CoherentPaaS : Providing transactional support for cloud data stores

Ricardo Jiménez Peris[1], Marta Patino Martínez[2], and Ivan Brondino[2]

[1] LeanXcale, Madrid Spain
rjimenez@leanxcale.com
[2] Universidad Politécnica de Madrid, Madrid, Spain
{mpatino, ibrondino}@fi.upm.es

Abstract. In the last decade it has been observed an exponential explosion of generated user data over the internet. Traditional data management solutions such as relational databases are simply not able to process this large amount of data in a reasonable time. A new need of high scalable data management tools emerged, the cloud data stores. These technologies are able to process petabytes of data but with an important trade-off: the lack of transactional consistency. The scenario becomes even more complex for those applications whose building blocks is on top of a hybrid data store ecosystem. This works presents a novel protocol to provide transaction semantics on top of heterogeneous data stores transparently to applications.

Keywords. cloud computing, data stores, transactions, ACID, SQL, NoSQL

1 Introduction

In the last decade it has been observed an exponential explosion of generated user data over the internet. Traditional data management solutions such as relational databases are simply not able to process this large amount of data in a reasonable time. A new need of high scalable data management tools emerged, the cloud data stores. Internet companies, like Facebook[3], Google[4] and Twitter[5], created data management solutions to satisfy this need, the NoSQL data stores. HBase[6], an open source implementation modeled after Google's Bigtable key value data store [1] or Cassandra [2], another key value data store, are only a couple of examples. These technologies are able process petabytes of data but with an important trade-off: the lack of strong transactional consistency. The secret behind this technologies is known as *sharding*, where data is partitioned and a single node is responsible to manage a reduced part of the data. There is no coordination between nodes when data items from different nodes are updated by the application.

The scenario becomes even more complex when other data store technologies come into play like document and graph data stores, i.e., MongoDB [3] and Sparksee [4] or Neo4j [7] respectively. Graph databases became very popular after the social networks explosion, in the beginning of this century. The problem remains the same: the lack or reduced support of transactional semantics. Applications whose building blocks rely on this type of technolgies, formally defined as polyglot persistence [5], are facing a big pain on providing consistency across data stores.

In this work we present a transactional processing architecture with special emphasis in the provisions that are needed in order to support transaction consistency across multiple data stores. The transactional processing leverages the ultra-scalable transactional management from the CumuloNimbo ultra-scalable transaction processing system [6].

The reminder of this work is organized as follows. Section 2 provides some background on transactional processing. Section 3 presents a simplified version of the CumuloNimbo transactional processing protocol. Then, Section 4 introduces the designed extensions on the transaction manager and the implementation efforts required on the data stores to provide full ACID support across heterogeneous data stores. Finally, Section 5 concludes this paper with some initial conclusions and defines a road map for the next steps in our research.

2 Transactions

2.1 ACID Properties

A transaction is a sequence of data operations that are executed in an atomic way. Transactions provide the so-called ACID properties [7], namely:

[3] http://www.facebook.com
[4] http://www.google.com
[5] http://www.twitter.com
[6] http://hbase.apache.org
[7] http://www.neo4j.com

– *Atomicity*: It provides all-or-nothing semantics in the advent of failures. That is, the effect of a transaction should be "all" it if succeeds (the transaction committed) or nothing if it does not succeed (the transaction aborted or rolled back).
– *Consistency*: It is provided by the application. The application code in a transaction should guarantee that if provided with a consistent state of the database, it should produce a new consistent state of the database.
– *Isolation*: It provides synchronization atomicity. It provides the illusion that the user is executing the transaction alone in the system even if multiple transactions are executed concurrently.
– *Durability*: It guarantees that the updates of a successful (committed) transaction are not lost even in the advent of failures.

2.2 Why Transactions?

Transactions are a very important abstraction to program since they remove two hard problems when programming applications. The first one is dealing with concurrency. Users do not need to take care about concurrency control when they program applications using transactions to bracket the access to shared data. The protocols in charge of implementing the isolation property will take care of concurrent accesses. Second, applications do not have to deal with failures. Atomicity and durability protocols provide automated recovery in the advent of failures yielding all-or-nothing semantics. The ACID properties simplify the task of programmers.

2.3 Implementation of transactional properties

Transactional properties are attained by a combination of different protocols. Atomicity provides the ability to undo aborted transactions. Durability provides the ability to redo successful transactions. Both atomicity and durability require having redundancy at the data level, typically in the form of a log, that is, they are implemented on top of a logging mechanism. The log has to be complemented with a recovery protocol that it is executed upon recovery after a failure. The recovery protocol is executed before the database becomes available after a failure and is in charge of restoring the database consistency by undoing updates of aborted transactions and redoing updates of committed transactions to start with a fully consistent database state.

Isolation requires having implicit concurrency control. The highest level of isolation is known as serializability [7]. Serializability guarantees that the concurrent execution of transactions is equivalent to a serial execution of them. Therefore, the result is as if there was not concurrency at all. That is, as if all reads and writes of a transaction would happen at a single point in time.

However, it is well known that serializability reduces dramatically the potential concurrency due to the conflicts between predicate reads (e.g. select where SQL statement) and writes. Basically, a predicate read conflicts with any write on the same table unless the predicate is made exclusively over indexed columns and the predicate can exploit the index ordering (equalities or inequalities over indexed columns). For this reason, other isolation levels have been proposed such as the ANSI isolation levels and snapshot isolation. Other ANSI isolation levels reduce too much the isolation resulting in many potential anomalies. However, snapshot isolation [8] has become very popular because it only introduces a single anomaly known as write skew that many applications do not even trigger. Snapshot isolation basically splits the synchronization atomicity of a transaction in two points, the start of the transaction at which all reads happen logically, and the end of the transaction at which all writes happen logically.

Snapshot isolation provides a very high isolation level thanks to the fact that transactions read from a snapshot of the database with the state as it was when the transaction was started. Snapshot isolation requires using multi-version concurrency control [7]. This mechanism lies in instead of storing a single version of each data item, a new version is created when a transaction that updated the item commits. Therefore, for a single data item multiple versions of it can exist at a given time. These versions need to be labeled in a way that they enable to choose the right version for a given transaction that tries to read a data item. Typically, logical timestamps are used for this labeling.

Snapshot isolation avoids all read-write conflicts including the aforementioned one between predicate reads and writes. However, it still forbids write-write conflicts. This requires for checking those conflicts with some conflict management system.

3 CoherentPaaS Transaction Management

We consider a transaction to be a sequence of read and write operations on data records. A read operation can read individual records or collections of records selected by means of an arbitrary predicate. We present a solution based on snapshot isolation that avoids conflicts between reads and writes and has been well-established

both for traditional relational database systems as well as transaction solutions on top of key-value data stores. Many commercial database systems provide snapshot isolation as their highest isolation level, such as Oracle, PosgreSQL.

We assume a multi-version system where each write operation $w_i(x_i)$ of transaction T_i on record x creates a new private version x, and each read operation $r_i(x_j)$ of transaction T_i reads the latest version of x, x_j created by a committed transaction T_j such that $j < i$ and there is no other committed transaction T_z, such that $j < z < i$.

With such a multi-version system, snapshot isolation requires the snapshot read and snapshot write properties. Snapshot read requires that a transaction T_i reads a snapshot of the database that reflects the latest committed versions of all records as of start time of T_i. In particular, this means that if T_i performs a read r_i on x, then it reads either the private version T_i previously created (read your own writes) or it reads the version x_j created by T_j such that T_j was the last transaction to write x and commit before T_i started. Snapshot write requires that no two concurrent transactions (i.e., neither committed before the other started) update the same entity. If this happens one of the two transactions will abort (typical strategies are either the first committer wins, or the first updater wins). When a transaction commits, the commit timestamp is increased and all the private versions of a transaction are tagged with that commit timestamp. If the transaction aborts, private versions are discarded.

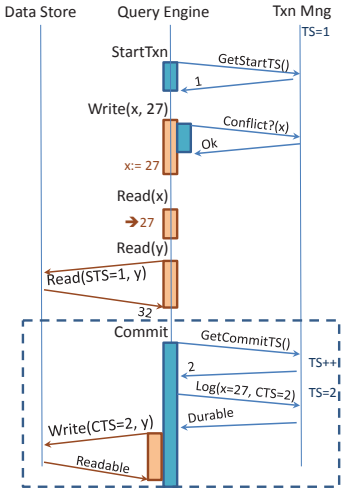

Fig. 1. Centralized Transaction Manager

In Figure 1, we illustrate a possible execution of transactions under snapshot isolation in a centralized system. The system is split into a query engine layer that parses and executes SQL queries, the data store layer that maintains the records (i.e., buffer and file system), and the transaction manager. In the figure, the actions associated with the transaction manager are indicated as blue boxes, the ones associated with the data store layer with brown boxes.

In most snapshot isolation implementations, the transaction manager maintains a counter that is used to tag transactions with start and commit timestamps. The value of the counter reflects the commit timestamp of the last committed transaction.

At start time of a transaction T_i, the transaction manager (Txn Mng in the figure) assigns the current counter value, TS, (reflecting the last committed transaction) as start timestamp. In the figure, the initial value of the counter is 1 (shown as $TS = 1$). When a transaction T_i wants to write a record x, the write operation will first perform a conflict check with the transaction manager. A conflict occurs, if there is a concurrent transaction T_j , i.e., T_j has not committed yet or its commit timestamp is larger than T_i's start timestamp ($C(T_j) > S(T_i)$), and T_j has written x. If there is no conflict, the write can proceed, creating a new private version of x ,so far only visible to T_i itself. If there is a conflict, T_i must be aborted to guarantee the snapshot write property. This simply means to discard the private versions T_i has created so far.

When a transaction T_i requests to read a record x, the data store has to provide the record created by transaction T_j with commit timestamp $C(T_j)$, such that $C(T_j) \leq S(T_i)$, and there is no version of x created by a transaction T_k such that $C(T_j) < C(T_k) < S(T_i)$. This provides the snapshot read property.

At commit time of transaction T_i, several things have to be accomplished. First, the transaction requests a commit timestamp $C(T_i)$ to the transaction manager, which is done by assigning it the next counter value. Second,

all record versions created by T_i must be labeled with $C(T_i)$. Then, the changes must be made durable, which is typically performed by persisting the redo-log to stable storage. The changes also must be integrated into the data management layer. Only then, the commit is confirmed to the user. In principle, this commit processing has to be an atomic action. In particular, the increment of the counter in the transaction manager and the integration of the new versions into the data store are tightly related because once the new counter value $C(T_i)$ is assigned as a start timestamp to a new transaction T_k, T_k must be able to see the updates performed by T_i, i.e., the updates must be visible in the data store.

CoherentPaaS Holistic Transaction Manager implements a distributed and ultra-scalable version of the described protocol. The details of the protocol are out of the scope of this work, more details can be found in [6].

4 Transactions across cloud data stores

CoherentPaaS has a set of subsystems that play an important role in transactional processing. They are: holistic transactional manager, *HTM*, data stores and common query engine [9], (*CQE*). We consider two kinds of data stores: 1) Data stores that fully delegate transactional processing to the holistic transactional manager; 2) Data stores that perform internally transactional processing and only delegate coordination of the transaction to the holistic transactional manager for global transactions (transactions handled by CoherentPaaS).

The holistic transactional manager provides all the transactional functionality for the first kind of data stores and transactional coordination for the second kind of data stores. The common query engine has a peculiar role. From the perspective of a CoherentPaaS application it looks like a complex data store that provides multiple functionalities (i.e. the aggregation of all CoherentPaaS data stores). From the perspective of the holistic transactional manager it looks as another application executing transactions across all data stores. Both the Application and the Common Query Engine do have a collocated Local Transaction Manager, *LTM*(see Figure 2).

The *LTM* is accessed via an Local Transaction Manager Client, *LTMc*, similar to a JDBC driver for a database that exposes the API to manage transactions and acts as an interface towards the holistic transactional manager. This means that the application and the common query engine do have collocated *LTMs*. Data stores are accessed by means of a client proxy as well (a JDBC driver or the equivalent for the data store) that are also collocated with the client application and the common query engine. Data stores clients implements a common interface that is used to interact with the *LTMc* in a transparent manner, independently on the nature of the data store. In order to support snapshot isolation, data stores must implement multi-versioning.

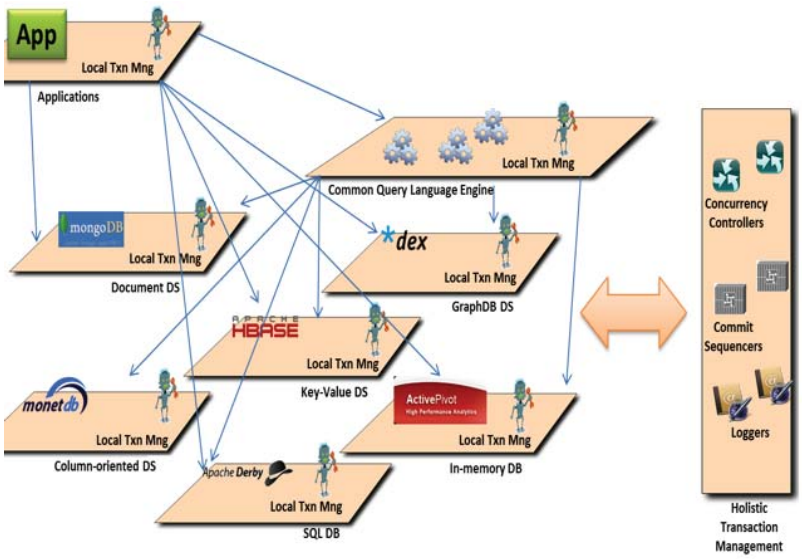

Fig. 2. CoherentPaaS architecture

4.1 Local Transaction Manager Client design

The Local Transaction Manager Client is the access point towards the CoherentPaaS Holistic Transaction Manager. At bootstrap, applications and common query engine instance a single *LTMc* and as many as needed data store

clients instances. Applications register data store clients instances in the *LTMc*. *LTMc* uses the references to invoke callbacks on the data store clients during the transaction execution.

In order to preserve the data store clients API, we have conceived the concept of connection, *Cnx*. The application creates a connection, just like in JDBC. It is through the connection where the transaction life cycle is managed. The application creates a *Cnx* to the *LTMc* invoking *getConnection*. There is a single connection per client thread.

A transaction is started through the invocation of *Cnx.startTransaction*. *startTransaction* produces a transaction context object, *TxnCtx*. The transaction context object is doted with a start timestamp, commit timestamp, a transaction context id and the list of involved data stores, which is empty when the transaction is started. The application associates one ore more data store clients to the current transaction, through *TxnCtx.associate(DSClient)*. Data store clients implement the interface *TransactionalDSClient*, the contract between data store clients and the *LTMc*. *TransactionalDSClient*is implemented by the clients and declares the following methods: $getWS$, $applyWS$, $notifyStartTransaction$, $redoWS$ and $terminate$. We will look at the details of the implementation details of the contract in the following section. A transaction is terminated (committed or rolled back) invoking *TxnCtx.commit* or *TxnCtx.abort* functions. The *TxnCtx* object is destroyed once the transaction is terminated.

The abstraction of connection is not present on the common query engine side. The common query engine is multi-threaded and having a single thread to execute queries against multiple data stores in a serial fashion is not desired. Instead, on the common query engine, the transaction context object is used directly. It provides the same API of *Cnx* allows concurrent access. For the reminder of this work, we provide the details of the implementation from the application perspective, that includes the whole transaction execution path.

4.2 Transaction Life Cycle

We deepen now in the details of the transaction life cycle. *Algorithm* 1 describes the initialization process of the system. The application instantiate one or more client connections to different data stores and the connection to the HTM, through the creation of a *LTMc*instance. The data store client connections are registered in the *LTMc*.

We describe the process with a transaction executed by the application against two data stores, *DataStore* and *DataStoreB*. We assume *DataStore* is a simplified version of a key value data, like HBase[8], and *DataStoreB* is a SQL data store, like LeanXcale[9]. *Algorithm* 2 presents the transaction code. The application creates a connection to the *LTMc* and starts the transaction (lines 1 and 2). Then, in lines 3 and 4, the data store clients instances are associated to the transaction. The association (*Algorithm* 4) tells the *LTMc* that these particular data stores participate in the transaction execution. Data stores are also notified that a new transaction is starting and they are participants of it.

Now the application executes transactionally and transparently two operations across heterogeneous data stores. The applications firstly reads some value X from the key value data store and them all rows in table $someTable.someColumn$ in the SQL data store. Finally the transaction commits. Every executed step could have failed, in such case rollback is invoked and the transaction is aborted. We describe now each of the executed steps one.

Algorithm 1 System bootstrap

1. *DSClient* $= connect(jdbcUrl)$
2. *DSClientB* $= connect(keyValueConnectUrl)$
3. *LTMc* $=$ *new LTMc*()
4. *LTMc*$.register($*DSClient*$)$
5. *LTMc*$.register($*DSClientB*$)$

Algorithm 2 Transaction execution example

1. *Cnx* $=$ *LTMc*$.getConnection()$
2. *TxnCtx* $=$ *Cnx*$.startTransaction()$
3. *TxnCtx*$.associate($*DSClient*$)$
4. *TxnCtx*$.associate($*DSClientB*$)$
5. $val =$ *DSClient*$.read(X)$
6. $sqlStmt =$ "$update\ someTable\ set\ someColumn = val$"
7. *DSClientB*$.execute(sqlStmt)$
8. *TxnCtx*$.commit()$

Fig. 3. System bootstrap and transaction execution example

[8] http://hbase.apache.org
[9] http://www.leanxcale.com

Cnx.startTransaction: *Algorithm* 3 creates a *TxnCtx*. The *LTMc* obtains a start timestamp and a context id and populates the *TxnCtx* object with them. This timestamp is the Snapshot Isolation version of the data that transaction will observe. The context id is used to globally uniquely identify this transaction.

TxnCtx.associate: *Algorithm* 4 associates a particular data store client to a *TxnCtx*. The association tells the *LTMc* that these particular data stores participate in the transaction execution. Data stores are also notified that a new transaction is starting and they are participants of it. Data store implementation of $notifyStartTransaction$ may vary depending on the data store nature. A non-transactional data store like HBase, just creates some internal data structure to keep track of the updates performed by the transaction. A SQL data store like LeanXcale, creates a SQL transaction and it is mapped to the CoherentPaaS global transaction. Either of the possible paths, the transaction context object is kept in a thread local variable. This decision was carefully though as usually transactions are executed by a single thread and thus there is no need to break the data store client API providing the transaction context in the signature of data store methods.

Algorithm 3 *Cnx.*$startTransaction()$

1. *TxnCtx* $=$ *new TxnCtx*()
2. $st =$ *LTMc.*$getStartTimestamp()$
3. $txnCtxId =$ *LTMc.*$getTxnCtxId()$
4. $ctx.startTimestamp = st$
5. $ctx.txnCtxId = txnCtxId$
6. **return** *TxnCtx*

Algorithm 4 *TxnCtx.*$associate($*TransactionalDSClient*$)$

1. *TxnCtx.*$addToInvolvedDataStore($*TransactionalDSClient*$)$ {make the data store client participant of this transaction}
2. *TransactionalDSClient.*$notifyStartTransaction($*TxnCtx*$)$ {notify the data store client that is participant of the transaction }

Fig. 4. Transaction start (3) and data store association to transactions (4).

DSClient.read(X): *Algorithm* 5 illustrates a simplification of the read operation. We assume the API does not change, but instead, the implementation needs to be extended to obtain the start timestamp and the transaction context id (lines 1 and 2). Then the native read is executed. Data stores that fully delegate the transaction processing on the CoherentPaaS , which is the case of HBase, retrieve the version corresponding to the provided start timestamp with the value of X. Data stores that delegate the transaction coordination to the CoherentPaaS transaction manager simply inform the global transaction context id. On the server side, this global transaction context id is mapped to an internal transaction and the operation is executed on the context of this internal transaction. In our example from *Algorithm* 2 (line 5), the transaction executes a simplified version of the read operation of the key value data store.

An extension of the read API that receives the context as a parameter is provided for multi-threaded environments such as the common query engine. In this case, the transaction context is resolved directly from the parameter instead of the thread local variable.

DSClient.write(X): *Algorithm* 6 illustrates a simplified version of the write operation. Likewise $read(X)$, We assume the API does not change, but instead, the implementation needs to changed to save the private version of the data item in some internal data structure (lines 6 to 10). Recall that private versions are visible only to the transaction that generates them and become public at commit time. Depending on the implementation, private versions may be kept on the client side or on the server side. In our example from *Algorithm* 2 (line 6 and 7), the transaction executes a simplified version of an update SQL statement on the JDBC client. Since the statement may update more than one item, a new private version of each data item is generated.

Apart for keeping the private version, and since under Snapshot Isolation the transaction manager needs to check for write-write conflicts, data stores that fully delegate the transactional processing on the CoherentPaaS transaction manager invoke the $checkWriteWriteConflict$ service of the *LTMc*. If a conflict is found, an error is thrown and the transaction needs to be aborted. Data stores that only delegate transaction coordination perform write write conflict checking on the server side. If a conflict is found, the data store returns an error signal and the transaction is aborted.

Likewise $read(X)$, an extension of the write API that receives the context as a parameter is provided for multi-threaded environments such as the common query engine. In this case, the transaction context is resolved directly from the parameter instead of the thread local variable.

DSClient.write(X): *Algorithm* 6 illustrates a simplified version of the write operation. Likewise $read(X)$, We assume the API does not change, but instead, the implementation needs to changed to save the private version of the data item in some internal data structure (lines 6 to 10). Recall that private versions are visible only to the transaction that generates them and become public at commit time. Depending on the implementation, private versions may be kept on the client side or on the server side. In our example from *Algorithm* 2 (line 6 and 7), the transaction executes a simplified version of an update SQL statement on the JDBC client. Since the statement may update more than one item, a new private version of each data item is generated.

Algorithm 5 *DSClient*.$read(x)$

```
1. st = TxnCtx.getStartTimestamp()
2. txnCtxId = TxnCtx.getTxnCtxId()
3. val = executeNativeReadForVersion(x, st, txnCtxId)
4. return val
```

Algorithm 6 *DSClient*.$write(x, val)$

```
1. ctxId = TxnCtx.getTxnCtxId()
2. conflict = LTMc.checkWriteWriteConflict(x)
3. if conflict == true then
4.    return abort on conflict
5. else
6.    ws = getPrivateWS(tid)
7.    if ws is empty then
8.       ws = createPrivateWS(tid)
9.    end if
10.   ws.addPrivateVersion(x, val, ctxId)
11.   return
12. end if
```

Fig. 5. Implementation example of *DSClient*.$read(x)$ and *DSClient*.$write(x, val)$.

TxnCtx.commit: In our example, *Algorithm* 2, the *commit* method is invoked and the commit phase is started after the transaction code is executed (line 8). The *commit* process is described in *Algorithm* 7. The first step is to certificate that the transaction is conflict free. This is done by invoking the *LTMc*.$certificate$ function. If there is not any conflict, the transaction may proceed to the next step otherwise an error is thrown and the transaction is aborted. Likewise in the commit process from Figure 1, the next step in commit phase is to persist the global transaction *write-set* in a durable log. Involved data stores are requested to provide the private write set, invoking $getWS$, in an array of bytes, the arrays of bytes are concatenated producing a unique and global array of bytes (lines 7 to 9). Data stores that delegate the transaction coordination to the CoherentPaaS transaction manager provides a byte array with a different semantic. The byte array is a handle to an internal commit log record. Also, some of these data stores perform conflict checking at commit time. If there is a write-write conflict $getWS$ returns an error and the transaction is aborted.

Read only transactions produce an empty *write-set*. If so, the transaction does not need to go through the commit process. Every data store is simply informed to cleanup temporary data (*TransactionalDSClient*.$terminate$, lines 18 to 20) associated to the transaction and the transaction is terminated.

Update transactions follow a different path. A commit timestamp is obtained from the CoherentPaaS Holistic Transaction Manager. This timestamp is used to label the private versions of the *write-set*on the data stores. Subsequently, the provided *write-set*' are concatenated and the global array of bytes is persisted in the log (lines 11 and 12). This is a no way back point. The transaction is considered to be *durable* and it must survive any kind of failure.

Once the transaction is *durable*, updates are reflected on the data stores. Involved data stores are requested to apply the write set, by invoking $applyWS$. Both $getWS$ and $applyWS$ implementations depend on the data store nature. Once the *write-set*are reflected on the data stores, the transaction is considered to be *readable*.

Note that a data store failure results in an error when $applyWS$ is invoked. Since the transaction is considered to be *durable*, and in consequence *committed*, this is simply ignored. *write-set* will be applied when the data store is recovered.

Finally, the *LTMc* informs that the transaction has been committed successfully. The transaction context id and the commit timestamp of the committed transaction are propagated to the CoherentPaaS holistic transaction manager. The transaction manager updates the conflict's table and eventually make this new version *visible*. Recall that he updates of a transaction T_i are *visible* to some transaction T_j when $ST(T_j \geq CT(T_i)$.

TxnCtx.rollback: In our example, *Algorithm* 2, instead of invoking *commit* the application could have decided to abort the transaction. This would have been done by invoking *abort*. The *abort* process is described in *Algorithm*

8. The procedure ask involved data stores and the *LTMc* to cleanup temporary data associated to the transaction (private write set, *TxnCtx*). Since the propagation of the updates to the data stores takes after the transaction is marked as $durable$ the cleanup is trivial. Data stores that delegate only the coordination of the transaction management, the *abort* process is mapped to an internal transaction *abort* execution.

Algorithm 7 $TxnCtx.commit()$

```
1. ctxId = ctx.getTxnCtxId()
2. conflict = LTMc.certificate(TxnCtx)
3. if conflict == true then
4.     return abort on conflict
5. end if
6. ws = {}
7. for dsc in TxnCtx.getInvolvedDS() do
8.     ws.append(dsc.getwrite-set(ctxId))
9. end for
10. if ws is not empty then
11.     ct = LTMc.getCommitTimestamp()
12.     LTMc.commitToLog(ws, ct, ctxId)
13.     for dsc in TxnCtx.getInvolvedDS() do
14.         dsc.applyWS(ct, ctxId)
15.     end for
16.     LTMc.commitTxn(txnCtxId, ct)
17. else
18.     for dsc in TxnCtx.getInvolvedDS() do
19.         dsc.terminate(ctxId)
20.     end for
21. end if
22. return
```

Algorithm 8 $TxnCtx.rollback()$

```
1. ctxId = ctx.getTxnCtxId()
2. for dsc in TxnCtx.getInvolvedDS() do
3.     dsc.rollback(ctxId))
4. end for
5. LTMc.rollback(TxnCtx))
6. return
```

Fig. 6. Implementation of *commit* and *abort*.

4.3 Recovery

In order to satisfy the durability property of the ACID acronym, the system must be able to survive after failures. During the recovery phase, the CoherentPaaS transaction manager opens the log file where the *write-set*are kept. *Write-sets* are read one by one and the *LTMc* requests the data store clients to redo the *write-set*. Data stores implement the $redoWS$ function. $redoWS$ receives the *write-set* provided at commit time, the transaction context id and the commit timestamp. The data store is in charge to recover the *write-set* from the array of bytes and apply the updates on the backing data store.

4.4 Failure scenarios

As stated in Section 4, CoherentPaaS transactions start from the moment the application invokes the *startTransaction* method. Then, the application associates one or more data store clients to the transaction. The application issues data manipulation operations on the different data stores. Changes made by the transaction (*write-set*) are kept as private versions on the data stores and are only visible to the transaction. Other transactions do not observe these updates until the transaction commits. When the application invokes the *commit* method, the following steps are executed:

1. Acquire a commit timestamp
2. Getting the *write-set* from each data store and persisting the transaction global *write-set*in the log. After this step is completed, the transaction is said to be $committed$,
3. Apply the *write-set*, that is, request every data store to make the private versions to become public
4. Inform the commit timestamp to the holistic transaction manager. This makes that the newly generated versions to become visible to other transactions.

The different components of CoherentPaaS participating in a transaction run on a distributed system, therefore, failures might affect to one or more components simultaneously. Any failure on any component that happens before the second step of the commit is completed, triggers the transaction rollback. After the second step is completed, the transaction is considered to be committed. The recovery process undoes the effects of aborted transactions and completes the transactions that executed successfully the second step.

4.5 Local Transaction Manager Client failures

Local Transaction Manager Client orchestrates the transaction life cycle. Internally it keeps a connection to every component of the CoherentPaaS transaction manager. It also keeps track every active transaction context object. *LTMc* persists the *write-set* of transaction on the transaction manager. Every transaction persisted is considered to be *committed*. When a *LTMc* fails all non-committed transactions are aborted.

When a *LTMc* is restarted after a failure, it connects to the CoherentPaaS transaction manager. Then, all data store clients registered on the *LTMc*, see *Algorithm* 1. The data store clients are registered at bootstrap to run the recovery process before making the system available to the application.

The recovery process is described in *Algorithm* 9. First, the log file is open (line 1). The recovery process starts reading one by one the log records from log file (line 2). The log record contains the following information: the transaction context id, the commit timestamp and the global write-set (the concatenation of all data store' *write-set*). The first byte of the *write-set* contains the data store id, $dsID$. The data store id is used to identify the data store at commit time.

For each log record, the *LTMc* reads one by one the *write-set* (line 6 to 10). For each *write-set*, the data store id is retrieved in order to know what data store has to redo the transaction. The data store is requested to redo the *write-set*, by the invocation of the $redoWS$.

Once the complete log record is reflected on the data stores, the *LTMc* informs the successfully committed transaction id and commit timestamp to the CoherentPaaS transaction manager to update the conflict's table and the snapshot isolation counter.

An observer reader may have observed already that a *write-set* could be applied more than once. Since we run snapshot isolation, overwriting some version of some data item is an idempotent operation. So no new values are generated every time $redoWS$ is invoked.

4.6 Data store failures

When a data store is restarted after a failure, it performs its internal recovery process and afterwards they are registered back in the *LTMc*. The *LTMc* triggers a data store recovery process, similar to the one described on the *LTMc* recovery process. The log file is open and it is iterated to read every log record, but instead of redoing all the *write-set* of the log record, only the recovering data store *write-set*, identified by $recoveringDsID$, are applied. The *Algorithm* 10 shows the recovery protocol executed by the *LTMc* when a data store is restarted and registered in the *LTMc*.

5 Conclusions and Future Work

We have presented a protocol to enhance transaction consistency semantics across heterogeneous data stores. The protocol dotes non-transactional data stores, such as MongoDB or HBase, with full ACID semantics compliance providing the dominant isolation level in the database market, Snapshot Isolation. Transactional data stores like LeanXCale, MonetDB have been easily incorporated to the platform extending their transactional engines to participate in CoherentPaaS transactions in a consistent and transparent manner to application developers.

We have extended the CumuloNimbo [6] transactional engine to support multiple data stores and we have defined and implemented an API to homogeneously integrate heterogeneous data stores. Data stores have implemented this API and implemented multi-versioning in the backing store to be able to support Snapshot Isolation.

We plan to do an exhaustive evaluation to measure the overhead of the transactional system on each data store individually. The evaluation will consist in running the Yahoo! Cloud Serving Benchmark (YCSB) [10], which is the de facto benchmark for cloud data stores. The evaluation will run the benchmark with different workloads (read-only, update-only, read and update with different percentages of each operation type) for every data store. Each data store will be evaluated with and without transactional support, thus we can measure the real overhead of our transactional system. Data stores that have implemented multi-versioning will be also evaluated with and without multi-versioning support to detect improvements opportunities in our implementation.

Algorithm 9 *LTMc* recovery

```
1. logFile = LTMc.openLogFile(ltmcId)
2. while not EOF logFile do
3.    logRecord = logFile.next()
4.    ct = logRecord.commitTimestamp
5.    txnCtxId = logRecord.txnCtxId
6.    for ws in logRecord do
7.       dsID = ws.id
8.       dsClient = LTMc.getDataStoreClient(dsID)
9.       dsClient.redoWS(txnCtxId, ct, ws)
10.   end for
11.   LTMc.commitTxn(txnCtxId, ct)
12. end while
13. return
```

Algorithm 10 Data store recovery

```
1. logFile = LTMc.openLogFile(ltmcId)
2. while not EOF logFile do
3.    logRecord = logFile.next()
4.    ct = logRecord.commitTimestamp
5.    txnCtxId = logRecord.txnCtxId
6.    dsClient = LTMc.getDataStoreClient(recoveringDsID)
7.    for ws in logRecord do
8.       dsID = ws.id
9.       if dsID == recoveringDsID then
10.         dsClient.redoWS(txnCtxId, ct, ws)
11.         break
12.      end if
13.   end for
14.   LTMc.commitTxn(txnCtxId, ct)
15. end while
16. return
```

Fig. 7. Implementation of the recovery protocol

Acknowledgments

This research has been partially funded by the European Commission under projects CoherentPaaS and LeanBigData (grants FP7-611068, FP7- 619606), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), and the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883).

References

1. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 15–15. USENIX Association, 2006.
2. Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
3. Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.
4. N. Martínez-Bazan, S. Gómez-Villamor, and F. Escalé-Claveras. Dex: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 124–127, April 2011.
5. Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.
6. Ricardo Jimenez-Peris, Marta Patino-Martinez, Bettina Kemme, Ivan Brondino, José Orlando Pereira, Ricardo Vilaça, Francisco Cruz, Rui Oliveira, and Muhammad Yousuf Ahmad. Cumulonimbo: A cloud scalable multi-tier sql database. *IEEE Data Eng. Bull.*, 38(1):73–83, 2015.
7. Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.
8. Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.
9. Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases*, pages 1–41, 2015.
10. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.