Transactional Replication in Hybrid Data Store Architectures

Hojjat Jafarpour
NEC Labs America
hojjat@nec-labs.com

Junichi Tatemura
NEC Labs America
tatemura@nec-labs.com

Hakan Hacıgümüş
NEC Labs America
hakan@nec-labs.com

ABSTRACT

We present a transactional and concurrent replication scheme that is designed for hybrid data store architectures. The system design and the requirements are motivated by the real business cases we encountered during the development of our commercial database product. We consider two databases where the original database handles read/write transactional application workloads while the second database handles read-only workloads from the same applications over the data periodically replicated from the original database. The main requirement is ensuring the application of the updates on the replica database in the exact same order they were executed in the original database, which is called execution-defined order. Although this requirement could easily be satisfied by the serial execution of the updates in the commit order, doing so in an efficient manner by exploiting concurrency is a challenging problem. We present a novel concurrency control algorithm to addresses that problem by also allowing the read-only workloads on the replica database to interleave with the concurrent replication. The extensive experiments show the efficacy of the proposed solution.

1. INTRODUCTION

Increasingly more organizations are using multiple database types side-by-side instead of trying to fit one database to all data management needs. The reason is each database product could be better fit for different business requirements. We call the use of multiple database types in the same computing environment hybrid data store architecture.

It is natural that data need to be replicated among those data stores, as the upstream applications ideally would like to use the underlying databases in a seamless fashion without worrying about and the availability of the data sets at a certain location.

An interesting hybrid architecture we are observing is using key-value stores along with relational databases. The relational databases have well known strengths and long, successful history in transactional data processing. Key-value stores have gained popularity for their seamless scalability and elasticity and also their lower-cost profile.

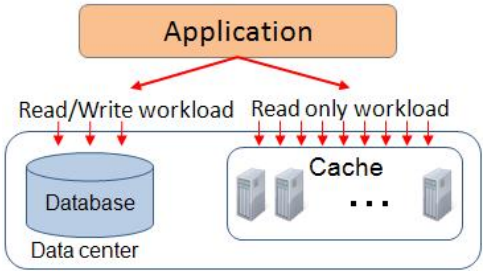

Figure 1: Caching for web applications.

In this work, we specifically focus on the replication, where data need to be replicated from a relational database to a key-value store. The system design and the requirements are motivated by the real business cases we encountered during the development of our commercial database product, Partiqle [20], which is commercialized under the name of IERS[1]. Partiqle is an elastic transactional SQL engine that is implemented on top of a key-value store. In a typical scenario the users already have a traditional relational databases product in use. They want to increase the scalability of the database with the increasing demand from the applications. However, the users don't prefer to scale-up the traditional relational database, instead they consider the scale-out approach through a key-value store, which is the replica of the relational database and serves a specific and demanding part of the workload. Naturally, the main requirement is ensuring the application of the updates on the replica database in the exact same order they were executed in the original database, which is called *execution-defined order*.

Another prominent example of such setting is caching, where data from relational database are cached in a large-scale cache cluster implemented as a key-value store, such as memcached [3]. Memcached is used to significantly reduce the read load on the database system by caching frequently read application generated data in a scale-out in-memory key-value cache. Figure 1 depicts the architecture of a web application that uses memcached. Indeed, most of the largest web applications including Facebook, YouTube, Twitter and Wikipedia are already using memcached as the key scale-out technology in their architecture.

An extreme case of application layer caching approach is

[1]http://www.nec.com/iers

10.5441/002/edbt.2015.58

to cache all the working set data instead of only parts of it. In fact, in this case all the working set data in database is replicated in the cache system that can either be a memory based system like memcached or disk based key-value store system like memcachedb [4] or membase [2] to provide data persistence and recovery. In addition to having the all advantages of normal caching system, such a replication-based system eliminates the cache miss possibility for applications. This is very beneficial when the access pattern to the data is such that reduces the probability of cache hit. Uniform distribution of data access is one of such patterns that may reduce usability of caching. The replication approach also simplifies the application developers' job by not requiring explicit cache value update or invalidation since all the updates are propagated to the replicated data automatically.

With all the benefits of such caching, a major issue that may arise is the exposition of stale data to application, which happens because the replica always lags behind the original data. If there is a high transactional update load on the original data, the replica may significantly be out of sync. Therefore, shortening the lag for the replica would significantly reduce the probability of exposing stale data to the application.

As shown in Figure 1, the relational database handles transactional read/write workload and the key-value store is responsible for handling a read-only workload. The transactional updates in the original database are shipped to the key-value store and applied in the same order to guarantee the correct state for the replica. We call this order as *execution-defined order*. Transactions may interleave during their execution against the original database. However their correct order is defined by their original execution and that order should be respected when doing the log replay for replication. Consequently we have the following requirements:

- The replication process should respect the execution-order of the transactions in the original database.
- The replication process should be efficient to increase replication speed and reduce replica lag.
- The read-only access should be allowed to interleave with an on-going replication process.

The first requirement states that the replication algorithm should guarantee that the resulting serialization order for transactions in the replica is exactly the same as the serialization order in the original database and no other serialization order is acceptable. To illustrate the problem, consider two transactions in Figure 2. If T_i is executed before T_{i+1} then the data item with key Key_k will not exist in the data store. On the other hand, if T_{i+1} is executed before T_i the data item with key Key_k with value *Object* will exist in the data store. Therefore, although both executions are correct from serialization point of view, the second execution is not acceptable since it does not result in the correct state considering the predefined execution order.

The above requirement could be trivially implemented by replaying the update values in commit order *serially*. However this kind of serial execution would be prohibitively in efficient – the second requirement. Therefore the replication algorithm should improve the efficiency by exploiting concurrency while still respecting the execution-defined order and allowing the read-only workload on the replica to see consistent database state concurrently.

As we discuss in the related work section, although there are related methods in the literature, none of them directly meets the requirements in the given system settings.

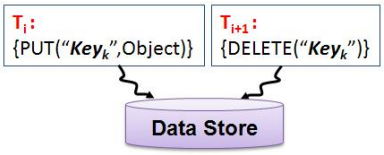

Figure 2: Impact of serialization order on system state.

The contributions of this paper can be summarized as follows:

- We present a replication algorithm that exploits concurrent execution for efficiency while guaranteeing the execution-defined order in the replica database and allowing read-only application workloads to interleave on the replica database.
- We present the architecture of the system, *TxRep*, we implemented based on the replication method presented in the paper and hybrid data store architecture. The actual system is used to generate our experimental results presented in the paper.
- Through extensive experiments we evaluate the performance of the proposed transactional replication algorithm under variety of parameters.

The rest of the paper is organized as follows. We review the related work in Section 2 and then present the system architecture and the consistency model in Section 3. Section 4 describes query translator followed by the details of our transaction manager and concurrency control algorithm in Section 5. We present our experimental results in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Application level caching systems such as memcached [3], memcachedb [4] and membase [2] have been used as scale out solution in many web applications. However, such systems do not provide any transactional consistency guarantees for data access and updates with the rest of the system. Transactional cache (TxCache) provides transactional access to application level caching systems such as memcached [19]. TxCache guarantees that any data accessed regardless of being in cache or database is consistent based on a valid snapshot of the database. It may result in stale snapshots which is acceptable for web applications. Unlike TxCache and other application level caching approaches, our proposed scale out approach replicate whole data base in the key/value store and prevents all read transactions from hitting the relational database.

In the relational database context, a common approach for scale-out is replication [13]. In [17] Manassiev et. al., present a replication technique for scaling and continuous availability of relational databases. The approach assigns a master for each conflict class where all update transactions

for the class are sent to the corresponding master. Each master node has a set of slaves that are its replicas and serve the read-only transactions in the system. Updates are disseminated from master to its slave nodes either eagerly upon their arrival or lazily by packing several updates and applying them together. This approach can be further improved by using a modified version of our algorithm in applying the updates on slave nodes.

Providing equivalence to a predefined serialization order has been explored in the context of relational databases. Conservative timestamp ordering guarantees the execution order based on the assigned timestamps to the transactions [10]. By delaying the execution of operations until completion of execution of conflicting operations with smaller timestamp, conservative T/O guarantees there will be no conflicts in execution of each operation. In practice, this approach serializes all write operations in the database. The improved version of this protocol, SDD-1, tries to provide more concurrency by using transaction classes [11]. Transactions are places in transaction classes and only potentially conflicting transaction classes should be dealt with conservatively.

Our concurrency control algorithm is very similar to concurrency control by validation [16]. This approach also assigns three states to transactions, START, VAL and FIN which are equivalent to START, COMITTED and COMPLETTED states in our algorithm. However, in our algorithm we have a predefined order that we should follow while in the validation-based case the order is decided in the validation phase. In the validation-based case if a transaction cannot be validated it is simply rolled back which will result in a different serialization order compared to the case where the transaction could be validated. But in our case this is not acceptable and we strictly should follow the predefined order.

Jiménez-Peris et. al., proposed a deterministic thread scheduling to enable replicas to execute transactions in the same order [14]. This approach requires careful consideration of the impact of interleaving of local threads and scheduled threads.

In [21] Thomson and Abadi propose a distributed database system that guarantees equivalence to a predetermined serial ordering of transactions by combining a deadlock avoidance technique with concurrency control schemes. All the transactions in the system go through a preprosessor component that determines the execution order and then are propagated to replicas using a reliable, totally ordered communication layer. The conflicting transactions are aborted and retried, however, all abort and retry actions are deterministic although the order may change by preprosessor. In our proposed approach, on the other hand, we do not have the possibility of changing serialization order and we should follow the predefined order strictly all the times.

Polyzois and Garcia-Molena proposed a similar algorithm for remote backup in transaction processing systems [18]. To execute transactions in the backup they use tickets to order transactions and two phase locking protocol for concurrency. Each transaction requests lock on the items that it needs, however, the locks are granted according to the transaction ticket number and the protocol ensures that no lock is granted to a transaction unless all the transactions with the smaller ticket that requested the same lock have been granted. Unlike this approach our concurrency control algorithm follows a technique similar to optimistic concurrency control.

3. SYSTEM ARCHITECTURE

Figure 3 illustrates the architecture our implemented system. We assume a standard relational database as the database that stores all the persistent application data. The database system provides a SQL interface for the applications and is responsible for handling read/write transactional workload. It is important to note that, we do not change the standard relational database's API's, query execution mechanisms, or optimizations.

To improve the performance of the database system for certain application workloads, the database is replicated into a distributed key-value store. The key-value store can be memory-based or disk-based store. The replicated key-value store plays similar role to cache for the database and is used to handle the read only workload while the read/write workload bypasses the key-value store and is run directly on the database. The system does not have any specific assumption about the key-value store and as long as the store provides standard PUT/ GET/ DELETE interface to access data, it can be used in our system. For instance, we can use memcached[3], memcachedb[4] or Dynamo [12] as the key-value store in our system. In our system we provide both key-value store API (PUT/ GET/ DELETE operations) and also SQL API to the key-value store. The key-value store API is the native API and it does not require any additional component. To enable SQL API to the key-value store we used our Partiqle system [20]. There are other examples of providing SQL-like APIs to key-value stores, such as UnQL[2] and CQL[3], which could also be used for this purpose.

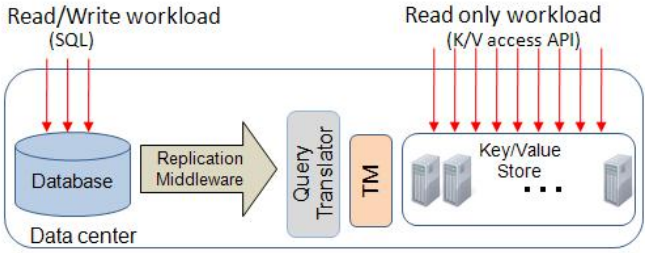

Figure 3: Scale out architecture of TxRep using key-value store.

Between the relational database and the key-value store we have three main components of our system that are responsible for synchronizing the key-value store with the relational database.

The Replication Middleware component is responsible for shipping the transaction log from the relational database to the replica in the key-value store. It periodically reads the transaction log in the database, packs the new transactions into a relocation message and ships the messages to the key-value store. Note that the transaction log only includes write statements and there is no need to apply read statements from the relational database in the replica. We used an MQ-based system (Apache ActiveMQ) as the replication middleware. Although it is an important part of the system

[2]http://www.unqlspec.org
[3]Cassandra Query Language: http://www.datastax.com

architecture, the replication middleware details are not directly relevant for the concurrent replication method, which is the main focus of the paper. Therefore, we give some details of the component in the appendix for the interested readers.

The Query Translator (QT) component is responsible for translating the update only SQL statements into native key-value store API operations that can be directly executed on the key-value store. Note that the replication workload only contains the write operations to key-value store. We will discuss details of the QT component shortly.

The Transaction Manager (TM) component is used to apply the transactions to the key-value store concurrently. The TM component essentially implements the proposed concurrent replication method in the paper. Note that when the transactions reach the TM they are in the form of native key-value store API as they have been translated by QT component. The result of applying the transactions should be exactly the same as applying them in serial manner with *execution-defined order*. We will provide more details on the TM in Section 5.

3.1 Consistency model

Our system provides both transactional and non-transactional access to the stored data. The read/write transactions bypass the key-value store and are run directly on the database. Non-transactional read only workload can be directly executed on the key-value store. This is the same access method that is provided in systems like memcached and memcachedb. For such workload the only consistency guarantee is the one that is provided by the key-value store and the read data may also be stale. Most of the existing key-value store systems can provide key level consistency guarantees where access to single key-value item (a single PUT, GET or DELETE operation) can be atomic.

4. QT: RELATIONAL DATA IN K/V STORE

In this section we present the details of the Query Translator component. We first present the data layout on a key-value store and then describe how transaction logs in SQL format are translated into key-value store API operations. In order to facilitate the discussion, we use a modified version of TPC-W benchmark [8] schema as a running example. Figure 4 depicts the modified relational schema. When a customer orders an item, a new tuple corresponding to the order is added into the order relation.

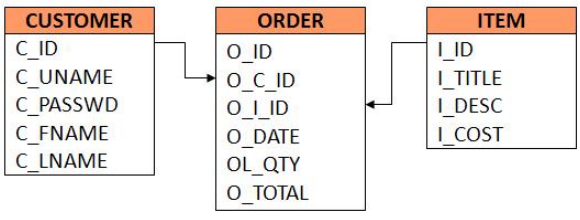

Figure 4: A relational schema for a web based store.

4.1 Relational data over key-value store

Since we replicate the relational data in RDBMS into a key-value store and the data layout on these two stores are different, we need to provide a mapping scheme to map the relational data layout into the key-value data layout.

The first step in storing relational data in the key-value store is to store data in relations. To store the tuples of a relation in the key-value store we represent each tuple as a key-value object. We construct the key for each tuple by combining the name of the relation and the primary key. This generates a unique key for each tuple in each relation. The value for the generated key is the set of all fields for tuple. For instance, consider the ITEM table in our example. Figure 5 depicts three tuples in this table. To represent each tuple as a key-value object, we first create a unique key for each tuple by concatenating the name of the relation with the primary key. For the first tuple the key will be "ITEM_1". The corresponding value for the first tuple will be the set of fields in the tuple, {1, 'Item1', 'Item1_Desc', 100}.

I_ID	I_TITLE	I_DESC	I_COST
1	Item1	Item1_Desc	100
2	Item2	Item2_Desc	150
3	Item3	Item3_Desc	75

Figure 5: Tuples in item table.

The above mapping of relational data into key-value objects provides primary key access to tuples where the application can query, read and write each tuple by its primary key. However, tuples cannot be accessed using any other attribute. For instance, a query cannot access an item based on its cost which is the value of "I_COST" column in ITEM table. This is because there is no key in the key-value store that provides access to item tuples using their cost value. Note that, usually, we are not allowed to scan the entire table: such an operation, which spans over the entire key-value nodes is very inefficient and is not affordable for web applications where the response time is limited. In order to provide access to the tuples through an attribute other than primary key, we create a *hash index* for the attribute in the key-value store. A hash index structure is composed of set of key-value objects. For each distinguished value for the indexed attribute, we create a key-value object. The key is constructed using the value of the indexed attribute and the value for the object is the set of keys for the tuples that the value of their corresponding attribute is the same as the value that was used to construct the key. As an example consider we want to provide access to items through the item cost. For each cost value in the ITEM table we should create a key-value object. The key for such an object is constructed using the relation name, which is "ITEM", the attribute name which is "COST" and the value of the cost column. The value of the object is the keys for the tuples that have the same cost. Figure 7 shows a hash index for the cost attribute in the ITEM relation. Assuming the cost of items with id 1 and 7 is 100, the value for the hash index object with key "ITEM_COST_100" will be "ITEM_1" and "ITEM_7" which are the keys to access the tuples corresponding to these items.

4.2 Range index using B-link tree

Although we can use a hash index to access tuples through any of their attributes, we cannot use it to answer range queries and queries that need to scan a whole table. To pro-

Key	Value
ITEM_1	{1,Item1, Item1_Desc, 100}
ITEM_2	{2,Item2, Item2_Desc, 150}
ITEM_3	{3,Item3, Item3_Desc, 75}

Figure 6: Key/value objects for the tuples in ITEM table.

Key	Value
ITEM_COST_100	[ITEM_1 , ITEM7]
ITEM_COST_150	[ITEM_2]
ITEM_COST_75	[ITEM_3 , ITEM14 , TEM21]

Figure 7: A hash index to access items with their cost value.

vide this capability over key-value store we propose another index structure that is based on B-link tree [15]. B-link tree is a concurrent B-tree that reduces the lock usage for efficiency. A B-link tree is a B^+-tree with an extra pointer in each node. This extra pointer in a node points to the right sibling of the node in the tree. Using this extra pointer, *look up* operation in B-link tree do not need to acquire any locks and *insert* and *delete* operations need to acquire locks on a small number of nodes. We create a key-value object for each B-link tree node. Hence, (1) conflicts among write operations are translated to conflicts on key-value store API operations, which are managed by the TM component (instead of locking), and (2) read-only transactions can access the B-link tree mapped on a key-value store without being blocked by updates.

4.3 SQL statement translation

The replication is done by shipping the transaction log from the relational database and applying the database update operations on the key-value store. Therefore the translation between the relational transactional log to key-value store API operations (such as PUT) involves translating INSERT / UPDATE / DELETE statements from the database log. This operation is not particularly difficult and we used our existing system components from Partiqle system [20] for this purpose.

5. TM: CONCURRENCY CONTROL FOR REPLICATION

The transaction manager (TM) component is responsible for concurrently executing transactions on the key-value store. A transaction starts with *start* statement and ends with *commit* statement. We consider the transactions that are executed by TM are the ones in transaction log of the relational database that were shipped by the replication middleware. However, read-only transactions from application can also be interleaved with the shipped update transactions if they need transactional access to the replicated data in the key-value store. To maintain the correctness of the replicated data in the key-value store the transaction manager should guarantee that the result of concurrent execution of the update transactions shipped from the database is exactly the same as serial execution of them in the same order as they were executed in the database. The simple way to provide such guarantee is to execute all the transactions in the key-value store side serially. However, if the update rate in the database is high, the replica could significantly lag behind the database and increase staleness of the data in the key-value store. It may also significantly reduce the throughput of the read only transactions that are being executed on the key-value store side.

To address this issue we can execute transactions concurrently, however, we need to provide concurrency control to guarantee correctness of the transaction executions. Such concurrency control mechanism is different from the ordinary concurrency control systems because of the execution-defined order of transactions. In an ordinary concurrency control algorithm when a set of transactions are executed, as long as the result of the execution is equal to *some* serial order of transaction execution the result is acceptable. However, in our case the concurrency control algorithm has to guarantee that the result of concurrent execution of transactions is exactly the same as the result of serial execution of them in the same order that they were already executed in the database. Therefore, the existing concurrency control algorithms cannot be used in our TM component.

We propose a new concurrency control algorithm that provides such a guarantee while executes transactions concurrently.

The algorithm receives a set of transactions as input and uses a set of threads in a threadpool to execute the transactions concurrently. Similar to the ordinary concurrency control algorithms we consider two types of conflicts, *read/write* conflict and *write/write* conflict. In read/write conflict two operations conflict if one of them is GET and the other is PUT or DELETE and both access the item with the same key. For instance, the following operations have read/write conflict: $GET(\text{“}key1'')$, $PUT(\text{“}key1'', object1)$. In write/write conflict two operations conflict if they are PUT or DELETE and both of them access the item with the same key. For instance, the following PUT operations have write/write conflict: $PUT(\text{“}key2''\ , object2)$, $PUT(\text{“}key2''\ , object3)$.

Two *concurrent* transactions conflict if and only if there is at least one read/write or write/write conflict between their corresponding PUT/ GET/ DELETE operations. Note that if two transactions are not concurrent, i.e., one starts *after* the other completes, they do not conflict even if there are conflicting operations. In order to define concurrency on the key-value store, we assume that the underlying key-value store provides consistent read-write, meaning that a write operation (PUT / DELETE) is atomic and its effect is immediately available for read (GET) operations (no stale data is read). This feature either is supported or can be added easily in most of key-value stores including Dynamo and HBase [12, 1].

Another important assumption that we have is that each transaction is assigned with a sequence number that indicates the place of the transaction in the ordered list of transactions. The sequence number for the update transactions can be easily assigned when the ordered list of transactions are generated by the publisher agent in the database side or when the list is received by the subscriber agent in the key-value store side. However, since we may also want to execute some read only transactions in the key-value store side

in a transactional manner we assign the sequence numbers for transactions in the subscriber agent along with assigning sequence numbers to the read only transactions. This guarantees that each transaction has a unique sequence number and the order of sequences for update transactions in the key-value store side is the same as database side while they may be interleaved with read only transactions in the key-value store side.

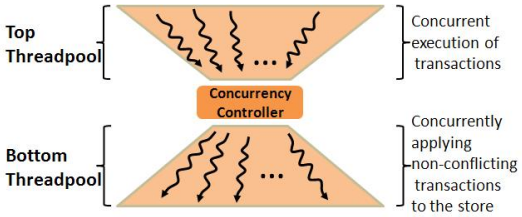

Figure 8: Transaction manager component.

Figure 8 depicts the transaction manager (TM) component in our system. There are two threadpools in the system that provide concurrent execution. The first threadpool which is shown on top of the concurrency controller is used for concurrent conversion of transactions into PUT / GET / DELETE operations with their corresponding data items. We refer to this threadpool as *top threadpool*. Each transaction is run over the key-value store using one thread from this threadpool and is represented as a set of PUT / GET / DELETE operations that is going to be evaluated by the concurrency controller. After evaluation of a transaction by the concurrency controller, if there is no conflict, the transaction is passed to the next threadpool that is shown in the bottom of the concurrency controller where another thread is used to apply the results of the transaction to the key-value store. We refer to this threadpool as *bottom threadpool*.

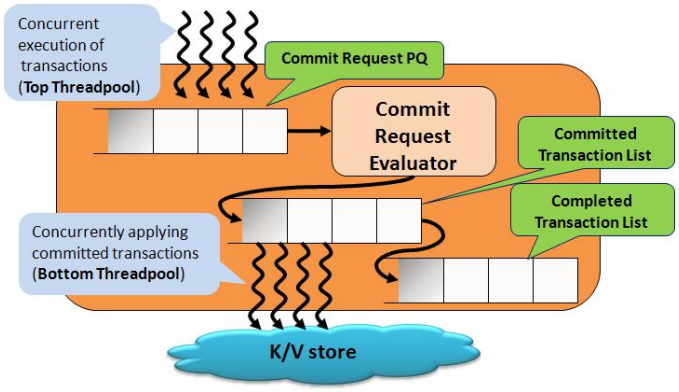

Figure 9: Internal architecture of concurrency controller.

We now present the details of transaction execution in the transaction manager component. The first step in execution of a transaction in the transaction manager is to detect the set of keys that the PUT / GET / DELETE operations in the transaction access. This is done in one thread that is acquired from the top threadpool shown in Figure 8. For each transaction, we find this set of keys using the query translator and a dedicated buffer. After a transaction starts we create an exclusive buffer for the transaction. For every GET operation in the transaction, if the value for the key does not exist in the buffer the value is retrieved from the key-value store. The value is also stored in the transaction buffer for future accesses. On the other hand, if the value for the key exists in the buffer, the GET operation uses the value in the buffer and does not access the key-value store. For every PUT operation in the transaction the corresponding key-value pair is added to the transaction buffer without accessing the key-value store. Therefore, until the commit statement in the transaction all the changes that are being done by a transaction are stored in the transaction buffer and the transaction does not affect data in the key-value store. In this step multiple transactions execute concurrently using the threads in the top threadpool in Figure 8. When a transaction reaches to its commit statement it is passed to the concurrency controller in the transaction manager. The concurrency controller uses our proposed concurrency control algorithm to detect the conflicts among transactions. If there is no conflict between a transaction and the transactions with lower sequence numbers, the updates of the transaction can be executed concurrently and the key-value store can be updated based on the new values in the transaction buffer. Note that a transaction is only checked for conflicts with its predecessors and there is no need to check for conflict with the transactions that have higher sequence number. If a transaction does not conflict with its predecessors, it can commit by applying its operations to the key-value store using one of the threads in the bottom threadpool as shown in Figure 8. Otherwise, if there is a conflict, because of predefined serialization order the transaction with higher sequence number should restart.

Algorithm 1 depicts our concurrency control algorithm in the transaction manager. Figure 9 also illustrates the internal architecture of the concurrency controller and different data structures that are used by concurrency controller. The algorithm receives the transactions in the form of PUT/ GET/ DELETE operations with the corresponding keys for each operation. These transactions are inserted into a priority queue that is used to sort transactions based on their sequence numbers. After generation of set of PUT/GET/DELETE operations for its corresponding transaction, each thread in the top threadpool puts its transaction in the priority queue. The priority queue, which is referred to as *CommitReqPQ* in the algorithm, is responsible for keeping the order of transactions based on ascending order of their sequence numbers. Each transaction can be in one of the following states:

- ACTIVE: An active transaction has started its execution but has not committed yet.
- COMMITTED: A committed transaction is the one that does not have any conflict with its predecessors. However, the updates in its buffer has not been applied to the key-value store.
- COMPLETED: A completed transaction is a committed transaction that the updates in its buffer have been applied to the key-value store.

In addition, each transaction is assigned with the following values:

- *startTime*: The time that the transaction starts its execution. This is the time when the transaction is assigned to a thread from the top threadpool in Figure 8.

Algorithm 1 Concurrency control in Transaction Manager

1: $CommitReqPQ$: Priority queue for commit request.
2: $CommittedTransactionList$: List of committed transactions.
3: $CompletedTransactionList$: List of completed transactions.
4: Transaction T_i is the first transaction in $CommitReqPQ$
5: **if** T_i's sequence number is NOT the next sequence number. **then**
6: goto line 4 (wait for the right transaction).
7: **end if**
8: Remove transaction T_i from $CommitReqPQ$.
9: **for all** $Tj \in CommittedTransactionList$ **do**
10: **if** T_i conflicts with T_j **then**
11: Add T_i to restart list of T_j (T_i will restarts when T_j is completed.)
12: **return**
13: **end if**
14: **end for**
15: **for all** $T_j \in CompletedTransactionList$ **do**
16: **if** T_i.startTime $< T_j$.completeTime **then**
17: **if** T_i conflicts with T_j **then**
18: Restart T_i.
19: **return**
20: **end if**
21: **end if**
22: **end for**
23: Add T_i to $CommittedTransactionList$.
24: Change the expected sequence number to $i + 1$.
25: In a new thread from bottom threadpool:
Execute T_i's statements.
Move T_i to $CompletedTransactionList$ when the execution is complete.
Restart the transactions in T_i's restart list.

- *commitTime*: The time that the concurrency control algorithm detects that the transaction does not have any conflict with its predecessors.
- *completeTime*: The time that the updates in the transaction buffer for a committed transaction are applied to the key-value store.

The algorithm also uses two lists, one for the committed transactions and one for the completed transactions. The committed transaction list holds the transactions that are in COMMITTED state and have committed successfully. Note that, although these transactions are considered committed, the effect of their execution which is stored in their corresponding buffers have not been applied to the key-value store. The completed transaction list contains the transactions that are in COMPLETED state which are the committed transactions that have also been applied to the key-value store.

The concurrency control algorithm starts by checking the first transaction in the *CommitReqPQ*. If this transaction's sequence number is not the expected sequence number the algorithm does nothing and waits until the transaction with the expected sequence number is put into the CommitReqPQ. If the transaction in the head of queue has the expected sequence number it is removed from the queue and is examined for conflict. Note that when the expected transaction is on top of the CommitReqPQ it means that all the preceding transactions have been evaluated by the algorithm and are in COMMITTED or COMPLETED state. Assuming that the removed transaction is T_i, the algorithm checks the conflicts with the transactions in both *CommittedTransactionList* and *CompletedTransactionList*. The conflict evaluation between T_i and the committed transactions is done in the for loop depicted in lines 9 to 14. If T_i conflicts with a committed transaction T_j, since the changes in T_j have not been applied to the key-value store, T_i may have not seen these changes and therefore, it needs to wait for T_j to apply the changes into the key-value store and restart its execution. In this case, the algorithm adds T_i to the restart list of T_j. The restart list for a transaction is the list of transactions that should be restarted after the transaction is completed and its effect is applied to the key-value store. In case of such conflict since T_i should restart after completion of T_j, the algorithm stops processing other transactions until completion of the conflicting committed transaction. All other transactions after T_i also are not processed since the expected transaction on top of the CommitReqPQ is T_i. After T_j completes, it then notifies all of its conflicting transactions to restart since now they can see the updates from T_j.

If T_i does not have any conflict with the committed transactions, the algorithm checks for conflicts with the completed transactions. This is done in the for loop depicted in lines 15 to 22 in the algorithm. However, as shown in line 16 the algorithm ignores the conflict test between T_i and the transactions that have completed before T_i started. Since we assume writes on the key-value store are immediately available for the readers, there is no concurrency between these transactions: This means that even if there is a conflict between T_i and such transactions, T_i used the updated data from these transactions. On the other hand, if T_i starts before completion of a completed transaction like T_j and T_i conflicts with T_j, then it is possible that T_i may have used out of date data. Therefore, the algorithm restarts T_i in order to make sure that T_i uses the correct data for its execution.

Finally, if T_i does not have any conflict with its predecessors, it can commit and be executed concurrently with them. The algorithm first changes the state of T_i into $COMMITTED$ and adds T_i to the list of committed transactions and updates the expected sequence number. Then, using a thread from the bottom threadpool, it applies the corresponding changes that should be made to data in the key-value store. When the thread finished updating the key-value store, the transaction is completed. At this point, the algorithm changes the state of the transaction to $COMPLETED$ and removes T_i from the CommittedTransactionList and adds it to the CompletedTransactionList. It also restarts all the transactions that have been waiting for completion of T_i. As mentioned, these transactions are stored in T_i's restart list.

5.1 Discarding completed transactions

In our concurrency control algorithm the CompletedTransactionList is the last list that stores transactions. However, by processing more and more transactions this list will grow larger. Therefore, we need to limit the size if this list and remove the completed transactions from the list if there is no need for them. The main reason to keep a completed trans-

action T_i in the CompletedTransactionList is that if another transaction T_j, starts before the completion of T_i and T_j has conflict with T_i, there is a possibility that T_j did not use the updated data resulted from T_j. Thus, in order to make sure that T_j observes the results of T_i, we need to make sure that T_j starts after completion of T_i. Based on this assumption, if there is no active transaction that has started before completion of a transaction T_i, there is no need to keep T_i and we can safely remove it from the CompletedTransactionList without jeopardizing the correctness of the algorithm.

Algorithm 2 Asynchronous removal of transactions from CompletedTransactionList.

```
1: ActiveTransactionList: List of active transactions.
2: CompletedTransactionList: List of completed transactions.
3: for all T_i ∈ CompletedTransactionList do
4:    boolean shouldBeRemoved = true;
5:    for all T_j ∈ ActiveTransactionList do
6:       if T_j.startTime < T_i.completeTime then
7:          shouldBeRemoved = false;
8:       end if
9:    end for
10:   if shouldBeRemoved then
11:      CompletedTransactionList = CompletedTransactionList - {T_i}
12:   end if
13: end for
```

We use an asynchronous algorithm to remove the completed transactions from the CompletedTransactionList. We consider a threshold for the CompletedTransactionList size and whenever the size of the list passes the threshold the transaction removal algorithm is called asynchronously. Algorithm 2 shows the process of detecting and removing completed transactions that are not required from the CompletedTransactionList. For each transaction in the CompletedTransactionList, the algorithm checks if there is any active transaction in the ActiveTransactionList. The ActiveTransactionList is the list that transactions are added when they start execution in the system. If there is at least one transaction that has started before completion of completed transaction T_i, the transaction T_i should not be removed from the CompletedTransactionList. Otherwise, the algorithm removes transaction from the CompletedTransactionList.

6. EXPERIMENTAL EVALUATION

The main goal of our experiments is to validate the advantage of using our proposed concurrency control algorithm and to analyze the effect of different tuning parameters in the performance of the algorithm. In particular we present the following results:

- The comparison of serial execution of transactions with concurrent execution based on our concurrency control.
- The effect of workload characteristics such as conflict ratio, read/write ratio and number of concurrent clients on our proposed concurrency control algorithm.
- The effect of system parameters such as degree of parallelism (number of threads) and key-value cluster size on our concurrency control algorithm.

6.1 Benchmark Description

Since we used web applications as one of the motivating applications for our proposed system, we use TPC-W benchmark [8], which is a transactional web e-commerce benchmark. The benchmark emulates an on-line book store with multiple on line browser sessions. The benchmark provides three different interaction types: browsing (5% of transactions are writes), shopping (20% of transactions are writes) and ordering (50% of transactions are writes). We modified an open source Java implementation of TPC-W benchmark to only emulate database transactions part of the benchmark [9]. The database contains eight tables: customer, address, orders, order line, credit info, item, author, and country. In our implementation we also have two auxiliary tables, shoppingcart and shoppingcartline. These tables are used to store the persistent state of the shoppingcart for each client. We used 2,000,000 items and 2880*1400 (4032000) customers, which results in a database with the size of 7.2GB.

To be able to test the specific parts of the system, we also create a synthetic workload on top of TPC-W database in such a way that we can create transaction conflicts at desired levels. In our synthetic workload each transaction has only one update statement where we update the quantity of an item in the database given the item id. We control the probability of conflict with selecting the item id value from a predefined range. The smaller the selection range, the higher the probability of selecting the same item id for different transactions and therefore, the higher the probability of conflict. Only accessing the same data item is not enough to generate a conflict for two transactions and the other necessary condition is the *concurrent* access to the item by both transactions.

6.2 Experiments Setup

We set up our experimental environment based on our scale out architecture shown in Figure 3. For relational database in the architecture we use MySQL [5] and for the key-value store we use Project Voldemort [6] which is an open source of Amazon Dynamo [12]. We implemented Replication Middleware, Query Translator and Transaction Manager components all in Java. We used Apache ActiveMQ for the messaging middleware in our replication middleware component. The publisher agent reads the transaction log from MySQL and constructs a replication message that is delivered to the replication agent through the ActiveMQ framework. We run the experiments on a set of up to 18 machines. We assign one machine to MySQL database where the publisher agent from the replication middleware also resides on it. Another machine is used as ActiveMQ broker. The Query Translator (QT) and Transaction Manager (TM) along with Subscriber Agent all reside in one machine. The rest of the machines in the experiment are used for key-value store. All the machines except the one that runs the Query Translator (QT) and Transaction Manager (TM) along with Subscriber Agent are Intel Xeon machines with 2.4GHz CPU and 16GB memory running CentOS 5.4. The machine that we used for Query Translator (QT) and Transaction Manager (TM) along with Subscriber Agent has an Intel Core(TM)2 Duo CPU with 3.16GHz speed and 4GB of memory running Ubuntu Linux kernel 2.6.32 and Sun's JDK 1.6.3. In all of the experiments, except the one for evaluating the effect of key-value cluster size, we use five

machines for Voldemort key-value cluster.

The metrics that we used in our evaluations are as follows:

Throughput: The throughput is the number of transactions that are executed in one time unit (second).

Execution time: The total execution time is the time it takes to execute all of the given transactions.

Number of Conflicts: As mentioned two transactions conflict when they access the same item concurrently during their execution. For a set of transactions, the number of conflicts is the total number of times that any two transactions conflict during the execution of the transaction set.

We run the workload on MySQL and then use the transaction log to construct the set of transactions with the predefined order that should be applied to the replica in the key-value store. Unless we specify explicitly, in all of the experiments we used 20 threads in top threadpool and 20 threads for bottom threadpool (as shown in Figure 8). The default key-value cluster size for the experiments also is five except for the last experiment.

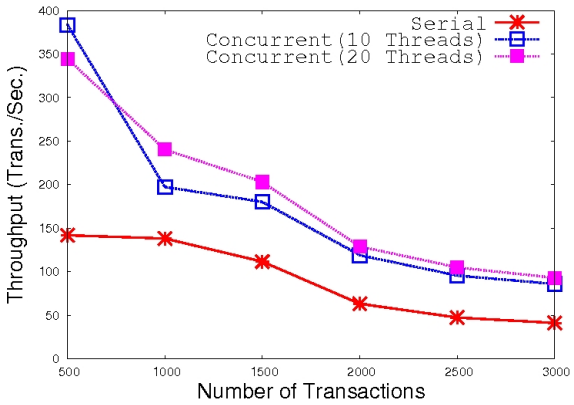

Figure 10: Throughput for Serial and Concurrent execution of transactions.

6.3 Experimental Results

Concurrent vs. Serial execution: The first set of experiments that we present is the comparison of serial execution of transactions with concurrent execution that uses the proposed algorithm. Most of the existing replication approaches use single threaded serial execution of updates in the replica so we use the serial execution as the base line in our experiment [7]. We measured throughput, total execution time and number of conflicts for serial execution and concurrent execution with 10 and 20 threads in our concurrency control algorithm. Figure 10 depicts the throughput for different number of transactions in the replication message that are applied to the key-value store. As it is seen our proposed concurrency control algorithm significantly increases throughput in all cases. Similarly, the total transaction execution time that is plotted in Figure 11 shows that the proposed concurrency control algorithm is at least twice as fast as executing the transactions in serial execution. This will obviously reduce the replica lag behind the original data and consequently the staleness of data in the replica.

As it is seen in both graphs, the benefit of using our concurrency control algorithm is more significant when there are fewer transactions in the replication message. In fact as it is seen, the throughput decrease and execution time increase

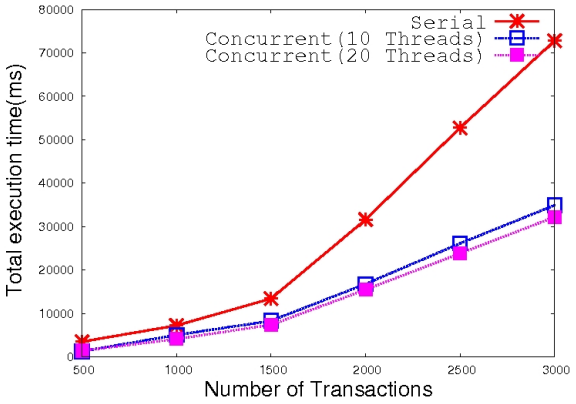

Figure 11: Total execution time for Serial and Concurrent execution of transactions.

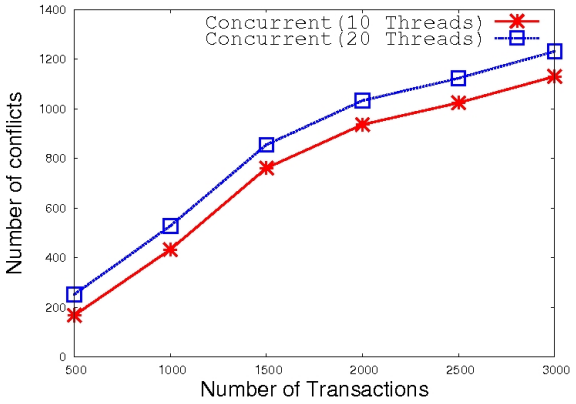

Figure 12: Conflict count for concurrent execution of transactions.

are not linear with respect to the number of transactions that are executed and by increasing the number of transactions in the replication message the throughput reduces faster and execution time increases faster too. This can be described by considering the number of conflicts in the execution process. Figure 12 depicts the number of conflicts occurred in the execution of different number of transactions in the replication message for 10 and 20 threads in the concurrency controller. The number of conflicts increases by increasing the number of transactions. This is expected since the more transactions results in higher probability of accessing same item by multiple transactions concurrently. As described in Section 5, when two transactions T_i and T_j conflict, the concurrency controller aborts the one which is behind in the execution-defined order. Assuming $i < j$, Transaction T_j should be aborted and restarted when the effects of T_i are applied to the key-value store. Note that, this would affect commit time for other transactions by delaying their commit time too. Therefore, as it is seen a conflict will not only slow down the conflicting transactions, but also the ones that are behind them too. Thus, a high number of conflicts reduces throughput more significantly.

Workload Read/Write ratio: We now represent the effect of read/write ratio in the workload on the performance of our concurrency control algorithm. Here read/write ratio is defined as the percentage of write transactions TPC-W

interactions. We performed our experiments for all three interaction types in TPC-W and present the results in Table 1. The three interaction types are Browsing where 5% of transactions are write transactions, Shopping where 20% of transactions are write transactions, and Ordering where 50% of transactions are write transactions. The algorithm has better throughput and execution time for browsing and shopping workloads compared to the ordering workload. As we discussed above, the larger number of write transactions increases the probability of conflicts that results in restarting transactions. This indeed increases the number of transactions that are being executed and therefore reduces throughput.

Effect of conflicts: Two transactions conflict if they both access the same data item *concurrently* and at least one of them updates the data item. To evaluate the effect of conflicts in our concurrency control algorithm we use our synthetic workload on TPC-W benchmark where we can control the number of conflicting transactions.

Figure 13 depicts the effect of conflicts on the algorithm throughput. In this figure we plot the improvement percentage over serial execution for 4500 transactions with different number of conflicts. The percentage of improvement in throughput is computed by dividing the difference between the measured throughput for the concurrency control algorithm and the serial execution to the throughput of serial execution and multiplying it by 100. When there is no conflict in the workload we have a steady value for the throughput improvement and the concurrent execution performs twice better than the serial execution (approximately 100% improvement). On the other hand, when we introduce conflicts in the workload the throughput declines as expected. This is caused by restarting the conflicting transactions that slow down the execution of other transactions.

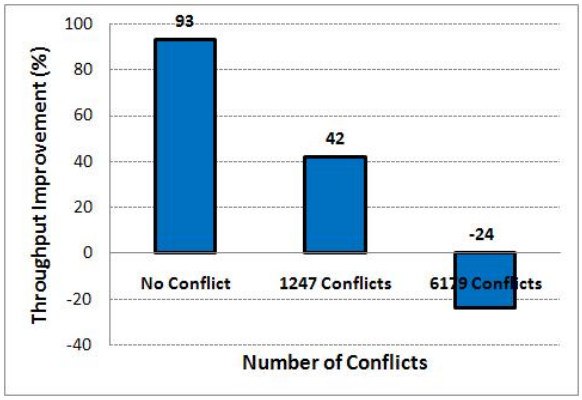

Figure 13: Impact of conflicts in workload on throughput.

The increase of conflicts can have more significant impact on the concurrency control algorithm as it is depicted for the case with 6179 conflicts (transaction restarts) in the graph. In this case, the throughput of concurrent execution is even less than serial execution which does not justify use of concurrent execution for workloads with high conflict ratio. Indeed, we evaluated this in Figure 14 where we show the throughput improvement for different number of conflicts. Similarly, we conclude that the concurrency control algorithm for concurrent execution of transactions is effective only when the number of conflicts in the workload is not too high. In case where the conflict ratio is too high it is better to use the serial execution instead of concurrent execution.

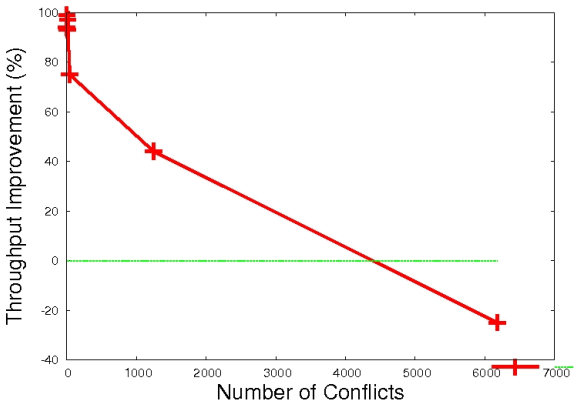

Figure 14: When to use concurrency.

Impact of number of threads in concurrency controller: One of the main tuning parameters in our proposed concurrency control algorithm is the number of threads that are used by the algorithm in initial execution of transactions to construct the set of PUT / GET / DELETE operations and the number of threads to apply non conflicting transactions to the key-value store. The more number of available threads for the transaction conversion to PUT / GET / DELETE operation will result in greater number of transactions requesting to be evaluated by concurrency controller. The larger number of threads for applying non conflicting transactions also should speed up concurrency controller by preventing it from waiting for a non conflicting transaction to be applied to the key-value store. We now present our experimental results on the impact of the number of threads on throughput and the number of conflicts.

Figure 15 plots the throughput of the serial execution along with the concurrency control algorithm for different number of transactions where we use 2, 5, 10 and 15 threads for each threadpool. The overall trend in this graph indicates that by increasing the number of threads we gain higher throughput in our concurrency control algorithm, however, this gain does not increase significantly by adding

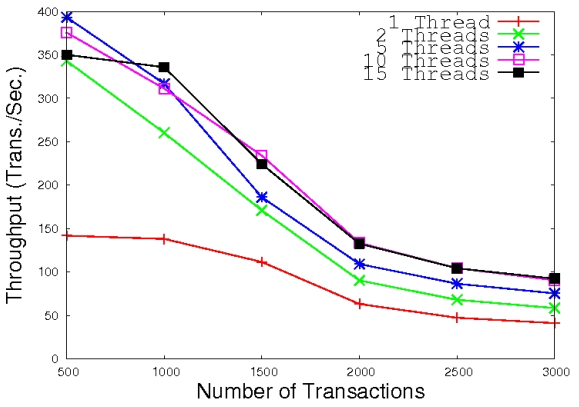

Figure 15: Thread count effect on throughput.

	Browsing (5% write)	Shopping (20% write)	Ordering (50% write)
Write Transactions	200	800	2000
Throughput (Tx/S)	247	256	129
Execution Time (ms)	808	3117	15503
Conflict Count	99	402	1033

Table 1: Results for different TPC-W workloads (4000 Transactions)

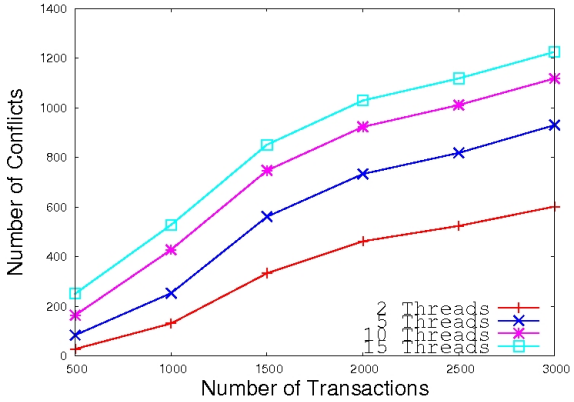

Figure 16: Thread count effect on conflicts.

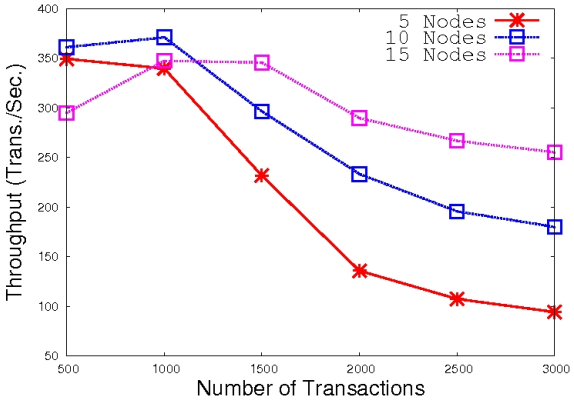

Figure 17: Key-value cluster size.

more threads to the system. As it is depicted the throughput gain for 10 and 15 threads is almost the same. The main reason is that the serial evaluation of transactions for conflicts in concurrency controller. Although we use concurrency in conversion of transactions to PUT / GET / DELETE operations and also in applying the transactions to the store, all the transactions should be evaluated according to their execution-defined order. Therefore, increasing the number of threads can improve throughput initially but at some point the serial evaluation of conflicts in concurrency controller will dictate the execution speed and therefore further increase of the number of threads will have negligible effect on the throughput.

The other factor in reducing the effect of more threads is the increased number of conflicts because of more threads in the system. To have two conflicting transactions not only they should access the same data item where at least one of them writes the data item, but also these accesses should be concurrent. Therefore, increased number of threads elevates the probability of conflict among transactions, which negatively affects the throughput. Figure 16 validates the impact of more threads on the number of conflicts encountered by the algorithm. As shown, by increasing the number of threads in the system, we will have more number of conflicts for the same number of transactions.

Impact of key-value cluster size: In order to analyze the impact of the key-value cluster size on our concurrency control algorithm we used Voldemort key-value store with three different setting, 5, 10 and 15 nodes. Figure 17 depicts the throughput for different number of transactions and different key-value cluster size. The overall trend in the figure is that the throughput is higher when there are more key-value nodes in the system. The larger number of nodes in key-value cluster results in smaller portion of load on each key-value node which in turn speeds up execution of PUT / GET / DELETE operations on each node. Therefore, by increasing the number of nodes in key-value system we can increase the throughput of our concurrency control algorithm.

7. CONCLUSIONS AND FUTURE WORK

We presented a scale out architecture based on fully replication of relational database on key-value store system where the key-value store is used for read-only transactions. Our proposed architecture ships transaction logs from relational database to the key-value store and applies them in such a way that the state of key-value store is exactly the same as the relational database. To reduce the replica lag in the key-value store side we proposed a novel concurrency control algorithm that guarantees a predefined serialization order (the one same as the order in transaction log). We empirically showed that the proposed algorithm significantly improves throughput compared to serial execution of transactions.

An interesting optimization to our concurrency control algorithm is to exploit transaction classes to speed up conflict detection and increase parallelism. By classifying transactions into transaction classes our algorithm would only evaluate conflicts for potentially conflicting transactions. This would eliminate many unnecessary operations and speeds up our concurrency controller.

8. ACKNOWLEDGMENTS

We thank Michael Carey and Hector Garcia-Molina for the insightful discussions and the contributions.

9. REFERENCES

[1] HBase. www.hbase.apache.org.
[2] Membase. www.membase.org.
[3] Memcached. www.memcached.org.
[4] Memcachedb. www.memcachedb.org.
[5] MySQL. www.mysql.com.
[6] Project Voldemort. www.project-voldemort.com.
[7] Replication in MySQL.

http://dev.mysql.com/doc/refman/5.6/en/replication-implementation-details.html.
[8] TPC-W. www.tpc.org/tpcw.
[9] TPC-W Java implementation. http://www.ece.wisc.edu/pharm/tpcw.shtml.
[10] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13:185–221, 1981.
[11] P. A. Bernstein, D. W. Shipman, and J. Rothnie. Concurrency control in a system for distributed databases (SDD-11). *ACM Trans. on Database Systems*, 1980.
[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.
[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25:173–182, June 1996.
[14] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000.
[15] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.
[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6:213–226, June 1981.
[17] K. Manassiev and C. Amza. Scaling and continuous availability in database server clusters through multiversion replication. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 666–676, Washington, DC, USA, 2007. IEEE Computer Society.
[18] C. A. Polyzois and H. García-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Trans. Database Syst.*, 19:423–449, September 1994.
[19] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.
[20] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacıgümüş. Partiqle: an elastic sql engine over key-value stores. In *SIGMOD Conference*, 2012.
[21] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

APPENDIX

A. REPLICATION MIDDLEWARE

The data in the key-value store is the replication of the data in the original database. To maintain the replicated data in the key-value store synchronized with the original data in the relational database we use our replication middeware. Details of the replication middleware are depicted in Figure 18. Our replication middeware is implemented on top of a publish/subscribe system. It includes a publisher agent that resides in the database system, a subscriber agent that resides in the key-value store side and a messaging middleware that provides communication framework between publishers and subscribers.The publisher agent periodically reads the transaction log from the database and packages the transactions in a message. The transaction log only includes write statements and does not contain the read statements in the transactions. The frequency of reading the log is a tunable parameter and can be adjusted based on different factors such as staleness limit for read only transactions in the key-value store.

The subscriber agent resides in the key-value store side. This agent receives the messages containing transactions from the publisher agent and applies them to the key-value store through the query translator and transaction manager. Since these transactions have already been executed in the database, the serialization order for the transactions is determined. The easiest way to guarantee such order is to execute these transactions serially over the key-value store. In this case, the subscriber agent issues the transactions to the key-value store through the query translator component using a single thread and each transaction starts after commit of its predecessor. However, as we describe in Section 5 our proposed concurrency control algorithm can guarantee such predefined serialization order while executing transactions concurrently. In this case, the subscriber agent uses a set of threads in a threadpool to concurrently issue the transactions to the key-value store through the query translator and transaction manager components.

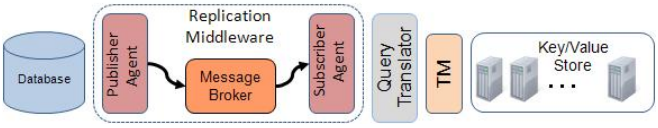

Figure 18: Replication middleware from RDBMS to Key-Value store.

We use a publish/subscribe system to route the transaction log from the publisher agent to the subscriber agent. This functionality is provided by the message broker component as shown in the Figure 18. We consider a single node as message broker, however, a federated set of message brokers can also be used to provide better scalability.

One of the main advantages of using a publish/subscribe system as communication framework for the replication middleware is the decoupling of the publisher agent and the subscriber agent. This eliminates the need for the publisher agent to know the subscriber agent and we can add more subscriber agents to provide multiple replicas without putting any extra load on the publisher agent. All the complexity and load of delivering transaction logs to the corresponding subscriber agents is handled by the message broker and the underlying publish/subscribe system.