is assigned as the start time at the beginning of the transaction and is used to define the snapshot for the transaction.

The transaction server maintains all the modified row keys for a given transaction in an in-memory table. It uses this information to detect the conflict among concurrent transactions. The transaction server uses the transaction start time (TS1) and the modified row key set (RS1) to identify whether any transactions that are committed after the time TS1 has modified any of the rows in RS1. If yes it means the transactions are conflicting. In this case the transaction server marks the transaction for abort.

On receipt of beginTransaction call, the transaction client contacts the transaction server to get transaction ID and LCT. The transaction server generates new transaction ID and adds it into its status table. The transaction client uses the transaction ID as the timestamp value for intermediate “put” operations. These “put” values are also cached in the transaction client.

While reading data, in “get” and “scan” operations, the transaction client first looks for the rows in the cache thus it will read its own changes. If the rows are not in the cache, the “get” and “scan” operations read the data from HBase table using LCT as timestamp value, which indicates the snapshot of the database at the beginning of the transaction.

At the time of commit the transaction client sends the modified records to the transaction server to determine conflicts by other concurrent transactions. The transaction server uses the modified records and the transaction’s start timestamp to determine whether any other transactions have modified and committed the same rows. If there are no conflicts it generates a transaction commit timestamp and sends it back to transaction client. The transaction server updates the status of the transaction to commit-ready.

The transaction client performs the final “put” operations using this commit timestamp. After completing the final “put” operations, the transaction client acknowledges transaction server and the server updates the transaction status as committed.

The transaction server updates the LCT value with the commit timestamp of this transaction. At this point the records are committed in HBase table and they are visible to other transactions. If it finds any conflicts then it marks the transaction for abort and updates the status as aborted. The transaction server sends abort information to transaction client. The transaction client aborts the transaction and returns to application. The client will not clean up the intermediate records. The transaction server takes care of this as explained below.

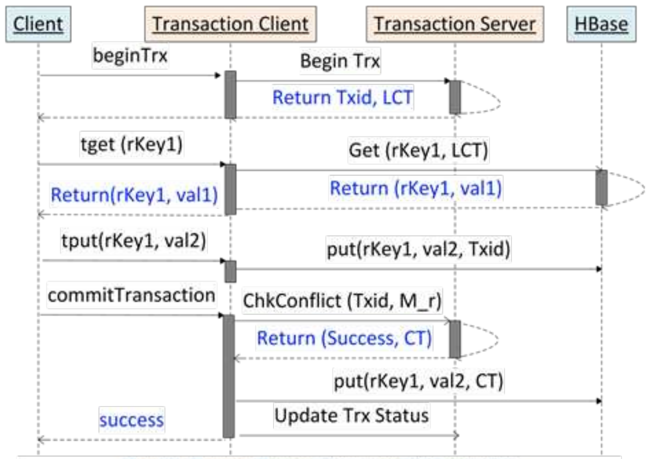


Figure 2: Sequence Diagram for a successful transaction

Figure 2, illustrates the sequence diagram for a successful transaction. During the begin transaction call from application, the transaction client calls transaction server to get the new transaction ID (Txid) and last commit timestamp (LCT). The transaction client intercepts the “tget” call from the application client (also called as client), checks for the corresponding record in its cache, if not found uses the LCT as timestamp to make “get” call to HBase. HBase returns the version of the record (rKey1, val1) that is visible at timestamp LCT. The client then executes the business logic and inserts the updated value (val2). The transaction client inserts this record using the Txid to ensure this intermediate record is not visible to other transactions. At the time of commit the transaction client sends the modified records set (M_r, in this case only rKey1) and the Txid to transaction server for detecting conflicts. In this example, there is no conflict and server returns success with the new commit timestamp (CT). The transaction client uses CT for final “put” operation and on completion acknowledges transaction server. The transaction server updates the transaction status and LCT. The CT becomes the new LCT.

The transaction client will not delete the intermediate records inserted with transaction ID as version. The transaction server implements a "purger thread" to clean up these residual records in the HBase table. The intermediate records modified by the transaction has to be removed for both committed and aborted transactions. The clean-up logic is same for both cases. The purger thread runs in background at specified frequency and deletes the residual records of the committed and aborted transactions.

The transaction servers implement a heartbeat mechanism to determine the client failures. If the client fails during the execution of transaction the server aborts the transaction. If the client fails after the commit-ready state and before commit acknowledgement the server makes the final “put” operation for the records modified by this transaction.

The transaction server will be implemented as process pair to provide high availability. If the primary server fails the backup will take over.

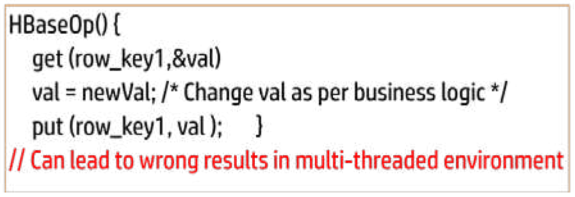
The transaction server logs the transaction state, including order in which they are okayed for commit into persistent space for recovery. The key of the records modified by the transaction are also logged for the commit-ready transactions. During the recovery of transaction server it reads the log to determine the transaction status. For the transactions that are in commit-ready state the server builds the list of records modified by this transaction from the log and inserts them into the HBase table with the transactions commit time. For the in-process transactions it cleans up the intermediate records by using the transaction id.

Our solution is non-intrusive and modular. It is not tightly coupled to HBase implementation. The transactions are supported both for the new tables as well as existing tables. The existing client applications must be modified to use the new APIs provided by transaction client to support transactions. The solution can be easily extended to other NoSQL products with minimal changes to transaction client.

3. EVIDENCE THE SOLUTION WORKS

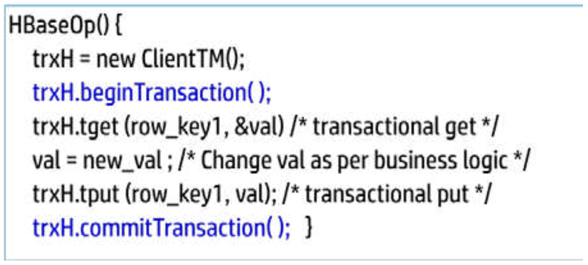
We have implemented the transaction server and transaction client. Tested it with a generic transactional application to validate the functionality.

Listing 1 and 2 shows the code snippet for a sample HBase application and the same application with transaction support.



```
HBaseOp() {
   get (row_key1,&val)
   val = newVal; /* Change val as per business logic */
   put (row_key1, val );      }
// Can lead to wrong results in multi-threaded environment
```

Listing 1: Code sample without transaction support



```
HBaseOp() {
   trxH = new ClientTM();
   trxH.beginTransaction( );
   trxH.tget (row_key1, &val) /* transactional get */
   val = new_val ; /* Change val as per business logic */
   trxH.tput (row_key1, val); /* transactional put */
   trxH.commitTransaction( );   }
```

Listing 2: Code sample with transaction support

The transactional application creates an instance of transaction client (TClient) and uses the methods exposed by it to create and commit the transaction.

Also it uses the "get" and "put" methods extended by transaction client. Without transaction support the sample application produces wrong results in multi-threaded environment.

To validate this we have developed a sample debit/credit application. The application operates on a single accounts table that contains the account identifier and the balance amount. The application transaction (a) deducts a fixed amount from one randomly selected account and (b) deposits the same amount in another randomly selected account. The application does some basic checks to ensure the debit and credit accounts are different, the balance never goes below a minimal value etc. The correctness of the transactions is measured by calculating the total sum of the value in all the accounts.

The application is tested with two different modes, (i) Transaction mode – wherein the application is linked with newly developed Transaction Client and uses the transaction API's offered by it, (ii) Non Transaction mode – wherein the application directly uses the API's exposed by HBase. The application is run with different configurations by varying (1) Total number of transactions (2) The number of simultaneous transactions (or threads).

We have run the application with multiple different values for (i) number of threads and (ii) number of transactions in a thread. The tables 1 and 2 show the results for multiple threads with each thread executing 100 transactions.

No. of threads	Transfers executed	Complete transfers	Incomplete transfers	Total Balance
1	100	100	0	100000000
10	1000	1000	0	99999850
100	9995	9996	0	99999750

Table 1: Without transactions

No. of threads	Transfers executed	Complete transfers	Incomplete transfers	Final Total Balance
1	100	100	0	100000000
10	1000	998	2	100000000
100	9998	9642	357	100000000

Table 2: With transactions

The first column represents the number of parallel threads; the second column indicates the total number of transaction that is tried. As mentioned earlier each thread runs 100 transactions. If the random number generator generates the same number for both debit and credit accounts then that transaction is not tried. Hence we see the value in this column to be less than the expected value for some cases – example row 3 (with transactions) has the value of 9998 instead of 10000. Third column indicates the number of completed transfers in case of default HBase and the number of committed transactions in the case of HBase-trx. The fourth column provides the number of failed transfers in the default HBase and number of aborted transactions in the case of HBase. The fifth column provides the total sum of balance in all the accounts.

As we can see form the table 1, there are inconsistencies in default HBase when run with parallel threads. The final balance is not same as the original and indicates the data loss/corruption due to failures. In case of "with transactions" (table 2) the transaction support guarantees the consistency and we always get the correct balance. We also see some failed transactions, indicated by "incomplete transfers" column, which are due to the conflicts.

4. COMPETITIVE APPROACHES

There are few attempts to provide transaction support for HBase. They fall into two main categories. One approach is to implement the transaction support on client side. HAcid [1] and HBaseSI [2] are two examples of this. They rely on additional metadata tables being created in HBase.

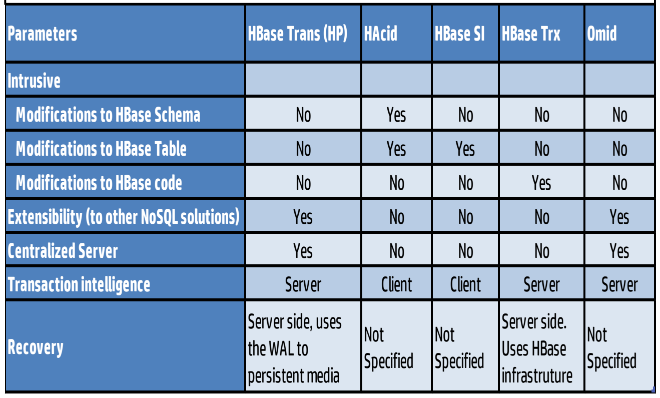
HAcid [1] is implemented as client library. It modifies the user tables in HBase to store additional metadata related to transaction management. Concurrency issues are handled at the client library by using the metadata information stored in user tables.

HBaseSI [2] is a client library that maintains special tables in HBase for supporting transactions. The transaction management logic is implemented in the client side based on the metadata in HBase tables.

The other approach is to implement a centralized transaction server. Omid [3][7] is an example of this, which is quite similar to our approach. Omid uses “transaction status Oracle” to manage the transactions. Omid caches the transaction metadata on client side to improve the performance. This results in multiple copies of metadata and increases the data traffic between client and server. Also maintaining the metadata adds additional overhead. The Omid clients cache the intermediate modification and hence need larger memory for long running transaction. This helps them to reduce the number of “put” operations.

HBase-Trx [8] was another open source attempt from Apache group to support transactions for HBase, which was later discontinued. HBase-Trx is tightly coupled to HBase and hence leverages the HBase code for transaction management and recovery. The concurrency is handled by HBase-Trx server library, which is implemented as an extension to HBase Region. The table 3 summarizes the characteristics of each of the solution.

Table 3: Comparison Table



Parameters	HBase Trans (HP)	HAcid	HBase SI	HBase Trx	Omid
Intrusive					
Modifications to HBase Schema	No	Yes	No	No	No
Modifications to HBase Table	No	Yes	Yes	No	No
Modifications to HBase code	No	No	No	Yes	No
Extensibility (to other NoSQL solutions)	Yes	No	No	No	Yes
Centralized Server	Yes	No	No	No	Yes
Transaction intelligence	Server	Client	Client	Server	Server
Recovery	Server side, uses the WAL to persistent media	Not Specified	Not Specified	Server side. Uses HBase infrastruture	Not Specified

5. Conclusion

We have presented a reliable and efficient implementation of transaction support library for HBase which is non-intrusive in nature. The approach does not need any changes to HBase schema or tables. It is implemented as a light weight centralized transaction server which provides the transaction management, conflict detection, logging and recovery services. A light weight transaction client library exposes the transaction support to users through transactional APIs. We have evaluated the approach for correctness.
As next steps we will measure the performance implications of the newly introduced transaction server and optimize it for both the latency and through-put.

6. References

[1] Mederos, A.:HAcid: A lightweight transaction system for HBase. Master’s Thesis, Espoo, September 24, 2012, Aalto University, School of Science, Degree program of Computer Science and Engineering

[2] Zhang, C., Sterck, H.D.:HBaseSI: Multi-row distributed transactions with strong snapshot isolation on clouds, Scalable Computing: Practice and Experience, Scientific International Journal for parallel and Distributed computing, Vol 12, No 2, 2011.

[3] Ferro, D.G.:Omid:Efficient Transaction Management and Incremental Processing for HBase, Yahoo Inc..

[4] HBase: http://hbase.apache.org/

[5] Cassandra: http://cassandra.apache.org/

[6] MongoDB: http://www.mongodb.org/

[7] Junqueira, F., Reed, B., Yabandeh, M.:Lock-free Transactional Support for Large-scale Storage Systems:IEEE/IFIP 41st International Conference, Dependable systems and Networks Workshop, Pages 176-181, June 2011..

[8] HBase-trx, https://github.com/hbase-trx, Git-Hub.