CumuloNimbo: Parallel-Distributed Transactional Processing

Ricardo Jiménez-Peris[1a], Marta Patiño-Martínez [a], Iván Brondino [a], Bettina Kemme[b], José Pereira[c], Rui Oliveira[c], Ricardo Vilaça[c], Francisco Cruz[c], Yousuf Ahmad[b]

Universidad Politecnica de Madrid[a], McGill Univ.[b], Univ. Minho[c]

1. Motivation

This paper describes a new cloud platform, CumuloNimbo, that envisions ultra-scalable transactional processing for multi-tier applications with the goal to process a million update transactions per second while providing the same level of consistency and transparency as traditional relational database systems. Most of the current approaches to attain scalability for transactional processing in the cloud resort to sharding. Sharding is a technique in which the database is split into many different partitions (e.g. thousands) that work as separate databases sharing the original schema of the database. Sharding is technically simple but neither syntactically nor semantically transparent. Syntactic transparency is lost because applications have to be rewritten as individual transactions are only allowed to access one of the partitions. Semantic transparency is lost, because the ACID properties provided previously by transactions over arbitrary data sets are lost. Alternatives to sharding have been proposed recently [BernRWY11, PengD10], but they are solutions for specialized data structures [BernRWY11] or are not designed for online systems that require fast response times [PengD10].

In this paper we present a new cloud platform whose goal is to support traditional relational online transaction processing applications with standard ACID demands, while at the same time is able to scale as much as sharded platforms. It provides the same programming environment in form of a standard application server as traditional multi-tier architectures, achieving full syntactic and semantic transparency for applications.

2. Global Architecture

The architecture of the system is depicted in Figure 1. The system consists of multiple tiers, each of them having a specific functionality. In order to achieve the desired throughput, each tier can be scaled by adding more servers. In order to achieve the required quality of service in terms of maximum response times, the goal is to be able to serve as many requests as possible by only the most upper layers. The main challenges are to maintain transactional properties across all layers and to make transactional tasks, such as concurrency control and persistence, scalable.

The application server instances are located at the top layer. Currently, the system focuses on Java EE, but in principle any application server technology could be integrated such as .NET. In our current prototype we use the JBoss application server together with the Hibernate object/relational mapping. JBoss's local transaction processing has been inactivated and replaced by our Cumulo transaction manager. The number of application servers will depend on the CPU consumption of the application and how many concurrent requests a single application server can handle.

The second tier is constituted by a distributed object cache. The cache is shared across all application server instances. The cache can consist of as many nodes as necessary to keep objects in the main memory, even the whole database, if necessary. Hibernate, when being requested for an object by the application will first look up the object in the distributed cache, and only if it is not available, request it from the database tier. The cache also supports query caching.

[1] R. Jiménez-Peris, M. Patiño-Martínez. System and method for highly scalable decentralized and low contention transactional processing. Patent filed at USPTO Application #61/561,508.

The third tier is the SQL query engine layer. This layer is accessed if (a) an individual object is requested but the object is not yet in the object cache, (b) an SQL query is submitted that cannot be answered by the cache, (c) updates have to be pushed down to the lower persistence layers at commit time. The number of query engines depends on the number of complex queries that are executed concurrently (and should be handled by different query engines), and generally on how many concurrent requests an individual query engine can handle. The query engine contains the traditional query planner, optimizer and executor components, but all transactional management is deactivated.

The fourth tier is constituted by the no-SQL data store. We currently rely on HBase as no-SQL data store. It provides us with an elastic key-value store that offers a tuple interface to the DB layer. It also maintains internally a block cache for fast execution. It sits on top of the parallel-distributed file system HDFS that constitutes the fifth tier and provides persistent storage to the data. Both data tables as well as index tables are materialized as HBase tables. The motivation for using HBase+HDFS was to scale storage and replicate at the storage level to have an elastic and fault-tolerant storage tier.

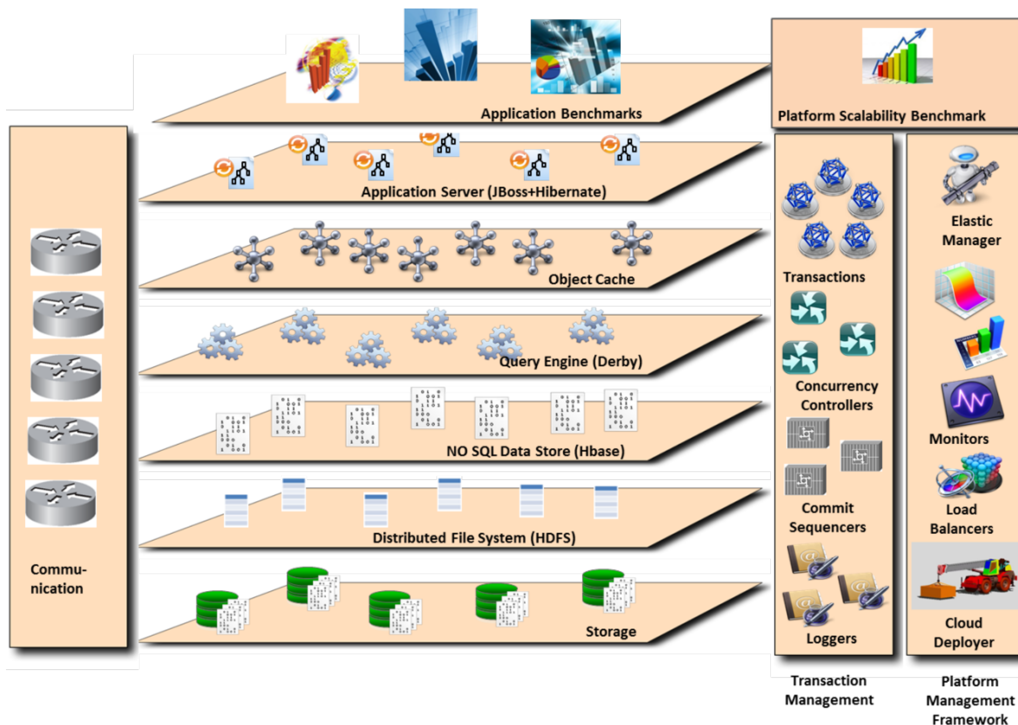

Figure 1: CumuloNimbo Architecture

Transaction management is done in a holistic manner and the different tiers collaborate to provide transactional properties. Transaction management furthermore relies on a set of subsystems that provide different functionality: commit sequencer, snapshot server, conflict managers, and loggers. This decomposition is crucial to attain scalability. Exactly how transaction management is performed across all tiers will be explained in detail in the next section.

There is an additional layer 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.

3. Ultra-Scalable Transactional Processing

Our transactional protocol is based on snapshot isolation [BerBGMNN95]. Snapshot isolation has great potential for scalability as it avoids read-write conflicts. Only if two concurrent update transactions access the same data item, a conflict occurs. Clearly, if the application has a few very hot data items that are updated by a large fraction of update transactions then no transactional system will be scalable. This means we assume that the conflict rate, i.e., the percentage of concurrent transactions that conflict, remains constant independently of the number of concurrent transactions. If this is true, our system will scale without any restrictions on the data items an individual transaction may access.

The potential of snapshot isolation for scalability has been shown by database replication approaches that pointed out the advantage of snapshot isolation over locking-based protocols (e.g, [KemJP10]). In these approaches, the limitation were not conflicts but the fact that updates had to be performed at all replicas, thus, the update load could not exceed the load any individual replica could handle. Thus, CumuloNimbo only includes replication for fault-tolerance at the persistence layer (HDFS). Otherwise, data is partitioned across nodes so that each node can handle the load submitted to the data it maintains.

Still, with transaction rates of hundreds of thousands of transactions, the concurrency control tasks needed for snapshot isolation can easily become a bottleneck. Let us have a short look at how snapshot isolation works. Each update operation on a data item X creates a new version of X. Each read operation of a transaction T on data item X reads the version of X that was created by a transaction T' such that T' was the last transaction to commit before T started and update X. That is, a transaction reads a committed snapshot of the database as of its start time. To implement these snapshot reads, each transaction receives a commit timestamp and a start timestamp determined by a commit counter. When a transaction commits the counter is incremented and the new value assigned as commit timestamp. At start time, the current counter value (reflecting the last committed transaction) is assigned as start timestamp. Data versions are labeled with the transactions that created them so that a transaction can determine the right version to be read. Snapshot isolation handles write conflicts with an abort. When a transaction T wants to update a data item that was updated by a concurrent transaction T', then T aborts. If not done in a smart way, generating start and commit timestamps and detecting write conflicts can become the bottleneck in the system. Therefore, our solution distributes these tasks into several correlated subsystems.

Conflicts are handled by a distributed conflict manager. Each manager takes care of a set of keys. When a transaction modifies a tuple it notifies the conflict manager in charge of its key. The conflict manager checks whether the key has been modified by a concurrent transaction and if so, it aborts the transaction. Otherwise, it keeps track that the key has been modified by the requesting transaction. In our system, we collocate the conflict manager of a set of keys with the cache node that is responsible to cache the objects with these keys, making the conflict check a local operation at the time the update is executed.

The core of the transactional processing is constituted by the commit sequencer and the snapshot server. The responsibilities are split between them. The commit sequencer provides commit timestamps to committing update transactions. The snapshot server provides sequences of snapshots (i.e., the start timestamps) that guarantee a consistent view of the database as mandated by snapshot isolation.

Our transaction manager, installed in each application server, receives the requests to start and commit transactions as well as intercepts the read and write operations. When a transaction is started, the transaction manager provides the transaction with the latest start timestamp it has received from the snapshot server. Then, all data operations are made on the associated snapshot identified by the start timestamp. As data versions are tagged with the transactions that created them, this can be done in a distributed fashion. We ensure this, whether the object resides already in the distributed cache or is retrieved from HBase via the DB layer. Each update operation returns as writeset the objects that were updated. This writeset might be empty. At commit time the transaction manager checks whether the transaction has a non-empty writeset. If this is the case, it is an update transaction, otherwise, it is a read-only transaction. In the latter case, the transaction is committed locally. Otherwise, the local transaction manager assigns a commit timestamp that it has received from the commit server. It then propagates the writeset to a special logging component (discussed further below) as well as to the DB layer which forwards it to HBase. The records in the writeset are tagged with the commit timestamp of the transaction. When the logger notifies that the writeset is durable, the local transaction manager confirms the commit to the client. When the DB layer confirms that the updates have been written to HBase and are visible, the local transaction manager reports to the snapshot server that the commit timestamp is visible and durable.

The snapshot server keeps track of which commit timestamps have been used, and which have been discarded. A commit timestamp is considered as used, when the writeset is durable and visible. The snapshot server also determines which is the most recent commit timestamp such that all previous timestamps have been used and/or discarded. It reports periodically to the local transaction managers about this current snapshot timestamp. This is the timestamp the local transaction managers use as start timestamp, as it is guaranteed that the updates of committed transactions with lower commit timestamp are reflected in the object cache and at the HBase layer, thus guaranteeing consistent reads.

The commit sequencer is responsible to assign commit timestamps. It does not do this on demand but in a proactive way in order to minimize latency and keep the overhead at the commit sequencer low. The idea is that the commit sequencer sends batches of commit timestamps to each local transaction manager in regular time intervals (e.g., every 10 milliseconds). In order to determine the proper batch size, each local transaction managers reports to the commit sequencer at each period the number of update transactions it has committed in the previous period. When a local transaction manager receives a new batch of commit timestamps, it discards all previously unused timestamps and notifies the snapshot server about them. Then, it uses commit timestamps from the new batch.

With this commit time is very fast. The commit timestamp already exists locally at the local transaction manager. Conflict detection was already performed when the update operations took place. The only delay is that the local transaction manager has to wait until the writeset is written to the logger. In order to avoid a bottleneck at the logger, we allow multiple logger instances, each taking care of a fraction of the writesets. Thus, the logger can be scaled as needed by adding additional instances.

4. Preliminary Results

An initial prototype has been built and integrated with JBoss, Hibernate, Derby as the query engine and HBase. The prototype has been evaluated in a cluster of 120 cores. The deployment was as follows: Five dual-core nodes were devoted to HBase region server+HDFS data node, 1 dual-core node to the HDFS name node+cache provisioner, 1 dual-core node to Zookeper+HDFS secondary server, 1 dual-core node to the HBase master server+application server provisioner, 1 dual-core node to commit sequencer, 1 dual-core node to snapshot server, 5 dual-core nodes for cache server instances and 5 dual-core nodes to logger instances. We co-located an instance of JBoss, Hibernate and modified Derby on a quad-core node and vary the number of quad-core nodes devoted to them from 1 to 20.

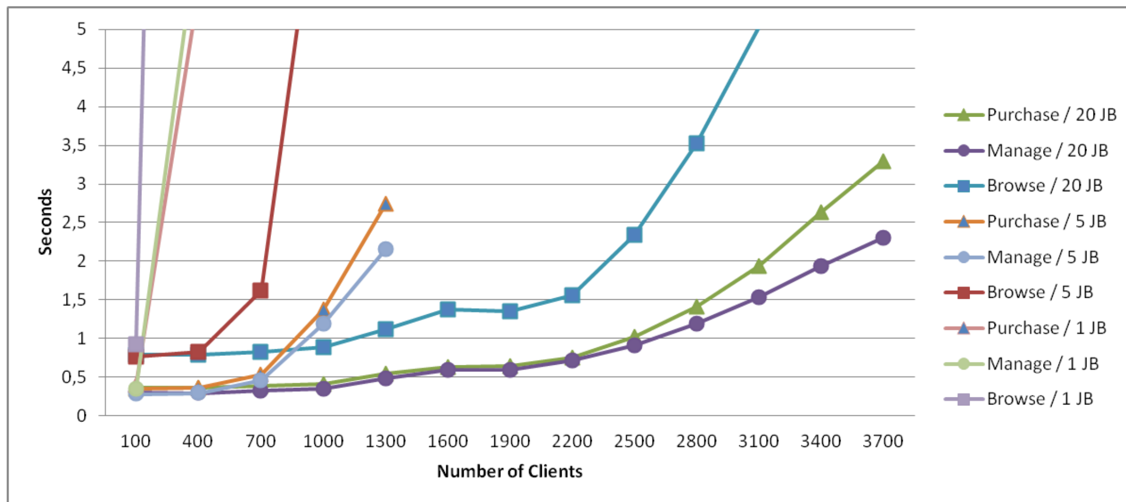

Figure 2: Evolution of Response time with 1 to 20 instances

The results are depicted in Figure 2. The threshold in the benchmark for the response time is 2 seconds. As can be seen for the management and purchase workloads the managed load with 1 JBoss+Hibernate+Derby node is able to handle only 100 clients, while with 5 and 20 nodes, it can handle 1000 and 3100-3400 clients. For the browsing workload, 1 JBoss+Hibernate+Derby node can handle 100 clients, and with 5 and 20 nodes it can handle 700 and 2200 clients. With database replication using the approach in [LinKPJ05] the maximum sustained load was 10 clients using 10 replicas.

5. Conclusions

This paper presents a new transactional cloud platform, CumuloNimbo. This system processes transactions fully in parallel, while at the same time it provides full ACID properties and coherence across all tiers. The system provides full transparency, syntactically and semantically, and it is highly distributed. The preliminary results show that the system has a substantial improvement in terms of scalability with respect to database replication. We believe that it has the potential to scale to thousands of cores, something that we are currently evaluating. Our current implementation has some issues with lack of sufficient concurrency and load balancing in HBase. Thus, we were not able to reach linear scalability but the results show a scalability substantially better than the one attained by the most modern database replication protocols.

6. References

[BerBGMNN95] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, P. E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD 1995

[PengD10] Daniel Peng, Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications", OSDI 2010.

[LinKPJ05] Y. Lin, B. Kemme, M. Patiño-Martínez, R. Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. ACM SIGMOD Int. Conf. on Management of Data. 2005

[KemJP10] B. Kemme, R. Jimenez-Peris, M. Patiño-Martinez. "Database Replication". Synthesis Lectures on Data Management #7. Morgan & Claypool Publishers. January 2010, vol (2): 1-153.

[BernRWY11] P.A. Bernstein, C. W. Reid, M. Wu, X. Yuan: Optimistic Concurrency Control by Melding Trees. Int. Conf. on Very Large Data Bases. 2011.