Transactional Processing for Polyglot Persistence

Ricardo Jimenez-Peris *, Marta Patiño-Martinez †, Iván Brondino † and Valerio Vianello †
*LeanXcale
Madrid
Email: rjimenez@leanxcale.com
† Universidad Politécnica de Madrid Spain Email:mpatino,ibrondino,vvianello@fi.upm.es

***Abstract*—NoSQL data stores have emerged in last years as a solution to provide scalability and flexibility in data modelling for operational databases. These data stores have proven that they are better suited for some kinds of problems than relational databases. In order to scale, they relaxed the properties provided by relational databases, mainly transactions. However, transactional semantics is still needed by most applications. In this paper we describe how CoherentPaaS provides scalable holistic transactions across SQL and NoSQL data stores such as document-oriented data stores, key-value data stores and graph databases.**

***Index Terms*—NoSQL, databases, transaction processing.**

I. Introduction

Companies have traditionally used transactional SQL databases to keep their operational data. In the last few years, with the advent of NoSQL, many companies are embracing new NoSQL data stores. These data stores include document oriented data stores, such as MongoDB [1], key-value data stores, such as Apache HBase [2] and graph databases, such as Neo4J [3] and Sparksee [4]. Companies use NoSQL technology due to different reasons. In some cases, because the flexibility of the data models that enables to store semi-structured data that has a high degree of variability that makes too hard to store them on relational databases oriented towards fully structured data. This is for instance the case for key-value data stores that are schemaless and enable to store tuples arbitrary schemas. In some other cases, it is because of the need of a different query language or API to access the data in a more comfortable manner. For instance, document-oriented data stores such as MongoDB offer an API that enables to query semi-structure documents in a flexible and comfortable way. Sometimes it is due to efficiency as happens with graph databases. Graph databases enable to perform traversals of a graph with a single invocation to the graph database what avoids a myriad of exchanges between the client and server side needed in a relational database to make a graph traversal. Another reason why NoSQL solutions have been adopted is due to some of them satisfy the scalability requirements of some applications that traditional transactional SQL databases fail to fulfill.

This trend has resulted in a situation where many companies now store their operational data in a diverse set of technologies both SQL and NoSQL leading to the so-called Polyglot Persistence [5]. Companies with Polyglot Persistence environments are facing now new challenges. The first one is that NoSQL data stores are non-transactional or at most allow transactions updating a single row. This fact creates many difficulties in the advent of failures and/or concurrent updates. It has been reported that BitCoin was stolen a lot of money because they were using NoSQL technology [1]. A hacker ran in parallel massive transfers of money over a set of accounts, since there was no transactional isolation, the hacker was able to transfer the original amount many times before the first transfer of money was able to update the value of the account to zero.

The lack of atomicity also causes many problems when multiple rows should be updated as part of a business operation, a failure in the middle will result in an inconsistent database since only a fraction of the rows would be updated. For instance, if a key-value data store such as HBase is used in a banking application, if there is a failure during a money transfer between two accounts it might happen that the withdrawal from the origin account completes but the deposit in the destination account is not performed. This scenario will lead to an inconsistent database.

The above problems become more acute in a Polyglot Persistence environment. It is very typical to store structured data in an operational SQL database and the semi-structured part of the data on a NoSQL data store such as a key-value or document-oriented data store. However, both pieces of data are part of an integral conceptual business operation. A failure after one data store is updated and before a second data store is updated results in an inconsistent logical database (the set of data stores being used).

The CoherentPaaS FP7 European project [6] is addressing the problems that companies are facing when they use Polyglot Persistence. This paper addresses the set of pains related to the lack of transactional ACID properties within and across data stores in these Polyglot environments. CoherentPaaS is leveraging the ultra-scalable transactional technology from LeanXcale [7] to bring scalable transactional processing to Polyglot Persistence environments and to NoSQL data stores. The rest of paper presents an overview of CoherentPaaS (Section II), then transactions are presented in Section **??**. Finally, we present an overview on how transactions have been scaled and the modifications needed on the data stores in order to provide transactional semantics and transactions across different data stores (Section IV).

[1] http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/

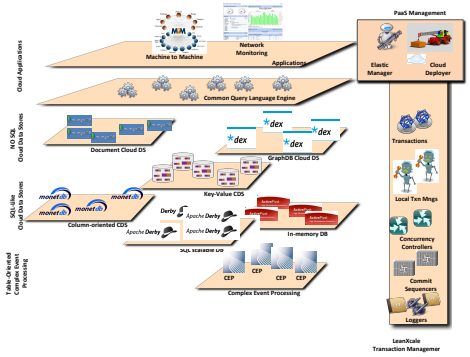

Fig. 1. CoherenPaaS components

II. CoherentPaaS

CoherentPaaS project goal is to reduce the effort required to build and increase the quality of the cloud applications using multiple cloud data management technologies via a single query language, a uniform programming model, and ACID-based global transactional semantics. Figure 1 shows the components of the transactional management system, LeanXcale, the common query language, CloudMdsQL [10], used to access all the data stores using a common programming language, and the different types of data stores that are integrated in CoherentPaaS. All types of data stores are being considered in CoherentPaaS, SQL databases, columnar, graph data stores and document oriented data stores.

III. Transactions

Transactions are the most convenient way to guarantee data consistency in the advent of failures and concurrent accesses. They provide the so-called ACID properties: Atomicity, Consistency, Isolation and Durability. Atomicity provides all-or-nothing semantics in the advent of failures. That is, in the advent of failures either all the updates in a transaction succeed or the final effect is as none of them have been performed. Consistency is enforced by the application. Basically, a transaction that starts with a consistent database should produce a new consistent database, that is, the application should be correct. Isolation states that the effect of executing concurrent transactions should guarantee certain properties. In the case of serializability a concurrent execution of transactions should be equivalent to a serial (sequential) execution of these transactions. Isolation simplifies the development of applications, since they access the database as a sequential application, that is, they do not have to program explicit concurrency control. Isolation enforces concurrency control implicitly and so the application developer simply programs a sequential applications that extremely much simpler than a concurrent application. Durability guarantees that once a transaction is committed, its updates cannot be lost, even if there are failures after the transaction has been committed.

Serializability is quite powerful. However, it has an inherent bottleneck. Serializability when there are predicate reads, reads that select a set of items based on a predicate, it highly constrains the concurrency of updates. The issue is that read and write operations of concurrent transactions conflict that is, an arbitrary predicate can conflict with any concurrent update on the same table. When the predicate is not performed over indexed columns the only way to guarantee serializability is by locking the whole table what prevents doing updates over the table concurrently with the predicate reads.

Different isolation levels have been proposed. For instance, the ANSI isolation levels enable to reduce the number of conflicts by reducing the level of isolation. Unfortunately, the ANSI isolation levels below serializability can exhibit serious anomalies in most applications that results in data inconsistencies. Fortunately, in 1995 a new isolation level was proposed called Snapshot Isolation (SI) [**?**] that is very close to serializability but, it avoids the conflicts between reads and writes, therefore avoiding on one hand the bottleneck between predicate reads and updates, and on the other hand, avoiding the interference between long read queries and updates that also result in highly constraining the potential concurrency across transactions.

Snapshot isolation provides for any running transaction a photograph or snapshot of the committed state of the database as of the start time of the transaction. In this way, it provides a high level of isolation since a running transaction does not observe any changes for concurrent transactions that commit. Snapshot isolation can be enforced by using two rules: the read rule and the write rule. The read rule ensures that a transaction observes all the updates of the transactions committed before the transactions starts. The write rule guarantees that no two concurrent transactions can update the same data item and commit. That is, at least one of them should be rollback. In this way, in snapshot isolation only write-write conflicts can happen, but never read-write conflicts. There are two ways for implementing the write rule: first updater wins or first committer wins. If there are two concurrent transactions trying to update the same item, the one that updates first the item will commit if the first updater rule is implemented. The second updater will abort. If the first committer rule is implemented, the transaction that commits in the first place is the one that will commit, the other transaction will abort.

IV. Scalable Transactional Processing

Transactions have been for decades the bottleneck that prevented databases from scaling-out. LeanXcale [**?**] implements a parallel-distributed transactional engine able to process millions of transactions per second. The transactional processing decomposes the ACID properties and scales each of them separately, but in a composable manner. The provided isolation is Snapshot Isolation (SI) [8]. This technology has been used to build an ultra-scalable full SQL full ACID database.

LeanXcale implements atomicity in a component so-called local transaction manager (LTM) that manages the lifecycle of a set of transactions. This component scales out by using as many LTM as needed to scale the number of concurrent transactions being submitted to the database. Isolation is decomposed in LeanXcale into two subproperties, isolation of

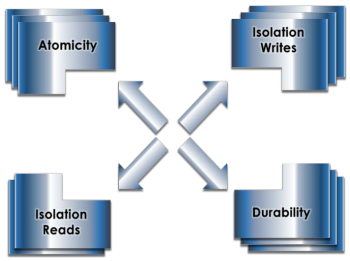

Fig. 2. Scaling-Out ACID Properties

writes and isolation of reads. Isolation of writes is enforced by conflict managers. A conflict manager takes care of detecting conflicts over a set of keys. Any two transactions modifying the same key will access the same conflict manager instance. Conflict management is scaled out by assigning the responsibility of different set of keys to different conflict managers.

Isolation of reads is the most sophisticated task. It is deal with two components that provide commit timestamps and start timestamps guaranteeing the snapshot isolation semantics. Durability in LeanXcale is enforced by loggers. A logger takes care of persisting the redo log records of committed transactions from a subset of local transaction managers. Loggers are scaled out by assigning to each of them the responsibility of making durable different subsets of executing transactions.

All above components can be scaled out except the two dealing with isolation of reads. However, these two components perform a tiny amount of work per update transaction and benchmarking shows that they are able to do process the work corresponding to many millions of transactions per second. A detailed description of the ultra-scale transactional processing is presented in [9].

The LeanXcale transaction manager is agnostic to the data management layer and only requires from a data manager to implement multi-versioning. Multi-versioning requires that every time a row is updated, a new version of the row is created tagged with a new version number. This means that multiple versions of the same row might exist at a particular point in time.

This fact has been exploited in CoherentPaaS to enrich NoSQL data stores with transactional semantics. In particular, MongoDB, HBase, and Sparksee have been extended with transactional semantics. For that purpose, each data store must provide two capabilities LeanXcale transactional manager:

- Being able to tag versions of updated data items with a commit timestamp provided by the transactional manager.
- To be able to read from a given snapshot providing a start timestamp. This requirement has implied to extend the integrated NoSQL data stores to support multi-versioning.

CoherentPaaS provides the individual NoSQL data stores with ACID semantics. For instance, a full ACID version of MongoDB has been produced as well a full ACID version of HBase that enables arbitrary multi-row transactions. At the same time CoherentPaaS provides global transactions across all data stores. An application can start a global transaction using the transactional manager API and then access any integrated data store such as LeanXcale SQL database, MongoDB, HBase, Sparksee and Neo4j, with full transactional guarantees. This means that global transactions across any combination of the previous data stores provide full isolation, atomicity and durability.

V. Data Access

CoherentPaaS also deals with another major issue in Polyglot Persistence environments, that is, how to query data across different data stores that provide different query languages/APIs and have different data models. This issue has been addressed in CoherentPaaS by developing a federated query language, CloudMdsQL [10], that combines SQL with the native query languages/APIs of the underlying data stores. Basically, CloudMdsQL enables to combine subqueries written in the native query languages/APIs for the individual data stores and global queries in SQL across the data stores to join and correlate data across data stores. A detailed presentation is provided in [10].

Acknowledgments

This research has been partially funded by the European Commission under project CoherentPaaS (grant agreement FP7-611068), 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

[1] "MongoDB," http://https://www.mongodb.org/.
[2] "HBase," http://www.hbase.org.
[3] "Neo4j," http://neo4j.com/.
[4] "Sparksee," http://sparsity-technologies.com/.
[5] M. Fowler and P. Sadalge, *NoSQL Distilled*. Pearson, 2013.
[6] "CoherentPaas," http://coherentpaas.eu/ .
[7] "LeanXcale," http://leanxcale.com.
[8] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ansi sql isolation levels." in *ACM SIGMOD International Conference on Management Of Data*, 1995.
[9] R. Jimenez-Peris and M. Patiño-Martinez, "System and method for highly scalable decentralized and low contention transactional processing. 2011. patent application number 61/561/508. us patent and trademark office."
[10] B. Koleb, P. Valduriez, C. Bondiombouy, R. Jimenez-Peris, J. O. Pereira, and R. Pau, "Cloudmdsql: Querying heterogeneous cloud data stores with a common language," 2015.