Transaction Support in the Cloud. Taking Advantage of Classic Approaches

José Enrique Armendáriz-Iñigo, Itziar Arrieta-Salinas
Dpto. Ing. Matemática e Informática
Universidad Pública de Navarra
31006 Pamplona, Spain
{enrique.armendariz, itziar.arrieta}@unavarra.es

Joan Navarro, August Climent
Distributed Systems Research Group
La Salle - Ramon Llull University
08022 Barcelona, Spain
{jnavarro, auge}@salle.url.edu

Abstract—Cluster-based databases have been traditionally used for providing fault tolerant storage solutions. However, the limited scalability of this kind of systems has motivated the creation of a variety of storage systems based on the cloud paradigm, which achieve high scalability and availability by giving up some functionalities such as strong consistency. While this is fairly acceptable for a wide range of applications, there are still many use cases which cannot take advantage of the cloud paradigm because they are unable to resign their transactional nature. The purpose of this paper is to review how these novel systems overcome typical scalability and availability issues and present a new architecture able to offer transactional support on the cloud. This hybrid architecture is based on well-known replication protocols used in classic transactional systems which are combined according to the application workload, thus providing different consistency levels depending on the needs of each application. Moreover, the system inherits the elasticity and dynamism properties of the cloud paradigm, which makes it able to offer high scalability and availability levels without sacrificing transactional support.

Keywords—Dynamic systems; eventual consistency; cloud computing; replication.

I. INTRODUCTION

The ambitious requirements regarding availability and fault tolerance in modern software distributed applications entail the need for replicating and persistently storing vast amounts of data from different sources. Relational databases (also called cluster-based databases) have been considered as the proper choice so far, either as primary copy [1] or update everywhere [2] systems using total order broadcast [3], despite their static and scalability limitations [4]. Replication is the main cause of the stringent scalability capabilities of relational databases, in the sense that the greater the number of replicas that have to perform an update is, the less efficient the system is due to the amount of synchronization messages that replicas have to exchange. In fact, Brewer’s CAP theorem [5] states that a partition-tolerant distributed system can guarantee only one of the following two properties: data consistency or availability. Consequently, it is not possible to provide strong consistency without limiting system scalability, and vice versa. Thus, even with the usage of partial replication, database clusters do not scale due to the strong consistency maintained among replicas, which leads to additional network and database stalls.

This context motivated the creation of a new class of data storage systems called NoSQL (Not Only SQL), that operate in cloud platforms. In terms of scalability, the cloud gives the perception of infinite resources that can be elastically removed or added according to the varying workload, with little cost in terms of latency. NoSQL systems achieve high performance and availability by giving up some functionalities such as joins and ACID transactions, thus offering poorer semantics than those in traditional SQL statements. Here we can mention basic key-value data stores such as Amazon’s Dynamo [6], which only supports simple read and write operations to a data item uniquely identified by a key; or more complex solutions like Google’s Bigtable [7], which enhances the basic model by adding structure to the data and offering range queries and single-row transactions.

In general, these systems have deliberately relaxed traditional ACID properties, relying on a special form of weak consistency named eventual consistency [8], which states that if no new updates are made to an item, eventually all accesses will return the last updated value. This kind of consistency is too weak for applications that require a stronger isolation level such as serializable or snapshot.

A. Related Work

Recently, several ways to provide transactional support in the cloud have been proposed. Since leaving application developers in charge of ensuring transactional consistency [9] is both costly and inefficient (thus only making sense for applications that rarely require to provide transactions with strong consistency guarantees [10]), most cloud-based solutions providing transactional support focus on managing transactions on the server side, making use of a variety of different approaches. Some of the most representative examples include: ecStore [11], which provides transactional semantics across multiple rows; ElasTraS [12], which is an elastic database system capable of scaling up and down according to the transaction workload, but it does only support a simplified type of transactions that are executed within one single data partition (which are named mini-transactions and were originally defined in Sinfonia [13]);
Microsoft SQL Azure [14] and Google Megastore [15], which support transactions over multiple records, although they require these records to be co-located in some way; or Deuteronomy [16], which supports ACID transactions by decomposing functions of the database storage engine kernel into a transactional component that manages transactions and their logical concurrency control and undo/redo recovery, and a data component that caches data, knows about the physical organization and supports a record-oriented interface with atomic operations.

The way to efficiently support transactions in the cloud is by reducing the interaction among replicas to the minimum. A common solution is to partition data. The challenge is how to partition data so that transactions can be entirely executed in a single partition (it is already well known that standard TPC benchmarks, like TPC-C [17], [12], [18] and TPC-E [17], can be split in such a way). However, there may be cases in which some transactions (named multi-partition transactions) need to access several partitions during their execution. The issue here is twofold: on the one hand, network stalls appear as transactions are fragmented (e.g., imagine that it is needed to write on item $x$ in partition $P_1$ the value read from item $y$ that belongs to partition $P_2$); and, on the other hand, some coordination has to be provided to commit a transaction and globally maintain consistency across partitions. With regard to the first issue, some solutions are to speculatively execute transactions (without notifying the clients to avoid cascading aborts) that are ordered after a fragment that is waiting for its commit, as done in [17], or to use transaction flow graphs to determine rendezvous points along the fragments of a multi-partition transaction [18]. In order to commit a multi-partition transaction and thus give a consistent view of data through different partitions, several techniques have been proposed so far: a single coordinator [17], [19]; multiple coordinators [20]; the two-phase commit rule and its variants [21], [13], [11]; or the Paxos algorithm and its variants [22].

In some cloud solutions, each replica maintains all its assigned partitions in main memory, therefore avoiding all writing to disk [23]. In these systems, the replication degree needs to be high enough to ensure durability, but disk stalls or the cost of a distributed storage [12] are saved. Other systems [11] rely on in-memory data structures to reduce response time, but periodically dump this information to disk. The thread-to-data policy has been shown to be effective through exploiting the regular pattern of data accesses [17], [18]: there exists one thread per partition that sequentially executes operations belonging to different transactions. Together with this, the use of stored procedures avoids any interaction with the user during the execution of a transaction [17], [18].

Due to the replication degree, the user will potentially access different versions of the same partition. It strongly depends on the asynchronous propagation of updates to different replicas [11]. We can have strong consistency in the core (where updates are firstly executed) and then these replicas can act as forwarders to other replicas holding that partition which in turn propagates to other ones and so on. Hence, we can have a hierarchy of replicas with different degrees of outdatedness. The system can attach a timestamp at the start of the transaction and try to find the appropriate consistency version according to it. In this case, the system is ensuring a special form of weak consistency with some freshness guarantees. From a business point of view, this can be best seen as a service where you pay for consistency; the stronger the more expensive the service is. Finally, from the correctness point of view, the system generates a one-copy multiversion schedule of each partition in the system [24]; recall that we are using a hierarchy version tree according to the replication degree of the given partition. Nevertheless, the consistency can vary from 1-copy-serializable [21] if the user accesses to the core, 1-copy-SI [24] if the access is in one of the branches of the hierarchy tree and to one-copy multiversion if the user accesses several partitions.

B. Contributions

The purpose of this paper is to propose a new integral (in the sense that it does not rely on third party solutions) architecture born with the same spirit as classic transactional systems but inspired by the novel cloud tendencies able to reach high availability and scalability levels. This architecture treats low layer servers as if they were in the cloud while implementing a dynamic transactional system on the top. This allows to ensure the persistence of transactional updates while providing high scalability and availability as if they were executed on a cloud database.

Furthermore, the top layer of our proposal is able to adapt itself to the current transactional demands by building a dynamic set of virtual cluster-based databases (also referred to as partitions). Each partition can implement different replication protocols and demand different consistency levels according to each application needs. Thus, it will be possible to offer a range of QoS guarantees to applications by varying the tradeoff between the consistency level and the availability conditions provided. This would contribute to fill the portability gap found when moving some applications to the cloud.

The remainder of this paper is structured as follows: Section II introduces the proposed system architecture. Section III describes possible alternatives for replicating data partitions. Section IV discusses the different types of service that can be provided by the presented system. Finally, Section V presents some concluding remarks.

II. System Model

Our proposal consists in adapting classical transactional approaches to a cloud-based storage infrastructure considering the following assumptions:
There are no limitations concerning hardware nor physical location of any device in our system. Hence, the system itself has to be able to recover properly from failures as a distributed transactional system would do.

Scalability is a major requirement; data has to be stored and processed using cloud-based storage techniques. Moreover, it has also to provide elasticity facilities to adapt the system infrastructure to user needs.

Data is arranged in multiple partitions within our system and multiple transactions access them concurrently. Each partition (also referred to as replication cluster) is managed by a replication protocol.

Balanced partitions are dynamically produced attempting to minimize the number of distributed transactions to achieve a perfect partitioning scheme [18], [17].

Transactions accessing different partitions are tolerated but do not have to be processed efficiently.

The overall system architecture of our proposal is depicted in Figure 1 and consists of (1) a single meta-data cluster and (2) multiple replication clusters. Network connections between devices are drawn with lines (continuous for in-use connections and dashed for idle ones) and data flows are plotted with arrows.

The meta-data cluster has three major functionalities: (1) persistently store and maintain the location(s), version, and time stamp of each data item existing in every partition (it uses Paxos [22] to provide fault tolerance), (2) decide to what cluster belongs a given server and what protocol must run on it, and (3) arrange replication clusters to achieve perfect partitioning inside them. This is achieved through the following entities:

- **Data Manager.** It decides which is the best data location according to the current workload and application demands. It also decides when servers from a given partition have to persistently store their data.
- **Protocol Manager.** It chooses the replication protocol that best fits on each partition. This is done by building up a knowledge model [25] that takes into account (1) the current data updates and queries, and (2) the global partition latency (e.g., a read intensive partition may provide a higher throughput with a primary-backup scheme rather than a 2PC multi-master).
- **Partition Manager.** It builds the aforementioned replication clusters by assigning to each server a partition. This is done when a server joins the system and also while it is belonging to an active partition providing system elasticity.
- **Speed dating manager.** It synchronizes data when a transaction is accessing data items stored in different partitions (i.e., it is violating the perfect partitioning principle). Moreover, it also puts servers joining a new partition up to date.

Each replication cluster provides transactional support and, roughly speaking, behaves as a classical distributed database able to support a traditional OLTP application. However, any server belonging to a replication cluster keeps all their data in main memory until the meta-data cluster orders it to perform a persistent storage action.

From the application point of view, the transaction is first sent to the meta-data cluster (marked with a red arrow in Figure 1) to know to what partition it has to be forwarded and then it exchanges all data directly to the partition it has been pointed out (marked with a blue line in Figure 1).

Recall that in order to provide both high scalability and transactional support, our architecture implements (1) persistent storage inherited from recent cloud based storage repositories [26], and (2) transactional support provided by traditional relational distributed databases.

### III. Replication Techniques

In general, cloud based databases have put aside classic techniques concerning replication. Data is partitioned and not replicated in all replicas. Instead, data is replicated up to a given \( K \) level [11], [12], [17]: i.e., there exist up to \( K \) physical copies of a certain partition.
As introduced in the previous section, the replication protocol that manages each partition in the proposed system is determined by the replication manager according to the global partition latency and current workload characteristics.

For instance, a partition whose items are frequently updated might benefit from an update everywhere replication solution based on total order broadcast, such as active replication or certification based replication [27]. In this kind of protocols, update operations can be performed by any replica (which will be responsible for propagating the corresponding changes to the rest of replicas in order to commit each transaction), thus avoiding the bottleneck that would be caused if all updates had to be handled by the same replica, as in primary-backup approaches.

However, update everywhere protocols suffer from a serious scalability limitation, as the cost of propagating updates in total order to all replicas is greatly affected by the number of involved replicas. This problem could be alleviated if some of the replicas involved in the update everywhere replication protocol acted as primaries for other backup replicas, which would asynchronously receive updates from their respective primaries. At the same time, backup replicas could act as primaries for other replicas, thus creating a hierarchy where updates are propagated in an epidemic way. In other words, only $n$ out of $m$ replicas storing a partition (probably with $n << m$) would participate in the update everywhere replication protocol, whereas the rest would form a hierarchy of backups.

Therefore, each partition can be seen as a tree of consistent versions, where the replicas of the highest level of the hierarchy have the most updated values for data items, whereas lower levels have older versions of the data items, as shown in Figure 2 we have that $V \geq V' \geq V''$. Depending on the consistency level demanded for a transaction, its operations will be forwarded to replicas containing newer or older versions. Versions can be associated with timestamps, so that transactions can execute queries stating the level of freshness of the returned data.

![Figure 2. Hierarchy of partition replicas.](image)

Moreover, replicas could be upgraded or downgraded in the hierarchy according to current system requirements. This configuration also facilitates the process of adding new replicas to the system and reduces its impact on the system’s overall performance, since a new replica could start in the lowest level of the hierarchy (after performing an initial state transferring) and be progressively upgraded depending on the needs of the system.

IV. DISCUSSION

Apart from dealing with real-time user operations (also referred to as online transaction processing, OLTP), typical data management systems also have to support periodical online analytical processing (OLAP) operations that involve massive queries, which serve as a basis for data mining and decision making tools.

Traditionally, OLTP and OLAP tasks have been managed separately: OLTP workloads are usually handled by relational database management systems (RDBMS); whereas OLAP tasks are executed in data warehouses that periodically collect data from the RDBMS and other sources by making use of ETL (extract, transform and load) procedures. As pointed out in [28], this system-level separation results in several inherent limitations such as lack of data freshness in OLAP, redundancy of data storage, high startup investment and high maintenance costs.

Cloud infrastructures, thanks to their high availability and scalability properties, provide the possibility of having integrated systems with both OLTP and OLAP capabilities. For instance, an elastic power-aware data-intensive cloud platform for supporting both OLTP and OLAP within the same storage and processing system, named epIC, is proposed in [28]. Due to the distinct characteristics of OLAP and OLTP workload, the query processing engine of epIC is loosely coupled with the underlying storage system and adopts different strategies to process queries from the two different workloads: OLAP queries are processed via parallel sequential scans, while OLTP queries are handled by indexing and localized query optimization.

The architecture proposed in this paper can also serve to support both OLTP and OLAP in an integrated system. The OLTP processes can be offered different service level agreements according to their needs. For instance, those applications that require strong consistency should access the nodes of the highest level of the replication hierarchy, so as to ensure that any access returns the last updated value. In contrast, there may be other applications that tolerate weaker consistency guarantees; hence, their transactions could involve nodes of lower levels of the hierarchy. For example, a very common form of eventual consistency is session consistency [8], in which the storage system is accessed in the context of sessions: as long as the session exists, the system guarantees read-your-writes consistency, i.e., after a client updates an item, it always accesses that updated value and never sees an older value. In order to
ensure this, queries belonging to a certain session should be performed at nodes that have already executed all the previous updates of that session.

On the other hand, OLAP queries should be directed to the lowest levels of the replication hierarchy, in order not to interfere with OLTP workload. Thus, OLAP queries would retrieve data with a certain degree of outdatedness; however, this drawback is usually admissible in most scenarios, given that OLAP is focused on analyzing patterns and trends of massive amounts of information. Moreover, if the OLAP tasks require a certain degree of data freshness, queries could be tagged with a timestamp so that the system would return data with a minimum freshness according to that timestamp.

V. CONCLUSIONS

We have presented a distributed storage integral architecture, dynamic enough, to efficiently perform both online transaction processing and online analytical processing without sacrificing system throughput nor data freshness.

Moreover, thanks to its lower layer cloud inspired scheme it provides high scalability and availability offering different levels of consistency. We also have proposed a new replication technique, based on epidemic updates, which is able to provide different consistency levels according to each application demands.

ACKNOWLEDGMENT

This work has been supported by the Spanish Government under research grant TIN2009-14460-C03-02.

REFERENCES