Scalable Join Queries in Cloud Data Stores

Zhou Wei
VU University Amsterdam
Tsinghua University Beijing
Email: zhouw@few.vu.nl

Guillaume Pierre
VU University Amsterdam
Email: gpierre@cs.vu.nl

Chi-Hung Chi
Tsinghua University Beijing
Email: chichihung@mail.tsinghua.edu.cn

***Abstract*—Cloud data stores provide scalability and high availability properties for Web applications, but do not support complex queries such as joins. Web application 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 with respect to subtle issues such as data consistency under concurrent read/write queries. We present join query support in CloudTPS, a middleware layer which stands between a Web application and its data store. The system enforces strong data consistency and scales linearly under a demanding workload composed of join queries and read-write transactions. In large-scale deployments, CloudTPS outperforms replicated PostgreSQL up to three times.**

***Index Terms*—Scalability, Web applications, Cloud Computing, Join Queries, Secondary-key Queries, NoSQL.**

I. Introduction

Web application workloads fluctuate widely according to predictable as well as unpredictable patterns [1], [2]. To accommodate load variations up to several orders of magnitude, many Web application providers turn to Cloud computing environments where computing capacity can be provisioned on demand. Although scaling a stateless application server tier is relatively easy, designing a data tier capable of increasing its throughput to arbitrary levels while retaining a familiar programming interface remains a challenge. Relational databases can be scaled to a certain extent using replication techniques [3], but this scalability remains limited (as we show later in this paper). On the other hand, Cloud data stores such as Bigtable [4], SimpleDB [5] and Cassandra [6] can always accommodate higher workloads by adding extra hardware. However, Cloud data stores have an important drawback: they support only 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.

Join queries are an essential feature for any database system, as they allow to query related informations from multiple tables in a single atomic operation. These queries are often the result of data normalization techniques, which have been used for decades to help guaranteeing semantic data integrity in large systems. The lack of join queries in Cloud data stores can often be mitigated using techniques such as rewriting a join query into a sequence of simple primary-key queries. However, such translation is not a trivial task at all. First, one must design data schemas carefully to allow such query rewriting. 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 discusses the design and implementation of join queries that are strongly consistent by design, relieving programmers from the burden of adapting their programs to the peculiarities of Cloud data stores. At the same time the system retains the good scalability properties of the cloud data stores. We implement join queries in CloudTPS, a middleware layer which sits between the Web application and the Cloud data store. CloudTPS in charge of implementing join queries and 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 [7]. The current paper focuses on CloudTPS's support for consistent join queries, while retaining the original scalability and fault-tolerance properties of the underlying Cloud data store[1].

CloudTPS supports a specific type of join queries known as foreign-key equi-joins. This is by far the most common type of joins in Web applications. For example, every join query issued by Wikipedia to its database belongs to this category[2] [8]. 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 is updated atomically 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

Chi-Hung Chi is supported by the National Natural Science Foundation of China under Project Number 61033006.

[1]Our prototype is available at http://www.globule.org/cloudtps.

[2]Wikipedia's query workload also contains aggregate queries, which are out of the scope of this paper.

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 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 queries Web applications would use if their Cloud data store supported them. We show that, with no change of the initial relational data schema nor the queries addressed to it, CloudTPS achieves linear scalability while enforcing data correctness automatically. In large-scale configurations, CloudTPS outperforms replicated PostgreSQL up to three times.

II. 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, these systems usually rely on full data replication techniques and therefore do not bring extra scalability improvement compared to a non-cloud deployment [3].

Cloud data stores such as Bigtable are praised for their scalability and high availability properties [4]–[6]. They however require highly skilled programmers capable of manually handling consistency issues [9]. The most advanced one for complex query support is Cassandra, which implements secondary-key queries by building local indexes at each node and translating secondary-key queries into scatter-gather operations. To our best knowledge, no existing Cloud data store supports join queries.

A number of recent systems support multi-item transactions. Percolator focuses on incremental processing of massive data processing tasks [10]. Megastore supports transactions within fine-grained partitions of data, but only limited guarantees across them [11]. Deuteronomy operates over a wide range of heterogeneous data sources [12]. G-Store proposes key grouping protocols, allowing for transactions to run within pre-defined groups [13]. Scalaris uses Paxos to support transactions across any number of key-value pairs [14]. ecStore supports range queries [15]. However, none of these systems discusses support of complex queries such as join queries.

H-Store [16] is a distributed in-memory OLTP database which supports the SQL language. However, its scalability relies on careful data partitioning across executor nodes. H-Store does not support join queries across multiple partitions. Similarly, ElasTraS automatically partitions data across a number of database nodes but supports complex queries only within a single partition [17].

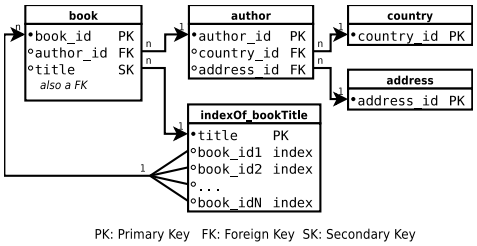


Fig. 1. An example data model for CloudTPS

Schism analyzes a query log to propose a data placement which minimizes the number of partitions involved in transactions [18]. This work is very complementary to the work on CloudTPS. However, it does not address the specific problem of join queries where the list of database nodes which should take part in a query can be found only while executing the transaction.

III. Database Model

A. Data Model

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 index table and the original table. 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. Other non-key attributes can be left undefined in the table schema.

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.

B. Join Query Types

CloudTPS supports a specific class of join queries known as foreign-key equi-join: the relationship between two records is expressed as an equality between a FK and a PK (in the same or another table). Equi-joins are by far the most common join queries, compared to relationships such as “less than” or “greater than.” These queries often result from database normalization methodologies. For example, we observed that all join queries in Wikipedia follow this structure.

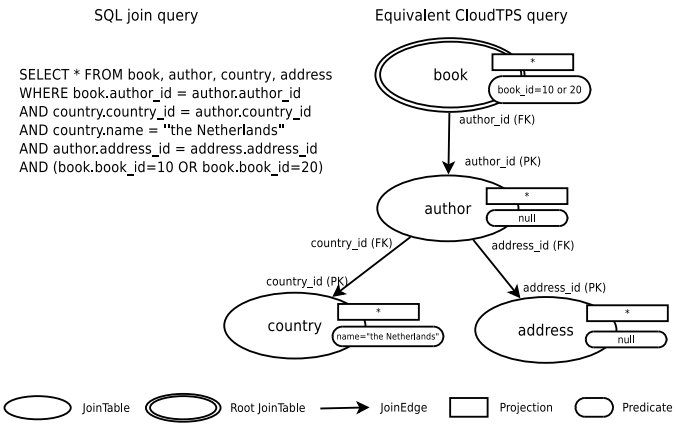


Fig. 2. CloudTPS's representation of a join query

Join queries must give an explicit list of primary keys referring to initial records found in one table. These records and the table where they are called the "root records" and "root table" of this query.

We consider only *inner-joins* which return all matching records, and where 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.

C. API

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

Join queries are 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 (a list of attributes to be returned) and possibly a predicate (a condition that a record must satisfy to be returned). A JoinEdge object represents a join operation between two tables. It contains references to the two JoinTable objects, the names and properties (i.e., PK or FK) of join attributes. A join query must designate one JoinTable object as the root table, which contains the list of primary keys of the root records. Multiple JoinTable objects are joined together using JoinEdge objects of which each matches the FK from one JoinTable to the PK of another. Self-join queries are supported 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" selects only books with an author from the Netherlands.

CloudTPS also handles read-only and read-write transactions defined as a "Transaction" java object containing a list of "SubTransaction" objects. Each sub-transaction represents an atomic list of operations 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.

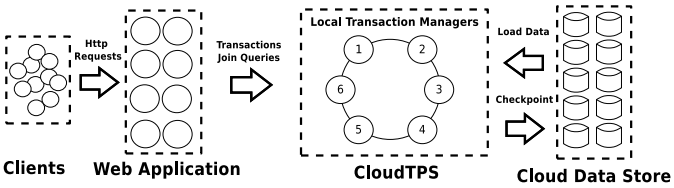


Fig. 3. CloudTPS system model

IV. 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. Note that the underlying Cloud data stores do not need to 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 [19] 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 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. LTMs employ a replacement policy so that unused data items can be evicted from their memory (the caching policy is discussed in details in [7]). CloudTPS expects LTMs to be connected by a low-latency network as in a data center.

A. Join Algorithm

1) Join queries: Intuitively, processing a join query spanning multiple tables requires to recursively identify matching records, starting from the root records (known by their primary keys) and following JoinEdge relationships. The method to identify the matched records, however, differs 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.

Forward query	SELECT * FROM author,book WHERE book.book_id =10 AND book.author_id=author.author_id
Backward query	SELECT * FROM author, book WHERE author.author_id =100 AND book.author_id=author.author_id

Table author

author_id (PK)	CloudTPS index entries
100	book.author_id (10,30)
101	book.author_id (20)

Table book

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

Fig. 4. Index data layout, with two types of join queries

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 is necessary to scan the full table and search for all records whose FK is equal to the PK of the known record. We name such join queries "backward join" queries. To avoid a prohibitively expensive full table scan for each backward join query, we use a technique similar to "join indices" in centralized databases [20], [21]. 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, for the same data schema as in Figure 1. The table book contains a FK author_id referring to the PK value of table author. The forward join query can be processed directly without any indexes, as the FK author_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. CloudTPS creates these indexes upon the declaration of the data schema, then maintains their consistency automatically. In Figure 4, the author record of PK(100) contains two index attributes showing that this record is referenced by two book records with PK 10 and 30. Using these indexes, the backward join query in Figure 4 can then efficiently identify the two matching book records.

2) Secondary-key queries: We use a similar solution by building explicit indexes. However, unlike joins, there exists no table where we can add index information. Instead, we create a separate index table for each SK. Each record of the index table has its PK equal to the SK of one or more records, of which the PKs are stored as its index attributes. A secondary-key query can then translate into a forward join query between the index table and the original table. Table I shows an example secondary-key query which searches records by the SK title of table book. The translated 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.

TABLE I
TRANSLATING A SECONDARY-KEY QUERY INTO A JOIN QUERY

Original	SELECT * FROM book WHERE book.title="bookTitle"
Translated query	SELECT * FROM book, indexOf_bookTitle WHERE indexOf_bookTitle.title="bookTitle" AND book.title=indexOf_bookTitle.title

B. 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 [7]. 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 transaction execution. 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. We therefore 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.

This paper uses the same mechanisms as in [7] to implement the Isolation, Consistency and Durability properties:

Consistency means that a transaction executing on a database that is internally consistent, will leave the database in an internally consistent state. We assume that consistency rules are applied within the logic of transactions, so Consistency is 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 concurrent transactions. In CloudTPS, each transaction is assigned a globally unique timestamp. LTMs are required to execute conflicting transactions in the order of their timestamps. Transactions which access disjoint sets of data items can execute concurrently.

Durability means 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. During the time between a transaction commit and the next checkpoint, durability is ensured by replicating the data items and transaction states across several LTMs.

We now turn to **Atomicity**: either all operations of a transaction succeed successfully, or none of them does. 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 issue a two-phase commit (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, 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.

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 identify matching records only after accessing the root records. We therefore 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. The LTM submits the new sub-transaction 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 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. The coordinator therefore does not need to send the "commit" messages to the involved LTMs. 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 root records first, then add matching records to the transaction during the query execution. For example, in Figure 5(b) the coordinator is initially only aware of two root records held by LTM 15 and LTM 66. After executing its sub-transaction, LTM 15 identifies a matching record 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 together and returning the result to the client.

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

2) Read-Write Transactions for Index Management: CloudTPS transparently creates indexes on all FKs and SKs. 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/updated/deleted, the corresponding index attribute must also be adjusted. To enforce strong data consistency, these affected index attributes must be updated within the same read-write transaction. Considering the example 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 indexes automatically, so 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 commit protocol for read-write transactions to dynamically add sub-transactions. 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 can commit 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 implement transparent index management with this protocol. Whenever a sub-transaction is 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. Initially, 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 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.

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

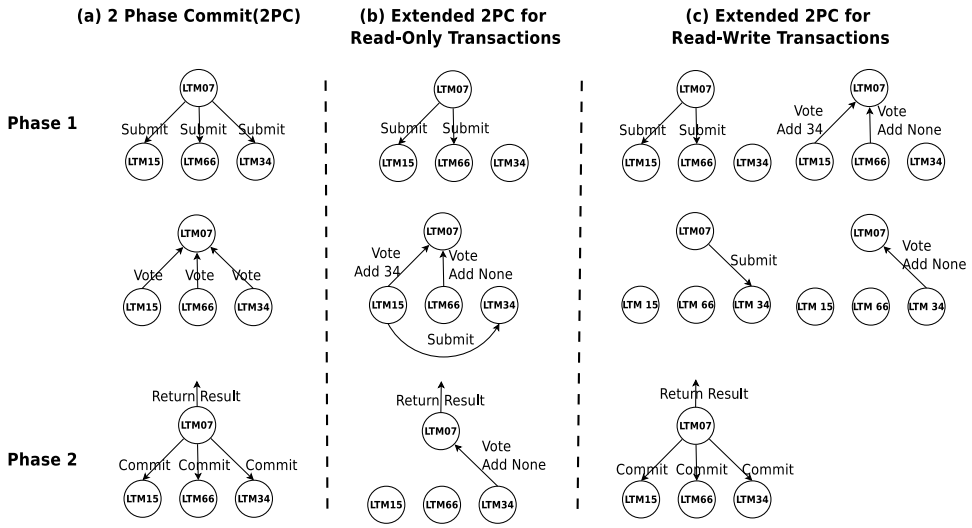


Fig. 5. Two-phase commit vs. the extended transaction commit protocols

as in our previous work. We therefore briefly introduce the main concepts here, and refer the reader to [7] 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.

During a network partition, LTMs are divided into multiple disjoint groups. In this case, according to the CAP theorem, we decide to guarantee consistency at the possible cost of a loss of availability. A partition may proceed accepting transactions provided that: (i) this partition is able to elect itself as the "majority" partition, of which the LTMs represent more than half of the previous membership; and (ii) its LTMs can recover the consistent states of all data items. In all other cases the system will reject incoming transactions until the partition is resolved and it fulfills the condition again. If a majority partition exists and manages to complete the recovery, the other LTMs will discard their entire states and rejoin when the network partition is resolved.

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.

V. Evaluation

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

A. Experiment Setup

1) System Configuration: We execute CloudTPS on top of SimpleDB running in the Amazon Cloud, and HBase v0.20.4 running in our local DAS-3 cluster [22]. 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 cluster. Each node 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 with a Gigabit LAN. In Amazon EC2 we use Medium Instances in the High-CPU family, which have 1.7 GB of memory, 2 virtual cores, and 350 GB of storage.

2) Throughput Measurement: Given a number of LTMs, we measure the maximum sustainable throughput under a constraint of response time. In DAS-3, we define a demanding constraint where 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 Amazon EC2, LTMs share a physical machine with other instances, and have less control over hardware resources. Furthermore, even multiple virtual

instances of the same type often exhibit different performance behavior [23]. To prevent these interferences from disturbing our evaluations, we relax the response time constraint in EC2: 90% of transactions must return within 100 ms.

To determine the maximum throughput of CloudTPS, we issue workloads with increasing request rates until the response time constraint is violated. Workloads are generated by a configurable number of Emulated Browsers (EBs), each of which issues requests from one simulated user. Each EB waits on average 1 second between receiving a response and issuing the next transaction.

Throughout the evaluation, we provision sufficient resources for clients and the cloud data store, so that CloudTPS remains the performance bottleneck. We configure CloudTPS to use one backup for each transaction and data item.

B. Microbenchmarks

We first study the performance of join queries and read-write transactions individually in CloudTPS using microbenchmarks. In this set of experiments, we deploy CloudTPS over 10 LTMs.

1) Workload: Two criteria influence the performance of join queries in CloudTPS: the number of data items 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 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 the same table. We can therefore generate join queries of arbitrary critical execution path length. Given the length of the critical execution path, we can also control the number of accessed records by varying the number of root records. This table contains 10,000 records.

CloudTPS uses caching strategies to prevent LTMs from memory overflow when loading application data. To avoid performance interferences, in all micro-benchmark we ensure that the hit rate remains at 100%.

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 6(a), when the number of accessed data items per transaction increases, the throughput expressed in accessed records per second remains largely constant. 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 lengths of critical execution paths. Figure 6(b) shows the impact of the critical execution path length. Simple primary key queries (with query length 0) outperform join queries (with query length at least 1) by roughly 50%. 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.

3) Read-Write Transactions: We now turn to 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.

Figure 6(c) shows that the number of record accesses per second remains roughly constant. The bottleneck is therefore the update of individual data items rather than the transaction overhead. We also note that transactions which only update data records and transactions which also update indexes exhibit very similar performance. This means the updating index records does not degrade the system performance significantly. One only needs to pay the price of updating the extra index records.

C. Scalability Evaluation

1) Evaluation setup: We now evaluate the scalability of CloudTPS under a demanding workload derived from the TPC-W Web application [24]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore. It is important to note that TPC-W was originally designed for relational databases. It therefore contains the same mix of join queries and read-write transactions as cloud-based applications would if their data store supported join queries.

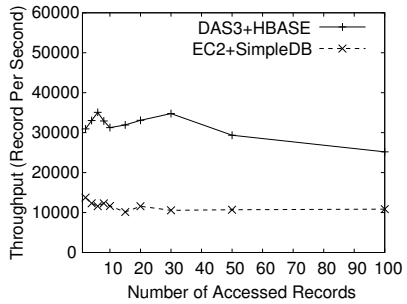
Deploying TPC-W in CloudTPS requires no adaptation to the database schema. We also kept all simple and complex queries unchanged, and merely translated them to CloudTPS's tree-based representation as discussed in Section III-C. TPC-W contains one secondary-key query which selects a customer record by 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.

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.

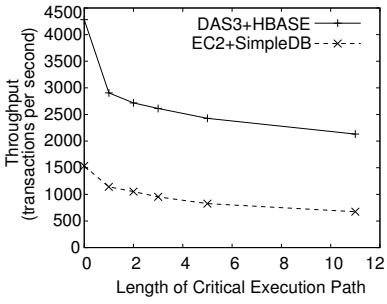
TPC-W contains 10 database tables. We populate the TPC-W database with 144,000 customer records in table "Customer" and 10,000 item records in table "Item." We then populate the other 8 tables proportionally according to the TPC-W benchmark requirements. TPC-W continuously creates new shopping carts and orders. During our scalability evaluation, we observe a hit rate around 80%.

2) Linear Scalability: Figure 7(a) depicts CloudTPS's scalability in the Amazon cloud. The overall system throughput grows linearly with the number of LTMs, which means that CloudTPS can accommodate any increase of workload with a proportional number of compute resources.

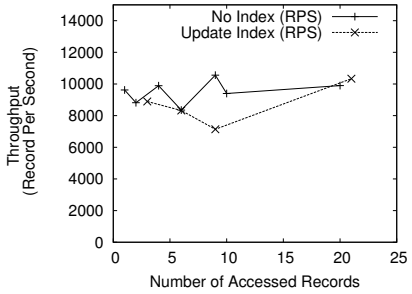
Figure 8 shows the distribution of the number of data items accessed per transaction under the configuration of 40 LTMs.



(a) Join queries with different numbers of accessed data items

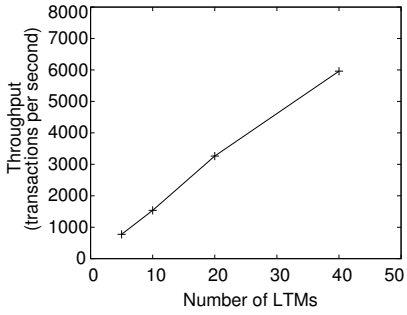


(b) Join queries with different length of execution path

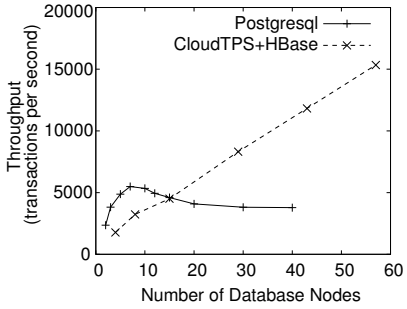


(c) Read/write transactions with different numbers of accessed data items

Fig. 6. Throughput with different types of join queries

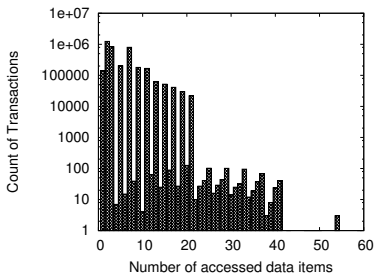


(a) Scalability of CloudTPS in Amazon EC2 + SimpleDB

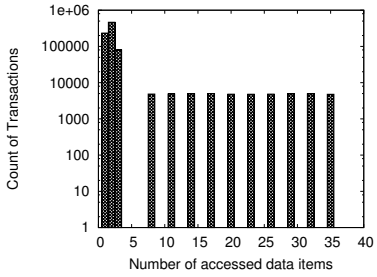


(b) Scalability of CloudTPS vs. PostgreSQL in DAS3

Fig. 7. Scalability of CloudTPS under TPC-W workload of join queries and read-write transactions only.



(a) Read-Only Transactions



(b) Read-Write Transactions

Fig. 8. Number of items accessed by transactions.

On average, read-only transactions access 4.92 data items and read-write transactions access 2.96 data items. 82% of transactions are join queries, while 18% are read-write. About 33% of read-only queries have a query length of one while the other 67% have a query length of two. About 84% of read-write transactions update indexes.

3) Comparison with a Relational Database: We compare CloudTPS with PostgreSQL v.9.0 on DAS-3. The PostgreSQL setup contains one master and N slaves, using the "Binary Replication" mechanism [25]. We issue all read-write transactions on the master, and balance read-only queries across the slaves. 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.

Figure 7(b) illustrates the differences between CloudTPS and a replicated relational database. In small systems, PostgreSQL significantly outperforms CloudTPS because each slave can execute read-only queries locally. PostgreSQL reaches a maximum throughput of 5493 TPS 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 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 TPS 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 TPS using 57 nodes (40 machines for CloudTPS and 17 machines for HBase). This clearly

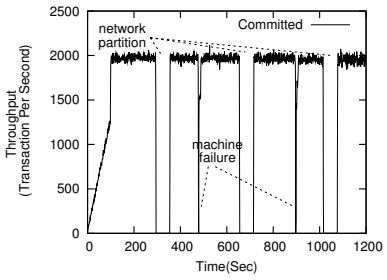


Fig. 9. Throughput across network partitions and node failures.

illustrates the scalability benefits of CloudTPS compared to a replicated relational database.

D. Tolerance of node Failures and Network Partitions

Finally, we illustrate CloudTPS's resilience to machine failures and network partitions, and more generally to any change in the system membership. We configure CloudTPS with 10 LTMs in DAS-3 and alternately create 3 network partitions and 2 machine failures. Each partition lasts 1 minute.

As shown in Figure 9, 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. Most of this time is spent recovering failed transactions, and would not appear during graceful reconfigurations. For network partitions, no data re-replication is necessary so CloudTPS recovers almost instantly after the network partition is resolved. In all cases the ACID properties are respected despite the failures.

During recovery from an LTM failure, the throughput drops to zero. This is due to a naive implementation which simply aborts all incoming transactions during recovery. A smarter implementation could abort only the transactions accessing the failed LTMs. We consider this as future work.

VI. 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 complex join queries do not necessarily contradict each other. CloudTPS exploits the fact that Web applications mostly use join queries which access 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 transactional protocols to address the needs of read-only join queries as well as read-write transactions which transparently update index values at runtime. The system scales linearly in our local cluster as well as in the Amazon Cloud.

References

[1] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[2] J. Elson, , and J. Howell, "Handling flash crowds from your garage," in *Proc. Usenix ATC*, 2008.

[3] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proc. VLDB*, 2000.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable : a distributed storage system for structured data," in *Proc. OSDI*, 2006.

[5] Amazon.com, "Amazon SimpleDB." 2010, http://aws.amazon.com/simpledb.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[7] W. Zhou, G. Pierre, and C.-H. Chi, "CloudTPS: Scalable transactions for Web applications in the cloud," *IEEE Transactions on Services Computing*, vol. PrePrints, 2011.

[8] "Wikibench – the realistic web hosting benchmark," http://www.wikibench.eu/.

[9] T. Hoff, "NoSQL took away the relational model and gave nothing back," High Scalablility blog, Oct. 2010, http://bit.ly/dCVu1a.

[10] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proc. OSDI*, 2010.

[11] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. CIDR*, 2011.

[12] J. J. Levandoski, D. Lomet, M. F. Mokbel, and K. K. Zhao, "Deuteronomy: Transaction support for cloud data," in *Proc. CIDR*, 2011.

[13] S. Das, D. Agrawal, and A. E. Abbadi, "G-Store: a scalable data store for transactional multi key access in the cloud," in *Proc. SoCC*, 2010.

[14] S. Plantikow, A. Reinefeld, and F. Schintke, "Transactions for distributed wikis on structured overlays," in *Proc. DSOM*, 2007.

[15] H. Vo, C. Chen, and B. Ooi, "Towards elastic transactional cloud storage with range query support," *Proc. VLDB*, 2010.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a high-performance, distributed main memory transaction processing system," in *Proc. VLDB*, 2008.

[17] S. Das, D. Agrawal, and A. E. Abbadi, "ElasTraS: An elastic transactional data store in the cloud," in *Proc. HotCloud*, 2009.

[18] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proc. VLDB*, vol. 3, pp. 48–57, September 2010.

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proc. STOC*, 1997.

[20] P. Valduriez, "Join indices," *ACM Trans. Database Syst.*, vol. 12, pp. 218–246, June 1987.

[21] P. O'Neil and G. Graefe, "Multi-table joins through bitmapped join indices," *SIGMOD Rec.*, vol. 24, pp. 8–11, September 1995.

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

[23] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 performance analysis for resource provisioning of service-oriented applications," in *Proc. NFPSLAM-SOC*, 2009.

[24] D. Menascé, "TPC-W: A benchmark for e-commerce," *IEEE Internet Computing*, vol. 6, no. 3, 2002.

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