Scalable Transactions for Web Applications in the Cloud

Zhou Wei[1,2]*, Guillaume Pierre[1] and Chi-Hung Chi[2]

[1] Vrije Universiteit, Amsterdam, The Netherlands
zhouw@few.vu.nl , gpierre@cs.vu.nl
[2] Tsinghua University, Beijing, China
chichihung@mail.tsinghua.edu.cn

Abstract. Cloud computing platforms provide scalability and high availability properties for web applications but they sacrifice data consistency at the same time. However, many applications cannot afford any data inconsistency. We present a scalable transaction manager for cloud database services to execute ACID transactions of web applications, even in the presence of server failures. We demonstrate the scalability of our system using a prototype implementation, and show that it scales linearly to at least 40 nodes sustaining a maximum throughput of 7286 transactions per second.

1 Introduction

Cloud computing offers the vision of a virtually infinite pool of computing, storage and networking resources where applications can be scalably deployed [1]. The scalability and high availability properties of Cloud platforms however come at a cost. First, the scalable database services offered by the cloud such as Amazon SimpleDB and Google BigTable allow data query only by primary key rather than supporting secondary-key or join queries [2, 3]. Second, these services provide only *eventual data consistency*: any data update becomes visible after a finite but undeterministic amount of time. As weak as this consistency property may seem, it does allow to build a wide range of useful applications, as demonstrated by the commercial success of Cloud computing platforms. However, many other applications such as payment services and online auction services cannot afford any data inconsistency. While primary-key-only data access is a relatively minor inconvenience that can often be accommodated by good data structures, it is essential to provide transactional data consistency to support the applications that need it.

A transaction is a set of queries to be executed atomically on a single consistent view of a database. The main challenge to support transactional guarantees in a cloud computing environment is to provide the ACID properties of

* This work is partially supported by the 863 project #2007AA01- Z122 & project #2008AA01Z12, the National Natural Science Foundation of China Project #90604028, and 973 project #2004CB719406.

Atomicity, Consistency, Isolation and Durability [4] without compromising the scalability properties of the cloud. However, the underlying data storage services provide only eventual consistency. We address this problem by creating a secondary copy of the application data in the transaction managers that handle consistency. Obviously, any centralized transaction manager would face two scalability problems: 1) A single transaction manager must execute all incoming transactions and would eventually become the performance bottleneck; 2) A single transaction manager must maintain a copy of all data accessed by transactions and would eventually run out of storage space. To support transactions in a scalable fashion, we propose to split the transaction manager into any number of Local Transaction Managers (LTMs) and to partition the application data and the load of transaction processing across LTMs.

Our system exploits two properties typical of Web applications to allow efficient and scalable operations. First, we observe that in Web applications, all transactions are short-lived because each transaction is encapsulated in the processing of a particular request from a user. This rules out long-lived transactions that make scalable transactional systems so difficult to design, even in medium-scale environments [5]. Second, Web applications tend to issue transactions that span a relatively small number of well-identified data items. This means that the two-phase commit protocol for any given transaction can be confined to a relatively small number of servers holding the accessed data items. It also implies a low (although not negligible) number of conflicts between multiple transactions concurrently trying to read/write the same data items.

A transactional system must maintain the ACID properties even in the case of server failures. For this, we replicate data items and transaction states to multiple LTMs, and periodically checkpoint consistent data snapshots to the cloud storage service. Consistency correctness relies on the eventual consistency and high availability properties of Cloud computing storage services: we need not worry about data loss or unavailability after a data update has been issued to the storage service. We assume a fail-stop failure model of the LTMs and do not address Byzantine failures.

It should be noted that the CAP dilemma proves that it is impossible to provide both strong Consistency and high Availability in the presence of network Partitions [6]. Typical cloud services explicitly choose high availability over strong consistency. In this paper, we make the opposite choice and prefer providing transactional consistency for the applications that require it, possibly at the cost of unavailability during network failures.

We demonstrate the scalability of our transactional database service using a prototype implementation. Following the data model of Bigtable, transactions are allowed to access any number of data items by primary key at the granularity of the data row. The list of primary-keys accessed by the transaction must be given before executing the transaction. This means for example that range queries are not supported. Our system supports both read-write and read-only transactions. We evaluate the performance of our prototype under a workload derived from the TPC-W e-commerce benchmark [7], and show that it scales linearly to 40 LTMs sustaining a maximum throughput of 7286 transactions per second. This means that, according to the principles of Cloud computing, any increase in workload can be accommodated by provisioning more servers. Our system also tolerates server failures, which only cause a few aborted transactions (authorized by the ACID properties) and a temporary drop of throughput during data reorganization.

This paper is structured as follows. Section 2 presents related work. Then, Section 3 presents the design of partitioned transaction manager, and shows how to guarantee ACID properties even in the case of server failures. Section 4 presents performance evaluations and Section 5 concludes.

2 Related Work

Current prominent cloud database services such as Amazon SimpleDB and Google Bigtable only support eventual consistency properties [2, 3]. To obtain a full database service on the cloud, one can easily deploy classical relational databases such as MySQL and Oracle, and thus get support for transactional consistency. However, these database systems rely on traditional replication techniques and therefore do not bring extra scalability improvement compared to a non-cloud deployment [8].

An alternative approach is to run any number of database engines in the cloud, and use the cloud storage service as shared storage medium [9]. Each engine has access to the full data set and therefore can support any form of SQL queries. On the other hand, this approach cannot provide full ACID properties. In particular, the authors claim that the Isolation property cannot be provided, and that only reduced levels of consistency can be offered.

The most similar system to ours is the Scalaris transactional DHT [10, 11]. Like us, it splits data across any number of DHT nodes, and supports transactional access to any set of data items addressed by primary key. However, each query requires one or more requests to be routed through the DHT, adding latency and overhead. Cloud computing environment can also be expected to be much more reliable than typical peer-to-peer systems, which allows us to use more lightweight mechanisms for fault tolerance.

3 System Design

Figure 1 shows the organization of our transactional system. Clients issue HTTP requests to a Web application, which in turn issues transactions to a Transaction Processing System (TPS). The TPS is composed of any number of LTMs, each of which is responsible for a subset of all data items. The Web application can submit a transaction to any LTM that is responsible for one of the accessed data items. This LTM then acts as the coordinator of the transaction across all LTMs in charge of the data items accessed by the transaction. The LTMs operate on an in-memory copy of the data items loaded from the cloud storage service.

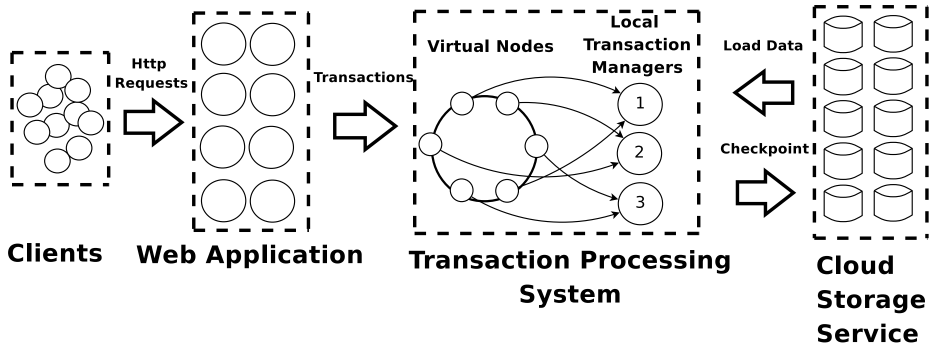


Fig. 1. System model

Data updates resulting from transactions are kept in memory of the LTMs and periodically checkpointed back to the cloud storage service.

We implement transactions using the two-phase commit protocol. In the first phase, the coordinator requests all involved LTMs and asks them to check that the operation can indeed been executed correctly. If all LTMs vote favorably, then the second phase actually commits the transaction. Otherwise, the transaction is aborted.

We assign data items to LTMs using consistent hashing [12]. To achieve a balanced assignment, we first cluster data items into virtual nodes, and then assign virtual nodes to LTMs. As shown in Figure 1, multiple virtual nodes can be assigned to the same LTM. To tolerate LTM failures, virtual nodes and transaction states are replicated to one or more LTMs. After an LTM server failure, the latest updates can then be recovered and affected transactions can continue execution while satisfying ACID properties.

We now detail the design of the TPS to guarantee the Atomicity, Consistency, Isolation and Durability properties of transactions. Each of the properties is discussed individually.

3.1 Atomicity

The Atomicity property requires that either all operations of a transaction complete successfully, or none of them do. To ensure Atomicity, for each transaction issued, our system performs two-phase commit (2PC) across all the LTMs responsible for the data items accessed. If an agreement of "COMMIT" is reached, the transaction coordinator can return the result to the web application without waiting for the completion of the second phase [13].

To ensure Atomicity in the presence of server failures, all transaction states and data items should be replicated to one or more LTMs. When an LTM fails, the transactions it was coordinating can be in two states. If a transaction has reached an agreement to "COMMIT," then it must eventually be committed;

otherwise, the transaction can be aborted. Therefore, we replicate transaction states in two occasions: 1) When an LTM receives a new transaction, it must replicate the transaction state to other LTMs before confirming to the application that the transaction has been successfully submitted; 2) After all participant LTMs reach an agreement to “COMMIT” at the coordinator, the coordinator updates the transaction state at its backups. This creates in essence in-memory “redo logs” at the backup LTMs. The coordinator must finish this step before carrying out the second phase of the commit protocol. If the coordinator fails after this step, the backup LTM can then complete the second phase of the commit protocol. Otherwise, it can simply abort the transaction without violating the Atomicity property.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. It is therefore necessary to re-replicate these data items to maintain N replicas. Once an LTM failure is detected, the failure detector issues a report to all LTMs so that they can carry out the recovery process and create a new consistent membership of the system. All incoming transactions that accessed the failed LTM are aborted during the recovery process. If a second LTM server failure happens during the recovery process of a previous LTM server failure, the system initiates the recovery of the second failure after the current recovery process has completed. The transactions that cannot recover from the first failure because they also accessed the second failed LTM are left untouched until the second recovery process.

As each transaction and data item has $N + 1$ replicas in total, the TPS can thus guarantee the Atomicity property under the simultaneous failure of N LTM servers.

3.2 Consistency

The Consistency property requires that a transaction, which executes on a database that is internally consistent, will leave the database in an internally consistent state. Consistency is typically expressed as a set of declarative integrity constraints. We assume that the consistency rule is applied within the logic of transactions. Therefore, the Consistency property is satisfied as long as all transactions are executed correctly.

3.3 Isolation

The Isolation property requires that the behavior of a transaction is not impacted by the presence of other transactions that may be accessing the same data items concurrently. In the TPS, we decompose a transaction into a number of sub-transactions. Thus the Isolation property requires that if two transactions conflict on more than one data item, all of their conflicting sub-transactions must be executed sequentially, even though the sub-transactions are executed in multiple LTMs.

We apply timestamp ordering for globally ordering conflicting transactions across all LTMs. Each transaction has a globally unique timestamp, which is monotonically increasing with the time the transaction was submitted. All LTMs then order transactions as follows: a sub-transaction can execute only after all conflicting sub-transactions with a lower timestamp have committed. It may happen that a transaction is delayed (e.g., because of network delays) and that a conflicting sub-transaction with a younger timestamp has already committed. In this case, the older transaction should abort, obtain a new timestamp and restart the execution of all of its sub-transactions.

As each sub-transaction accesses only one data item by primary key, the implementation is straightforward. Each LTM maintains a list of sub-transactions for each data item it handles. The list is ordered by timestamp so LTMs can execute the sub-transactions sequentially in the timestamp order. The exception discussed before happens when an LTM inserts a sub-transaction to the list but finds its timestamp smaller than the one currently being executed. It then reports the exception to the coordinator LTM of this transaction so that the whole transaction can be restarted. We extended the 2PC with an optional "RESTART" phase, which is triggered if any of the sub-transactions reports an ordering exception. After a transaction reached an agreement and enters the second phase of 2PC, it cannot be restarted any more.

We are well aware that assigning timestamps to transactions using a single global timestamp manager can create a potential bottleneck in the system. We used this implementation for simplicity, although distributed timestamp managers exist [14].

3.4 Durability

The Durability property requires that the effects of committed transactions would not be undone and would survive server failures. In our case, it means that all the data updates of committed transactions must be successfully written back to the backend cloud storage service.

The main issue here is to support LTM failures without losing data. For performance reasons, the commit of a transaction does not directly update data in the cloud storage service but only updates the in-memory copy of data items in the LTMs. Instead, each LTM issues periodic updates to the cloud storage service. During the time between a transaction commit and the next checkpoint, durability is ensured by the replication of data items across several LTMs. After checkpoint, we can rely on the high availability and eventual consistency properties of the cloud storage service for durability.

When an LTM server fails, all the data items stored in its memory that were not checkpointed yet are lost. However, as discussed in Section 3.1, all data items of the failed LTM can be recovered from the backup LTMs. The difficulty here is that the backups do not know which data items have already been checkpointed. One solution would be to checkpoint all recovered data items. However, this can cause a lot of unnecessary writes. One optimization is to record the latest checkpointed transaction timestamp of each data item and replicate these timestamps to the backup LTMs. We further cluster transactions into groups, then replicate timestamps only after a whole group of transactions has completed.

Another issue related to checkpointing is to avoid degrading the system performance at the time of a checkpoint. The checkpoint process must iterate through the latest updates of committed transactions and select the data items to be checkpointed. A naive implementation that would lock the whole buffer during checkpointing would also block the concurrent execution of transactions. We address this problem by maintaining a buffer in memory with the list of data items to be checkpointed. Transactions write to this buffer by sending updates to an unbounded non-blocking concurrent queue [15]. This data structure has the property of allowing multiple threads to write concurrently to the queue without blocking each other. Moreover, it orders elements in FIFO order, so old updates will not override younger ones.

3.5 ReadOnly Transactions

Our system supports read-write and read-only transactions indifferently. The only difference is that in read-only transactions no data item is updated during the second phase of 2PC. Read-only transactions have the same strong data consistency property, but also the same constraint: accessing well identified data items by primary key only. However, our system provides an additional feature for read-only transactions to support complex read queries such as range queries executed on a consistent snapshot of database.

We exploit the fact that many read queries can produce useful results by accessing an older but consistent data snapshot. For example, in e-commerce Web applications, a promotion service may identify the best seller items by aggregating recent orders information. However, it is not necessary to compute the result based on the absolute most recent orders. We therefore introduce the concept of Weakly-Consistent Read-only Transaction (WCRT), which is defined as follows: 1) A WCRT allows any type of read operations, including range queries; 2) WCRTs do not execute at the LTMs but access the latest checkpoint in the cloud storage service directly. A WCRT always executes on an internally consistent but possibly slightly outdated snapshot of the database. WCRTs are supported only if the underlying data storage service supports multi-versioning, as for example Bigtable [3]. To implement WCRTs, we introduce a snapshot mechanism in the checkpoint process, which marks each data update with a specific snapshot ID that is monotonically increasing. This ID is used as the version number of the new created version when it is written to the cloud storage service. A WCRT can thus access a specific snapshot by only reading the latest version of any data item of which the timestamp is not larger than the snapshot ID.

We define that the updates of every M transactions with subsequent timestamps constitute a new snapshot. Assuming that the transaction timestamp is implemented as a simple counter, the first snapshot reflects all the updates of committed transactions $[0, M)$. The next snapshot reflects updates from transactions $[0, 2M)$, and so on. A snapshot is "ready" after all the updates of transactions it reflects have been written back to cloud storage service and have been made visible to the Web application. The TPS publishes the latest snapshot that is "ready," so the web application can access the consistent view of data from cloud storage service.

The difficulty in implementing the snapshot mechanism is that each LTM performs checkpoints independently and that any transaction may update data items in multiple LTMs. We introduce a Master node which collects reports from each LTMs about its latest *locally* "ready" snapshot. So the Master node can determine the latest *globally* "ready" snapshot. This Master node also carries out the task of publishing the latest snapshot ID to Web applications. If the underlying cloud data storage service satisfies "Read-Your-Writes" consistency, writing back successfully will be a sufficient condition to consider the snapshot as "ready." Otherwise, the LTMs must check if the issued updates is readable.

4 Evaluation

We demonstrate the scalability of our transactional database system by presenting the performance evaluation of a prototype implementation under the workload of TPC-W [7]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com. This paper focuses on the scalability of the system with increasing number of LTMs rather than on its absolute performance for a given number of LTMs. Our evaluation assumes that the application load remains roughly constant and studies the scalability in terms of the maximum sustainable throughput under a response time constraint.

4.1 Experiment Setup

All experiments are performed on the DAS-3 at the Vrije Universiteit, an 85-node Linux-based server cluster [16]. Each machine in the cluster has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are connected to each other with a Myri-10G LAN such that the network latency between the servers is negligible. We use Tomcat v5.5.20 as application server and HBase v0.2.1, an open-source clone of BigTable [17] as backend cloud storage service. The TPC-W application and load generator are deployed in separate application servers.

We generate an evaluation workload composed of transactions issued by the TPC-W Web application. The workload is generated by a configurable number of Emulated Browsers (EBs), each of which issues requests from a simulated user. We observe that most of the read queries and read-only transactions of TPC-W can tolerate a slightly old version of data, so they can directly access a consistent snapshot from the storage service using WCRTs. The workload that an Emulated Browser issues to the TPS mainly consists of read-write transactions that require strong data consistency. Figure 2 shows the workflow of transactions issued by an Emulated Browser, which simulates a typical shopping process of customer.

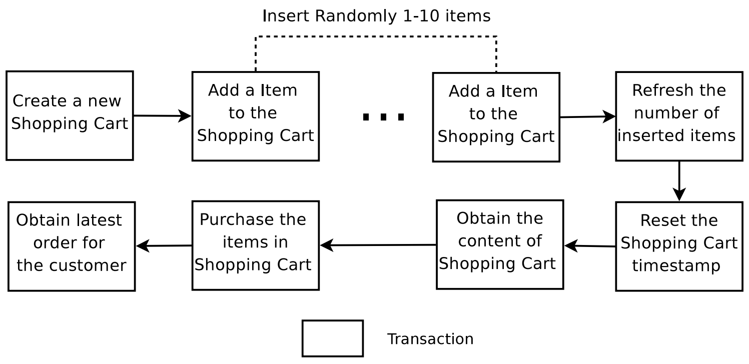

Insert Randomly 1-10 items
Create a new Shopping Cart
Add a Item to the Shopping Cart
...
Add a Item to the Shopping Cart
Refresh the number of inserted items
Reset the Shopping Cart timestamp
Obtain the content of Shopping Cart
Purchase the items in Shopping Cart
Obtain latest order for the customer
Transaction

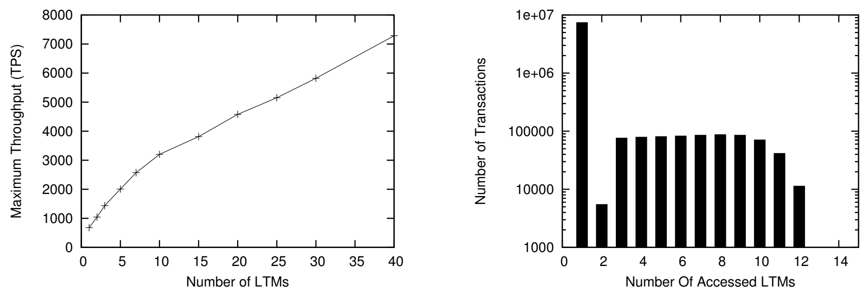
Fig. 2. Workflow of transactions issued by each Emulated Browser (EB) of TPC-W

Each EB waits for 500 milliseconds between receiving a response and issuing the next transaction.

We adapted the original relational data model defined by TPC-W to the Bigtable data model, so that the application data can be stored into HBase. Using similar data denormalization techniques as in [18], we designed a Bigtable data model for TPC-W that contains only the columns accessed by the transactions in Figure 2. The relational data model of TPC-W comprises six tables that are accessed by these transactions. To adapt this data model to Bigtable, we first combine five of them ("Orders, Order_Line, Shopping_Cart, Shopping_Cart_Entry, CC_XACTS") into one *bigtable* named "Shopping." Each of the original tables is stored as a column family. The new *bigtable* "Shopping" has the same primary key as table "Shopping_Cart." For table "Order_Line," multiple rows are related to one row in table "Order," they are combined into one row and stored in the new *bigtable* by defining different column names for the values of same data column but different rows. Second, for the remaining table "Item," only the column "i_stock" is accessed. We thus can have a *bigtable* named "Item_Stock" which only contains this column and has the same primary key. Finally, for the last transaction in Figure 2 which retrieves the latest order information for a specific customer, we create an extra index *bigtable* "Latest_Order" which uses customer IDs as its primary key and contains one column storing the latest order ID of the customer.

Before each experiment, we populate 144,000 customer records in the "Latest_Order" *bigtable* and 10,000 item records in the "Item_Stock" *bigtable*. We then populate the "Shopping" *bigtable* according to the benchmark requirements.

In our experiments, we observed a load balancing problem in HBase because TPC-W assigns new shopping cart IDs sequentially. Each HBase node is responsible for a set of contiguous ranges of ID values, so at any moment of time, most newly created shopping carts would be handled by the same HBase node. To address this problem, we horizontally partitioned the *bigtables* into 50 sub-bigtables and allocated data items in round-robin fashion.



(a) Maximum Sustainable Throughput

(b) Number of LTMs accessed by the transactions of TPC-W

Fig. 3. Throughput Scalability of the Transaction Processing System

4.2 Scalability Evaluation

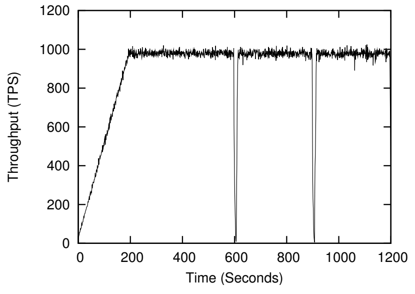
We study the scalability of the Transaction Processing System by measuring the system performance in terms of throughput. We first define a response time constraint that imposes that the 99% of transactions must return within 100 ms. For a given number of LTMs we then measure the maximum sustainable throughput of the system before the constraint gets violated.

We configure the system so that each transaction and data item has one backup in total, and set the checkpoint interval to 1 second. We start with one LTM and 5 HBase servers, then add more LTM and HBase servers. In all cases, we deliberately over-allocated the number of HBase servers and client machines to make sure that the Transaction Processing System remains the performance bottleneck.

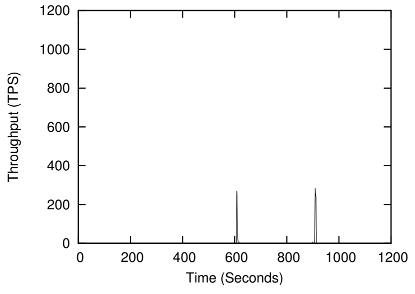
Figure 3(a) shows that our system scales nearly linearly. When using 40 LTM servers it reaches a maximum throughput of 7286 transactions per second generated by 3825 emulated browsers. In this last configuration, we use 40 LTM servers, 36 HBase servers, 3 clients to generate load, 1 server as global timestamp manager, and 1 Master server for managing snapshots for WCRTs and coordinating the recovery process. This configuration uses the entire DAS-3 cluster so we could not extend the experiment further. The maximum throughput of the system at that point is approximately 10 times that of a single LTM server.

Figure 3(b) shows the number of LTMs that participate in the transactions[3], when there are 40 LTMs servers in the system. Transactions access up to 12 LTMs, which corresponds to a shopping cart containing the maximum number of 10 items. We however observe that the vast majority of transactions access only one LTM. In other words, most of the transactions of TPC-W execute within one LTM and its backups only. We expect this behavior to be typical of Web applications.

[3] The LTMs that act only as backup of transactions or data items are not counted in.



(a) Committed transactions



(b) Aborted transactions

Fig. 4. Throughput of system in the presence of LTM server failures

4.3 Fault Tolerance

We now study the system performance in the presence of LTM server failures. We configure the system so that each transaction and data item has one backup and set the checkpoint interval to 1 second. We start with 5 LTMs and generate the workload using 500 EBs so that the system would not overload even after two LTM servers failures.

We first warm up the system by adding 25 EBs every 10 seconds. The full load is reached after 200 seconds. After running the system normally for a while, one LTM server is shutdown to simulate a failure. Figure 4(a) shows the effect of two LTM server failures at times 600 and 900 seconds. When an LTM server fails, the system recovers within a few seconds and the transaction throughput returns to the previous level. At the same time, as shown in Figure 4(b), a few transactions are aborted because the incoming transactions that accessed the failed LTM must be rejected during the recovery process. After two LTM server failures, the remaining 3 LTM servers can still sustain a throughput of about 1000 transactions per second.

5 Conclusion

Many Web applications need strong data consistency for their correct executions. However, although the high scalability and availability properties of the cloud make it a good platform to host Web content, scalable cloud database services only provide eventual consistency properties. This paper shows how one can support ACID transactions without compromising the scalability property of the cloud for web applications, even in the presence of server failures.

This work relies on few simple ideas. First, we load data from the cloud storage system into the transactional layer. Second, we split the data across any number of LTMs, and replicate them only for fault tolerance. Web applications typically access only a few partitions in any of their transactions, which gives our system linear scalability. Our system supports full ACID properties even in the presence of server failures, which only cause a temporary drop in throughput and a few aborted transactions.

Data partitioning also implies that transactions can only access data by primary key. Read-only transactions that require more complex data access can still be executed, but on a possibly outdated snapshot of the database. Lifting this limitation is on our immediate research agenda.

References

1. Hayes, B.: Cloud computing. Communications of the ACM **51**(7) (July 2008) 9–11
2. Amazon.com: Amazon SimpleDB. `http://aws.amazon.com/simpledb`.
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable : a distributed storage system for structured data. In: Proc. OSDI. (2006) 205–218
4. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
5. Transaction Processing Performance Council: TPC benchmark C standard specification, revision 5 (December 2006) http://www.tpc.org/tpcc/.
6. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
7. Smith, W.D.: TPC-W: Benchmarking an ecommerce solution. White paper, Transaction Processing Performance Council
8. Atwood, M.: A MySQL storage engine for AWS S3. In: MySQL Conference and Expo. (2007) `http://fallenpegasus.com/code/mysql-awss3/`.
9. Brantner, M., Florescu, D., Graf, D., Kossmann, D., Kraska, T.: Building a database on S3. In: Proc. ACM SIGMOD. (2008) 251–264
10. Moser, M., Haridi, S.: Atomic Commitment in Transactional DHTs. In: Proc. CoreGRID Symposium. (2007)
11. Plantikow, S., Reinefeld, A., Schintke, F.: Transactions for distributed wikis on structured overlays. In: Proc. Intl. Workshop on Distributed Systems: Operations and Management. (2007) 256–267
12. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proc. ACM Symposium on Theory of Computing. (1997) 654–663
13. Hvasshovd, S.O., Torbjornsen, O., Bratsberg, S.E., Holager, P.: The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In: Proc. VLDB. (1995) 469–477
14. Raz, Y.: The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In: Proc. VLDB. (1992) 292–312
15. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. ACM symposium on Principles of distributed computing. (1996) 267–275
16. DAS3: The Distributed ASCI Supercomputer 3. `http://www.cs.vu.nl/das3/`.
17. HBase: An open-source, distributed, column-oriented store modeled after the Google Bigtable paper. `http://hadoop.apache.org/hbase/`.
18. Wei, Z., Dejun, J., Pierre, G., Chi, C.H., van Steen, M.: Service-oriented data denormalization for scalable web applications. In: Proc. Intl. World Wide Web Conf. (2008)