

CloudTPS: Scalable Transactions for Web Applications in the Cloud

Zhou Wei^{*,†} Guillaume Pierre^{*}

Chi-Hung Chi[†]

^{*} Vrije Universiteit, Amsterdam, The Netherlands

[†] Tsinghua University, Beijing, China

February 15, 2010

Technical report IR-CS-53, Vrije Universiteit, February 2010.

Abstract

NoSQL Cloud data services provide scalability and high availability properties for web applications but at the same time they sacrifice data consistency. However, many applications cannot afford any data inconsistency. CloudTPS is a scalable transaction manager to allow cloud database services to execute the ACID transactions of web applications, even in the presence of server failures and network partitions. We implement this approach on top of the two main families of scalable data layers: Bigtable and SimpleDB. Performance evaluation on top of HBase (an open-source version of Bigtable) in our local cluster and Amazon SimpleDB in the Amazon cloud shows that our system scales linearly at least up to 40 nodes in our local cluster and 80 nodes in the Amazon cloud.

Keywords: Scalability, web applications, cloud computing, transactions, NoSQL.

Contents

1	Introduction	3
2	Related Work	6
2.1	Data Storage in the Cloud	6
2.2	Distributed Transactional Systems	8
3	System Model	9
4	System Design	12
4.1	Atomicity	12
4.2	Consistency	13
4.3	Isolation	13
4.4	Durability	14
4.5	Membership	15
4.5.1	Membership Updates	16
4.5.2	Dealing with Network Partitions	16
5	System Implementation	18
5.1	Portability	19
5.2	Memory Management	20
5.3	ReadOnly Transactions	22
6	Evaluation	23
6.1	Migration of TPC-W to the Cloud	24
6.2	Experiment Setup	25
6.3	Scalability Evaluation	26
6.4	Tolerance of LTM Failures and Network Partitions	31
6.5	Memory Management	32
7	Conclusion	37

1 Introduction

Cloud computing offers the vision of a virtually infinite pool of computing, storage and networking resources where applications can be scalably deployed [22]. In particular, NoSQL cloud database services such as Amazon SimpleDB [2] and Google Bigtable [13] offer a scalable data tier for applications in the cloud. These systems typically partition the application data to provide incremental scalability, and replicate the partitioned data to tolerate server failures.

The scalability and high availability properties of Cloud platforms however come at a cost. First, these scalable database services allow data query only by primary key rather than supporting secondary-key or join queries. Second, these services provide only weak consistency such as 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 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 Atomicity, Consistency, Isolation and Durability [20] without compromising the scalability properties of the cloud. However, the underlying cloud data storage services provide only eventual consistency. We address this problem by creating a secondary temporary 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 and availability 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.

CloudTPS exploits three 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 [49]. Second, Web applications tend to issue transactions that span a relatively small number of well-identified data items. This means that the 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. Third, many read-only queries of Web applications can produce useful results by accessing an older yet consistent version of data. This allows to execute complex read queries directly in the cloud data service, rather than in LTMs.

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.

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 [19]. Typical cloud services explicitly choose high availability over strong consistency. In this article, 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.

To implement CloudTPS efficiently, we must address two additional issues:

Firstly, there exists a wide variety of cloud data services [2, 13, 14, 37]. CloudTPS should be portable across them, and the porting should require only minor adaptation. On the one hand, as current cloud data services use different data models and interfaces, we build CloudTPS upon their common features: our data model is based on attribute-value pairs. The implementation only demands a simple primary-key-based “GET/PUT” interface from cloud data services. On the other hand, current cloud data services provide different consistency levels. For instance, Bigtable supports transactions on single data items while SimpleDB only provides eventual consistency. To

ensure the correctness and efficiency of CloudTPS, we implement various mechanisms for different underlying consistency levels.

Secondly, loading a full copy of application data into the system may overflow the memory of LTMs, forcing one to use many LTMs just for their storage capacity. This is, however, not necessary as only the currently accessed data items contribute to maintaining ACID properties of transactions. Other unaccessed data items can be evicted from the LTMs if we can fetch their latest stored versions from the cloud storage service. We observe that Web applications exhibit temporal locality where only a portion of application data is accessed at any time [51, 50]. We can therefore design efficient memory management mechanisms to restrict the number of in-memory data items in LTMs while still maintaining strong data consistency. Data items being accessed by uncommitted transactions must stay in the LTMs to maintain ACID properties; others depend on a tradeoff between memory size and access latency. We use a cost-aware replacement policy to dictate which data items should remain in the LTMs.

We demonstrate the scalability of our transactional database service using a prototype implementation. Following the data models of Bigtable and SimpleDB, 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 a transaction must be given explicitly before executing the transaction. This means for example that range queries are not supported within a transaction. CloudTPS 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 [35]. We implemented our approach on top of the two main families of scalable data layers: HBase, an open-source clone of BigTable [23], running in our local cluster; and SimpleDB, running in the Amazon Cloud [3]. We show that CloudTPS scales linearly to at least 40 LTMs in our local cluster and 80 LTMs in the Amazon Cloud. This means that, according to the principles of Cloud computing, any increase in workload can be accommodated by provisioning more servers. CloudTPS tolerates server failures, which only cause a few aborted transactions (authorized by the ACID properties) and a temporary drop of throughput during transaction recovery and data reorganization. In case of network partitions, CloudTPS may need to reject incoming transactions to maintain data consistency. It recovers and becomes available again as soon as the network is restored, while still maintaining ACID properties. We finally present the performance of our memory management mechanism and

show that it can effectively control the buffer sizes of LTMs and only cause minor performance overhead.

This article is an extended version of a previous conference paper [54]. The additional contributions of this article are as follows: (i) We describe the membership management and failure recovery protocol in detail. This protocol maintains ACID properties in the case of machine failures and network partitions. (ii) We present the memory management mechanism, which prevents LTMs from memory overflow and reduces the required number of LTMs. (iii) We describe our prototype implementation on top of SimpleDB and discuss the port of CloudTPS to other cloud data services. (iv) We further demonstrate the scalability of our transactional system by evaluating the performance of our prototype implementation on top of SimpleDB in the Amazon Cloud.

This article is structured as follows. Section 2 presents related works and Section 3 describes the system model. Section 4 then presents the design of partitioned transaction manager, and shows how to guarantee ACID properties even in the case of server failures and network partitions. Section 5 discusses the implementation details of CloudTPS and two optional optimizations for memory management and read-only transactions. Finally, Section 6 presents performance evaluation and Section 7 concludes.

2 Related Work

2.1 Data Storage in the Cloud

The simplest way to store structured data in the cloud is to deploy a relational database such as MySQL or Oracle. The relational data model, typically implemented via the SQL language, provides great flexibility in accessing data. It supports sophisticated data access operations such as aggregation, range queries, join queries, etc. RDBMSs support transactions and guarantee strong data consistency. One can easily deploy a classical RDBMS such as MySQL and Oracle in the cloud and thus get support for transactional consistency. However, the features of flexible data querying and strong data consistency prevent one from partitioning data automatically, which is the key for performance scalability. These database systems rely on replication techniques and therefore do not bring extra scalability improvement compared to a non-cloud deployment [29, 41, 6].

On the other hand, a new family of cloud database services such as Google Bigtable [13], Amazon SimpleDB [2], Yahoo PNUTS [14], and Cassandra [31], uses simplified data models based on attribute-value pairs.

Application data are organized into tables, of which each is a collection of data items. Data items are typically accessed through a “GET/ PUT” interface by primary key. Additional sophisticated data access operations, such as range queries in SimpleDB or data item scanning in Bigtable, are limited within a table. None of them supports operation across multiple tables, such as join queries. This data model allows such systems to partition application data into any number of tables efficiently. Furthermore, these cloud database services relax data consistency: they disallow any consistency rules across multiple data partitions and provide little support for transactions. For example, SimpleDB and Cassandra only support eventual consistency, which means that data updates can be visible after an undetermined amount of time [52]. Bigtable and PNUTS support transactions but only over a single data item, which is not sufficient to guarantee strong data consistency.

Google Megastore [26] is a transactional indexed record manager on top of BigTable. Megastore supports ACID transactions across multiple data items. However, programmers have to manually link data items into hierarchical groups, and each transaction can only access a single group. In CloudTPS, transactions can access any set of data items together. Megastore replicates data items and uses Paxos algorithms to guarantee strong consistency in case of replica failures. Paxos can address Byzantine failures, but it also requires $2F+1$ servers to support the simultaneous failure of any F servers, which introduces high costs for a transaction. In contrast, we apply relatively light-weight master-slave mechanism to push updates to replicas, thus improving the performance of normally-committed transactions.

Microsoft SQL Azure Database [37] is a scalable cloud data service which supports the relational data model and ACID transactions containing any SQL queries. However, similar to Megastore, it requires manual data partitioning and does not support distributed transactions or queries across multiple data partitions located in different servers.

An alternative approach to implement cloud database is to run any number of database engines in the cloud, and use the cloud file system as shared storage medium [11]. 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.

Sudipt [15] proposes a similar approach to ours to support scalable transactions in the cloud. It also splits the transaction manager into multiple ones, where each one loads a specific data partition from the cloud stor-

age service and owns exclusive access to it. However, the system does not address the problem of maintaining ACID properties in the presence of machine failures. Furthermore it allows only restricted transactional semantic, similar to the one of Sinfonia [1], for distributed transactions across multiple data partitions.

2.2 Distributed Transactional Systems

There have been decades of research efforts in efficiently implementing distributed transactions for distributed database systems [42, 38, 39]. A number of distributed commit protocols [33, 43, 46, 21] and concurrency control mechanisms [7, 8, 9] have been proposed to maintain the ACID properties of distributed transactions. However, as distributed databases use the same relational data model as RDBMS, they also cannot partition the data automatically and thus lack scalability. On the other hand, we can apply these techniques as building blocks in designing CloudTPS. We rely on 2-Phase Commit (2PC) [33, 43] as the distributed commit protocol for ensuring Atomicity, and on timestamp-ordering [7] for concurrency control.

H-Store [47, 27] is a distributed main memory OLTP database, which executes on a cluster of shared-nothing main memory executor nodes. H-Store supports transactions accessing multiple data records with SQL semantics, implemented as predefined stored procedures written in C++. H-Store also replicates data records to tolerate machine failures. H-Store focuses on absolute system performance in terms of transaction throughput, and achieves very high performance on one executor node. However, the scalability of H-Store relies on careful data partition across executor nodes, such that most transactions access only one executor node. On the other hand, we prefer to focus on achieving linear scalability specifically for Web applications, such that any increase in workload can be accommodated by provisioning more servers. Also note that H-Store does not maintain persistent logs or keep any data in the non-volatile storage of either the executor nodes nor any backing store. CloudTPS checkpoints the updates back to the cloud data service to guarantee durability for each transaction.

Sinfonia [1] is a distributed message passing framework which supports transactional access to in-memory data across the distributed system. It addresses fault tolerance by primary-copy replication and by writing transactionally consistent backups to disk images. In contrast with our work, Sinfonia provides a low-level data access interface based on memory address and only supports transactions with restricted semantics. Besides, it requires the applications to manage data placement and caching themselves

across the distributed system. Sinfonia targets infrastructure applications which require fine-grained control of data structure and placement to optimize performance. On the other hand, Web applications usually require rapid and flexible development, so we prefer accessing a logical and location-transparent data structure with rich-semantic transactions.

Transactional memory (TM) systems relate to our work as they support transactional access to in-memory data. They traditionally target single multiprocessor machines [24], but recent research works extend them to distributed systems and support distributed transactions across in-memory data of multiple machines [34, 30, 10]. Distributed TM systems however provide no durability for transactions and do not address machine failures. The reason is that TM systems are mainly designed for parallel programs that solve large-sized problems. In this case, only the final results are valuable and required to be durable. On the other hand, TM systems usually execute in a managed environment with few machine failures. They thus provide no durability for intermediate transactions and do not address machine failures to maximize the system performance. However for Web applications which are usually interactive, the result of each transaction is critical. TM systems are therefore not suitable for Web applications.

Another similar system to ours is the Scalaris transactional DHT [40]. It splits data across any number of DHT nodes, and supports transactional access to any set of data items addressed by primary key. However, it supports no durability for the stored data while our system provides durability for transactions by checkpointing data updates into the cloud data service. On the other hand, Scalaris guarantees strong consistency by using Paxos algorithms, which can address Byzantine failures, but introduces high costs for a transaction. Moreover, each query requires one or more requests to be routed through the DHT, potentially adding latency and overhead. Cloud computing environments 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 Model

Figure 1 shows the organization of CloudTPS. 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

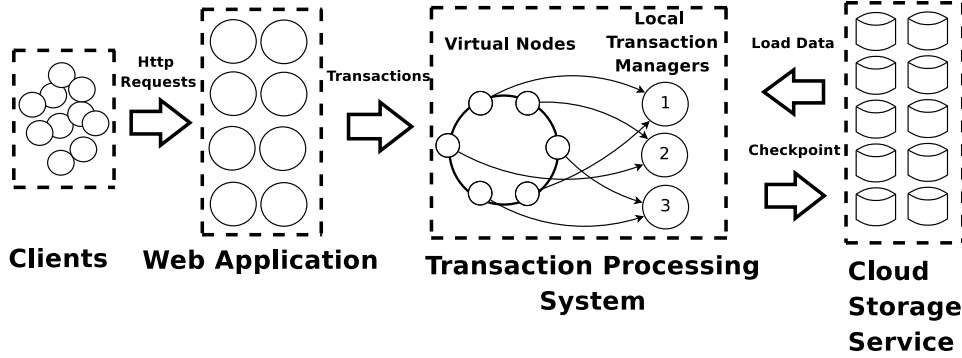


Figure 1: CloudTPS system model

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. Data updates resulting from transactions are kept in memory of the LTMs. To prevent data loss due to LTM server failures, the data updates are replicated to multiple LTM servers. LTMs also periodically checkpoint the updates back to the cloud storage service which is assumed to be highly-available and persistent.

We implement transactions using the 2-Phase Commit protocol (2PC). In the first phase, the coordinator requests all involved LTMs and asks them to check that the operation can indeed be executed correctly. If all LTMs vote favorably, then the second phase actually commits the transaction. Otherwise, the transaction is aborted.

Transactions in CloudTPS are short-lived and access only well-identified data items. CloudTPS allows only server-side transactions, which consist of predefined procedures stored at all LTMs. Each transaction consists of one or more sub-transactions, which operate on a single data item each. The Web application must provide the primary keys of all accessed data items when it issues a transaction.

Concretely, a transaction is implemented as a Java object containing a list of sub-transaction instances. All sub-transactions are implemented as sub-classes of the “SubTransaction” abstract Java class. As shown in Figure 2, each sub-transaction contains a unique “className” to identify itself, a table name and primary key to identify the accessed data item, and input parameters organized as attribute-value pairs. Each sub-transaction imple-

```

public abstract class SubTransaction {
    //Input by Web Application
    public String className;
    public String tableName;
    public String primaryKey;
    public Hashtable<String ,String> parameters;

    //Input by LTMs
    public Hashtable<String ,String> dataItem;

    //Output
    public String [][] dataToReturn;
    public String [][] dataToPut;
    public String [] dataToDelete;

    //Data Operations of the SubTransaction
    public VoteResult run();
}

```

Figure 2: The parent class of all sub-transactions classes

ments its own data operations by overriding the “run()” operation. The return value of the “run()” operation specifies whether this sub-transaction is able to commit. The execution of “run()” also generates the data updates and the results for read data operations, which are stored in the “Output” attributes.

The bytecode of all sub-transactions is deployed at all LTMs beforehand. A Web application issues a transaction by submitting the names of included sub-transactions and their parameters. LTMs then construct the corresponding sub-transaction instances to execute the transaction. In the first phase of 2PC, LTMs load the data items of each sub-transaction and execute the “run()” operation to decide on their votes and generate proposed data updates. If an agreement to “COMMIT” is reached, LTMs apply the updates to the data items.

We assign data items to LTMs using consistent hashing [28]. 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.

4 System Design

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. We then discuss the membership mechanisms to guarantee the ACID properties even in case of LTM failures and network partitions.

4.1 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, CloudTPS performs two-phase commit (2PC) across all the LTMs responsible for the data items accessed. As soon as an agreement of “COMMIT” is reached, the transaction coordinator can simultaneously return the result to the web application and complete the second phase [25].

To ensure Atomicity in the presence of server failures, all transaction states and data items should be replicated to one or more LTMs. LTMs replicate the data items to the backup LTMs during the second phase of transaction. Thus when the second phase completes successfully, all replicas of the accessed data items are consistent. The transaction state includes the transaction timestamp (discussed in Section 4.3), the agreement to either “COMMIT” or “ABORT”, and the list of data updates to be committed.

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 still 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 with the agreement of “COMMIT” and all the data updates. The participant LTMs piggyback their data updates with their vote messages. 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 ACID properties.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. It is necessary to re-replicate these data items to maintain N backups. 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 first 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.

4.2 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.

4.3 Isolation

The Isolation property requires that the behavior of a transaction is not disturbed by the presence of other transactions that may be accessing the same data items concurrently. The TPS decomposes a transaction into a number of sub-transactions, each accessing a single data item. Thus the Isolation property requires that if two transactions conflict on any number of data items, all 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 into 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 reaches 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 and a single point of failure in the system. We used this implementation for simplicity, although distributed timestamp managers exist [32].

4.4 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 4.1, 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 an extra 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 [36]. 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.

4.5 Membership

To correctly execute transactions, all LTMs must share the same view of system membership to determine the assignment of data items consistently. The system membership changes when LTMs join, depart, fail or recover from failures. These events may happen at any time, including during the execution of transactions. To ensure the ACID properties, changes in system membership must not take place during the 2PC execution of any transaction. When an LTM fails, other LTMs must therefore first complete the recovery of all ongoing transactions before updating the system membership.

In addition to LTM failures, the system may also encounter network failures, which can temporarily split the LTMs into multiple disconnected partitions. In such a case, according to the CAP dilemma, we decide to guarantee consistency at the possible cost of a loss of availability. In the case of system partitioning, transactions may still proceed provided that: (i) one of the partitions is able to elect itself as the “majority” partition; and (ii) its available LTMs can recover the consistent states of all data items. In all other cases the system will reject incoming connections until it fulfills the condition again.

This section presents our mechanism to recover the system consistently from network partitions.

4.5.1 Membership Updates

To ensure a consistent membership, all membership changes are realized through a 2PC across all available LTMs. All LTMs block incoming transactions until the new system membership has been committed consistently. In the first phase of a membership change, each LTM waits for all of its coordinated ongoing transactions to terminate, and then votes “COMMIT.” After reaching an agreement to “COMMIT,” the second phase updates the system membership and applies the new data assignment through data replication/movement.

Each membership change creates a new membership version attached to a monotonically increasing timestamp. Each LTM attaches the timestamp of its current membership to all of its messages. If an LTM receives a message with a higher timestamp than its own, this means that the other LTMs consider it as having failed. The concerned LTM discards its entire state and rejoins.

After each membership change, the new timestamp is stored in a special “Membership” table in the cloud data service. By scanning through this “Membership” table, any new LTM or any Web application instance can locate the currently available LTMs. One issue is that the cloud data services may return a stale membership. However, one can contact the TPS and obtain the latest membership as long as the stale membership contains at least one LTM currently in the TPS.

Any LTM may initiate a membership update if it wants to join the system or it detects the unavailability of other LTMs. This means that multiple updates may be issued simultaneously. To guarantee the isolation of such updates, we use a simple optimistic concurrency control mechanism that only one membership update can take place at a time [44]. If an LTM receives a request for membership update before a previous one has finished, then this LTM will vote “ABORT” to the latter. To avoid continuous conflicts and aborts, LTMs may insert a random time delay before initiating the previously aborted membership update again.

4.5.2 Dealing with Network Partitions

In case of a network partition, multiple system subsets may consider that the other unreachable LTMs have failed, recover from their “failures” and carry on with processing the application workload independently from each other. However, this would violate the ACID properties and must therefore be avoided.

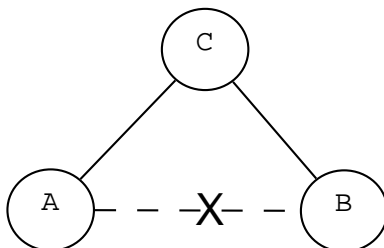


Figure 3: An example of unclean network partitions

For simplicity, we assume that no network partition occurs during the recovery of another partition. Supporting this latter case requires additional algorithms that we consider out of the scope of this article.

We use the Accessible Copies algorithm [18] to recover the system consistently during network partitions. This algorithm ensures that only one partition may access a given data item by allowing access to a given data item only within a partition that contains a majority of replicas. Instead of using a majority partition for each data item, we adapt the “majority” rule such that only the partition that contains more than half of the previous membership can access all data items. Minority partitions are forbidden access to any data item. It may happen that the majority partition lacks more than N LTM servers from the previous membership¹, and thus cannot recover all data items; in this case it rejects all incoming transactions until it can recover all data items.

Once a majority partition is established, it can recover all the ongoing transactions and accept new incoming transactions. After recovery, all LTMs in the majority partition have the new system membership with an increased timestamp. The other ones, which still have the previous membership timestamp, can detect after network partition recovery that they have been excluded from the membership, and rejoin as new members.

When an LTM discovers that other LTMs are unreachable because of LTM crashes and/or network partitions, it identifies its new partition membership through a 2PC across all LTMs. In the first phase, it sends an “invitation” to all LTMs; any responding LTM which vote “COMMIT” belongs to its partition membership. After all LTMs either respond or time-out, the second phase updates the partition membership of all LTMs in the partition of the coordinator. One optimization is to exclude the discovered unreachable LTMs from the first 2PC of building partition membership. This op-

¹Assuming that each transaction and data item has $N + 1$ replicas in total.

timization is effective for the scenario of LTM failures, avoiding a possible delay of time-out in waiting for responses from these failed LTMs. In case of network partitions, the first 2PC may fail to establish a majority partition to recover the system. LTMs should then include these excluded LTMs back into the following periodical 2PCs of building partition membership.

The above mechanism can organize the TPS into a number of disjointed partitions, provided that the network is “cleanly” partitioned: any two LTMs in the same partition can communicate, and any two LTMs in different partitions cannot. However, a network may also be “uncleanly” partitioned due to the lag of reconstructing routing tables. Figure 3 shows an example of unclean partition where each LTM has a different view of reachable LTMs: $\text{view}(A) = \{A,C\}$, $\text{view}(B) = \{B,C\}$ and $\text{view}(C) = \{A,B,C\}$. In this case, LTM C may join two different partitions: either $\{A,C\}$ or $\{B,C\}$, which both turn out to be majority partitions. To ensure that an LTM can only belong to one partition at a time, we define that if an LTM has already joined a partition, it will vote “ABORT” to any “invitation” of joining a different partition.

Minority partitions periodically try to rejoin the system by checking if previously unavailable nodes become reachable again. Receiving an “ABORT” vote for an “invitation” indicates that partitions are reconnected. In this case, the two partitions can be merged through a 2PC across all LTMs in the two partitions. The first phase is to push the memberships of two partitions to all the participant LTMs. A participant LTM votes “COMMIT” if the received membership matches its current latest partition membership. Otherwise, it votes “ABORT”. If an agreement to “COMMIT” is reached, the second phase updates the partition membership of all participant LTMs into the combined membership of two partitions. If any participant LTM votes “ABORT” or fails to respond, the 2PC is aborted.

5 System Implementation

This section discusses implementation details of CloudTPS, in particular how to support various cloud data storage services. We also present two optional optimizations: memory management to prevent memory overflow in the LTMs, and handling of read-only transactions containing complex read queries.

	SimpleDB	Bigtable	PNUTS
Data Item	Multi-value attribute	Multi-version with timestamp	Multi-version with timestamp
Schema	No schema	Column-families	Explicitly claimed attributes
Operation	Range queries on arbitrary attributes of a table	Single-table scan with various filtering conditions	Single-table scan with predicates
Consistency	Eventual consistency	Single-row transaction	Single-row transaction

Table 1: Key differences between cloud data services

5.1 Portability

CloudTPS relies on a cloud data storage service to ensure transaction durability. However, current cloud data storage services support different data models, consistency guarantees, operation semantics and interfaces. Adapting CloudTPS to all of them is a challenge. We compare three prominent and typical cloud data services: Amazon SimpleDB, Google Bigtable and Yahoo PNUTS. Our implementation is compatible with SimpleDB and Bigtable. Porting CloudTPS to other data services requires only minor adaptations.

SimpleDB, Bigtable and PNUTS have a number of similarities in their data models. They all organize application data into tables. A table is structured as a collection of data items with unique primary keys. The data items are described by attribute-value pairs. All attribute values are typed as strings. Data items in the same table can have different attributes. Data items are accessed with “GET/PUT” by primary key. Operations across tables, such as join queries, are not supported.

On the other hand, as shown in Table 1, the three cloud data services also have some key differences:

1. SimpleDB supports multiple values per attribute of a data item, while Bigtable and PNUTS only allow one. To be compatible with all of them, our data model allows only one value per attribute.
2. SimpleDB does not impose a predefined schema for its tables. PNUTS requires explicit claims of all attributes in a table, but it is still compatible with SimpleDB, as it does not require all records to have values for all claimed attributes and new attributes can be added any time without halting query or update activity. On the other hand, Bigtable

groups attributes into predefined column-families. To access an attribute, one should include its column-family name as its prefix. We address this difference by always prepending attribute names with the column-family name for Bigtable.

3. All three cloud data services support sophisticated data access operations within a table, but via different APIs. SimpleDB supports range queries inside a table with its specific language; Bigtable and PNUTS provide similar functionality with table scanning using various filtering conditions or predicates. This difference is irrelevant to the system design described before, as it accesses data items only by primary key. However, the optimization of read-only transactions, as described in Section 5.3, allows Web applications to access consistent data snapshots in cloud data services directly via their APIs. Therefore, the implementation of this optimization depends on the interface of the underlying cloud data service.
4. SimpleDB provides eventual consistency so that applications may read stale data. In contrast, Bigtable and PNUTS support single-row transactions, so they can guarantee returning the latest updates. We assume that when CloudTPS starts and loads a data item from the cloud data service for the first time, all the replicas of this data item are consistent. So CloudTPS can obtain the latest updates in this case, regardless of the consistency level of underlying cloud data service. However, this is not true for reloading a data item that has been recently updated. Different data consistency models of cloud data services require additional adaptations to implement our performance optimizations, as discussed in the following sections.

5.2 Memory Management

For efficiency reasons we keep all data in the main memory of the LTMs. However, maintaining a full copy of all application data may overflow the memory space, if the size of the data is large. One would thus have to allocate unnecessary LTM servers just for their memory space, rather than for their contributions to performance improvement. On the other hand, we notice that Web applications exhibit temporal data locality so that only a small portion of application data is accessed at any time [51, 50]. Keeping unused data in the LTMs is not necessary for maintaining ACID properties, so LTMs can evict these data items in case of memory shortage, and reload them from the cloud data service when necessary.

The key issue is that the eviction of any data items from LTMs must not violate the ACID properties of transactions. Obviously, the data items that are currently accessed by ongoing transactions must not be evicted until the transaction completes and the data updates have been checkpointed. After evicting a data item from the LTMs, future transactions may require it again. To guarantee strong consistency for these transactions, LTMs have to guarantee that the latest version of the evicted data items can be obtained from the cloud data service in the next read. The solution to this issue, however, depends on the consistency level guaranteed by the underlying cloud data service. To ensure that the latest version of a data item is visible, CloudTPS requires that the underlying cloud data service supports at least “Monotonic-reads” consistency [48]. If the data service provides the “Read-your-writes” consistency, checkpointing back the latest updates successfully is sufficient to evict a data item. For instance, Google Bigtable and Yahoo Pnuts support single-row transactions and thus provide “Read-your-writes” consistency. If the data service provides only eventual consistency, such as in SimpleDB, then LTMs may still obtain stale data even after a “GET” returned the latest version. To address this problem, we store the timestamps of the latest versions of all data items in LTMs, which can then determine if the newly loaded version of data item is up-to-date. If it is not, LTMs will abort the transactions and maintain ACID properties at the cost of rejecting these transactions.

Storing the latest timestamps of all data items in memory may also overflow the memory if the number of data items is extremely large. Storing them in the cloud data service is not an option, since they must maintain strong consistency. A simple solution could be to store them in the local hard drive of the LTM.

Another difficulty is that SimpleDB does not support multi-versions with timestamp, but multi-values for an attribute. We address this by attaching a timestamp at the end of the value of each attribute and so transform “multi-values” into “multi-versions”.

Another issue is to minimize the performance overhead of memory management, and carefully select which data items should be evicted. To maintain reasonable performance, we should maximize the hit rate of transactions in LTMs. Standard cache replacement algorithms, such as LRU, assume that data items have identical sizes. However, in CloudTPS, the data items have very different sizes, leading to a poor performance. So we adopt the cost-aware GreedyDual-Size (GDS) algorithm [12] which leverages knowledge of data item sizes to select data items to evict. The GDS algorithm associates a value H to each data item p : $H(p) = L + cost/size$, where L is the H

value of the latest evicted data item. We set the *cost* parameter to 1 for all data items as this optimizes hit rate. The parameter *size* refers to the size of data item p . Each time that an LTM needs to replace a data item, it selects the data item with the lowest H value and updates its L value to the H value of this evicted data item. When a data item is accessed, the H value of this data item is recalculated with the updated parameters: the latest L value and its possibly changed *size*.

5.3 ReadOnly Transactions

CloudTPS 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 as read-write transactions, but also the same constraint: accessing well identified data items by primary key only. However, CloudTPS provides an additional feature for read-only transactions to support complex read queries such as range queries.

We exploit the fact that many read queries can produce useful results by accessing a possibly stale 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 may not be necessary to compute the result based on the absolute most recent orders. We therefore introduce the concept of Weakly-Consistent Read-only Transaction (WCRT): A WCRT contains any number of read operations offered by the cloud data service, e.g., table scan for Bigtable. Web applications issue WCRTs directly to the cloud data service, bypassing the LTMs. All read operations of a WCRT executes on the same internally consistent but possibly slightly outdated snapshot of the database.

To implement WCRTs, we introduce a snapshot mechanism in the checkpoint process of LTMs, which marks each data update with a specific snapshot ID that is monotonically increasing. This ID is used as the version number of the newly 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 group transactions in sets of M transactions with subsequent timestamps, of which each group constitutes 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. At the finest granularity, with $M = 1$, each read-write transaction creates a new snapshot.

The key issue in this snapshot mechanism is to determine whether a consistent snapshot is fully available in the cloud data service such that WCRTs can execute on it. A consistent snapshot contains all the updates of the transactions which it reflects. It is fully available only after all these updates have been checkpointed back. The main difficulty is that a transaction may update data items across multiple LTMs, where each LTM performs checkpoints for its own data items independently from the others. Therefore, CloudTPS must collect checkpoint progress information from multiple LTMs. To address this issue, we use the cloud data service as a shared medium for collecting checkpoint progress information. The system creates an extra table named “Checkpoint”, where each LTM writes its latest completed snapshot ID into a separate data item using its membership ID as the primary key value. So the minimal snapshot ID stored in the “Checkpoint” table represents the latest snapshot of which the updates are all checkpointed.

Even though all the updates of a snapshot have been checkpointed successfully, the availability of this snapshot still depends on the consistency level provided by the cloud data service. The data services must provide at least “Monotonic-reads” consistency, so that LTMs can verify the visibility of the updates before claiming the snapshot is available. Bigtable and PNUTs support single-row transactions and thus provide the “Read-Your-Writes” consistency. Therefore, the snapshot is immediately available after writing all checkpoints back. Lastly, Amazon SimpleDB only supports eventual consistency, so it is impossible to guarantee the visibility of certain writes in the next read. Therefore, this feature cannot be applicable for cloud data services with eventual consistency only.

6 Evaluation

We demonstrate the scalability of CloudTPS by presenting the performance evaluation of a prototype implementation on top of two different families of scalable data layers: HBase running in our local DAS-3 cluster [23] and SimpleDB running in the Amazon Cloud. We then show that CloudTPS can recover from LTM failures and network partitions efficiently by presenting the throughput of CloudTPS under these failures. Lastly, we demonstrate the effectiveness of the memory management mechanism and discuss the tradeoff between system performance and buffer sizes.

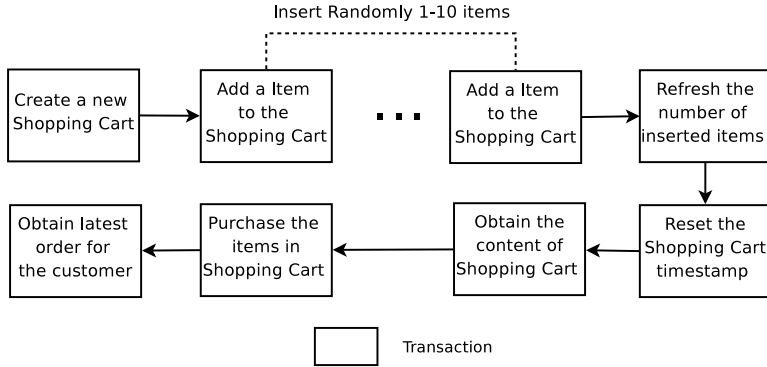


Figure 4: Workflow of transactions issued by each Emulated Browser (EB) of TPC-W

We evaluate our prototype under a workload derived from TPC-W [35]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com.

6.1 Migration of TPC-W to the Cloud

TPC-W is originally designed as a Web application using a SQL-based relational database as backend. However, CloudTPS is based on HBase and SimpleDB. Both of them use a different data model other than the relational data model, and do not support the SQL language. We therefore need to adapt the original relational data model of TPC-W into the data models of BigTable and SimpleDB. As described in Section 5.1, we can easily adapt the Bigtable data model into SimpleDB data model by using the exact same attribute names, which are prepended with the column family names. Therefore, we first adapt the relational data model of TPC-W into the Bigtable data model.

Using similar data denormalization techniques as in [53], we design a Bigtable data model for TPC-W that contains the data accessed by the transactions in Figure 4. 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 tables (“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 can thus have a bigtable named “Item_Stock” which only contains this column and has the same primary key. Finally, for the last transaction in Figure 4 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.

For both HBase and SimpleDB, 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. As shown in Figure 4, the workload continuously creates new shopping carts. Thus, the size of the “Shopping” bigtable increases continuously during the evaluation, while the other two bigtables remain constant in size. In the evaluation of the memory management mechanism, we also evaluate the system performance with 1,000,000 item records in the “Item_Stock” bigtable.

In the performance evaluation based on HBase, we observed a load balancing problem. TPC-W assigns new shopping cart IDs sequentially. However, 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 to subtables in round-robin fashion.

As for Amazon SimpleDB, we organize application data into a number of domains (i.e., tables), while each domain can only sustain a limited amount of update workload. So we also have to horizontally partition a table in round-robin fashion and place each partition into a domain. However, different from HBase, we can use at most 100 domains for the whole application. We therefore partition the three tables into different number of sub-tables according to our estimated data access loads. We horizontally partition the “Shopping” bigtable into 80 domains and the other two bigtables into 5 domains each. This way SimpleDB can provide sufficient capacity for both writes and reads, while CloudTPS remains the performance bottleneck for performance evaluation.

6.2 Experiment Setup

We perform the scalability evaluation on top of two different scalable data layers: 1) HBase v0.2.1 [23] running in the DAS-3 cluster [16] in our uni-

versity; 2) SimpleDB in the Amazon Cloud [3]. In both platforms, we use Tomcat v5.5.20 as application server. The LTMs and load generators are deployed in separate application servers.

DAS-3 is an 85-node Linux-based server cluster. 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 Gigabit LAN such that the network latency between the servers is negligible.

Amazon EC2 offers various types of virtual machine instances, which may share the resources of a same physical machine with other instances. As of September 2009, there are two families of instance types: Standard and High-CPU. We perform scalability evaluation with both families. We choose the Small Instance in the Standard family, which has 1.7 GB memory, 1 virtual core with 1 EC2 Compute Unit, and 160 GB instance storage. At the time of our experiment, Standard Small instances cost \$0.10 per instance-hour. Besides, we also select Medium Instance in the High-CPU family, which has 1.7 GB of memory, 2 virtual cores with 2.5 EC2 Compute Units each, and 350 GB of instance storage. High-CPU Medium instances cost \$0.20 per instance-hour. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

TPC-W workload is generated by a configurable number of Emulated Browsers (EBs) which issue requests from one simulated user. Our evaluations assume that the application load remains roughly constant. The workload that an Emulated Browser issues to the TPS mainly consists of read-write transactions that require strong data consistency. Figure 4 shows the workflow of transactions issued by an Emulated Browser, which simulates a typical customer shopping process. Each EB waits for 500 milliseconds on average between receiving a response and issuing the next transaction.

6.3 Scalability Evaluation

We study the scalability of CloudTPS in terms of maximum sustainable throughput under a response time constraint. We perform the scalability evaluation in both DAS-3 and the Amazon cloud. In DAS-3, we assign a physical machine for each LTM, and have low contention on other resources such as network. Therefore, for the evaluations in DAS-3, we define a demanding response time constraint that imposes that the 99% of transactions must return within 100 ms. On the other hand, in the public Amazon cloud, our LTMs have to share a physical machine with other instances, and we have less control of the resources such as CPU, memory, network, etc. Fur-

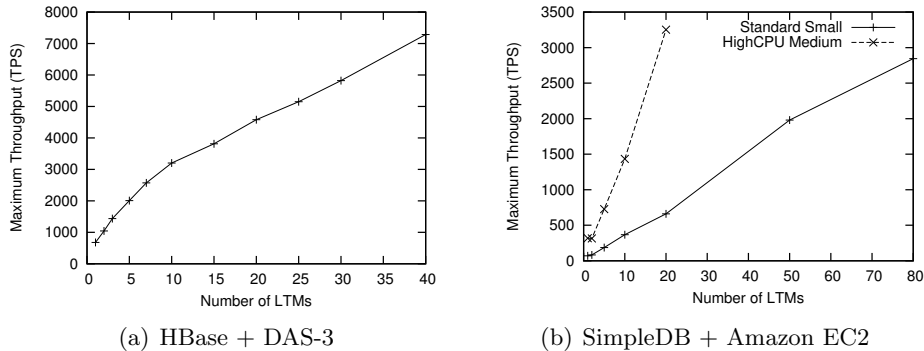


Figure 5: Maximum system throughput under a response time constraint

thermore, even multiple instances of the exact same type may exhibit different performance behavior [17]. Therefore, to prevent these interferences from disturbing our evaluation results, we relax the response time constraint for the evaluations in the Amazon cloud: 90% of transactions must return within 100 ms.

We perform the scalability evaluation by measuring the maximum sustainable throughput of the system consisting of a given number of LTMs before the constraint gets violated. In DAS-3, we start with one LTM and 5 HBase servers, then add more LTM and HBase servers. We carry out each round of the experiment for 30 minutes to measure the performance of system under a certain number of EBs. In all cases, we deliberately over-allocated the number of HBase servers and client machines to make sure that CloudTPS remains the performance bottleneck. We perform similar steps in the Amazon cloud by starting with one LTM. CloudTPS remains the performance bottleneck, as SimpleDB can provide sufficient capacity for both writes and reads. We configure the system so that each transaction and data item has one backup in total, and set the checkpoint interval to 1 second.

Figure 5(a) shows that CloudTPS scales nearly linearly in DAS-3. 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, and 1 server as global timestamp manager. 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.

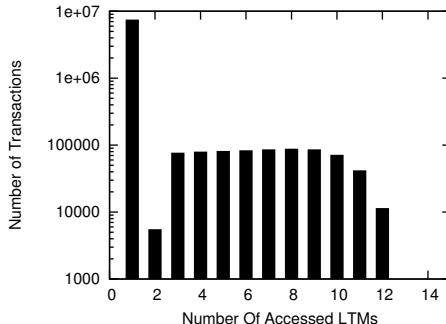


Figure 6: Number of LTMs accessed by the transactions of TPC-W

Figure 5(b) shows the scalability evaluation in the Amazon cloud. Here as well, CloudTPS scales nearly linearly with both types of EC2 virtual instances. When using 80 “Standard Small” instances, CloudTPS reaches a maximum throughput of 2844 transactions per second generated by 1600 emulated browsers. The maximum throughput of the system at that point is approximately 40 times that of a single LTM server. When using 20 “High-CPU” Medium instances, CloudTPS reaches a maximum throughput of 3251 transactions per second generated by 1800 emulated browsers. This is a 10-fold improvement compared to one LTM.

Furthermore, we explore the cost-effectiveness of the two EC2 instance types for CloudTPS. The “High-CPU medium” instances cost 2 times more than “Standard Small” instances. As show in Figure 5(b), 20 “High-CPU medium” instances, which together cost \$4 per hour, can sustain a higher maximum throughput than 80 “Standard Small” instances, which together cost \$8 per hour. Therefore, for this application, using “High-CPU medium” instances is more cost-effective than “Standard Small” instances.

The linear scalability of CloudTPS relies on the property that transactions issued by Web applications only access a small number of data items, and thus span only a small number of LTMs. We illustrate this property by measuring the number of LTMs that participate in the transactions with the configuration of 40 LTMs servers. As shown in Figure 6, the vast majority of transactions access only one LTM². 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. The purchase transaction in Figure 4 is the only transaction that accesses more than 1 data item.

²The LTMs that act only as backup of transactions or data items are not counted in.

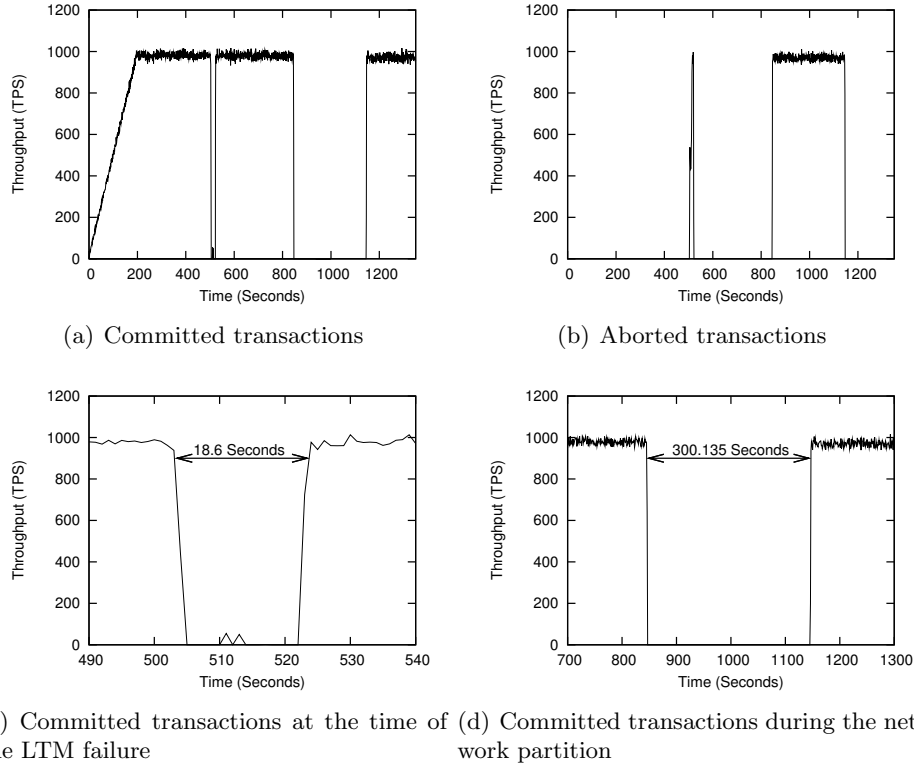
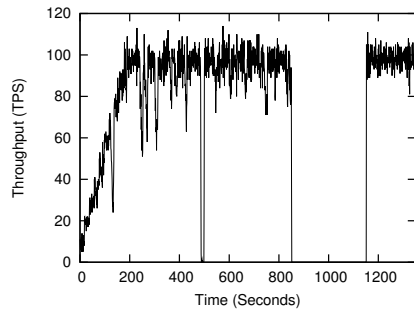
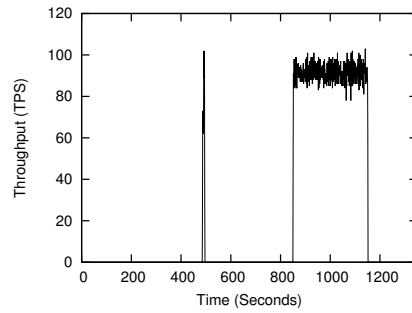


Figure 7: Throughput of system in the presence of LTM server failures in DAS-3

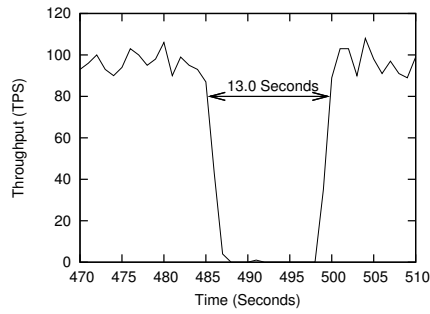
It first creates an order and clears the shopping cart inside the data item of the “Shopping” bigtable, then updates the stocks of all purchased items in the “Item_Stock” bigtable, and lastly updates the latest order ID of the customer in the “Latest_Order” bigtable. As the number of items contained in a shopping cart is uniformly distributed between 1 and 10, the number of data items accessed by a purchase transactions also has an uniform distribution between 3 and 12. Figure 6 shows that fewer transactions access 11 or 12 LTMs compared to the ones accessing 3 to 10 LTMs, and some transactions even access 2 LTMs. This is because the accessed data items may be located within the same LTM, so the number of accessed LTMs may be smaller than the number of accessed data items.



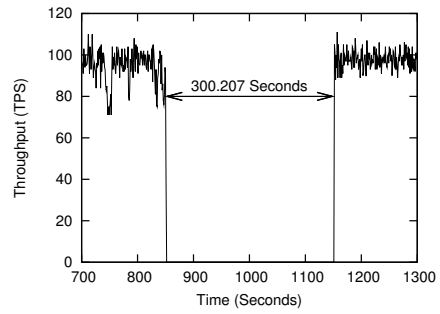
(a) Committed transactions



(b) Aborted transactions



(c) Committed transactions during the LTM failure



(d) Committed transactions during the network partition

Figure 8: Throughput of system in the presence of LTM server failures in the Amazon Cloud

6.4 Tolerance of LTM Failures and Network Partitions

We now study the system performance in the presence of LTM server failures and network partitions. We perform the evaluation in both DAS-3 and the Amazon Cloud. We configure CloudTPS with 5 LTM servers, and each transaction and data item has one backup. We generate a workload using 500 EBs in DAS-3 and 50 EBs in the Amazon cloud, such that the system would not overload even after an LTM server failure. After the system throughput is stabilized, we first kill one LTM server. Several minutes later, we simulate a 5-minutes-long network partition where each partition contains one LTM server.

After detecting the failures, all alive LTMs continuously attempt to contact with other LTMs. The time delay between two attempts of contact follows a uniform distribution between 200 and 1200 milli-seconds.

Figure 7 illustrates the evaluation in DAS-3. 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 at time 504 seconds. Figure 7(a) shows that the system recovers from the failure and the transaction throughput returns to the previous level. At the same time, as shown in Figure 7(b), a few transactions are aborted because the incoming transactions that accessed the failed LTM must be rejected during the recovery process. Afterwards, at time 846 seconds, we simulate a network partition lasting for 5 minutes. When we restore the network partition, the system recovers and returns to the previous level of transaction throughput.

Figure 7(c) shows the performance details of the recovery processes of the LTM failure. After the LTM failure it takes 18.6 seconds for the system to return to the previous level of transaction throughput. This duration is composed of:

- 0.5 second to rebuild a new membership: including a delay of 382 milli-seconds before the 2PC to avoid conflicts and then build the new membership in 113 milli-seconds;
- 12.2 seconds to recover the blocked transactions which were accessing the failed LTM;
- 5.9 seconds to reorganize the data placement of LTMs to match the new system membership.

At times 511 and 513 seconds, LTMs recover respectively 55 and 50 blocked transactions. However there is still one last blocked transaction left, which

is recovered only at time 517 seconds. The most time of recovering blocked transactions is spent in waiting for a time-out exception which detects that an LTM has failed. In our implementation each transaction detects such failures independently, which explains why transactions recover at different times.

Figure 7(d) shows the system throughput during the network partition which occurs from time 846 to 1146 seconds, lasting for 300 seconds. As shown in Figure 7(d), after the end of network partition, the system recovers in 135 milliseconds and returns to the previous level of transaction throughput. The reason why the system recovers so fast is because there is no LTM failure along with the network partition, so all blocked transactions can resume execution without recovery, and no data redistribution is necessary.

Figure 8 depicts the same evaluation in the Amazon cloud. Similar to the evaluation result in DAS-3, Figure 8(a) shows that the system recovers from the LTM server failures and the network partition, and Figure 8(b) shows that transactions are aborted during system recovery. Figure 8(c) shows the performance detail of recovering the LTM failure: the LTM server fails at time 486 seconds, after detecting the LTM failure, the system spends 13 seconds to recover and the transaction throughput returns to the previous level at time 499 seconds. During the failure recovery, the remaining 4 LTMs firstly merge into one partition in about 1 second. Then the system recovers transactions in 4 seconds and reorganizes data placement in 8 seconds. After the LTM failure, the system encounters a network partition lasting for 5 minutes. Figure 8(d) shows the detail of recovering from the network partition, which starts from time 851 seconds to 1151 seconds. No LTM failure happens along with the network partition. After the network partition is restored, the system recovers in 207 milliseconds and returns to the previous level of transaction throughput at time 1152 seconds. Note that the system throughput in the Amazon cloud fluctuates more than in DAS-3 because we have less control of virtualized resources in the Amazon cloud.

6.5 Memory Management

Lastly, we demonstrate that our memory management mechanism can effectively prevent LTMs from memory overflow. Furthermore, we study the performance of CloudTPS with different buffer sizes and data sizes. We carry out the evaluation in DAS-3 on top of HBase, which provides “Read-your-writes” consistency just like Bigtable. We configure the system such that before evicting a data item, LTMs obtain the data item from HBase

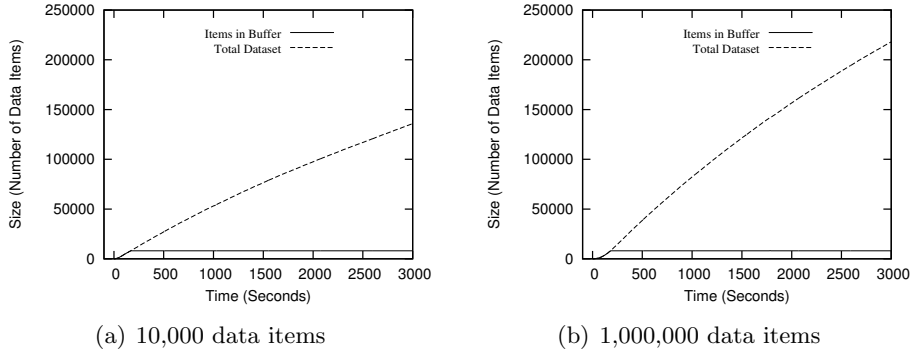


Figure 9: Buffer size vs. total accessed data items of LTM #1

and verify that the obtained value reflects the latest in-memory updates. Therefore, this performance evaluation represents the system implementation for the cloud data services supporting “Monotonic-reads” consistency level.

We first deploy a system with 3 LTMs and impose a constant workload for one hour. We configure the system so that each LTM can maintain at most 8000 data items in its buffer. We then evaluate the system under two different scales of data set sizes: either 10,000 or 1,000,000 records in the “Item_Stock” table. For the data size of 10,000 items, we impose a workload of 500 EBs. For the data size of 1,000,000 items, we impose a workload of 250 EBs.

Figure 9(a) and Figure 9(b) show that under both data set sizes, our mechanism effectively maintains the buffer size of LTM #1 within the limit of 8000 data items. As shown in the Figure 9(a), using 10,000 items, without memory management, after an hour, this LTM would have to maintain almost 140,000 data items in memory. As for the data size of 1,000,000 items, Figure 9(b) shows that, after an hour, the size of total data set increases to an even larger number of more than 200,000 data items. In both cases, without memory management, the size of the total accessed data set increases almost linearly, which would eventually cause a memory overflow.

We then compare the performance of the system under different data set sizes. Figure 10 shows that the hit rate of LTM #1 stabilizes around 90% for 10,000 items, and about 60% for 1 million items. The other LTMs in the system behave similarly. Figure 11(a) and Figure 11(b) show the total transaction throughput of the system during the 1 hour evaluation. The

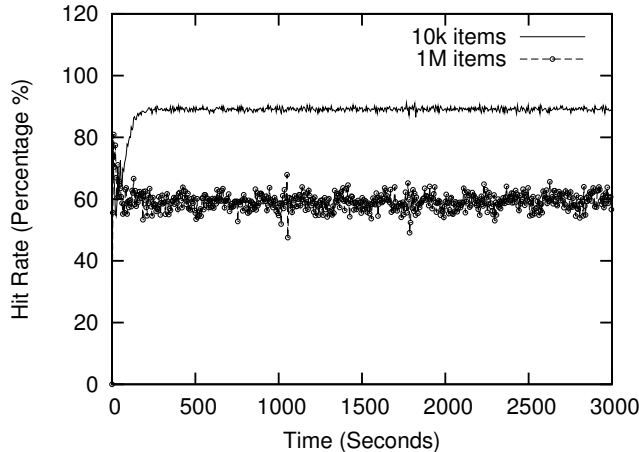


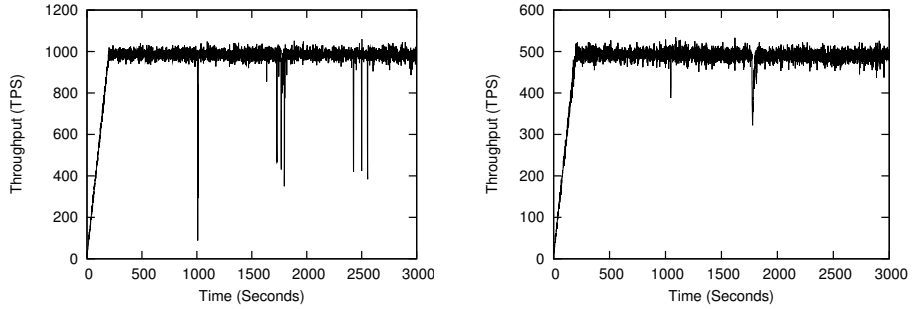
Figure 10: Hit rate of LTM #1

drops of throughput at some points are due to the JVM garbage collection, which temporarily block the LTMs. With 10,000 items, the system sustains a transaction throughput of about 1000 TPS and 99.44% of transactions complete within 100 ms. For 1 million items, the system sustains about 500 TPS, but only 97.02% of transactions satisfy the performance constraint.

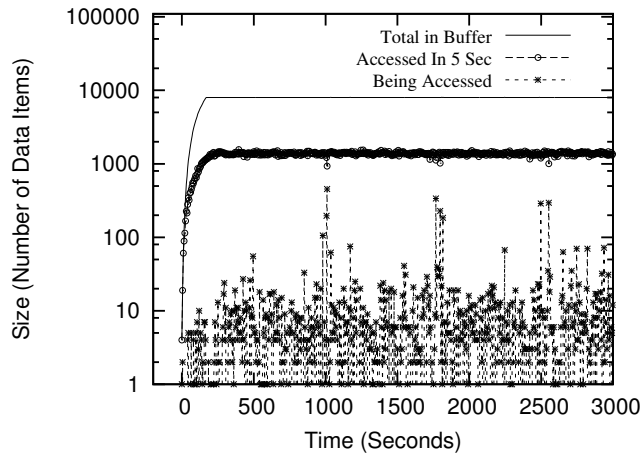
The efficiency of our memory management mechanism depends on the data locality of the Web application. Figures 11(c) and 11(d) show that only very few data items are being accessed at a time in the two different scenarios. Note that Figures 11(c) and 11(d) are in log scale. Figure 11(c) shows that less than 2000 distinct data items are accessed in each 5 seconds interval. Comparing to the total accessed data items shown in Figure 9(a), this application shows strong data locality which implies that our mechanism can only introduce minor performance overhead³.

Finally, we study the system performance with different buffer sizes of LTMs in terms of 99th percentile of response times of the system. We configure the system with 20 LTMs and impose a workload of 2400 EBs, which issues about 4800 transactions per second. We start with the minimum buffer size required by LTMs to maintain the ACID properties, where only the absolutely necessary data items remain in the buffer. To achieve this, we

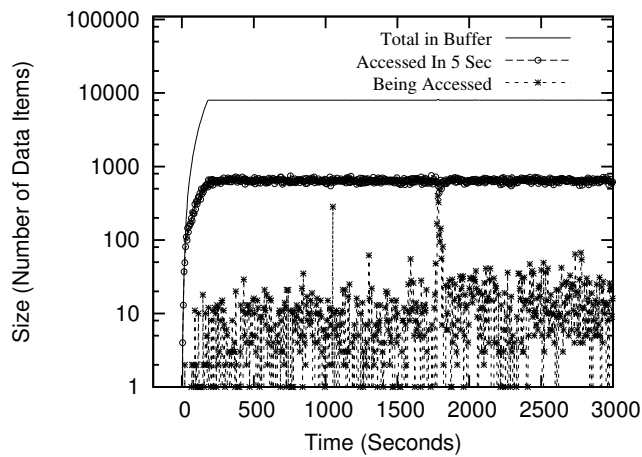
³Note that TPC-W randomly selects books to add into a shopping cart with uniform distribution. Several works consider that this behavior is not representative of real applications and create extra locality artificially [45, 5, 4]. We can therefore consider unmodified TPC-W as a somewhat worst-case scenario.



(a) Total system throughput with 10,000 items (b) Total system throughput with 1,000,000 items



(c) Data locality of LTM #1 with 10,000 items



(d) Data locality of LTM #1 with 1,000,000 items

Figure 11: Memory management evaluation

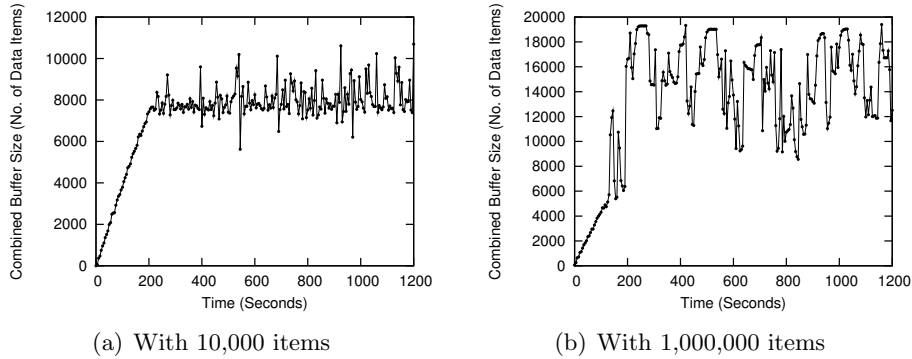


Figure 12: The minimum required overall buffer size of total 20 LTMs under the load of 2400 EBs

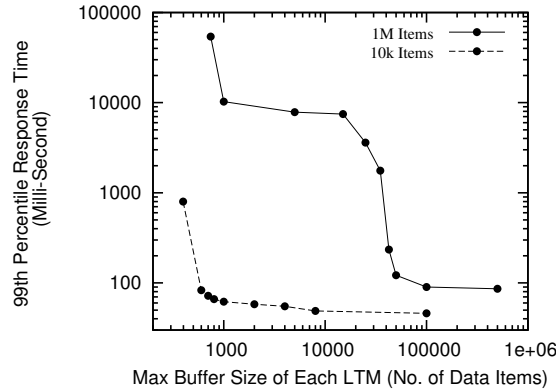


Figure 13: Response time with different buffer sizes

implement an “Evict-Now” algorithm that evicts any evictable data items as soon as possible. We then increase the buffer size until no data item is evicted during the evaluation. Similar to the previous evaluation, we evaluate the system performance with 10,000 and 1 million records in the “Item_Stock” table.

Figure 12(a) shows the combined buffer size of all LTMs when applying the “Evict-Now” algorithm. For the data size of 10,000 items, the average buffer size is 7957 data items, which means 397 data items per LTM. For the data size of 1 million items, the average buffer size is 14837, so 741 data items per LTM. Figure 13 shows performance of our system under different buffer sizes. The initial value of each line in Figure 13 indicates the 99th percentile of response times of the system using “Evict-Now” algorithm. Therefore,

We adopt 397 and 741 as the initial values for the X-axis in Figure 13. Note that we plot Figure 13 in log scale.

We first study the 99th percentile of response times with 10,000 items. As shown in Figure 13, when we increase the buffer size from the minimum size of 397 to 1000 data items per LTM, the 99th percentile response time decreases dramatically from 799 ms to 62 ms. When we continue increasing the buffer size to 100,000 data items where no data items have to be evicted, the 99th percentile response time only improves to 46 ms. In other words, increasing the buffer size from 397 to 1000 data items, the response time of the system decreases by an order of magnitude. Increasing the buffer size even further by two orders of magnitude to 100,000 data items can only achieve 25% reduction of response time. It is because that at the point of 1000 data items, the overall buffer size of the system reaches 20,000 data items, which is already large enough to contain almost all 10,000 “item_stock” data items and other currently accessed data items from other two tables. Increasing the buffer size even further can only allow to store more seldomly accessed data items, and thus cannot effectively improve the hit rate of the system.

We then study the performance result with 1 million items. As shown in Figure 13, in the beginning, similar to the result with 10,000 items, the 99th percentile of response times decreases dramatically from 54 seconds to 10 seconds, when we increase the buffer size from the minimum size of 741 to 1000 data items for each LTM. However, increasing the buffer space from 1000 to 15,000 does not bring much performance improvement, because the total data size is so large that the hit rate remains roughly the same. If we continue increasing the available storage from 15,000 to 100,000, the 99th percentile of response times decreases dramatically from 7486 ms to 90 ms. After the point of 100,000 data items, continue increasing the buffer size further does not bring significant performance improvement.

Comparing the two lines in Figure 13, we notice that a good buffer size for 10,000 items could be 1000 data items. For the line of “1M items”, we can find a similar point of 100,000 data items. In both cases, this represents about 10% of the total data set size.

7 Conclusion

Many Web applications need strong data consistency for their correct execution. 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 relatively weak consistency properties. This

article 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 and network partitions.

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 CloudTPS linear scalability. CloudTPS supports full ACID properties even in the presence of server failures and network partitions. Recovering from a failure only causes a temporary drop in throughput and a few aborted transactions. Recovering from a network partition, however, may possibly cause temporary unavailability of CloudTPS, as we explicitly choose to maintain strong consistency over availability. Our memory management mechanism can prevent LTM memory overflow. We expect typical Web applications exhibit strong data locality and our mechanism will only introduce minor performance overhead. 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.

CloudTPS allows Web applications with strong data consistency demands to be scalably deployed in the cloud. This means that Web applications in the cloud do not need any more to compromise consistency for scalability.

References

- [1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of Symposium on Operating Systems Principles*, pages 159–174, 2007.
- [2] Amazon.com. Amazon SimpleDB., 2010. <http://aws.amazon.com/simpledb>.
- [3] Amazon.com. EC2 elastic compute cloud, 2010. <http://aws.amazon.com/ec2>.
- [4] Khalil Amiri, Sanghyun Park, and Renu Tewari. DBProxy: a dynamic data cache for web applications. In *Proceedings of International Conference on Data Engineering*, pages 821–831, 2003.

- [5] Christopher Olston Amit, Amit Manjhi, Charles Garrod, Anastassia Ailamaki, Bruce M. Maggs, and Todd C. Mowry. A scalability service for dynamic web applications. In *Proceedings of Conference on Innovative Data Systems Research*, pages 56–69, 2005.
- [6] Mark Atwood. A MySQL storage engine for AWS S3. In *MySQL Conference and Expo*, 2007. <http://fallenpegasus.com/code/mysql-awss3/>.
- [7] Philip A. Bernstein and Nathan Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th international conference on Very Large Data Bases*, pages 285–300, 1980.
- [8] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.
- [9] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.
- [10] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, 2008.
- [11] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 251–264, 2008.
- [12] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, page 18, 1997.
- [13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : a distributed storage system for structured data. In *Proceedings of The 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.
- [14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel

- Weaver, and Ramana Yerneni. PNUTS: Yahoo!’s hosted data serving platform. In *Proceedings of the 34th International Conference on Very Large Data Bases*, pages 1277–1288, 2008.
- [15] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic transactional data store in the cloud. In *Workshop on Hot Topics in Cloud Computing*, 2009.
- [16] DAS3. The Distributed ASCII Supercomputer 3, 2007. <http://www.cs.vu.nl/das3/>.
- [17] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *The 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, 2009.
- [18] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th symposium on Principles of Database Systems*, pages 215–229, 1985.
- [19] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
- [20] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [21] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 486–497, 1997.
- [22] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, July 2008.
- [23] HBase. An open-source, distributed, column-oriented store modeled after the Google Bigtable paper, 2006. <http://hadoop.apache.org/hbase/>.
- [24] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th annual symposium on Principles of distributed computing*, pages 92–101, 2003.

- [25] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The clustra telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [26] JJ Furman, Jonas S Karlsson, Jean Michel Leon, Steve Newman, Alex Lloyd, and Philip Zeyliger. Megastore: A Scalable Data System for User Facing Applications. *Proceedings of the ACM SIGMOD international conference on Management of Data*, 2008.
- [27] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. In *Proceedings of the 34th international conference on Very large data bases*, pages 1496–1499, 2008.
- [28] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of ACM Symposium on Theory of Computing*, pages 654–663, 1997.
- [29] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, 2000.
- [30] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A Software Transactional Memory Framework for Clusters. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 51–58, 2008.
- [31] Avinash Lakshman, Prashant Malik, and Karthik Ranganathan. Cassandra: A structured storage system on a P2P network. In *Keynote talk at the ACM SIGMOD international conference on Management of Data*, 2008.
- [32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

- [33] M. L. Liu, D. Agrawal, and A. El Abbadi. The performance of two phase commit protocols in the presence of site failures. *Distributed Parallel Databases*, 6(2):157–182, 1998.
- [34] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, 2006.
- [35] Daniel A. Menascé. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3), 2002.
- [36] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of ACM symposium on Principles of distributed computing*, pages 267–275, 1996.
- [37] Microsoft.com. Microsoft SQL Azure Database., 2010. <http://www.microsoft.com/azure/data.aspx>.
- [38] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [39] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 2nd edition, February 1999.
- [40] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. In *Proceedings of International Workshop on Distributed Systems: Operations and Management*, pages 256–267, 2007.
- [41] Mikael Ronström and Lars Thalmann. MySQL cluster architecture overview. MySQL Technical White Paper, April 2004.
- [42] J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):1–17, 1980.
- [43] George Samaras, Kathryn Britton, Citron andrew, and C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distributed Parallel Databases*, 3(4):325–360, 1995.

- [44] Gunter Schlageter. Optimistic methods for concurrency control in distributed database systems. In *Proceedings of the 7th international conference on Very Large Data Bases*, pages 125–130, 1981.
- [45] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, January–February 2007.
- [46] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.
- [47] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it’s time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160, 2007.
- [48] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd international conference on Parallel and distributed information systems*, pages 140–150, 1994.
- [49] Transaction Processing Performance Council. TPC benchmark C standard specification, revision 5, December 2006. <http://www.tpc.org/tpcc/>.
- [50] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009.
- [51] Werner Vogels. Data access patterns in the amazon.com technology platform. In *Proceedings of the 33rd international conference on Very large Data Bases, Keynote Speech*, page 1, 2007.
- [52] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.
- [53] Zhou Wei, Jiang Dejun, Guillaume Pierre, Chi-Hung Chi, and Maarten van Steen. Service-oriented data denormalization for scalable web applications. In *Proceeding of the 17th international conference on World Wide Web*, pages 267–276, 2008.

- [54] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Scalable transactions for web applications in the cloud. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 442–453, August 2009.