Transaction management across data stores

Ricardo Jimenez-Peris
LeanXcale,
IMDEA Software Institute,
Campus de Montegancedo, Madrid, Spain
Email: rjimenez@leanxcale.com

Marta Patiño-Martinez*
Universidad Politécnica de Madrid, ETSIINF,
Campus de Montegancedo, Madrid, Spain
Email: mpatino@fi.upm.es
* Corresponding author

Iván Brondino
LeanXcale,
IMDEA Software Institute,
Campus de Montegancedo, Madrid, Spain
Email: ivan.brondino@leanxcale.com

Valerio Vianello
Universidad Politécnica de Madrid, ETSIINF,
Campus de Montegancedo, Madrid, Spain
Email: vvianello@fi.upm.es

Abstract: Companies have evolved from a world where they only had SQL databases to a world where they use different kinds of data stores such as key-value data stores, document-oriented data stores and graph databases. This scenario rose new challenges such as data model heterogeneity and data consistency. There could be inconsistencies in case of failures during business actions requiring to update data scattered across different data stores due to the lack of transactional consistency across data stores. In this paper we propose an ultra-scalable transactional management layer that can be integrated with any data store with multi-versioning capabilities. This layer has been developed in the context of the FP7 CoherentPaaS European Project where it was integrated with six different data stores, three NoSQL data stores and three SQL-like databases. We particularly focus on the ultra-scalable transaction management API and how it can be easily integrated in any versioned data store.

1 Introduction

Companies have evolved from a world where they only had SQL databases to a world where they use different kinds of data stores such as key-value data stores, document-oriented data stores and graph databases. The reason why they have started to introduce this diversity of persistency models is because different NoSQL technologies bring different data models with associated query languages and/or APIs. However, they are confronted now with a problem in which they have the data scattered across different data stores. This problem lies in that one a business action requires to update the data, the data reside in different data stores, and they are subject to inconsistencies in the advent of failures and/or concurrent accesses. These inconsistencies appear due to the lack of transactional consistency that was guaranteed in their traditional SQL databases but it is not guaranteed within the NoSQL data stores neither across data stores and databases. A second problem that appears in this polyglot persistence environments is that applications cannot correlate data across data stores without replicating the functionality of the database at the application level. CoherentPaaS comes to remedy this situation.

CoherentPaaS provides an ultra-scalable transactional management layer that can be integrated with any data store with multi-versioning capabilities. The layer has been integrated with six different data stores. It has been integrated with three different kinds of NoSQL data stores: a document-oriented, MongoDB, a key-value, HBase, and a graph database, Sparksee. It has also been integrated with three databases, an OLTP SQL database, LeanXcale, a columnar ROLAP SQL database, MonetDB, and an in-memory MOLAP active database, ActivePivot.

CoherentPaaS also addresses the issue of querying across data stores. This is the topic of a different paper (Kolev et al., 2016).

In what follows, we describe this generic ultra-scalable transactional management layer and focus on its API and how it can be integrated in different ways with different data stores and databases.

2 Transactions and ACID properties

2.1 Why transactions

Transactions are a very important abstraction to program database applications since they remove two very hard problems for application developers. The first one is dealing with concurrency. Programming concurrent applications is complex, requiring highly qualified engineers and it is also quite error prone with errors difficult to reproduce and to debug. Transactions solve this problem by providing implicit concurrency control. Developers just need to bracket with transactions the access to the database that they want to have implicit concurrency control, for instance, the operations for making a money transfer between two bank accounts. The implicit concurrency control provided by transactions provides isolation across different users and/or applications accessing the shared data on the database. The second problem that transactions solve is to guarantee data consistency in the advent of failures. Programming an application that has to tolerate failures and guarantee the consistency of the persisted data is very hard and extremely complex, totally out of scope for regular business application developers. Transactions fully solve this issue by guaranteeing data consistency across all operations executed within a bracketed transaction. In the next section, we describe in more technical detail the properties provided by transactions, so-called ACID properties.

2.2 ACID properties

A transaction can be described formally as a sequence of data operations that are executed in an atomic way. Transactions provide the so-called ACID properties (Bernstein et al., 1987), namely:

- *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 (then it is said that the transaction committed) or nothing if it does not succeed (then it is said that 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 synchronisation 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.

2.3 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 serialisability (Berenson et al., 1995). Serialisability 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 serialisability 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 (CumuloNimbo Project, http://cumulonimbo.eu) has become very popular because it only introduces a single anomaly known as write skew that many applications do not trigger. There has been described several methods on how to attain serialisability on top of snapshot isolation (Fekete et al., 2005; Cahill et al., 2008).

Snapshot isolation basically splits the synchronisation 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. 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 labelled 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 labelling.

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 transactional processing

In this section, we present the CoherentPaaS transactional processing. We present first a centralised naive implementation and then the holistic transactional processing.

3.1 Centralised transactional processing

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 (Jiménez-Peris et al., 2012; CumuloNimbo Project, http://cumulonimbo.eu) 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.

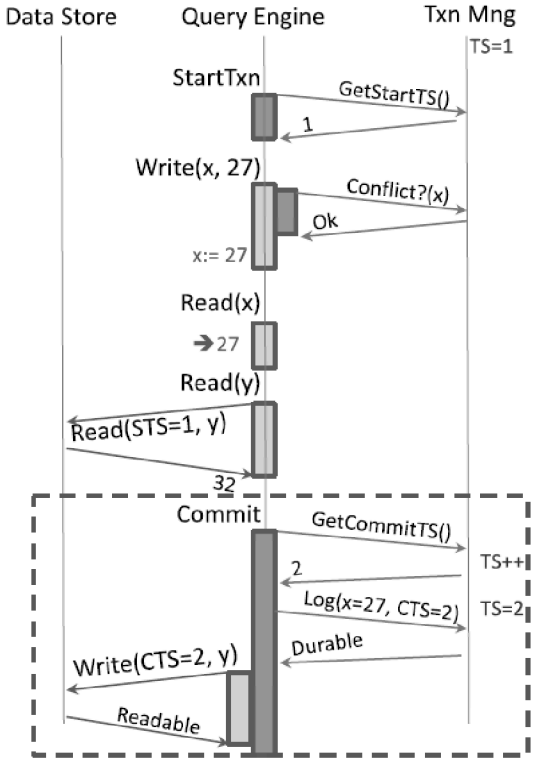
We assume a multi-version system where each write operation wi(xi) of transaction Ti on record x creates a new private version x, and each read operation ri(xj) of transaction Ti reads the latest version of x, xj created by a committed transaction Tj such that $j < i$ and there is no other committed transaction Tz, 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 Ti reads a snapshot of the database that reflects the latest committed versions of all records as of start time of Ti. In particular, this means that if Ti performs a read ri on x, then it reads either the private version Ti previously created (read your own writes) or it reads the version xj created by Tj such that Tj was the last transaction to write x and commit before Ti 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.

In Figure 1, we illustrate a possible execution of transactions under snapshot isolation in a centralised 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.

Figure 1 Transaction processing with snapshot isolation (see online version for colours)



At start time of a transaction Ti, the transaction manager (Txn Mng in the figure) assigns the current counter value, TS, (reflecting the last committed transaction) as start timestamp (ST). In the figure, the initial value of the counter is 1 (shown as TS = 1). When a transaction Ti 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 Tj, i.e., Tj has not committed yet or its commit timestamp is larger than Ti's ST (C(Tj) > S(Ti)), and Tj has written x. If there is no conflict, the write can proceed, creating a new private version of x, so far only visible to Ti itself. If there is a conflict, Ti must be aborted to guarantee the snapshot write property. This simply means to discard the private versions Ti has created so far. When a transaction Ti requests to read a record x, the data store has to provide the record created by transaction Tj with commit timestamp C(Tj), such that C(Tj) ≤ S(Ti), and there is no version of x created by a transaction Tk such that C(Tj) < C(Tk) < S(Ti). This provides the snapshot read property.

At commit time of transaction Ti, several things have to be accomplished. First, the transaction requests a commit timestamp C(Ti) to the transaction manager, which is done by assigning it the next counter value. Second, all record versions created by Ti must be labelled with C(Ti). 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 store. 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(Ti) is assigned as a ST to a new transaction Tk, Tk must be able to see the updates performed by Ti, i.e., the updates must be visible in the data store.

While existing solutions might differ from above outline in the way timestamps are assigned or when conflict detection is done, the principle execution steps are conceptually similar.

4 Transaction management in CoherentPaaS

CoherentPaaS has a set of subsystems that play an important role in transactional processing. They are: holistic transactional manager, data stores and common query engine. 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 an interposed transactional coordinator. All components dealing with transactions do have a collocated local transaction manager (LTM) (see Figure 2). The LTM is accessed via an local transaction manager client (LTMClient) (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, the common query engine and the data stores 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. The common query engine itself also has its proxy client that it is collocated with the application (data stores do not this client since they never invoke directly the common query engine, simply reply to its requests).

In the following sections, we describe how the application interacts with an LTMClient, the common query engine client (CQEClient) and the different data store clients.

4.1 Transaction manager API

In CoherentPaaS, an application may access the data stores either through the common query language (via the CQEClient) or directly through the data store clients. In both situations, the application will bracket the transactions explicitly.

In order to execute a transaction the next steps must be performed:

1 Initialisation of the LTMClient.

2 Register the data store clients with the LTMClient.

3 Get a transaction connection.

4 Execute one or more transactions using the transaction connection.

The first step enables to access the LTM, as it happens with a JDBC driver. The second step is performed to enable the LTM to propagate transparently the transactional context relieving the application from this burden. The third step provides the handle to manage transactions for a CoherentPaaS session.

Figure 2 Interaction across subsystems involved in transactional management (see online version for colours)

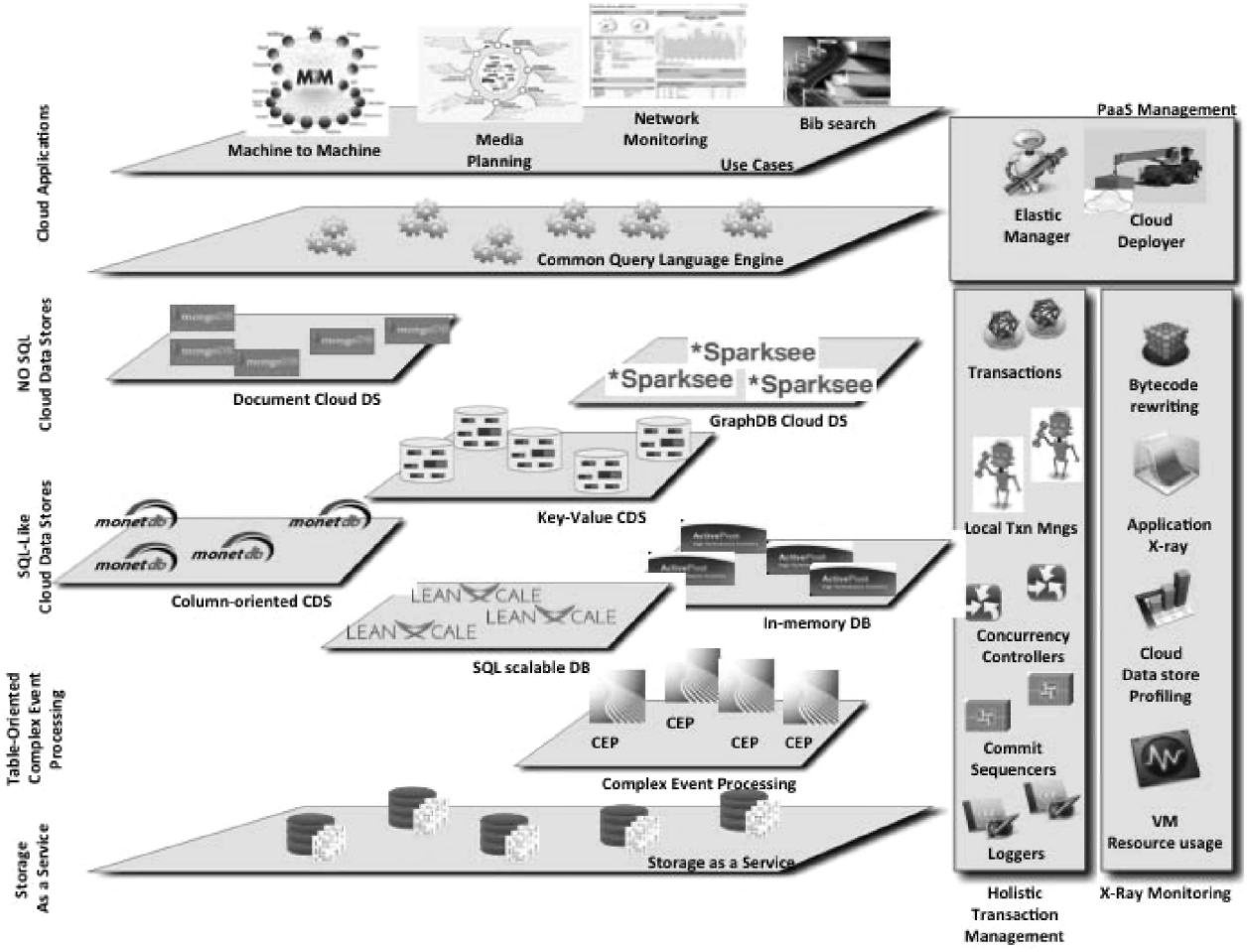


Figure 3 Registration

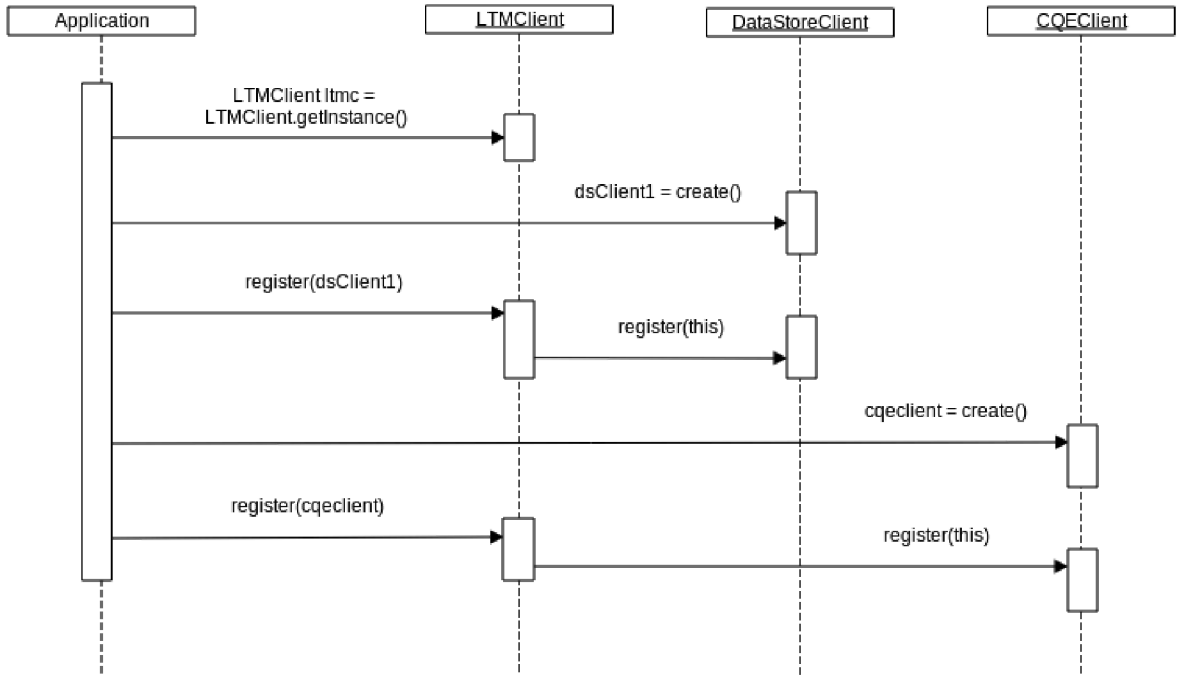
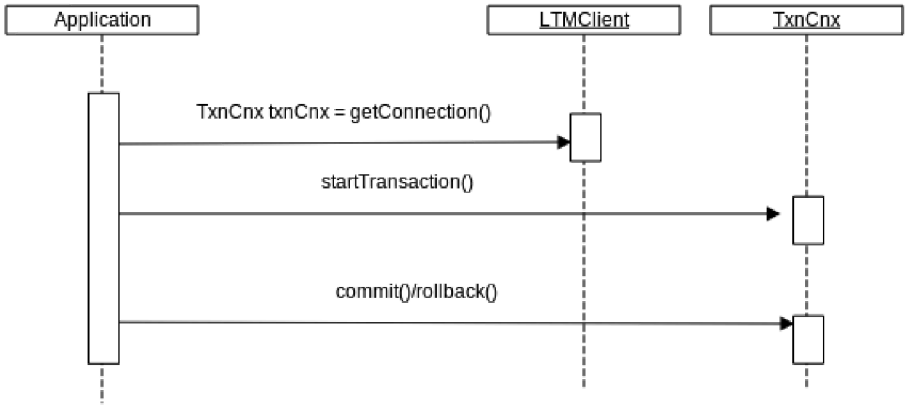


Figure 4 Getting a connection



4.2 Initialisation and registration with the transaction manager

The first step lies in initialising the system and registering with the transactional manager. Figure 3 shows how an application initialises the LTMClient. The application is going to access both to the data stores directly and through the common query engine. The application first requests an instance of the LTMClient using the *getInstance*() method. Internally the LTMClient connects to the LTM. Then, the application instantiates one or more data store clients (*create*) and registers (*register*) them with the LTMClient. This makes the LTMClient aware of the existence of the data store clients.

The common query engine is treated as another data store. The application also requests an instance of the proxy client, the CQEClient, and registers it.

4.3 Getting a transaction connection

The application needs to obtain a transaction connection before submitting any transactional operation. The transaction connection (*TxnCnx*) is obtained from the LTMClient (LTMC) and is analogue to a JDBC connection. The transaction connection is in charge of keeping the transactional context (*TxnCtx*). It is also in charge of interacting with the LTMClient in a transparent way to the application. The application contacts directly the data store clients without being aware of existence of the transaction connection. The data store clients interact with the connection to perform all the interaction with the transactional management. There is a single transaction context active at a time per transaction connection.

Figure 4 shows the interaction with the transaction connection (TxnCnx). First, the application requests the

LTMClient a connection. The LTMClient returns a transaction connection. Now the application is able to execute transactions in CoherentPaaS.

There are two ways of managing transactions and transactional context propagation. The transactional context can be propagated explicitly to the data stores by the application. However, it is more convenient, when feasible, to propagate it implicitly. In latter case, the application is relinquished from this responsibility. Here, we leverage the thread state management to allow data store clients to find out the connection associated to the application thread and the associated transactional context. Implicit propagation is the preferred way to manage transactional context propagation since it simplifies the application development. The explicit management is provided for those data stores that are not willing or cannot perform (they do not support sessions) the implicit transactional context propagation.

4.4 Executing transactions in CoherentPaaS

There are two ways to execute transactions in CoherentPaaS. The application may either access the data using the data stores directly or through the common query engine (or even a combination of both). In this section, we explain both scenarios, as they are slightly different from the point of view of the internal interactions in CoherentPaaS.

4.5 Executing transactions using the data store clients

In this section we show how an application executes transactions accessing the data stores directly.

Figure 5 illustrates an example where the application executes a transaction using the CoherentPaaS Transaction Manager and two different data stores, *DataStore1* and *DataStore2*. The figure shows the interactions once a transaction connection has been created.

The application starts the transaction invoking the start primitive. Then, it reads a record *X* from *DataStoreClient1*, to do so, the application executes *read*(*X*) on *DataStoreClient1* using the standard API provided by *DataStore1*. Internally, *DataStoreClient1* asks the transaction connection (*TxnCnx*) for the transaction context (*TxnCtx*). *DataStoreClient1* accesses the *TxnCnx* using a thread local variable. *TxnCtx* has the information related to the transaction, namely, the transaction id (*TID*) and the *ST*. *DataStoreClient1* gets the ST from the *TxnCtx* (*getStartTimestamp*). Finally, *DataStoreClient1* contacts the data store *DataStore1* to read the right version (*read*(*X*, *TS*)). Read operations on different stores will be executed in the same way.

Write operations are similar to read operations however, conflicts need to be checked. The *DataStoreClient1* asks to the transaction connection, *TxnCnx*, for the transactional context. Then, it gets the *ST* from the transactional context, *TxnCtx*. The *DataStoreClient1* asks again to the transaction connection, *TxnCnx*, to check if there is any write-write conflict (*hasConflict*) providing the identifier of the *DataStoreClient1* and the key to be modified. If the data store provides write-write conflict detection, this invocation is not executed. If there is no conflict, *DataStoreClient1* requests the transaction context, *TxnCtx*, for the TID and stores in an internal buffer (*writetoWriteSet*) the write operation with key *Y*, value *Y*1 and the transaction identifier, *TID*.

Finally, the transaction commits. To do so, the application invokes the commit procedure at the LTMC. The first step in the commit phase is to log the write-sets. The transaction connection, *TxnCnx*, requests every data store client involved in the transaction (*DataStoreClient1* and *DataStoreClient2*) the write-set (*getWS*). *DataStoreClient1* asks again for the transaction context to the connection, *TxnCnx*, which returns the commit timestamp *CSj* of the transaction. The clients of the data stores can provide either the writeset or a handle to the writeset in its internal log, if the data store does logging by itself. Once the LTMC collects all the writesets involved in the transaction, it provides the LTM the write-sets and they are all flushed to the CoherentPaaS log. Then, the transaction is marked as committed and the LTMC returns the control to the application.

Finally, the LTMC asks every data store client involved in the transaction to apply the writeset (*applyWS*) to the corresponding data store in order to make the writeset visible to future transactions. Again, every involved data store client uses the transaction context, *TxnCtx*, from the transaction connection, to get the transaction identifier, *TID*. Then, the data store clients get the corresponding writeset and apply every change done by the transaction in the underlying data store. Once the write-set application phase finishes, the LTMC informs the LTM that the transaction is now durable and readable. The LTM will take into account to advance the snapshot counter when possible.

4.6 Executing transactions from the common query engine

In this section, we show how the application performs transactional access to the data stores through the common query engine.

The application uses the LTMClient and the CQEClient. The CQEClient behaves as the data store clients in the previous section but, instead of executing data store requests, redirects the MdbQL queries to the common query engine (see Deliverable 3.1). The common query engine is in charge of parsing the queries and generating an execution plan that may access to the different data stores in CoherentPaaS.

Figure 5 Transaction execution accessing the data stores

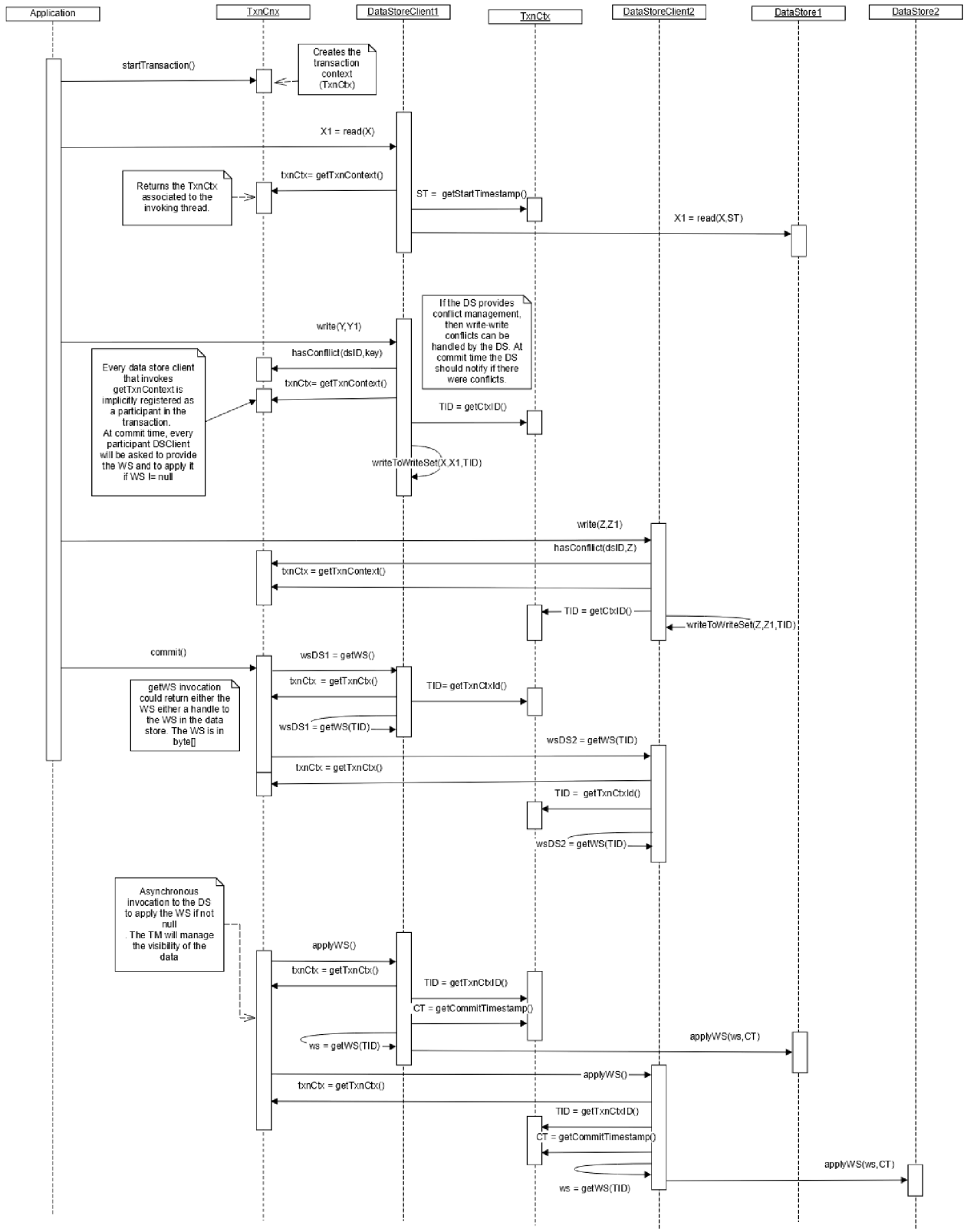
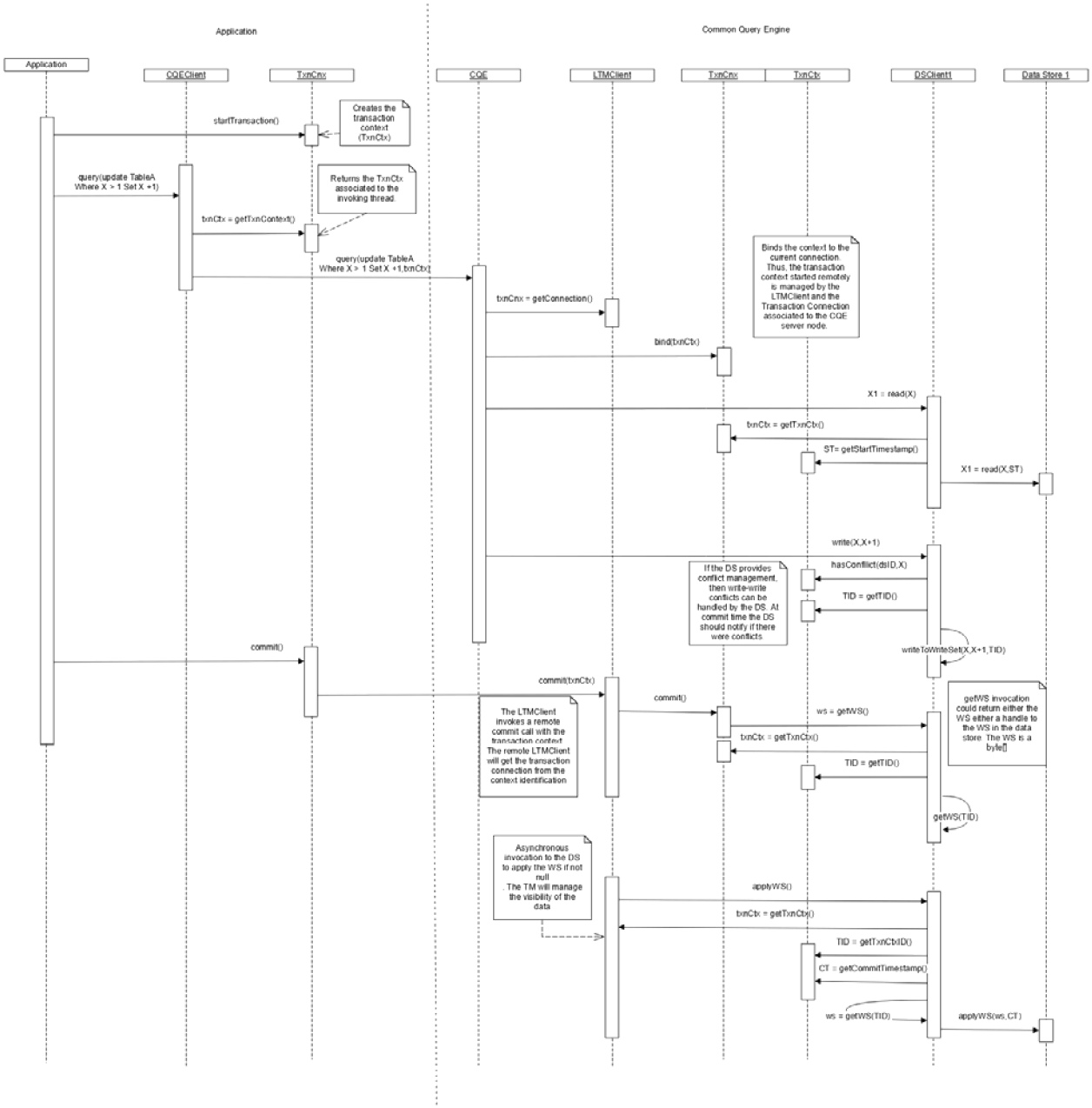


Figure 6 Transaction execution with the common query engine



In this setup, the LTMClient is deployed both at the application side and also at the common query engine side. The LTMClient deployed at the application side acts as a proxy between the application and the LTMClient deployed at the common query engine. The LTMClient co-located with the common query engine is in charge of solving the requests to the data store client and communicates with its LTM. From the transactional point of view, this is a delegated (or interposed) coordinator of the transaction.

Figure 6 illustrates this scenario. The figure presents the steps after the LTMClient is created and a transaction connection is obtained. The application starts the transaction invoking *startTransaction* from *TxnCnx*. This invocation creates a transaction context and binds it to the connection. Then, the application requests the execution of a query, '*update X where X* > 1 set *X* + 1', to the CQEClient. The CQEClient redirects the request to the CQE server. This invocation includes the transaction context (*TxnCtx*) as a

parameter. The CQE server requests a connection to its LTMClient and binds the transaction context received as a parameter to the new transaction connection (*bind*). The CQE parses the query and finds that two operations will be executed, read the value of X and then increment it.

At this point, the CQE plays the same role that the application did in the previous section. That is, the CQE requests operations to the data store clients within a transactional context. Once the CQE finishes the execution of the query, it returns the control the CQEClient and the CQEClient returns the control to the application. When the application invokes the commit operation on the transaction connection, the commit is redirected to the LTMClient located at the CQE server. That LTMClient will coordinate the commit phase just like in the previous scenario. Once the commit phase is over, the LTMClient informs the LTMClient deployed at the application side that the transaction has committed.

5 Conclusions

In this paper, we have presented how holistic transactional management, transactions across multiple data stores, has been achieved in CoherentPaaS. With this holistic transaction management now it becomes possible to get the same transactional consistency guarantees within NoSQL data stores and across data stores and it was possible within a single transactional SQL database. With this functionality, now it becomes possible to store each data item under the most appropriate data model, relational, key-value, document, graph, …, without sacrificing the data consistency. By combining these holistic transactions with MdsQL, then data can also be queried across data stores and therefore fully solving the issues that has emerged in polyglot persistency environments.

Acknowledgements

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

References

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P. (1995) 'A critique of ANSI SQL isolation levels', *SIGMOD Rec.*, May 1995, Vol. 24, No. 2, pp.1–10, doi: 10.1145/568271.223785.

Bernstein, P.A., Hadzilacos, V. and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN: 0-201-10715-5.

Cahill, M.J., Röhm, U. and Fekete, A.D. (2008) 'Serializable isolation for snapshot databases', *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp.729–738.

CumuloNimbo Project [online] http://cumulonimbo.eu (accessed 14 November 2016).

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P. and Shasha, D. (2005) 'Making snapshot isolation serializable', *ACM Transactions on Database Systems*, Vol. 30, No. 2, pp.492–528, doi: 10.1145/1071610.1071615, ISSN: 0362-5915.

Jiménez-Peris, R., Patiño-Martínez, M., Magoutis, K., Bilas, A. and Brondino, I. (2012) 'CumuloNimbo: a highly-scalable transaction processing platform as a service', *ERCIM NEWS No. 89, Special Theme Big Data*, April, pp.34–35.

Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R. and Pereira, J. (2016) 'CloudMdsQL: querying heterogeneous cloud data stores with a common language', *Distributed and Parallel Databases*, Vol. 34, No. 4, pp.463–503, doi: 10.1007/s10619-015-7185-y.