CumuloNimbo: A Cloud Scalable Multi-tier SQL Database

Ricardo Jimenez-Peris*
Univ. Politécnica de Madrid

Marta Patiño-Martinez*
Univ. Politécnica de Madrid

Bettina Kemme
McGill Univ.

Ivan Brondino
Univ. Politécnica de Madrid

José Pereira
Univ. de Minho

Ricardo Vilaça
Univ. de Minho

Francisco Cruz
Univ. de Minho

Rui Oliveira
Univ. de Minho

Yousuf Ahmad
McGill Univ

Abstract

This article presents an overview of the CumuloNimbo platform. CumuloNimbo is a framework for multi-tier applications that provides scalable and fault-tolerant processing of OLTP workloads. The main novelty of CumuloNimbo is that it provides a standard SQL interface and full transactional support without resorting to sharding and no need to know the workload in advance. Scalability is achieved by distributing request execution and transaction control across many compute nodes while data is persisted in a distributed data store. In this paper we present an overview of the platform.

1 Introduction

With cloud infrastructure becoming increasingly popular, scalable data management platforms that are able to handle extremely high request rates and/or data sets have started to appear on the market. One development line achieves scalability through scale-out – using clusters of commodity computers and dynamically assigning as many of these servers to the application as are needed for the current load. While this has proven to be tremendously successful for batch analytical processing (through the map-reduce paradigm) and for applications that only require a simple key/value API, scaling out transactional applications has shown to be much more challenging. Business applications are often built on widely used software stacks (e.g. Java EE, Ruby on Rails, etc.), and scaling them out transparently, offering a full-fledged scalable Platform as a Service (PaaS) is much more complex due to the data and transaction management that must be spread across and between different layers in the software stack. For this reason, the most common current industrial solutions fall back to sharding where the data is split into many different partitions and transactions are expected to access only one partition. Some solutions support distributed transactions, but then often rely on standard distributed commit protocols to handle them, making these distributed transactions expensive. Thus, application developers are encouraged to avoid distributed transactions which might require rewriting existing applications. At the same time, finding proper

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work has been partially funded by the Spanish Research Council (MICCIN) under project BigDataPaaS (TIN2013-46883), and by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) cofunded by FSE & FEDER and CoherentPaaS and LeanBigData FP7 projects funded by European Commission under contracts FP7-611068, FP7-619606.

partitions that lead to few distributed transactions and an equal load on all servers is just the same challenging. All this requires expertise that many developers and system administrators do not have. Therefore, there is a huge need for a PaaS that allows developers to continue writing applications as they do today, with standard software stacks such as Java EE, relying on relational database technology providing standard SQL and full transactional properties without concerns for data storage scalability needs.

In the last years, several new proposals have appeared as alternatives to sharding [35, 9, 31]. All of them, however, have certain restrictions. For instance, [35] assumes transactions are written as stored procedures and all data items accessed by a transaction are known in advance, [9] has been designed for very specialized data structures, and [31] is optimized for high throughput at the cost of response time.

In this paper, we present CumuloNimbo, a new scalable fault-tolerant transactional platform specifically designed for OLTP workloads. CumuloNimbo provides SQL and transactional properties without resorting to sharding. This means that an existing application can be migrated to our platform without requiring any single change, and all transactional properties will be provided over the entire data set. Our current PaaS prototype follows the traditional multi-tier architecture that allows for the necessary separation of concerns between presentation, business logic and data management in business applications. At the highest level, our prototype supports one of the most common application development environments, Java EE, as development layer. Beneath it, a SQL layer takes care of executing SQL queries. Data management is scaled by leveraging current state of the art NoSQL data stores, more concretely, a key-value data store, HBase. The key-value data store scales by partitioning, but this partitioning is totally transparent to applications that do not see it neither get affected by it. Additionally, the NoSQL data store takes care of replicating persistent data for data availability at the storage level (file system) outside the transaction response time path. Transaction management is located between the SQL and the data store layer, providing holistic transaction management.

A challenge for a highly distributed system in which each subsystem resides on multiple nodes as ours is the cost of indirection. The more subsystems a request has to pass through, the longer its response time. We avoid long response times by collocating components of different layers when appropriate. Furthermore, we make calls to lower levels or remote components (e.g., for persisting changes to the storage layer) asynchronously whenever possible so that they are not included in the response time of the transactions.

In this paper, we give an overview of the overall architecture of CumuloNimbo, the individual components and how they work together. We outline how transaction processing is performed across all software stacks.

2 CumuloNimbo Architecture Overview

The architecture of the CumuloNimbo PaaS is depicted in Figure 1. The system consists of multiple tiers, or layers, each of them having a specific functionality. In order to achieve the desired throughput, each tier can be scaled independently by adding more servers. Nevertheless, instances of different tiers might be collocated to improve response time performance. The whole stack provides the same functionality as the traditional tiers found in multi-tier architectures. However, we have architected the multi-tier system in a very different way.

Application Server Layer The highest layer depicted in the figure, the application layer, consists of application servers (such as Java EE or Ruby on Rails) or any application programs connecting to the SQL query layer via standard interfaces (such as JDBC or ODBC). Transaction calls are simply redirected to the SQL query layer. For our current implementation, we have taken the open-source JBoss application server that we left unchanged. Our architecture allows for the dynamic instantiation of application server instances depending on the incoming load.

SQL query layer The query engine layer provides the standard relational interface (SQL) including the transaction commands (commit, abort, and begin). The query layer itself is responsible for planning, optimizing and executing queries. However, it does not perform transaction management nor is it responsible for data storage.

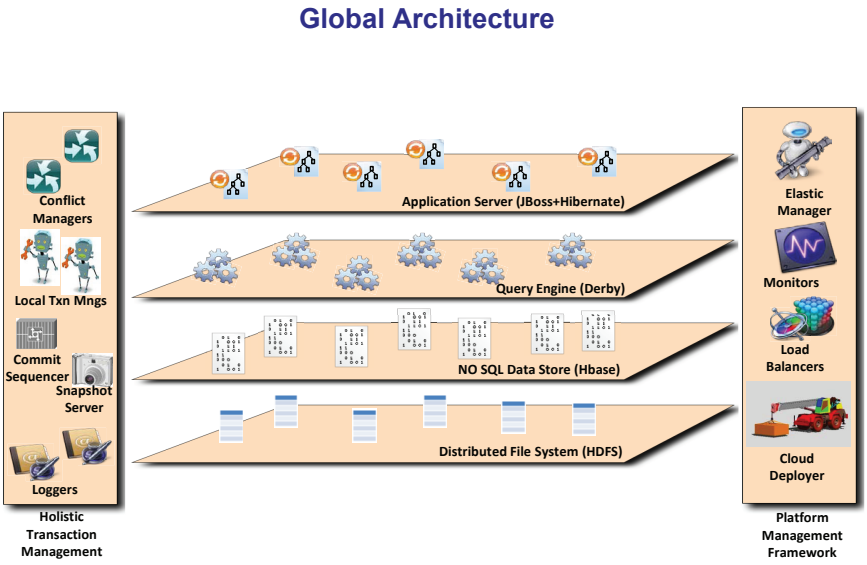

Figure 1: CumuloNimbo Architecture

Our implementation uses the Apache Derby query engine (i.e., compiler and query plan runtime). Substantial parts of Derby have been rewritten to replace the original storage system based on local files with the NoSQL data storage system. This includes the mapping of the relational schema into an appropriate schema of the NoSQL data store, replacing the standard scan operators with scan operators that interact with the NoSQL data store, re-implementing indices, and distributing cost estimation over the SQL query layer and the NoSQL data store. Furthermore, the transactional management in Derby has been replaced by our transaction management.

NoSQL data store layer This layer uses an advanced data store that is designed for inherent scalability in terms of data size and processing requirements. As the currently available NoSQL data stores differ greatly in terms of functionality, the choice of NoSQL data store has a considerable influence of the task distribution between SQL query layer and data store layer. Our current system is based on HBase, an elastic key-value store. The HBase API is provided in form of a library to the application, which in our case is an instance of the SQL query layer, i.e., a Derby instance. In the following, we refer to this library as HBase client as it resides on the client.

HBase offers a tuple interface, i.e., it has the concept of tables (HTable) with a primary key and a set of column families. Having column families allows separate storage and tuple maintenance of each family. HBase already provides advanced query facilities such as simple scans that return all tuples of a table on an iteration basis, and predicate filters on individual columns of a table. Our Derby extension takes advantage of the HBase features to push parts of the query execution to the NoSQL data store layer in order to reduce the amount of tuples that have to be transferred over the network, and to implement secondary indices as tables in HBase.

HBase splits tables horizontally into regions, each region consisting of rows with continuous primary key values. Regions are then distributed onto region servers. HBase also maintains internally an in-memory cache for fast execution. It sits on top of a parallel-distributed file system, HDFS [4], that provides persistent and fault-tolerant data storage. Our CumuloNimbo system can take the default HDFS implementation as all additional semantics needed is embedded in HBase.

To provide durability, HBase persists each incoming update to its write-ahead log (also stored in HDFS). When an HBase server fails, its regions are re-assigned to other servers. These servers run a recovery procedure that replays lost updates from the log, bringing these regions back to a consistent state. For performance reasons,

HBase allows the deactivation of a synchronous flush of the write-ahead log to HDFS, so that the server may return from an update operation before the update is persisted to HDFS. In this case, the recovery of HBase cannot guarantee that all updates executed before a server failure will be recovered.

We have extended HBase to help providing transactional guarantees both in terms of isolation and durability while at the same time offering low response times for demanding OLTP workloads.

Generally, it is possible to use other scalable data stores at this layer. The choice might affect the binding to the SQL query layer but it should not have any influence on the other layers.

Configuration In principle, each layer can be scaled individually, that is, one can instantiate as many application server instances, query engine instances and HBase region servers as needed. However, depending on the configuration and execution costs of the components some components can be collocated on the same machine. In particular, it might make sense to put an instance of the application layer and an instance of the SQL query engine (together with its HBase client) on the same node. Then, communication between these instances remains local. What is more, the SQL query engine in this case is run in embedded mode avoiding all remote invocations to it. If large data sets have to be transferred, this collocation avoids one indirection that can be costly in terms of communication overhead.

In contrast, HBase servers are always independent of these two upper layers, as this is where the data resides. Given that OLTP workloads have stringent response time requirements, one possible configuration is to have as many region servers as necessary to hold all data, or at least all hot data, in main memory.

Transaction management Transactions are handled in a holistic manner providing full ACID properties across the entire stack. In particular, we made the design decision to invoke our advanced transaction management from the NoSQL data store, therefore enabling transactions for both SQL and NoSQL applications of the same data in case parts of the application do not require the upper layers of the stack. Holistic transaction management relies on a set of specifically designed transaction subsystems each of them providing a specific transactional property. The fundamental idea is that scaling out each of the transactional properties individually is not that hard. The difficulty lies in ensuring that they work together in the correct way.

Platform management There is an additional component taking care of platform management tasks such as deployment, monitoring, dynamic load balancing and elasticity. Each instance of each tier has a monitor collecting data about resource usage and performance metrics that are reported to a central monitor. This central monitor provides aggregated statistics to the elasticity manager. The elasticity manager examines imbalances across instances of each tier and reconfigures the tier to dynamically balance the load. If the load is balanced, but the upper resource usage threshold is reached, then a new instance is provisioned and allocated to that tier to diminish the average usage across instances of the tier. If the load is low enough to be satisfied with a smaller number of instances in a tier, some instances of the tier transfer their load to the other instances and are then decommissioned. The deployer enables the configuration of an application and its initial deployment. After the initial deployment the elasticity management takes care of dynamically reconfiguring the system to adjust the resources to the incoming load. Figures 2 and 3 show such an adjustment. Figure 2 shows CPU usage and response time of one query engine. As response time goes beyond the agreed level for a predefined time, the system deploys an additional query engine. Figure 3 now shows that the CPU load on both query engines is significant lower and the latency well beyond the service level agreement.

3 Transactional Processing

Efficient and scalable transaction management is the corner stone of our architecture. This section provides a high-level overview of transaction management across the entire stack. Our transactional protocol is based on snapshot isolation. With snapshot isolation, each transaction reads the data from a commit snapshot that represents all committed values at the start time of a transaction. Conflicts are only detected on updates. Whenever

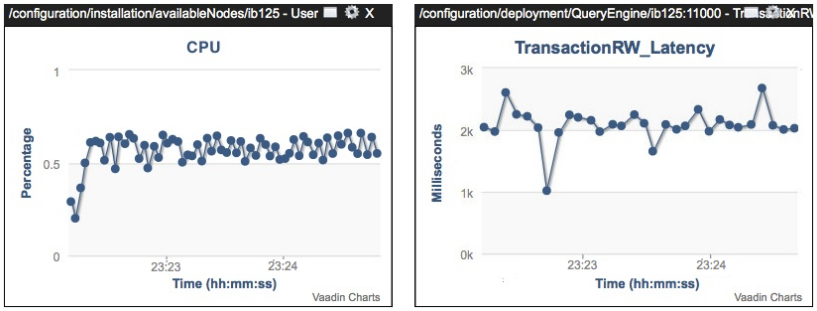

Figure 2: Query engine and latency monitoring

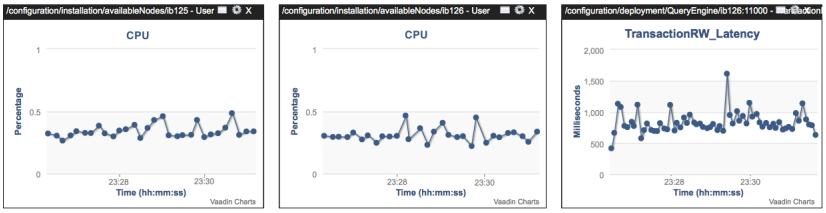

Figure 3: Query engine provisioning

two transactions are concurrent (neither commits before the other starts), and they want to write a common data item, one of them has to abort. Snapshot isolation has been very popular for distributed transaction management as it avoids conflicts between predicate reads and writes.

Transaction Execution We control transaction execution within the HBase layer. With this, transactions can be used directly on HBase if an application does not need the services of the application server and query layers. For that, we extended the HBase client to provide a transaction interface (start, commit/abort). Each read and write request is associated with a transaction. The actual transaction control is embedded in a local transaction manager module that is collocated with the HBase client. Upon transaction start the local transaction manager provides the start timestamp, and the transaction enters the *executing* stage. Write operations create new versions that are only visible to the local transaction that creates them. In fact, we use a deferred-update approach. Upon a write request the HBase Client caches the write locally as private version in a transaction specific cache. When the query engine submits a commit request, the local transaction manager coordinates a conflict check and aborts the transaction in case of conflict with a concurrent transaction that already committed. Abort simply requires discarding the transaction specific cache. If there are no conflicts the transaction obtains its commit timestamp, the updates are persisted to a log and a confirmation is returned to the query engine. At this time point the transaction is considered *committed*. The HBase client then asynchronously sends the versions created by the transaction to the various HBase region servers. The HBase region servers *install* the updates in their local memory store but do not immediately *persist* them to HDFS. This only happens asynchronously in a background process. This is ok, as the updates are already persisted in a log before commit. In summary, an update transaction is in one of the following states: *executing*, *aborted*, *committed*, *installed*, *persisted*.

In order to read the proper versions according to snapshot isolation, we tag versions with commit timestamps and whenever a transaction T requests a data item, we return the version with the largest commit timestamp that is smaller or equal to T's start timestamp, or, if T has already updated the data item, with the version in T's transaction specific cache. As we write versions to the HBase region servers only after commit, start timestamps

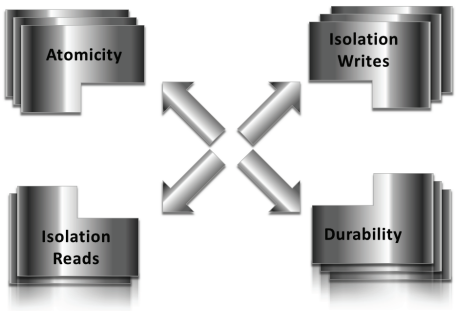

Figure 4: Scaling and distributing transactional tasks

are chosen so that we can ensure that all data that needs to be read is actually installed.

A crucial aspect of our approach is that we take the expensive flushing of changes to the data store out of the critical path of the transaction while still guaranteeing that transactions see a consistent snapshot of the data.

Distributing transaction tasks There are as many local transaction managers as there are HBase clients (i.e., as there are query engine instances), each managing a subset of all transactions in the system. Thus, a local transaction manager cannot independently create timestamps or perform conflict checks. Instead, these tasks are performed by highly specialized components each of them being scaled and distributed independently (see Figure 4): loggers, timestamp managers, conflict managers. These components interact with the local transaction managers in a highly optimized way to keep message volume, message delay and the tasks to be performed by any individual instance as small as possible.

Failures Being a highly distributed system, many different kinds of failures can occur: a query engine together with the HBase client and the local transaction manager can fail, an HBase region server can fail, or any of the transaction components can fail. We have implemented failure management in a recovery manager which is described in detail in [2]. When an HBase client fails, the recovery manager's recovery procedure replays from the transaction manager's log any of the client's transactions that were committed but not yet completely installed in all participating servers. This is necessary because our client holds the updates until commit time, so we may lose the committed updates if a client failure occurs before or during the install phase. Note that updates that were not yet committed are permanently lost when the client fails. These transactions are considered aborted. This is not a problem because only the durability of committed transactions must be guaranteed.

When an HBase server fails, first HBase assigns the server's region to another server and executes its own recovery procedure. Then CumuloNimbo's recovery procedure replays all committed transactions that the failed server participated in that were installed by the client but not persisted to HDFS before the server failure occurred. This is necessary because updates are persisted asynchronously to HDFS, so we may lose updates if a server failure occurs before or during the persist phase. Note that for transactions that were fully persisted before the crash, we can rely on HBase's own recovery procedure to guarantee the durability of the transaction.

During normal processing, HBase clients and HBase servers inform the recovery manager on a regular basis of what they have installed and persisted, respectively. This information is sent periodically to reduce the overhead and avoid that the recovery manager becomes a bottleneck. It serves as a lower bound on which transactions have to be replayed in case of recovery.

4 Query engine

While the query engine layer has no tasks in regard to transaction management, it is the component that enables the execution of arbitrary SQL queries on top of a key-value data with limited query capabilities. Now, we shortly outline how we couple Derby with HBase.

We have taken some of the components of Derby and adjusted and extended them to our needs: SQL compiler and optimizer, and generic relational operator implementations. The major modifications that we performed were (1) a reimplementation of major data definition constructs in order to provide an automatic mapping from the relational data model to the one of HBase; (2) the implementation of sequential and index scan operators to use HBase; (3) implementation of secondary indexes within HBase. Finally, we deactivated Derby's transaction management and integrated it with the our transaction management. This last part was fairly easy as the only task of the query engine is to forward any start and commit requests to the HBase client. In the following we briefly describe these modifications. More details can be found in [37].

Data Model Mapping The relational data model is much richer than the key-value data model and a non-trivial mapping was required. HBase offers a table interface (HTable), where a table row consists of a primary key and a set of column families. Data is maintained in lexicographic order by primary key. A table is split into regions of contiguous primary keys which are assigned to region servers.

We have adopted a simple mapping. Each relational table is mapped to an HTable (base table). The primary key of the relational table becomes the (only) key of the HTable. The rest of the relational columns are mapped as a single column family. We believe that this approach is plausible for OLTP workloads[1]. Each secondary index is mapped as an additional HTable that translates the secondary key into the primary key. For unique indexes the row has a single column with its value being the key of the matching indexed row in the primary key table. For non-unique indexes there is one column per matching indexed row with the name of the column being the matching rows key. Ideally, the data in the secondary index is sorted by the key value as we can then restrict the rows for a given scan by specifying start and stop keys. The problem is that HBase stores row keys as plain byte arrays which do not preserve the natural order of the data type of the key (e.g., char, date, integer, etc.). Therefore, we developed a specific encoding from key values into plain bytes that preserves the lexicographic byte order of the original data type. For composite indexes, we encode multiple indexed columns in the same byte array through concatenation using pre-defined separators.

Query Execution Since the interaction between the query engine and the key-value data store is through the network it is important to keep network traffic and network latency low. Simple database scans of the entire table use the standard HBase scan. Transfer of tuples is done in blocks by the HBase client to avoid individual fetches. When a query contains a range or equality predicate the query optimizer checks whether a secondary index exists on that column. If this is the case, we first access the secondary index table to retrieve all primary keys of all matching tuples (specifying the start and stop keys for the range). Then, we retrieve all the rows of the table with a single query specifying the matching primary keys. When there is no index available that would allow retrieving exactly the matching tuples, we could use an index that provides us with a superset. However, this would mean that we have to retrieve tuples from the base table and check additional conditions in Derby, resulting in unnecessary network traffic. Therefore, in this case, we restrain from using the secondary indexes and perform a scan on the base table using the filter capacity of HBase (i.e., concatenating several single column filters into a conjunctive normal form that is equivalent to the original SQL predicate query). The join operators are completely executed in the query engine as HBase does not offer join operators.

[1] For OLAP workloads that we have not targeted yet, it might be interesting to split the relational columns in different column families to reduce the overhead.

5 Related Work

Several approaches have been proposed to implement ACID transactions in the cloud. Most of the approaches are based on sharding (i.e., partitioning) [8, 35, 7, 40, 17, 14, 16, 33, 13] and some of them even limit transactions to access a single data partition.

CloudTPS [40] supports scalable transactions by distributing the transaction management among a set of local transaction managers and partitioning the data. The list of keys being accessed by a transaction must be provided at the beginning of the transaction. Microsoft SQL Azure [8] supports SQL queries but requires manual data partitioning and transactions are not allowed to access more than one partition. ElasTraS [17] provides an elastic key-value data store where transactions execute within a single data partition (static partitions). Dynamic partitioning is also supported using *minitransactions* [1] which piggyback the whole transaction execution in the first phase of the two-phase commit protocol executed among data partitions. Google Megastore [7] allows programmers to build static entity groups of items (sharding). Megastore provides an SQL-like interface and is built on top of Bigtable [11] providing transactional properties on an entity group. Two-phase commit is used, although not recommended, for cross entity group transactions.

Calvin [35] is a fault-tolerant transactional system that provides ACID transactions across multiple data partitions. Serializability is achieved by resorting to a deterministic locking mechanism that orders transactions before they are executed based on the data they are going to access (read and write sets). Relational Cloud [14] provides automatic workload partitioning [15] and distributed SQL transactions. G-Store [16] provides transactions over partitioned groups of keys on top of a key-value store. Groups are dynamically created. Granola [13] uses a novel timestamp-based coordination mechanism to order distributed transaction but is limited to one-round transactions which do not allow for interaction with the client.

After the limitations presented in [10] several approaches were suggested for providing transaction isolation on top of a scalable storage layer [18]. For instance, Deuteronomy supports ACID transactions over arbitrary data [23]. It decouples transaction processing from data storage [26, 25]. Data stores can be anywhere. The system caches data residing on secondary storage and avoids data transfer between the transaction management component and the data store whenever possible. In fact, similar to our approach described in [2], the data store is only updated after transaction commit. With this it exhibits a performance similar to the one of main memory systems. Deuteronomy implements a multi-version concurrency control mechanism that avoids blocking. The design exploits modern multicore hardware. Google Percolator [31] provides transactions with snapshot isolation on top of Bigtable. Percolator provides high throughput and is intended to be used for batch processing. Omid follows a similar approach providing transactions on top of HBase [19]. In [30, 29], serializable executions are guaranteed in a partial replicated multi-version system. Update transactions, however, require a coordination protocol similar to 2-Phase Commit. In [36], the authors use a novel form of dynamic total order multicast and rely on some form of determinism to guarantee that transactions on replicated data are executed in the proper order. cStore [38] implements snapshot isolation on top of a distributed data store. At commit time write locks are requested for all the data items updated by a transaction. If all the locks are granted, the transaction will commit. If some locks are not granted, the granted locks are kept, the read phase of the transaction is re-executed and those locks that were not granted are requested again. More recently, Google has proposed F1 [33], a fault-tolerant SQL database on top of MySQL. As reported by the authors the latency of writes is higher than a traditional SQL database (e.g., MySQL) but they are able to scale the throughput of batch workloads. F1 requires sharding and the business applications need some knowledge about it. Spanner [12] is a distributed and replicated database that shards data across datacenters. It provides ACID transactions based on snapshot isolation and a SQL-based interface. Synchronized timestamps are used to implement snapshot isolation. Two-phase commit is used when data from different shards is accessed from a transaction. MDCC [21] also replicates data across multiple data centers providing read committed transactions by default. Replicated commit [27] provides ACID transactional guarantees for multi-datacenter databases using two-phase commit multiple times in different datacenters and Paxos [22] to reach consensus among datacenters. Walter [34] implements transactions

for geo-replicated systems for key-value data store. Transactions use 2-phase commit and data is propagated across sites asynchronously. MaaT [28] proposes a solution for coordination of distributed transactions in the cloud eliminating the need for locking even for atomic commitment, showing that their approach provides many practical advantages over lock-based methods (2PL) in the cloud context.

While strong consistency is sufficient to maintain correctness it is not necessary for all applications and may sacrifice potential scalability. This has lead to a whole range of solutions that offer weaker consistency levels, without requiring coordination [6, 24, 3, 5, 20]. Moreover, in [32] the authors developed a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution.

In [39] the authors studied the scalability bottlenecks in seven popular concurrency control algorithms for many-core CPUs. The results shows that all existing algorithms fail to scale in a virtual environment of 1000 cores. However the analysis is restricted to a single many-core CPU machine.

In contrast, CumuloNimbo provides a transparent fault tolerant multi-tier platform with high throughput and low latencies without partitioning data or requiring a coordination protocol. CumuloNimbo is highly scalable by separation of concern and scaling each component individually.

6 Conclusions

This paper presents a new transactional cloud platform, CumuloNimbo. This system provides full ACID properties, a standard SQL interface, smooth integration with standard application server technology, and coherence across all tiers. The system provides full transparency, syntactically and semantically. CumuloNimbo is highly distributed and each component can be scaled independently.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[2] M. Y. Ahmad, B. Kemme, I. Brondino, M. Patiño-Martínez, and R. Jiménez-Peris. Transactional failure recovery for a distributed key-value store. In *ACM/IFIP/USENIX Int. Middleware Conference*, pages 267–286, 2013.

[3] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *EuroSys*, 2013.

[4] Apache. HDFS. http://hadoop.apache.org.

[5] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, 2013.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *ACM Int. Conference on Management of Data (SIGMOD)*, pages 761–772, 2013.

[7] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data System Research (CIDR)*, pages 223–234, 2011.

[8] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kallan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL Server for cloud computing. In *IEEE Int. Conference on Data Engineering (ICDE)*, pages 1255–1263, 2011.

[9] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[10] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *ACM Int. Conference on Management of Data (SIGMOD)*, pages 251–264, 2008.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 15–15, 2006.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[13] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 223–235, 2012.

[14] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *Conference on Innovative Data Systems Research (CIDR)*, pages 235–240, 2011.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[16] S. Das, D. Agrawal, and A. El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *ACM Symposium on Cloud Computing (SoCC)*, pages 163–174, 2010.

[17] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)*, pages 5:1–5:45, 2013.

[18] A. Dey, A. Fekete, and U. Röhm. Scalable transactions across heterogeneous NoSQL key-value data stores. *PVLDB*, 6(12):1434–1439, 2013.

[19] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *IEEE Int. Conference on Data Engineering (ICDE)*, pages 676–687, 2014.

[20] W. Golab, M. R. Rahman, A. A. Young, K. Keeton, J. J. Wylie, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ACM Symposium on Cloud Computing (SoCC)*, pages 28:1–28:2, 2013.

[21] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *EuroSys*, pages 113–126, 2013.

[22] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[23] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*, 2015.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symp. on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[25] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *PVLDB*, 2(1):265–276, 2009.

[26] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *Conference on Innovative Data System Research (CIDR)*, 2009.

[27] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9):661–672, 2013.

[28] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB*, 7(5):329–340, 2014.

[29] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *ACM/IFIP/USENIX Int. Middleware Conference*, pages 456–475, 2012.

[30] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *2012 IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 455–465, 2012.

[31] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2010.

[32] A. F. Peter Bailis, M. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3), 2015.

[33] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, and P. Tong. F1: The fault-tolerant distributed rdbms supporting google's ad business. In *ACM Int. Conference on Management of Data (SIGMOD)*, pages 777–778, 2012.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 385–400, 2011.

[35] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[36] P. Unterbrunner, G. Alonso, and D. Kossmann. High availability, elasticity, and strong consistency for massively parallel scans over relational data. *VLDB J.*, 23(4):627–652, 2014.

[37] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira. An effective scalable SQL engine for nosql databases. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 155–168, 2013.

[38] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010.

[39] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8:209–220, 2014.

[40] W. Zhou, G. Pierre, and C.-H. Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing (TSC)*, 5(4):525–539, Jan. 2012.