Consistent Join Queries in Cloud Data Stores

Zhou Wei[*,†] Guillaume Pierre[*]

Chi-Hung Chi[†]

[*] Vrije Universiteit, Amsterdam, The Netherlands

[†] Tsinghua University, Bejing, China

Technical report IR-CS-68, Vrije Universiteit, January 2011.

Abstract

NoSQL Cloud data stores provide scalability and high availability properties for web applications, but do not support complex queries such as joins. Developers must therefore design their programs according to the peculiarities of NoSQL data stores rather than established software engineering practice. This results in complex and error-prone code, especially when it comes to subtle issues such as data consistency under concurrent read/write queries. CloudTPS implements support for join queries and strongly consistent multi-item read-write transactions in a middleware layer which stands between the Web application and its data store. CloudTPS supports the two main families of scalable data layers: Bigtable and SimpleDB. Performance evaluations show that our system scales linearly under a demanding workload composed of join queries and read-write transactions.

Keywords: Scalability, web applications, cloud computing, join queries, secondary-key queries, NoSQL.

Contents

1 Introduction

Non-relational Cloud databases such as Google's Bigtable [5], Amazon's SimpleDB [1] and Facebook's Cassandra [17] are at the heart of many famous Websites. The increasing popularity of these Cloud data stores (also often called NoSQL databases) originate in their near-infinite scalability properties: one can always accommodate higher workloads by simply adding extra hardware. This good property however comes at a cost: Cloud data stores only support very simple types of queries which select data records from a single table by their primary keys. More complex queries such as joins and secondary-key queries are not supported.

The success of Cloud data stores demonstrates that one can program a wide range of useful applications using this limited query API. For example, a join query can often be rewritten into a sequence of primary-key queries. However, such translation is not a trivial task at all. First, one must design data schemas carefully to allow such query rewrite. Second, and more importantly, programmers need sufficient understanding of subtle concurrency issues to realize and handle the fact that a sequence of simple queries is equivalent to the original join query only in the case where no update of the same data items is issued at the same time. Although skilled programmers can effectively develop good applications using this data model, we consider that program correctness should not be an optional feature left under the sole responsibility of the programmers. Correctness should as much as possible be provided out of the box, similar to the idiotproof strong consistency properties of relational databases.

This paper aims to support join queries that are strongly consistent by design, relieving programmers from the burden of adapting their programs to the peculiarities of NoSQL data stores. At the same time we must retain the good scalability properties of the cloud data stores. We implement join queries in a middleware layer which sits between the Web application and the Cloud data store. This layer, called CloudTPS, is in charge of enforcing strict ACID transactional consistency on the data, even in the case of server failures and network partitions. We presented the transactional functionalities of CloudTPS in a previous publication [26]. However, in this early work, join queries were not supported. This paper discusses CloudTPS's support for consistent join queries, while retaining the original scalability and fault-tolerance properties of the underlying Cloud data store.

CloudTPS supports a specific type of join queries known as foreign-key equi-joins. These queries start by extracting one or more database records by their primary keys, then join with other tables by following references to other primary keys found in the first record. A typical foreign-key equi-join is "SELECT * FROM book, author WHERE book.id = 10 AND book.a_id = author.id." This query first retrieves one record of table "book" by its input primary key, then uses attribute "a_id" to identify the related record in table "author".

Note that support for this family of join queries also allows us to implement secondary-key queries: CloudTPS only needs to maintain a separate index table that maps secondary key values back to their corresponding primary keys.

Secondary-key queries are then translated into equivalent join queries. When the main table is updated, its associated index table must be atomically updated as well.

The scalability properties of CloudTPS originate in the fact that most queries issued by Web applications (including transactions and join queries) actually access a small number of data items compared with the overall size of the database. This property is verified in all real-world Web applications that we studied: because database queries are embedded in the processing of an end-user HTTP request, programmers tend to naturally avoid complex and expensive queries which would for example scan the entire database.

We demonstrate the performance and scalability of CloudTPS using a realistic workload composed of primary-key queries, join queries, and transactions issued by the TPC-W Web hosting benchmark. This benchmark was originally developed for relational databases and therefore contains a mix of simple and complex queries similar to the way Web applications would exercise their NoSQL data store if they were given the opportunity. We show that, with no change of the initial relational data schema nor the queries addressed to it, CloudTPS achieves linear scalability while enforcing database correctness automatically.

2 Related Work

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, including support for sophisticated join queries. 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 full data replication techniques and therefore do not bring extra scalability improvement compared to a non-cloud deployment [16, 2].

Cloud data stores, such as Google Bigtable [5], Amazon SimpleDB [1], Yahoo PNUTS [6], and Cassandra [17], are being praised for their scalability and high availability properties. They achieve these properties by using simplified data models based on attribute-value pairs, and by supporting only very restrictive types of queries to retrieve data items by their unique key. This allows data stores to automatically partition application data, which is the key for performance scalability.

Cloud data stores however also receive heavy criticism for the demands they put on programmers to manualy handle consistency issues (see for example [13]). Recently, a number of systems appeared to support strongly-consistent multi-item ACID transactions: Percolator [21], Megastore [3], Deuteronomy [18], G-Store [10], Scalaris [22] and ecStore [25], each with their own focus. Percolator focuses on incremental processing of massive data processing tasks. Megastore supports transactional consistency within fine-grained partitions of data, but only limited consistency guarantees across them. Deuteronomy operates over a wide range of heterogeneous data sources. G-Store proposes key grouping

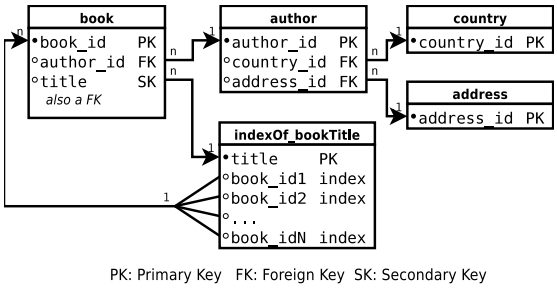


Figure 1: An example data model for CloudTPS

protocols, allowing for transactions to run within pre-defined groups. Scalaris is a DHT which uses Paxos to support transactions across any number of key-value pairs. ecStore supports range queries. However, none of these systems discusses support of complex queries such as join queries.

H-Store [24, 14] is a distributed main memory OLTP database. A salient feature of H-Store is that it implements the SQL language, and therefore supports join queries. However, H-Store's scalability relies on careful data partition across executor nodes, such that most transactions access only one executor node. H-Store does not support join queries which span multiple data partitions.

Similar to H-Store, ElasTraS automatically partitions data across a number of database nodes, and supports complex queries within each partition [9, 8]. Schism analyzes a query log to propose a data placement which minimizes the number of partitions involved in transactions [7]. These approaches are interesting in that they reduce the number of servers that need to be involved in any particular transaction. As such, they are complementary to the work on CloudTPS. However, they do not address the specific problem of join queries where the list of nodes which take part in a query can only be found while executing the transaction.

3 Database Model

Join queries, in the most general case, can range from simple operations matching two records from different tables to very complex data-mining queries which scan the entire database for hours. CloudTPS focuses on the most common types of joins only. We now detail the CloudTPS data model, the type of join queries it supports, and the way they are expressed by programmers in our system.

3.1 Data Model

Different Cloud data stores employ similar yet different data models to define how the data are organized. However, one goal of CloudTPS is to offer consistent join queries for a wide range of data stores indifferently. For example, Bigtable

and SimpleDB use similar data models with tables, rows and columns; however, Bigtable requires defining column-families as a prefix for column names, while SimpleDB does not impose any schema; SimpleDB supports multiple values for a column, while Bigtable supports only a single value for one column but with multiple versions. Similar to AppScale [4] (which also aims to unify access to many different NoSQL databases), CloudTPS must define a single data model to be mapped over different data store models. To handle different requirements of Cloud data stores, CloudTPS automatically transforms the "logical" attribute names from queries into "physical" attribute names used in the specific Cloud data store it is using. CloudTPS can be deployed to new Cloud data stores easily by developing a new "adapter" which maps CloudTPS's data model and APIs onto specifics of the new data store. However, while AppScale defines its unified data model as simple key-value pairs with queries spanning only a single table, CloudTPS needs a more structured data model to support complex join operations across multiple tables.

CloudTPS defines its data model as a collection of tables. Each table contains a set of records. A record has a unique Primary Key (PK) and an arbitrary number of attribute-value pairs. An attribute is defined as a Foreign Key (FK) if it refers to a PK in the same or another table. Applications may use other non-PK attributes to look up and retrieve records. These attributes are defined as Secondary Keys (SK) and are supported in CloudTPS by creating a separate index table which maps each SK to the list of PKs where this value of the SK is found. A secondary-key query can thus be transformed into a join query between the original table and the index table. To support join queries, CloudTPS expects applications to define the table schema in advance, with the table names, the PK, all the SKs and FKs together with their referenced attributes.

Figure 1 shows an example data model which defines four data tables and one index table. The table book defines book_id as its primary key. The FK author_id of table book refers to the PK of table author. To support secondary-key queries which select books by their titles, CloudTPS automatically creates an index table indexOf_bookTitle. Each record of table book matches the record of table indexOf_bookTitle of which the PK value equals its SK title. Therefore, the SK title is also a FK referring to the PK of the index table indexOf_bookTitle. Other non-key attributes can be left undefined in the table schema.

3.2 Join Query Types

"A join query combines records from two or more tables in a database. It is a means for combining fields from two tables by using values common to each other. A programmer writes a join predicate to identify the record for joining. If the evaluated predicate is true, the combined record is then produced" [28]. CloudTPS restricts this very general definition to support a specific class of join queries known as foreign-key equi-join. These queries have the property of accessing relatively small numbers of data items (which is essential for preserving

system scalability as we shall see). Meanwhile, they are by far the most common type of join queries in real-world Web applications.

Join queries in CloudTPS must explicitly give the primary keys of a list of initial records found in one table. These records and the table where they are stored are referred to as the "root records" and "root table" of this join query. This restriction excludes performing full table scans to join two tables completely.

CloudTPS specifically supports equi-joins queries: the matching relationship between two records is expressed as an equality between a FK and a PK (in the same or another table). For example, one such constraint can be that the author name found in the "book" record matches the author name found in the matching "author" record. Equi-join queries are by far the most common join queries, compared to other relationships such as "less than" or "greater than." Equi-join queries are often the result of database normalization methodologies.

This paper considers only *inner-joins* which return all records that have at least one matching record, while the final combined record contains merged records from the concerned tables. Other types of join, such as outer-join (which may return records with no matching record), and semi-join (which only returns records from one table), are out of the scope of this paper.

Some applications may require referential integrity, which means that the value of a foreign key always refers to an existing record. We assume that the referential integrity property is enforced in the logic of each read-write transaction. Therefore, CloudTPS maintains the referential integrity as long as the ACID properties for transactions are guaranteed.

3.3 API

Web applications access CloudTPS using a Java client-side library, which offers mainly two interfaces to submit respectively join queries and transactions.

CloudTPS expects join queries to be expressed as a collection of "JoinTable" and "JoinEdge" Java objects. A JoinTable object identifies one table where records must be found. It contains the table name, the projection setting (i.e., the list of attributes to be returned) and possibly a predicate (i.e., a condition that a record must satisfy to be returned). A join query must designate one JoinTable object as the root table, which contains the explicit list of primary keys of the root records. Multiple JoinTable objects are joined together using a JoinEdge object which matches the FK from one JoinTable to the PK of another. One can write a self-join query by creating two JoinTable objects with the same table name.

Figure 2 shows the SQL and CloudTPS representations of a join query which retrieves information about two books and their authors. The book object is the root table, and the primary keys of root records are 10 and 20. A JoinEdge starts from JoinTable "book" to "author" indicating the FK "author_id" in table "book" refers to the PK of JoinTable "author". The predicate of JoinTable "country" shows that only books whose author comes from the Netherlands should be returned.

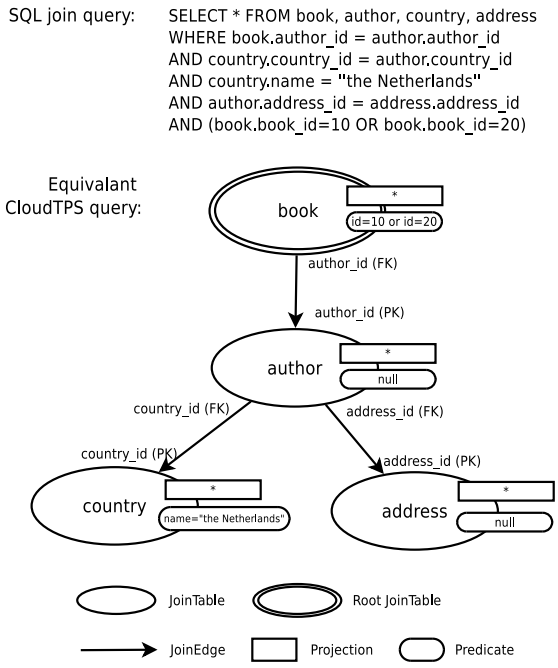


Figure 2: The input data structure representation of an example join query

CloudTPS also handles read-only and read-write transactions defined as a "Transaction" java objet containing a list of "SubTransaction" objects. Each sub-transaction represents an atomic operation on one single record. 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.

4 System Design

CloudTPS considers join queries as a specific kind of multi-row transactions. It therefore enforces full transactional consistency to the data they access, even in the case of machine failures or network partitions. However, the underlying Cloud data stores do not guarantee strong consistency across multiple data items.

CloudTPS is composed of a number of Local Transaction Managers (LTMs). To ensure strong consistency, CloudTPS maintains an in-memory copy of the accessed application data. Each LTM is responsible for a subset of all data items. We assign data items to LTMs using consistent hashing [15] on the item's primary key. This means that any LTM can efficiently compute the identity of the LTM in charge of any data item, given its primary key. Transactions and

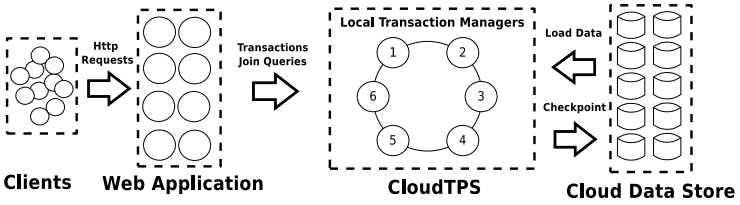


Figure 3: CloudTPS system model

Forward	SELECT * FROM author, book WHERE book.id = author.auth_id AND book.id = 10	**Root:** book(10)
Backward	SELECT * FROM author, book WHERE book.id = author.auth_id AND author.id = 100	**Root:** author(100)

author

id(PK)	(CloudTPS index attribute)	(value)
100	Ref::book::auth_id::10 Ref::book::auth_id::30	10 30
101	Ref::book::auth_id::20	20

book

id (PK)	author_id	title
10	100	title1
20	101	title2
30	100	title3

Figure 4: The index data layout for the example application data

join queries operate on this in-memory data copy, while the underlying cloud data store is transparent to them.

Figure 3 shows the organization of CloudTPS. Clients issue HTTP requests to a Web application, which in turn issues queries and transactions to CloudTPS. A transaction or join query can be addressed to any LTM, which then acts as the coordinator across all LTMs in charge of the data items accessed by this query. If an accessed data item is not present in the LTM's memory, the appropriate LTM will load it from the cloud data store. Data updates resulting from transactions are kept in memory of the LTMs and later checkpointed back to the cloud data store[1]. LTMs employ a replacement policy so that unused data items can be evicted from their memory (the caching policy is discussed in details in [26]).

4.1 Join Algorithm

Foreign-key equi-join is the most common type of join query in Web applications. Considering the case of equi-join on two tables, for each pair of matching records, one contains a FK which must be equal to the PK of the other. Intuitively, processing a join query spanning multiple tables consists of recursively identifying matching records, starting from the root records (known by their primary keys) and following JoinEdge relationships.

The methods to identify the matched records, however, differ according to the role of the given records. If an already known record contains a FK which references the PK of a new record, then the new record can be efficiently located by its PK. We call this type of join queries "forward join" queries. On the other hand, if the PK of an already known record is *being referenced* by the FK of a new record to be found, then in principle it becomes necessary to scan the full table and search for all records whose FK is equal to the PK of the known record. Such table scans are however very expensive and should therefore be avoided. We name such join queries "backward join" queries. To avoid a full table scan for each "backward join" query, we complement the referenced table with direct links to the PKs of matching records. This allows to translate such queries into "forward join" queries. On the other hand, we now need to maintain these indexes every time the tables are updated. If a data update changes the reference relationships among records, the update query must be dynamically translated into a transaction in which the indexes are updated as well.

Figure 4 shows an example index data layout to support a forward and a backward join query. The underlying data schema is the same as in Figure 1. The table book contains a FK auth_id referring to the PK value of table author. The forward join query can be processed directly without any indexes, as the FK auth_id of its root book record identifies that the PK of its matched author is 100. The backward join query, on the contrary, starts by accessing its root record in table author and requires additional indexes to identify the matching record. The indexes are stored as arbitrary number of "index attributes" in each record of the referenced table. Doing this does not require to change the data schema as all Cloud data stores support the dynamic addition of supplementary fields onto any data item. Each index attribute represents one matched referring record with the corresponding FK. We name the index attribute name as Ref::{table name}::{FK name}::{PK value}, while the value of the index attribute is the PK of the matching record. CloudTPS creates these indexes upon the declaration of the data schema, then maintains their consistency automatically. In Figure 4, in the author record of PK(100), the index attribute Ref::book::author_id::10 means that this record has a matching book record of PK(10) referring by FK auth_id. With these complemented index attributes, the backward join query in Figure 4 identifies two matching book records with PK 10 and 30 for its root record.

[1]Each LTM checkpoints data periodically irrespective of the current load of the cloud data store. Implementing more sophisticated strategies such as [19] is part of our short-term agenda.

Secondary-Key query	SELECT * FROM book WHERE book.title="bookTitle"
Translated join query	SELECT * FROM book WHERE indexOf_bookTitle.title="bookTitle" AND book.title=indexOf_bookTitle.title

Table 1: Translating a secondary-key query into a join query

Secondary-key queries are similar to the previous case: we need to efficiently identify records using one of its non-PK attributes. We apply a similar solution by building explicit indexes. However, unlike foreign-key joins, there exists no table where we can simply add index information. To support secondary-key queries, we therefore create a separate index table for each SK. A secondary-key query then translates into a forward join query between the index table and the original table. Table 1 shows an example secondary-key query which searches records by the SK title of table book. The separate index table indexOf_bookTitle for this SK is defined in Figure 1. The translated join query first locates the root record in the index table by using the given SK value "bookTitle" as the PK value. It then retrieves the PKs of the matched book records. The result of the secondary-key query is obtained by retrieving each of these matched book records.

4.2 Consistency Enforcement

To ensure strong consistency, CloudTPS implements join queries as multi-item read-only transactions. Our initial implementation of CloudTPS already supported multi-item transactions [26]. However, it required the primary keys of all accessed data items to be specified at the time a transaction is submitted. This restriction excludes join queries, which need to identify matching data items during the execution of the transaction. Besides, it also prohibits transparent index management as programmers would be required to provide the primary keys of the records containing the affected index attributes. To address this issue, we propose two extended transaction commit protocols: (i) for read-only transactions to support join queries, and (ii) for read-write transactions to support transparent index management.

Transactional consistency is defined according to the four ACID properties: Atomicity, Consistency, Isolation and Durability. This paper uses the same mechanism as in [26] to implement the Isolation, Consistency and Durability properties:

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. We assume that the consistency rules are applied within the logic of transactions. Consistency is therefore ensured as long as all transactions are executed correctly.

Isolation means that the behavior of a transaction is not impacted by the presence of other transactions that may be accessing the same data items concurrently. In CloudTPS each transaction is assigned a globally unique timestamp by a centralized timestamp manager. LTMs are required to execute conflicting transactions in the order of their timestamps. If an LTM receives a transaction after having committed a conflicting transaction with a greater timestamp, it votes "ABORT" to restart the late transaction with a new timestamp. Transactions which access disjoint sets of data items can execute concurrently or in any order.

The **Durability** property requires that the effects of committed transactions will not be undone, even in the case of server failures. CloudTPS checkpoints the updates of committed transactions back to the cloud data store, which is assumed to be highly-available and persistent. During the time between a transaction commit and the next checkpoint, durability is ensured by the replication of data items and transaction states across several LTMs.

We now focus on the **Atomicity** property: either all operations of a transaction succeed successfully, or none of them does. Specifically, for a read-write transaction, Atomicity means that the data updates must either all be committed or all aborted. For a read-only transaction, Atomicity means that, if committed, all required records must be retrieved so that the client receives a correct and complete result.

In CloudTPS, a transaction is composed of any number of sub-transactions, where each sub-transaction accesses a single data item atomically. To enforce Atomicity, transactions follow the two-phase commit protocol (2PC) across all LTMs responsible for the accessed data items. As shown in Figure 5(a), in the first phase, the coordinator submits all the sub-transactions to the 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. To implement join queries, we however need to extend 2PC into two different protocols respectively for join queries as read-only transactions, and for transparent index management in read-write transactions.

4.2.1 Read-Only Transactions for Join Queries

The 2PC protocol requires that the identity of all accessed data items is known at the beginning of the first phase. However, join queries can only identify the matched records after the root records are accessed. To address this issue, we need to extend the 2PC protocol. During the first phase, when the involved LTMs complete the execution of their sub-transactions, besides the normal "COMMIT" and "ABORT" messages, they can also vote "Conditional COMMIT" which requires more sub-transactions to be added to the transaction for accessing new records. To add a new sub-transaction, the LTM submits it to both the responsible LTM and the coordinator. The responsible LTM executes this new sub-transaction, while the coordinator adds it to the transaction and

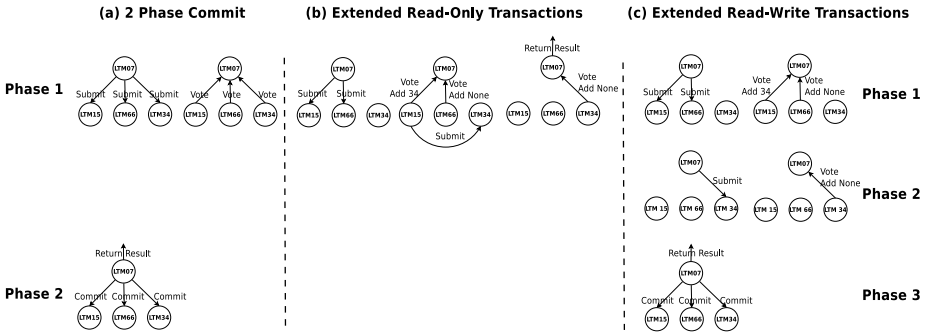


Figure 5: Two-phase commit vs. the extended transaction commit protocols

waits for its vote. The coordinator can commit the transaction only after no sub-transaction requests to add new sub-transactions.

As read-only transactions do not commit any updates, LTMs can terminate these sub-transactions immediately after all concerned LTMs return their votes rather than waiting until the second phase. The coordinator therefore no longer needs to send the "commit" messages to all involved LTMs. In this case, transactional consistency is enforced by the timestamp ordering protocol: concurrent read-only transactions which access non-disjoint sets of data items are executed in the same order at all LTMs.

This extension allows join queries to access the root records first, and then add matching records to the transaction during the query execution. Figure 5(b) shows an example of the extended protocol to execute a join query. At the beginning, the coordinator is only aware of the root records which are held by LTM 15 and LTM 66. After receiving and executing the sub-transactions, LTM 15 successfully identifies a matching record which is hosted by LTM 34. LTM 15 submits the new sub-transaction to LTM 34 directly and also returns the new sub-transaction along with its "Conditional COMMIT" vote to the coordinator. On the other hand, LTM 66 identifies no matching record so it simply returns "COMMIT." Finally, LTM 34 executes the new sub-transaction and also returns "COMMIT" with no more new sub-transactions. The coordinator can then commit the transaction by combining the records into the final result and returning them to the client.

In case of machine failures or network partitions, LTMs can simply abort all read-only transactions without violating the ACID properties.

4.2.2 Read-Write Transactions for Index Management

CloudTPS transparently creates indexes on all FKs and SKs to execute join queries efficiently. To ensure strong data consistency, when a read-write transaction updates any data items, the affected index attributes must also be updated atomically. As each index attribute stands for a referring record matching to its belonging record, when the FK of this referring record is inserted/updat-

ed/deleted, the corresponding index attribute must also be adjusted. Specifically, if the FK value is updated from value "A" into a new value "B", its corresponding index attribute stored in the referenced record "A" must be removed, and a new index attribute must be added to the new referenced record "B." To enforce strong data consistency, these affected index attributes must be updated within the same read-write transaction. Considering the example as defined in Figure 4, a read-write transaction could insert a book record which matches an existing record in table author. When this read-write transaction commits, the primary key of this new book record must already be stored as an index attribute into the corresponding author record.

CloudTPS creates the indexes automatically, so the index maintenance must also be transparent to the programmers. Here as well, this means that transactions must be able to identify data items to be updated during the execution of the transaction. For example, a query which would increment a record's secondary key needs to first read the current value of the secondary key before it can identify the records it needs to update in the associated index table.

To implement transparent index management, we extend the transaction commit protocol for read-write transactions to dynamically add sub-transactions to the transaction for updating the affected index attributes. Similar to the extension for read-only transactions, during the first phase, LTMs can generate and add more sub-transactions to access new data items. However, unlike in read-only transactions, LTMs should not submit new sub-transactions to the responsible LTMs directly. In read-write transactions, if any sub-transaction votes "ABORT," the coordinator sends abort messages to all current sub-transactions immediately, in order to minimize the blocking time of other conflicting transactions. Allowing LTMs to submit new sub-transactions directly to each other opens the door to ordering problems where the coordinator received the information that new sub-transactions have been added after it has aborted the transaction. Therefore, in read-write transactions, the involved LTMs submit new sub-transactions to the coordinator only. The coordinator waits until all current sub-transactions return before issuing any additional sub-transactions. The coordinator enters the final phase and commits the transaction when all sub-transactions vote "COMMIT" and do not add any new sub-transactions. If any sub-transaction in any phase votes "ABORT," then the coordinator aborts all the sub-transactions.

We can easily implement transparent index management with this protocol. Whenever a sub-transaction is submitted and executed, the LTM in charge of this data item automatically examines the updates to identify the affected index attributes. If any FKs or SKs are modified, the LTM then generates new sub-transactions to update the affected index attributes.

Figure 5(c) shows an example of the extended read-write transaction. In phase 1, the coordinator LTM 07 submits sub-transactions to update data items hosted in LTM 15 and LTM 66. LTM 15 identifies an affected index attribute hosted by LTM 34, while LTM 66 identifies none. LTM 15 thus generates a new sub-transaction for updating this index attribute and returns it back to the coordinator along with its vote of "COMMIT." After both LTM 15 and

LTM 66 vote "COMMIT," the coordinator then starts a new phase and submits the new sub-transaction to LTM 34. After LTM 34 also votes "COMMIT," the coordinator finally commits the transaction in phase 3.

4.2.3 Fault Tolerance

CloudTPS must maintain strong data consistency even in the case of machine failures and network partitions. CloudTPS uses the same fault-tolerance mechanism as in our previous work. We therefore briefly introduce the main concepts here, and refer the reader to [26] for full details.

To execute transactions correctly all LTMs must agree on a consistent membership, as this is key to assigning data items to LTMs. Any membership change is therefore realized by a transaction across all LTMs. When one or more LTMs fail to respond to transaction messages, and the remaining LTMs represent less than half of the previous membership, then there is a possibility that no machine has failed but the network has been partitioned. In this case CloudTPS rejects all incoming transactions until the partition has been resolved. Otherwise, it can exclude the unresponsive LTMs from the membership, and start accepting transactions again after the system recovers from the LTM failures.

Recovering from an LTM failure implies that some surviving LTM fulfills the promises that the failed LTM made before failing. Such promises belong to two cases. In the first case, a coordinator initiated a transaction but failed before committing or aborting it. To recover such transactions, each LTM replicates its transaction states to one or more "backup" LTMs (chosen by consistent hashing through the system membership). If the coordinator fails, its backups have enough information to finish coordinating the ongoing transactions.

In the second case, a participant LTM voted "COMMIT" for some read-write transactions but failed before it could checkpoint the update to the cloud data store. Here as well, each LTM replicates the state of its data items to one or more "backup" LTMs so that the backups can carry on the transactions and checkpoint all updates to the data store. Assuming that each transaction and data item has N backups in total, CloudTPS can guarantee the ACID properties under the simultaneous failure of up to N LTM servers.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. Upon any change in membership it is therefore necessary to re-replicate data items to maintain the correct number of replicas. Following an LTM failure, CloudTPS can return to its normal mode of operation after all ongoing transactions have recovered, a new system membership has been created, and the relevant data items have been re-replicated.

5 Implementation

5.1 CloudTPS Architecture

CloudTPS is composed of any number of identical LTMs. The internal architecture of an LTM, shown in Figure 6, consists of four core components running

inside a Tomcat application server. This potentially allows to run CloudTPS on the same application servers as the Web application itself for improved communication performance.

Each LTM has a "Read-Write" and a "Read-Only" transaction manager to carry out the logics of transaction coordinators. The Read-Write transaction manager also maintains coordination with other LTMs in charge of backup data item replicas. To locate data items across CloudTPS, the membership manager which maintains a consistent view of system membership. The data manager manages the in-memory copy of data and executes sub-transactions in sequential order according to transaction timestamps.

The implementation of LTMs is inspired from the SEDA architecture [27]. Similarly to SEDA, CloudTPS is designed to handle large number of concurrent transactions with strong response time constraints. We implement the four core components as single-threaded self-contained service components. Each service component maintains a FIFO message queue for accepting messages. The service component continuously listens to its message queue, and handles incoming messages sequentially. Each service component is single-threaded to avoid the need to lock private data structures.

Executing a transaction consists of sending a sequence of messages between service components in the local or remote LTMs. Service components send messages via the message router, which is not an active service component but simply a passive set of Java classes. If the destination service component resides in the local LTM, the message router simply forwards the message to the corresponding message queue. Otherwise, the message router dispatches the message to one of its "worker" threads to perform network messaging. Each worker is implemented as a single-threaded service component which continuously waits for network messages to send. At the other end, the message receiver is implemented as a regular java servlet deployed in the application server.

Besides interacting with other components, the data manager also needs to access the underlying cloud data store to load and checkpoint data items. The data manager performs these operations by invoking the Adapter manager, which dispatches each operation to one of its own worker service components. The data items loading operations have higher priority than checkpointing. Loading a data item is in the critical path of a transaction which needs this item to progress, while checkpointing can be delayed with no impact on transaction latencies. The adapter manager therefore maintains two separate pools of workers for loading and checkpointing data items respectively. Loading different data items can be done completely independently, so the adapter manager selecting workers to load a data item in round-robin fashion. However, for checkpointing updates, the updates of multiple conflicting transactions on the same data item must be checkpointed in the order of transaction timestamps. To guarantee sequential order, the adapter manager dispatches all updates of the same data item to the same worker.

The workers of the adapter manager access the underlying cloud data store via adapters, which transform the logical data model of CloudTPS into the physical data model used by the cloud data store. The Web application does

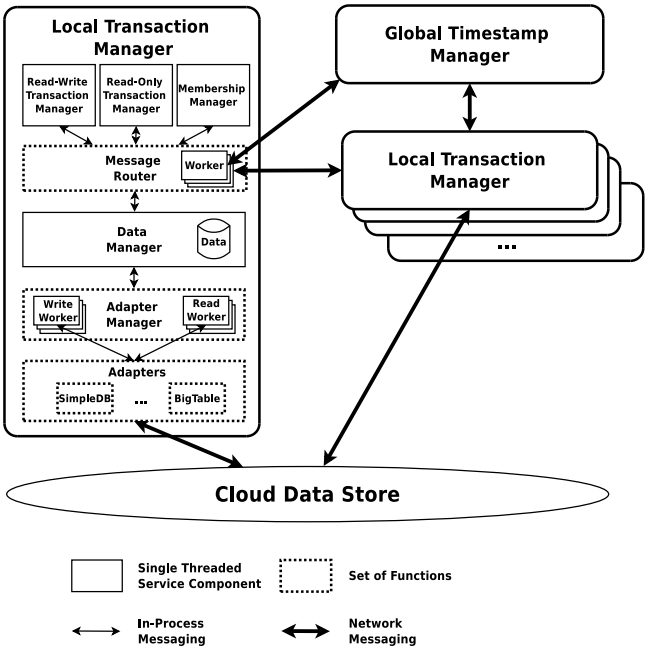


Figure 6: The internal architecture of CloudTPS

not need any adjustment depending on the type of underlying cloud data store. The current implementation supports SimpleDB and HBase [12] (an open source clone of Bigtable). Migrating CloudTPS between these two cloud data stores requires only to change the adapter configuration. One can also easily implement new adapters for other cloud data stores as we discuss in the next section.

5.2 Adapters

CloudTPS aims to provide a uniform data access overlay that allows Web applications to transparently access different cloud data stores. To be compatible with a broad range of cloud data stores, CloudTPS requires that table names only include letters and digits. The primary key and all attributes are of type "String." All table names and attribute names are case-insensitive. CloudTPS automatically transforms all input parameters into lower-case.

Different cloud data stores have different rules for the definition of attribute names. For example, Bigtable requires the column-family name to be a prefix to the attribute name, while SimpleDB does not. For Bigtable, we store the application data in the "Data:" column family while the index attributes are in "Ref:". For SimpleDB, the application data have the same attribute names as in queries. The index attributes are prepended with the prefix "Ref:" so as to differentiate them from the application data.

Some Cloud data stores, such as SimpleDB, require to horizontally partition tables into multiple sub-tables for higher throughput. Even for cloud data stores that partition tables automatically, such manual partitioning is often still necessary for better load balancing. For example, HBase automatically partitions tables according to the size of the data rather than the load they receive. CloudTPS transparently splits tables horizontally into a configurable number of partitions. The physical table name for the cloud data store is then appended with the partition index.

Supporting a new data store in CloudTPS only requires developing a new Java class which implements the Adapter interface. The core of an adapter is two methods: loadItem and checkpointItem. This is simple, straightforward code. The two adapters currently implemented in CloudTPS are about 400 lines long each.

6 Evaluation

We now evaluate the performance and scalability of CloudTPS in three scenarios: micro- and macro-benchmarks, and a scenario with node failures and network partitions.

6.1 Experiment Setup

6.1.1 System Configuration

We execute CloudTPS on top of two different families of scalable data layers: SimpleDB running in the Amazon Cloud, and HBase v0.20.4 running in our local DAS-3 cluster [12]. In both platforms, we use Tomcat v6.0.26 as application server. The LTMs and load generators are deployed in separate machines.

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. We use Medium Instances in the High-CPU family, which have 1.7 GB of memory, 2 virtual cores with 2.5 EC2 Compute Units each, and 350 GB of storage. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

6.1.2 Throughput Measurement

Given a specific workload and number of LTMs, we measure the maximum sustainable throughput of CloudTPS under a constraint of response time. For the evaluations in DAS-3, we define a demanding response time constraint which imposes that 99% of transactions must return within 100 ms. DAS-3 assigns a physical machine for each LTM, and has low contention on other resources

such as the network. On the other hand, in the public Amazon cloud, LTMs have to share a physical machine with other instances, and we have less control of the resources such as CPU, memory, network, etc. Furthermore, even multiple virtual instances of the exact same type may exhibit different performance behavior [11]. 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.

To determine the maximum sustainable throughput of CloudTPS, we perform several rounds of experiments with different request rates. The workload is generated by a configurable number of Emulated Browsers (EBs), each of which issues requests from one simulated user. Each EB waits for 1000 milliseconds on average between receiving a response and issuing the next transaction. Our evaluations assume that the application load remains roughly constant. In each round, we configure different numbers of EBs and measure the throughput and response time of CloudTPS. We start with a small number of EBs, and increase the number of EBs until the response time of CloudTPS violates the response time constraint. Each round lasts 30 minutes.

Throughout the evaluation, we provision sufficient resources for clients and underlying cloud data stores, to ensure that CloudTPS remains the bottleneck of the system.

6.2 Microbenchmarks

We first study the performance of join queries and read-write transactions in CloudTPS using microbenchmarks.

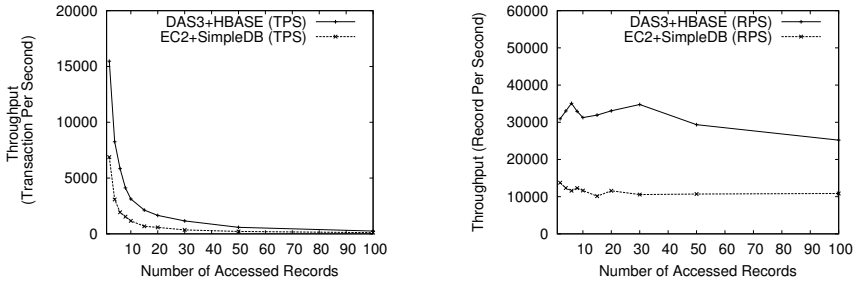
6.2.1 Workload

Two criteria influence the performance of join queries in CloudTPS: the number of data items that they access, and the length of the critical execution path (i.e., the height of the query's tree-based representation). For example, a join query joining two tables has a critical execution path of one. We first evaluate the performance of CloudTPS under workloads consisting purely of join queries or read-write transactions with specific number of accessed records and length of critical execution path.

The microbenchmark uses only one table, where each record has a FK referring to another record in this table. We can therefore generate a join query with arbitrary length of its critical execution path by accessing the referenced record recursively. Given the length of the critical execution path, we can control the number of accessed records by defining the number of root records. We generate 10,000 records in this table.

CloudTPS applies a cache replacement strategy to prevent LTMs from memory overflow when loading application data. In our evaluation with microbenchmarks, we configure the system such that the hit rate is 100%.

In this set of experiments, we deploy CloudTPS with 10 LTMs.



(a) Expressed in transactions/second

(b) Expressed in record accesses/second

Figure 7: Throughput of join queries with different number of accessed data items

6.2.2 Join Queries

Here we study the performance of CloudTPS with join queries only. We first evaluate CloudTPS with join queries all having the same length of critical execution path of one, but access different numbers of data items. As shown in Figure 7(a), in both DAS-3 and EC2 platforms, when the number of accessed data items increase, the throughput in terms of Transaction Per Second (TPS) decreases dramatically. This is to be expected, since the complexity of join queries largely depends on the number of data items they access. Figure 7(b) shows the same throughput expressed in numbers of accessed records per second. The result remains close to the ideal case, where the lines stay perfectly horizontal. We can also see that instances in DAS-3 perform approximately three times faster than medium High-CPU instances in EC2.

We then evaluate the system with join queries that access the same number of data items (12 items), but with different length of critical execution path. Figure 8 shows that as the length of the critical execution path increases, the maximum sustainable throughput decreases slightly. This is expected as longer execution paths increase the critical path of messages between LTMs, and therefore imply higher transaction latencies. To maintain the strict response time constraint, the system must reduce throughput.

6.2.3 Read-Write Transactions

We now study the performance of CloudTPS with a workload composed of read-write transactions (including read-write transactions which update index records). The updated index records are included in the count of accessed records of a transaction. We perform this evaluation in the DAS-3 platform.

Similar to join queries, as shown in Figure 9(a), the throughput in terms of TPS decreases dramatically when the number of accessed records increases. However, Figure 9(b) shows that the throughput in record accesses per second remains roughly constant. This shows that the performance bottleneck is the

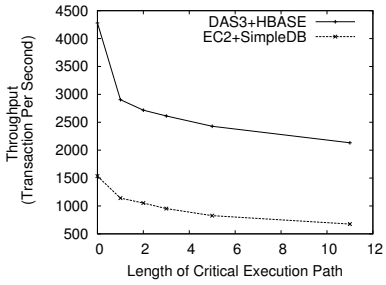
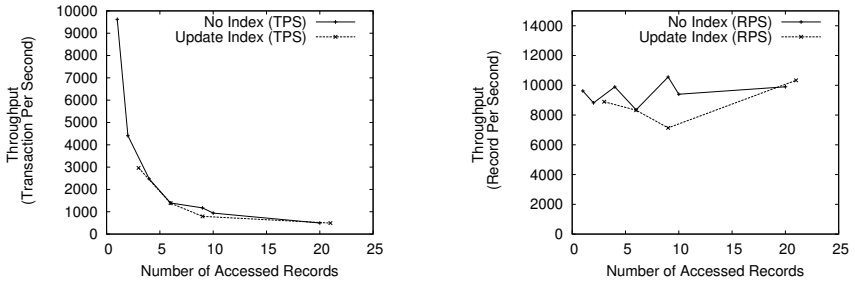


Figure 8: Throughput of join queries with different length of execution path



(a) Transaction Per Second

(b) Record Per Second

Figure 9: Throughput of join queries with different number of accessed data items

update operation of individual data items rather than the cost of the transaction itself.

We also note that in Figure 9, the line for transactions which only update data records and the line for transactions which also update indexes are very close to each other. This means the extra phase of updating index records does not degrade the system performance significantly. One only needs to pay the price of updating the extra index records.

6.3 Scalability Evaluation

6.3.1 TPC-W Web Application

We now evaluate the scalability of CloudTPS under a demanding workload derived from the TPC-W Web application [20]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com. It is important to note that TPC-W was originally designed and developed for relational databases. Therefore, it contains the same mix of join queries and read-write transactions as cloud-based applications would if their data store supported join queries. TPC-W contains 10 database tables.

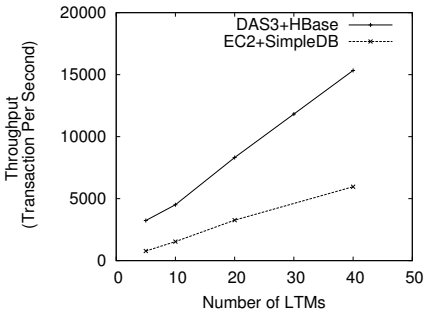


Figure 10: Scalability of CloudTPS under TPC-W workload

Deploying TPC-W in CloudTPS requires no adaption to the database schema except converting all data types to "string." We also kept all simple and complex queries unchanged, and merely translated them to CloudTPS's tree-based representation as discussed in Section 3.3.

TPC-W contains a secondary-key query which selects a customer record by its user name. CloudTPS therefore automatically creates an index table indexOf_customerC_uname referring to the SK "c_uname" of data table "customer." This query is then rewritten into a join query across the two tables. The index table is the root table and the input user name is the primary key of the root record.

We derive a workload from TPC-W containing only join queries and read-write transactions. This workload excludes all simple primary-key read queries, which are the most common query type for Web applications. This creates a worst-case scenario for CloudTPS's performance and scalability.

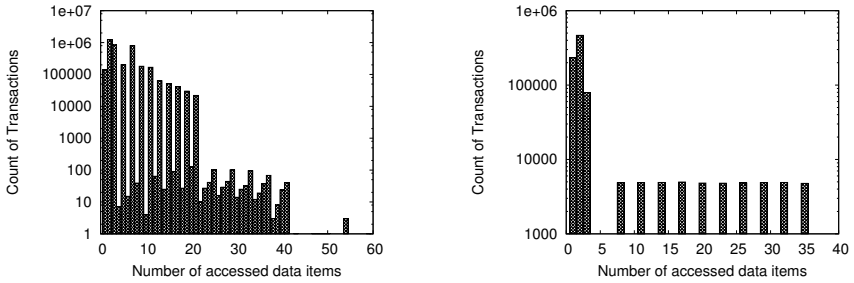
We populate the TPC-W database with 144,000 customer records in table "Order" and 10,000 item records in table "Item." We then populate the other tables according to the TPC-W benchmark requirements.

TPC-W continuously creates new shopping carts and orders. Each insert triggers one cache miss. On the other hand, as the size of affected data tables keeps increasing, this eventually results in more record evictions from the LTMs, which in turn potentially triggers more cache misses. During our scalability evaluation, we observe a hit rate around 80%.

6.3.2 Scalability Results

Figure 10 depicts the results of the scalability experiments in DAS-3 and the Amazon cloud. We can see that the overall system throughput grows linearly with the number of LTMs. This means that CloudTPS can accommodate any increase of workload with a proportional number of compute resources.

In DAS-3, with 40 LTMs, CloudTPS achieves a maximum sustainable throughput of 15,340 TPS. For this experiment, we also use 30 machines to host HBase, 1 machine as timestamp manager and 8 clients. This configuration uses the



(a) Read-Only Transactions (b) Read-Write Transactions

Figure 11: Number of data items accessed by transactions.

complete DAS-3 Cluster, so we cannot push the experiment further. Figure 11 shows the distribution of the number of data items accessed per transactions under this configuration of 40 LTMs (note that the y axis is in log scale). On average, a read-only transaction accesses 4.92 data items and a read-write transaction accesses 2.96 data items. Within all input transactions, 82% transactions are join queries, and 18% are read-write transactions. As for the length of critical execution path, 33.3% of the read-only transactions have a length of one, while the other 66.7% have two. For read-write transactions, 84.2% of them need to update indexes, while the other 15.8% do not.

In EC2, with 40 LTMs, CloudTPS achieves a maximum sustainable throughput of 5,960 TPS. CloudTPS achieves three times better throughput in DAS-3 than in EC2 with High-CPU medium instances.

This evaluation shows that CloudTPS scales linearly under a demanding workload typical of a Web application designed with no restriction regarding join queries. We expect CloudTPS to continue scaling linearly with even larger numbers of LTMs.

6.3.3 Scalability comparison with relational databases

We now compare the scalability of CloudTPS with that of PostgreSQL, a high-performance open-source relational database. Once again we measure the maximum throughput that the system can sustain while respecting the constraint that 99% of transactions must return under 100 ms. We run these experiments on the DAS-3 cluster. When running CloudTPS, we count both CloudTPS and HBase nodes as "database nodes." Running the same experiment in EC2 is not possible as we cannot measure the number of machines used by SimpleDB.

We compare CloudTPS with a PostgreSQL v9.0 setup. The database is configured to use one master database and N slaves. Synchronization between master and slave databases is enforced by the "Binary Replication" mechanism [23]. We issue all read-write transactions on the master, and balance join queries across the slaves.

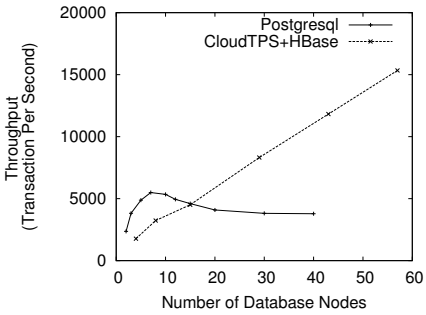


Figure 12: Scalability of CloudTPS vs. PostgreSQL

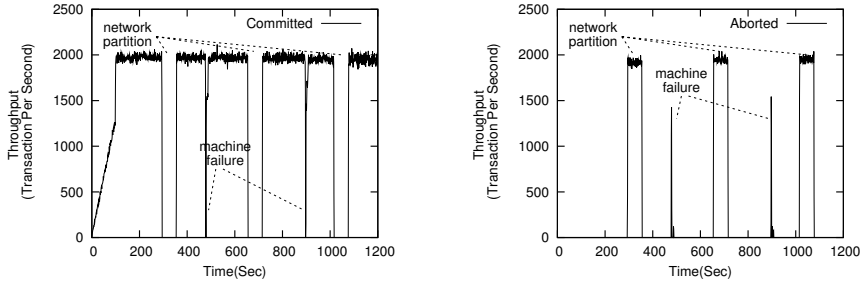
Figure 12 illustrates the fundamental differences between CloudTPS and a replicated relational database. In small systems, PostgreSQL performs much better than CloudTPS. This is due to the fact that each database server can run read-only join queries locally, without needing any communication with other servers. PostgreSQL reaches a maximum throughput of 5493 transactions per second using one master and six slaves. However, at this point the master server becomes the bottleneck as it needs to process all update operations and send the binary operations to its slaves. The master's throughput eventually decreases because of the growing number of slaves to which it must send updates. On the other hand, CloudTPS starts with a modest throughput of 1770 transactions per second in its smallest configuration of 4 machines (two machines for CloudTPS and two machines for HBase). However, its throughput grows linearly with the number of database nodes, reaching a throughput of 15,340 transactions per second using 57 nodes (40 machines for CloudTPS and 17 machines for HBase). This clearly shows the scalability benefits of CloudTPS compared to a replicated relational database.

6.4 Tolerating Network Partitions and Machine Failures

Finally, we illustrate CloudTPS's behavior in the presence of machine failures and network partitions. We configure CloudTPS with 10 LTMs and then alternately create 3 network partitions and 2 machine failures. Each network partition lasts 1 minute. We run this experiment in DAS-3.

As shown in Figure 13, in case of single-machine failures, CloudTPS recovers within about 14 seconds before failed transactions are recovered and the responsible data items of the failed LTM are re-replicated to new backup LTMs. On the other hand, for network partitions, no data re-replication is necessary. CloudTPS recovers almost instantly after the network partition is restored. In all cases the transactional ACID properties are respected despite the failures.

We note that during the recovery from an LTM failure, the system throughput drops to zero. This is due to a naive implementation of our fault tolerance mechanism which simply aborts all incoming transactions during recovery. A



(a) Committed Transactions

(b) Aborted Transactions

Figure 13: CloudTPS tolerates 3 network partitions and 2 machine failures

smarter implementation could avoid such degration by only aborting the transactions accessing the failed LTMs. We consider this as future work.

7 Conclusion

Cloud data stores are often praised for their good scalability and fault-tolerance properties. However, they are also strongly criticized for the very restrictive set of query types they support. As a result, Web application programmers are obliged to design their applications according to the technical limitation of their data store, rather than according to good software engineering practice. This creates complex and error-prone code, especially when it comes to subtle issues such as data consistency under concurrent read/write queries.

This paper proves that scalability, strong consistency and relatively complex join queries do not necessarily contradict each other. CloudTPS exploits the fact that most join queries access only a small fraction of the total available data set. By carefully designing algorithms such that only a small subset of the LTMs is involved in the processing of any particular transactions, we can implement strongly consistent join queries without compromising the original scalability properties of the cloud data store. We designed specific transactional protocols to catter for the special needs of read-only join queries as well as read-write transactions which transparently update indexe values at runtime. The system scales linearly in our local cluster as well as in the Amazon Cloud.

Providing support for strongly consistent join queries essentially brings cloud data stores much closer in functionality to classical full-featured relational databases. We hope that CloudTPS can help in making cloud data stores safer and easier to use for regular application developers, and thereby contribute to the success of these extraordinary technologies. At the same time, there remains a large feature gap between CloudTPS and any mature SQL database: outer-joins, semi-joins, aggregation queries, data management tools, etc. It remains an open research topic to identify which of these missing features could be

added to cloud data stores without compromising their initial good properties of scalability and high availability.

References

[1] Amazon.com. Amazon SimpleDB., 2010. `http://aws.amazon.com/simpledb`.

[2] Mark Atwood. A MySQL storage engine for AWS S3. In *MySQL Conference and Expo*, 2007. `http://fallenpegasus.com/code/mysql-awss3/`.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. CIDR*, 2011.

[4] Chris Bunch, Navraj Chohan, Chandra Krintz, Jovan Chohan, Jonathan Kupferman, Puneet Lakhina, Yiming Li, and Yoshihide Nomura. An evaluation of distributed datastores using the AppScale cloud platform. In *Proc. IEEE CLOUD*, 2010.

[5] 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 *Proc. OSDI*, 2006.

[6] 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 *Proc. VLDB*, pages 1277–1288, 2008.

[7] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB*, 2010.

[8] Sudipto Das, Shashank Agarwal, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an elastic, scalable, and self managing transactional database for the cloud. Technical report, CS, UCSB, 03/2010 2010.

[9] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic transactional data store in the cloud. In *Proc. HotCloud*, 2009.

[10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *Proc. SoCC*, 2010.

[11] 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.

[12] HBase. An open-source, distributed, column-oriented store modeled after the Google Bigtable paper, 2006. `http://hadoop.apache.org/hbase/`.

[13] Todd Hoff. NoSQL took away the relational model and gave nothing back. High Scalablility blog, October 2010. `http://highscalability.com/blog/2010/10/28/nosql-took-away-the-relational-model-and-gave-nothing-back.html`.

[14] 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 *Proc. VLDB*, 2008.

[15] 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*, 1997.

[16] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. VLDB*, 2000.

[17] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[18] Justin J. Levandoski, David Lomet, Mohamed F. Mokbel, and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *Proc. CIDR*, 2011.

[19] John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren. Stout: an adaptive interface to scalable cloud storage. In *Proc. USENIX ATC*, 2010.

[20] Daniel A. Menascé. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3), 2002.

[21] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. OSDI*, 2010.

[22] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. In *Proc. DSOM*, 2007.

[23] PostgreSQL.org. Binary replication tutorial, 2010. `http://wiki.postgresql.org/wiki/Binary_Replication_Tutorial`.

[24] 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 *Proc. VLDB*, 2007.

[25] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *Proc. VLDB*, 2010.

[26] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for Web applications in the cloud. Technical Report IR-CS-053, Vrije Universiteit Amsterdam, February 2010. `http://www.globule.org/publi/CSTWAC_ircs53.html`.

[27] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. SOSP*, 2001.

[28] Wikipedia. Join (SQL), 2010. `http://en.wikipedia.org/wiki/Join_(SQL)`.