A Taxonomy of Partitioned Replicated Cloud-based Database Systems*

Divy Agrawal
University of California Santa Barbara

Amr El Abbadi
University of California Santa Barbara

Kenneth Salem
University of Waterloo

Abstract

The advent of the cloud computing paradigm has given rise to many innovative and novel proposals for managing large-scale, fault-tolerant and highly available data management systems. This paper proposes a taxonomy of large scale partitioned replicated transactional databases with the goal of providing a principled understanding of the growing space of scalable and highly available database systems. The taxonomy is based on the relationship between transaction management and replica management. We illustrate specific instances of the taxonomy using several recent partitioned replicated database systems.

1 Introduction

The advent of the cloud computing paradigm has given rise to many innovative and novel proposals for managing large-scale, fault-tolerant and highly available processing and data management systems. Cloud computing is premised on the availability of large data centers with thousands of processing devices and servers, which are connected by a network forming a large distributed system. To meet increasing demands for data storage and processing power, cloud services are scaled out across increasing numbers of servers. To ensure high availability in the face of server and network failures, cloud data management systems typically replicate data across servers. Furthermore, the need for fault-tolerance in the face of catastrophic failures has led to replication of data and services across data centers. Such geo-replication holds the promise of ensuring continuous global access.

This paper considers the problem of providing scalability and high availability for *transactional database systems*. These are database systems that support the on-going operations of many organizations and services, tracking sales, accounts, inventories, users, and a broad spectrum of similar entities and activities. A key feature of these systems is support for transactional access to the underlying database. Transactions simplify the development and operation of complex applications by hiding the effects of concurrency and failures.

To provide scalability and high availability, databases are typically partitioned, replicated, or both. Partitioning splits a database across multiple servers, allowing the database system to scale out by distributing load across

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 38(1), March 2015, pages 4-9.

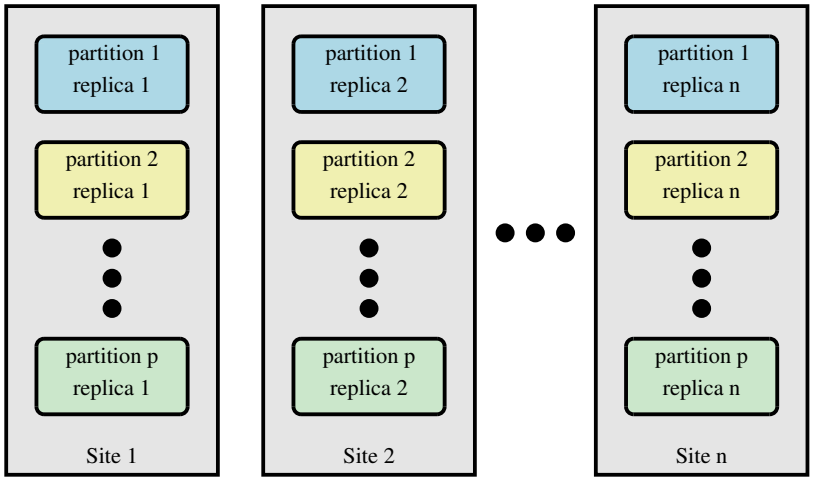

Figure 1: A Replicated Partitioned Database

more and more servers. Replication stores multiple copies of a partition on different servers. If the replicas are distributed across multiple data centers, they can provide fault-tolerance even in the presence of catastrophic failures, such as earthquakes or hurricanes. Such replication is often referred to as geo-replication.

Many modern database systems operate over partitioned, replicated databases, and they use a variety of different techniques to provide transactional guarantees. Our objective in this paper is to present a simple taxonomy of such systems, which focuses on *how* they implement transactions. By doing so, we hope to provide a framework for understanding the growing space of scalable and highly available database systems. Of course, these database systems differ from each other in many ways. For example, they may support different kinds of database schemas and provide different languages for expressing queries and updates. They also target a variety of different settings, e.g., some systems geo-replicate so as to survive catastrophes, while others are designed to scale out within a single machine room or data center. We will ignore these distinctions as much as possible, thus allowing the development of a simple taxonomy that focuses on transaction mechanisms.

2 Replicated Partitioned Databases

We begin with an abstract model of a distributed, replicated transactional database system. This model will provide a common context for the subsequent presentation of the taxonomy. We assume that the system stores a large database. To allow the system to scale up, the database is divided into p non-overlapping *partitions*, so that each partition can be managed by a separate server. Partitioning is accomplished in different ways in different systems, but we will not be concerned here with the way that the partitions are defined.

For the sake of availability, each partition of the database is replicated n times. As shown in Figure 1, each complete copy of the database is referred to as a *site*. Because there are n complete copies of the database, there are n sites. In a geo-replication setting, these sites may be housed in geographically distributed data centers.

Applications perform queries and updates against the database. A transaction is an application-defined group of database operations (queries and updates) that can be executed as an indivisible atomic unit. The database servers that manage the database partition replicas work together as a distributed database management system to handle the application's transactions. The "gold standard" for the distributed database system is to provide a *one-copy serializability* [3] guarantee for application transactions. Such a guarantee says that transactions will behave as if they executed sequentially on a single (non-replicated) copy of the database. In practice, however, some of the systems that we will describe will fall short of this gold standard in one or more ways.

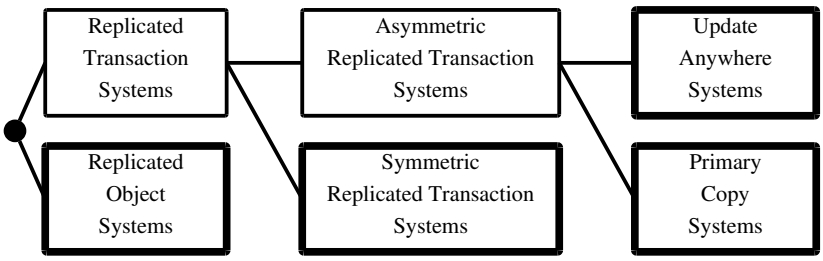

Figure 2: Taxonomy of Partitioned, Replicated Database Systems

3 Taxonomy

Database systems that manage replicated, distributed databases face a two-dimensional problem. First, even if the database were not replicated, it would be necessary to coordinate the database partitions to enforce transactional (ACID) guarantees. Many concurrency control, recovery, and commitment techniques exist to solve this *transaction management* problem. Second, the database system must somehow synchronize the database replicas so that replication can be hidden from the application, and so that the system can remain available even when some of the replicas are down. A variety of existing replica synchronization techniques can be used to solve this *replica management* problem.

Our taxonomy, which is shown in Figure 2, is based on the relationship between transaction management and replica management in these systems. Since there are well-known techniques for both transaction management and replica management, the challenge in designing a distributed replicated database system is how to combine techniques to arrive at an effective design that will address both problems. Thus, the relationship between transaction management and replica management is a natural basis for our taxonomy.

The top level of the taxonomy distinguishes between two general types of systems: *replicated object systems* and *replicated transaction systems*. In replicated object systems, replica management is used to make replicated database partitions look like non-replicated partitions. Transactions are then implemented over the logical, non-replicated partitions, using any of the well-known transaction management techniques. Thus, replicated object systems can be thought of as implementing transaction management on top of replica management.

Replicated transaction systems, in contrast, implement replica management on top of transaction management. In replicated transaction systems, transactions run over individual partition replicas, rather than logical partitions. That is, each *global transaction* is implemented using n separate *local transactions*, one at each site. Each site has a local transaction manager that ensures that its local transactions have ACID properties with respect to other local transactions at that site. (In contrast, replicated object systems have a single *global* transaction manager.) Each local transaction is responsible for applying the effects of its parent global transaction to the replicas at its own local site. We will refer to the local transactions as *children* of their (global) parent.

We further divide replicated transaction systems into two subclasses, *symmetric* and *asymmetric*. In symmetric replicated transaction systems, all of the local transactions of a given parent transaction are the same. Typically, they can run concurrently at the different sites. In contrast, in an asymmetric system, one local *master transaction* runs first at a single site. If that local transaction is able to commit, then the remaining local transactions are run at the other sites. We will refer to these remaining transactions as *update propagation transactions*. Typically, the update propagation transactions perform only the updates of the master transaction, and do not perform any reads.

Finally, we distinguish two types of asymmetric replicated transaction systems. In *primary copy* systems, all master transactions run at a single *primary* site. The remaining sites only run update propagation transactions. In *update anywhere* systems, master transactions may run at any site.

4 Examples

In this section, we briefly discuss several recent partitioned, replicated database systems, describing them in terms of the our taxonomy. We have included an example from each of the "leaf" categories of the taxonomy shown in Figure 2.

4.1 Replicated Object Systems

A prominent example of a replicated object system is Google's Spanner [5], which is designed to support geo-replication. Other systems that use the replicated object approach are the Multi-Data Center Consistency (MDCC) [7] system, which is also designed to allow geo-replicated data, and Granola [6], which is not.

Spanner stores keyed records, with related records grouped by the application into directories. A *tablet* in Spanner is a collection of directories or fragments of directories. Tablets are replicated. Thus, they correspond to partitions in our generic architecture (Figure 1) - we will refer to them here as partitions. Spanner uses Paxos [8] to synchronize the replicas of each partition across sites. Spanner uses a separate instance of Paxos, with a long-lived leader, for each partition.

To implement transactions, including transactions that involve multiple partitions, Spanner uses two-phase locking for concurrency control, and two-phase commit. The Paxos leader in each partition is responsible for participating in these protocols on behalf of that partition. It does so in much the same way that it would if its partition were not replicated, except that changes to the partition's state are replicated using Paxos instead of just being stored locally at the leader's server. The leaders serve as the link between the transaction protocols and Paxos, by ensuring that both database updates and changes in the state of the transaction are replicated.

The transaction protocol described here is a simplified version of the full protocol used by Spanner. The full protocol uses a system called TrueTime to associate timestamps with transactions, and ensures that transactions will be serialized in the order in which they occur in time. Spanner also uses TrueTime (and versioned objects) to support point-in-time read-only transactions, which "see" a snapshot of the database at a particular time. However, these features of the full Spanner protocol do not change its underlying nature, which is that of a replicated object system.

4.2 Symmetric Replicated Transaction Systems

An example of a symmetric replicated transaction approach is UCSB's replicated commit protocol [9]. Under the replicated commit protocol, each site includes a transaction manager capable of implementing transactions over the database replica local to that site. Each global transaction is implemented as a set of local child transactions, one at each site.

As is the case with Spanner, a global transaction's update operations are deferred until the client is ready to commit the transaction. To commit a transaction, the client chooses a *coordinating partition* for that transaction. It then contacts the replica of that partition at each site, and requests that the transaction commit locally at that site, providing a list of the transaction's updates.

At each site, the coordinating partition is responsible for determining whether that site is prepared to commit its local child of the global transaction. If so, the coordinating partition notifies the coordinators at other sites, and the client, that its site is prepared to commit the transaction. The global transaction will be committed if the local coordinators at a majority of the sites vote to commit it.

Under this protocol, it is possible that a transaction that commits globally may not commit at some sites, since only a majority of sites must agree on the commit decision. Thus, some transactions' updates may be missing at some sites. In order to ensure that it will observe all committed updates, a client that wishes to read data from a partition sends its read request to *all* replicas of that partition, waits for a majority of the

replicas to respond, and chooses the latest version that it receives from that majority. To ensure global one-copy serializability, each local replica acquires a (local) read lock when it receives such a request.

4.3 Primary Copy Systems: Cloud SQL Server

A recent example of a primary copy system is Microsoft's Cloud SQL Server [2], which is the database management system behind the SQL Azure [4] cloud relational database service. Cloud SQL Server supports database partitioning (defined by application-specified partitioning attributes) as well as replication. However, transactions are limited to a single partition unless they are willing to run at the read committed SQL isolation level. Cloud SQL server is not designed for geo-replication, so all of database replicas (the "sites" shown in Figure 1) are located in the same datacenter.

In Cloud SQL Server, one replica of each partition is designated as the primary replica. Clients direct all transactions to the primary replica, which runs them locally. When a transaction is ready to commit at the primary site, the transaction manager at the primary assigns it a commit sequence number. It then sends a commit request to each of the secondary replicas. Each secondary replica then starts a local transaction to apply the updates locally at that replica, and sends an acknowledgment to the primary when it has succeeded. Thus, Cloud SQL Server is an asymmetric replicated transaction system since each transaction runs first (to the commit point) at the primary site before being started at the secondaries. Once the primary has received commit acknowledgments from a majority of the secondary replicas, it commits the transaction at the primary and acknowledges the commit of the global transaction to the client.

4.4 Update Anywhere Systems: Megastore

In primary copy asymmetric systems like Cloud SQL Server, the master children of all global transactions run at the same site. In update anywhere asymmetric systems, the master children of different global transactions may run at different sites. Google's Megastore [1] is an example of an asymmetric update anywhere systems. In Megastore, database partitions, which are called *entity groups*, are replicated and geographically distributed.

In Megastore, a client can initiate a single-partition transaction at any replica of that partition - typically, the client will use a nearby replica. For each partition, Megastore manages a transaction log, which is replicated to all sites using Paxos. The client reads the partition log to determine the sequence number of the log's next free "slot", then runs the transaction locally at its chosen replica, deferring updates until it is ready to commit the transaction. When the client is ready to commit the transaction, its chosen replica attempts to write a transaction record to the replicated log, using Paxos, into the slot that it read before starting the transaction. If this succeeds, the transaction is committed. The initiating site applies the updates to its replica, and all other sites read the transaction's updates from the shared log and apply them to their replicas using local transactions. Thus, the replicated transaction log serves to coordinate the sites, ensuring the transactions commit in the same order everywhere.

5 Conclusion

In this paper we proposed a taxonomy for partitioned replicated database systems that reside in the cloud. Our focus was on databases that support both transactions and replication. We used several state of the art systems to illustrate specific instances of the taxonomy. The taxonomy is skewed, in the sense that there are several subcategories of Replicated Transaction Systems, but only one category of Replicated Object Systems. This taxonomy thus represents a first principled attempt to understand and classify the transaction mechanisms of many of the proposed state of the art systems. We hope this will lead to a better understanding of the trade-offs offered, as well as a potentially more elaborate and extensive taxonomy in the future.

References

[1] Jason Baker, Chris Bond, James Corbett, J.J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. Conf. on Innovative Database Research (CIDR)*, January 2011.

[2] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kallan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for cloud computing. In *Proc. IEEE Int'l Conf. on Data Engineering*, 2011.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 1021–1024, 2010.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, 2012.

[6] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Proc. USENIX Annual Technical Conf.*, June 2012.

[7] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proc. EuroSys*, pages 113–126, April 2013.

[8] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

[9] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.