Strong Consistency in Cache Augmented SQL Systems

Shahram Ghandeharizadeh, Jason Yap, Hieu Nguyen
Computer Science Department
University of Southern California
Los Angeles, California 90089
shahram,jyap,hieun@usc.edu

ABSTRACT

Cache augmented SQL, CASQL, systems enhance the performance of simple operations that read and write a small amount of data from big data. They do so by looking up the results of computations that query the database in a key-value store (KVS) instead of processing them using a relational database management system (RDBMS). These systems incur undesirable race conditions that cause the KVS to produce stale data. This paper presents the IQ framework that provides strong consistency with no modification to the RDBMS. It consists of two non-blocking leases, Inhibit (I) and Quarantine (Q). Ratings obtained from a social networking benchmark named BG show the proposed framework has minimal impact on system performance while providing strong consistency guarantees.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Distributed databases, Transaction processing*

General Terms

Design

Keywords

Concurrency, Consistency, Transaction

1. INTRODUCTION

Organizations extend a relational database management system (RDBMS) with a key-value store (KVS) to enhance system performance for workloads consisting of simple operations that exhibit a high read to write ratio [27, 29, 17, 20, 15, 14], e.g., interactive social networking actions [8]. The key insight is that query result lookup using the KVS is faster and more efficient than processing the same query using the RDBMS. A challenge of the resulting Cache Augmented SQL (CASQL) system is how to maintain the cached query results consistent in the presence of updates to the RDBMS.

One approach is for the developer to provide software to either invalidate, refresh, or incrementally update the key-value pairs, see Figure 1. To describe these techniques, we define a *session* as a sequence consisting of an RDBMS operation followed by one or more KVS operations (or with the KVS operations being performed first). With invalidate, the application computes the key impacted by the SQL Data Manipulation Language (DML) commands[1] and deletes them from the KVS. With refresh, the application reads the impacted key-value pairs, updates them, and writes them back to the KVS. With incremental update, the application transmits the changes for a key-value pair to the KVS and the KVS applies them to update the key-value pair. One may implement these techniques using triggers in the RDBMS, reducing a session to an RDBMS operation that performs the KVS operation as a part of its execution.

With multiple concurrent sessions executing simultaneously, all three techniques result in a variety of race conditions that cause the KVS to produce stale data. Table 1 shows the percentage of read requests that observe unpredictable data (stale values) reported by a social networking benchmark named BG [6]. (See Section 6 for details of BG and the imposed workload.) With one session, there is no stale data because the developer provided a correct implementation of a session. As we increase the number of concurrent sessions that results in a higher system load, different techniques result in various undesirable race conditions that insert stale data in the KVS. BG quantifies the percentage of read operations that observe this unpredictable (stale) data as reported in Table 1.

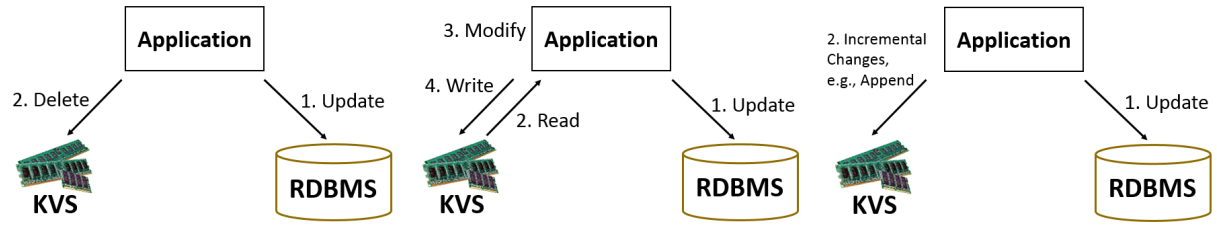
On may address this limitation using two different approaches. With the first, the software developer identifies the undesirable race conditions and implements a session in a manner that prevents them. The second approach provides a general purpose framework that prevents the undesirable race conditions for all possible sessions and application use cases. This approach enhances the productivity of a software developer by enabling them to focus on the application requirements instead of identifying and solving race conditions with each possible application use case. This second approach is the focus of this paper.

The primary contribution of this paper is the IQ framework and its simple programming model that employs Inhibit (I) and Quarantine (Q) leases to prevent race conditions that may insert stale data in the KVS. The IQ frame-

[1]Insert, delete, and update commands.

Middleware '14, December 08 - 12 2014, Bordeaux, France

http://dx.doi.org/10.1145/2663165.2663318 ...$15.00.



(a) Invalidate (b) Refresh (c) Incremental Update (δ)

Figure 1: Three techniques to maintain the key-value pairs of the KVS consistent with updates to the tabular in the RDBMS.

work provides *strong consistency* and is desirable as it makes systems easier for a programmer to reason about [24]. In addition to detailing the semantics of these leases with both invalidate, refresh and incremental update, we describe an implementation using a variant of memcached. Experimental results show the IQ framework reduces the amount of stale data to zero with minimal impact on system performance. A possible limitation of the framework is the *potential* for starvation under a heavy system load when leases are employed using a certain programming paradigm, see Section 6. We show starvation can be avoided by performing the KVS operations as a part of the RDBMS transaction that constitutes a session.

In [27], Facebook describes the concept of a lease to avoid undesirable race conditions when invalidate is implemented in an application. The I lease of the IQ framework is identical to this lease. The IQ framework is different as it introduces a Q lease to provide strong consistency with invalidate, refresh and incremental update independent of its implementation in either the application or the RDBMS trigger. Both the I and Q leases are different than the Shard (S) and eXclusive (X) locks [26, 19, 32]. To illustrate, with a key-value pair, multiple I leases are not allowed on this key while a lock manager grants multiple S locks on the same key-value pair. These and other related work are detailed in Section 7.

The rest of this paper is organized as follows. Section 2 introduces terms and their definitions as used in this paper. Sections 3 and 4 detail the alternative scenarios that cause the KVS to produce stale data. Each section presents how the I/Q leases are employed to prevent the identified scenarios. While Section 5 presents an implementation of the I/Q leases using Twemcache, Section 6 evaluates this implementation using a social networking benchmark named BG. We review related work in Section 7. Brief conclusions and future research directions are presented in Section 8.

System Load	Invalidate	Refresh	Incremental Update
1 session	0%	0%	0%
Low 10 concurrent sessions	0.5%	0%	0.01%
Moderate 100 concurrent sessions	1.1%	1.4%	0.2%
High 200 concurrent sessions	1.3%	1.8%	2.9%

Table 1: Percentage of unpredictable data with Twemcache. These percentages are reduced to zero with the IQ framework.

2. OVERVIEW

Our proposed IQ framework targets CASQL systems realized using an off-the-shelf RDBMS and a key-value store that supports simple operations such as get, set, compare-and-swap, delete, append, prepend (and other incremental change operators). It requires no change to the RDBMS software. Instead, it extends the KVS with new commands that implement two leases, I and Q. In addition, it includes a simple programming model that implements the concept of sessions. A session acquires and releases leases in a manner similar to the two phase locking protocol [26, 19, 32]. These leases must be obtained either prior to or as a part of the RDBMS transaction. The framework is non-blocking and deadlock free. It may delete key-value pairs and abort and re-start sessions to realize strong consistency.

The IQ framework implements the ACID (atomicity, consistency, isolation, durability) properties of transactions at the granularity of sessions, imposing the serial schedule observed by the RDBMS onto the KVS. It guarantees atomicity by ensuring both the RDBMS and the KVS operations of a session are applied to data in both the RDBMS and the KVS. We implement this using the insight that the KVS contains a subset of the data in the RDBMS and it is acceptable to delete a key-value pair to provide this property. Consistency means a session transitions the data in both the RDBMS and the KVS from one valid state to another. If the RDBMS aborts a session's RDBMS transaction then the session's KVS operations should not be applied. Isolation requires each session to appear to have executed by itself even though it executed concurrently with many other sessions. Durability is provided by the RDBMS with an in-memory KVS that has a key-value representation of a subset of the relational database.

The IQ framework ensures a session that changes data observes its own update. For example, once a session updates a key-value pair using either refresh or incremental updates, should the session read the value of the key again, it observes the latest value produced by itself. The session's change is not visible to other sessions until the session commits. A session either aborts or commits explicitly. Moreover, the framework provides serial schedules that provide *equilibrium* defined as equal effect on data in both the RDBMS and the KVS.

This section provides an abstraction of the different operations supported by the KVS and the RDBMS. We use these to formally define a session. Subsequently, we present the I and Q leases used to implement the ACID properties.

The focus of this study is on simple read (R), write (W), delete, incremental change (δ), and read-modify-write (R-

Term/notation	Definition
KVS	A key value store such as memcached.
RDBMS	A relational database management system such as MySQL.
RDB	A relational database.
δ	Incremental update operation such as append, prepend, increment, decrement.
Operation	Read (R), Write (W), Delete, Read-Modify-Write (R-M-W), δ using either the KVS or the RDBMS, see Table 3.
Transaction	A logical sequence of one or more RDBMS operations executed atomically.
BG Action	An interactive social networking activity such as invite friend, see Table 5.
Session	A sequence of operations consisting of at most one RDBMS transaction and one or more KVS operations.
Command	An atomic implementation of an operation using either a KVS or an RDBMS, see last two columns of Table 3.

Table 2: List of terms/notations and their definitions.

M-W) operations that manipulate a small amount of data. Table 3 shows the commands of memcached and SQL equivalent to these abstractions. The R operation is equivalent to the get command of memcached and the SELECT statement of SQL. The W operation is equivalent to the set command of the KVS and either the insert or update command of SQL. It may produce a new data item or overwrite the value of an existing data item. The Delete operation removes a data item and has a trivial mapping for both memcached and SQL, see Table 3. With an incremental change, δ, the application propagates a change to an existing data item. With a KVS such as memcached, it might be an operation such as append and prepend. With SQL, it is implemented using the update command.

The R-M-W operation reads a data item, updates it, and writes it back. At a conceptual level, it resembles the SQL update command. However, it is not the same physically because the update command pushes the modification to be processed by the RDBMS server, resembling an incremental update. With a KVS, say memcached, the application implements the R-M-W operation by issuing a get command to retrieve a key-value pair, update the value in its memory, and write it back using the set command, see Figure 1.b. Instead of the set command, one may use compare-and-swap (cas) to implement R-M-W atomically as follows. Simply maintain the old value (v_{old}) retrieved by the get (R) command, apply the M to compute a new value (v_{new}), and implement the W operation using cas with v_{old} and v_{new}. Should the cas fail because v_{old} does not match the current value of the referenced key (changed by another concurrent W/R-M-W operation), the application may re-try its R-M-W operation starting with the R.

The use of cas does not provide strong consistency. This is illustrated in Figure 2 with two write sessions, S1 and S2. The RDBMS operations of each session are denoted with a disk while its KVS operations are denoted with DRAM sticks. Both S1 and S2 consist of 5 operations shown on top and below the time line, respectively. Operations of S2 occur after S1's RDBMS operations and prior to S1's KVS operations. Assume S1 increments the value of a data item by 50 while S2 multiplies the value of the same data item by 10. If the original value of the data item is 100 then the interleaved schedule of Figure 2 produces the final value of 1500 and 1050 for the data item in the RDBMS and the KVS, respectively. This is undesirable because the RDBMS and the KVS should reflect the same value for the data item in order for the next session(s) that reads the value of the data item to be serialized. Hence, the cas fails to provide strong consistency. As detailed in Section 4, The

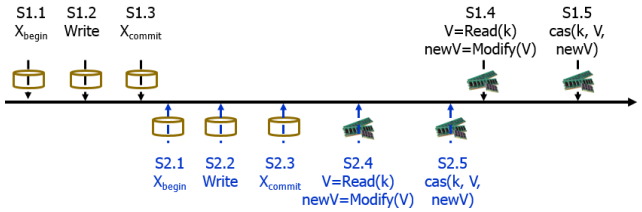


Figure 2: Use of compare-and-swap (cas) does not provide strong consistency.

IQ framework prevents schedules of Figure 2.

Table 4 shows the application of invalidate, refresh, and incremental update with the different KVS operations. Invalidate does not use the R-M-W operation or a δ operation (such as append) as it always deletes a key-value pair that is impacted by a change to the RDBMS. With refresh, the application may fetch a key-value pair from the KVS, modify it in its memory, and write it back to the KVS, see Figure 1. Hence, it does not apply to a δ operation such as append that pushes the modification (change) to the KVS. Finally, a δ operation is different than both invalidate and refresh as it does not delete or R-M-W a key in the KVS.

Sessions are categorized into read and write sessions. A read session retrieves one key-value pair from the KVS. If the KVS does not have a value for the referenced key (a KVS miss), then the session proceeds to query the RDBMS to compute a key-value pair that it inserts into the KVS for future reference. A write session performs an RDBMS write operation that may impact one or more key-value pairs. With invalidate, the system deletes the impacted key-value pairs from the KVS. With refresh, the application computes a new value for the impacted keys and inserts them in the KVS. An example of a read session is to retrieve the profile of a member of a social networking site. An example of a write session is for a member of a social networking site to extend an invitation to another member for friendship.

To provide strong consistency, we introduce two leases named Inhibit (I) and Quarantine (Q). The KVS grants an I lease to a session that observes a KVS miss, enabling it to query the RDBMS and populate the KVS with a key-value pair. The KVS grants at most one I lease on a key. Hence, at most one session may query the RDBMS to populate the KVS. All other concurrent read sessions referencing the same key must back off and try again. They observe either the value produced by the current lease holder or one of them will be granted an I lease to query the RDBMS and populate the KVS.

Operation	memcached command	SQL command
Read	get	SELECT ... FROM ... WHERE ...
Write	set	INSERT INTO tblname UPDATE tblname SET ... WHERE ...
Delete	delete	DELETE FROM tblname WHERE ...
R-M-W	get set/cas	
δ	append prepend	UPDATE tblname SET ... WHERE ...

Table 3: Alternative actions and their implementation with memcached and a SQL system.

Operation	Invalidate	Refresh	Incremental Update
Read	✓	✓	✓
Write	✓	✓	✓
Delete	✓	✓	✓
R-M-W	✗	✓	✗
δ	✗	✗	✓

Table 4: Presence of KVS operations with invalidate, refresh, and incremental update.

A session that intends to either write, delete, δ, or R-M-W one or more keys must obtain a Q lease on each key explicitly. When a Q lease request for a key encounters an existing I lease held by a read session, S_R, the IQ framework invalidates the I lease of S_R to grant the Q lease. Subsequently, when S_R writes its computed value to the KVS, the KVS ignores its write operation because its I lease is no longer valid.

Refresh and δ handle collisions of two Q lease requests in a different manner when compared with the I lease. See Figure 5 and discussions of Sections 3 and 4.

A lease for a key has a fixed life time and is granted to one KVS connection (thread) at a time. The finite life time enables the KVS to release the lease and continue processing operations in the presence of node failures hosting the application. This is particularly true with refresh and δ due to how they use the Q leases: If the KVS holds Q leases indefinitely (similar to locks) then both node failures and intermittent network connectivity may degrade system performance severely. With a finite life time for leases, the KVS recovers from node failures hosting an application with granted leases.

The next two sections detail how invalidate and refresh employ the I/Q leases to provide strong consistency.

3. INVALIDATE

This section describes how invalidate incurs an undesirable race condition with snapshot isolation to insert stale data in the KVS. Section 3.2 presents how the I/Q framework prevents this race condition. Due to lack of space, we refer the interested reader to [18] for a formal proof of correctness.

3.1 Undesirable Race Condition

To improve performance, many RDBMSs offer snapshot isolation, a multi-version concurrency control [10] technique that enhances simultaneous execution of transactions. Snapshot isolation guarantees (1) all reads made in a transaction observe a consistent snapshot of the RDB and (2) the transaction commits only if none of its updates conflict with a concurrent update made since that snapshot. Snapshot isolation may result in an undesirable race condition [28] between a read and a write session, inserting stale data in the KVS. An example is shown in Figure 3 with a write session S1 using triggers to delete impacted key-value pairs as a part of the RDBMS transaction that is invalidating the KVS. S2 is a read session that performs its KVS read after S1's delete, observes a KVS miss, queries the RDBMS to compute a key-value pair, and insert this key-value pair in the KVS. Snapshot isolation enables S2's RDBMS query to reference a version of the RDB prior to the execution of S1's RDBMS transaction. S2 inserts this stale value in the KVS in Step 2.5 with S1 committing in Step 1.4. A subsequent KVS read for this key-value pair observes a stale value, inconsistent with the RDB.

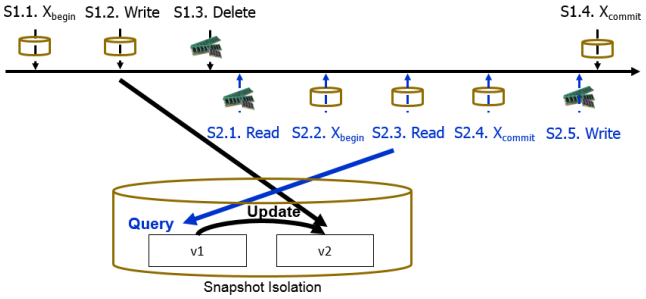


Figure 3: S2 inserts a stale value in the KVS.

Changing S1 to perform its delete operation as either a part of or after its RDBMS transaction commit does not resolve the undesirable race condition of Figure 3. To illustrate, consider moving Step 1.3 of Figure 3 to occur either as a part of or after Step 1.4. It is still possible for an adversary to move Step 2.5 to occur after this step to insert a stale value in the KVS.

3.2 Solution

The IQ framework consists of two leases, I and Q, that prevent the race condition of Figure 3. Sessions employ these leases as follows. When a read session such as S2 observes a KVS miss for a key k_j, as long as there is no pending I or Q lease on k_j, the KVS server grants an I lease on k_j to S2. This enables S2 to query the RDBMS to compute a value v_j and insert its k_j-v_j in the KVS as long as its I lease has not been invalidated by a Q lease, see below.

With an existing I or Q lease on k_j, the KVS server does not grant an I lease to S2. Instead, it informs S2 to back off and try its read request for k_j again, see Figure 5.a. The duration of back off may increase exponentially with S2's repeated KVS lookups. With an existing I lease, this back off ensures at most one concurrent session queries the RDBMS for the same key k_j with many sessions consuming the value v_j inserted in the KVS by that one session [27]. With an existing Q lease, S2's back off is appropriate as it is referencing a key that is in the process of being changed in the RDBMS. This prevents S2 from querying the RDBMS simultaneously. Once the pending Q lease is released by its write sessions, should there be no other pending lease on k_j

and S2 looks up k_j to observe a miss, the KVS server grants an I lease to S2 to query the RDBMS.

A write session requests a Q lease on a key k_j that it intends to invalidate. An example is session S1 of Figure 3. The KVS grants the Q lease always, see Figure 5.a. Should there be an existing I lease on k_j then the Q lease invalidates this I lease, preventing the read session that owns this I lease from inserting its key-value pair in the KVS. With an existing Q lease on k_j, the Q lease is granted as multiple KVS delete operations are idempotent. There is no undesirable race condition incurred with multiple write sessions racing to delete the same key twice or more.

In addition to the I and Q leases, a write session must explicitly commit after its successful execution of RDBMS and KVS operation. This commit releases the Q leases obtained by the session. To illustrate, consider the two sessions shown in Figure 4. S1 is a write session and is modified in two ways. First, Step 1.3 acquires a Q lease that succeeds always. This lease is granted even if the referenced key is not KVS resident. Second, a new step, Step 1.5, is added to commit this write session which causes the KVS to delete the invalidated keys. With these changes and the compatibility matrix of Figure 5.a, Step 2.1 of S2 that observes a KVS miss must obtain an I lease on its referenced key. This key was quarantined by Step 1.3 of S1 and the KVS notifies S2 to back off and try again, pushing this step to succeed once S1 commits, i.e., deletes its referenced key and releases its Q lease. This prevents S2 from computing and inserting a stale value in the KVS. (S2 does not have an explicit commit as it is a read session.)

3.3 An Optimization

Assuming[2] a read session looks up the value of only one key, a possible optimization is to defer deletion of the key-value pairs performed by a write session to when it commits (and release its Q leases after deleting the corresponding keys). This enables the read sessions that reference the impacted key-value pair to observe a KVS hit consuming a value that is in the process of being invalidated (and updated in the RDBMS). This is acceptable because the read sessions can be serialized to have occurred prior to the mid-flight write session. Moreover, the overhead of implementing this optimization is minimal as an I lease is requested by a read session once there is a KVS miss.

With this optimization, Step 1.3 of S1 in Figure 3 obtains a Q lease without deleting its referenced key. S1 deletes its key in Step 1.5 after Step 1.4 once it commits, see Figure 4. Assuming the referenced key is KVS resident, Step 2.1 of S2 observes a KVS hit, eliminating its remaining steps 2.2 to 2.5. This eliminates the undesirable race condition. Moreover, it results in a valid serial schedule with S2 occurring prior to S1 which is mid-flight and not committed.

With this optimization, to enable a session to observe its own update, an implementation must force S1 to observe a KVS miss when it references its own key, see Section 5 for an implementation. This causes S1 to query the RDBMS and observe its own update that invalidated the key. One may conceptualize this optimization as a versioning technique that maintains the latest version of the key-value pair that is being invalidated for use by all sessions except the one that is currently deleting it. Once this session commits, this version is removed from the KVS.

[2]To the best of our knowledge, this assumption holds true in all cases.

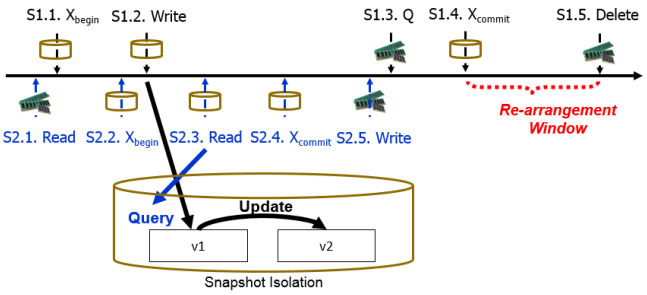


Figure 4: Re-arrangement window.

Another possible extension is an abort command for those write sessions that encounter exceptions (perhaps due to some constraint violation). This command releases the Q leases and leaves the current version of the key-value pairs in the KVS. Without an abort command, a write session that encounters an exception terminates without releasing its Q leases. All Q leases obtained by the session time out and the KVS deletes these key-value pair.

The proposed optimization results in a higher probability of concurrent read sessions being re-arranged to have occurred before a write session in a serial schedule. This is due to a longer re-arrangement window that is non-existent without the optimizations of this section. This is shown in Figure 4 where S2 races with S1 to insert a value in the KVS. Once S1 commits its RDBMS transaction, the version of the value for the impacted key in the KVS is rendered obsolete. All the KVS hits for this key observe an older version of its value, requiring them to be re-organized to have occurred prior to S1. Without the optimization, the re-arrangement window would have shrunk down to zero: The window of time between S2.5 and S1.3 cannot be considered as a re-arrangement window because the faith of S1 is not clear at time S1.3 as S1 may abort.

The implementation of Section 5 contains the optimizations detailed in this section.

4. REFRESH AND INCREMENTAL UPDATE

In addition to the race conditions of Section 3.1, the refresh and incremental update techniques suffers from undesirable race conditions attributed to R-M-W and δ operations. Section 4.1 presents these race conditions. Subsequently, Section 4.2 modifies the semantics of the I/Q leases and how they prevent these race conditions. Due to lack of space, we refer the interested reader to [18] for a formal proof of correctness.

4.1 Problem definition

Section 2 provided an example to show the use of compare-and-swap (cas) by itself is insufficient to provide strong consistency, see Figure 2 and its discussion. In Figure 2, it is possible to perform the KVS write operations either prior to or as a part of the RDBMS transaction. However, if the RDBMS transaction aborts, the developer must provide additional software to restore the modified key-value pairs to their original values. Otherwise, during the time that this key-value pair exists in the KVS, the system may produce dirty reads. This is shown in Figure 6 with the read Ses-

Existing Lease / Requesting Lease	I	Q
I	KVS miss, Back off	Not compatible, Back off
Q	Grant Q and void I	Grant Q

(a) Invalidate

Existing Lease / Requesting Lease	I	Q
I	KVS miss, Back off	Not compatible, Back off
Q	Grant Q and void I	Reject and Abort requester

(b) Refresh

Figure 5: Compatibility matrices of I/Q leases.

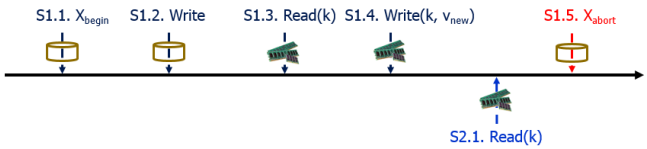


Figure 6: Dirty read with refresh.

sion S2 consuming the key-value pair produced by the write Session 1 that aborts in its Step 1.5. Both Figures 2 and 6 highlight the importance of producing the same serial schedule with both the RDBMS and the KVS in the presence of R-M-W and δ operations.

The race condition attributed to snapshot isolation is shown with δ operations in Figure 7, with the read session S2 overwriting the value of the key produced by the write session S1. Switching the KVS operation of S1 to occur after the RDBMS transaction may result in a different race condition as shown in Figure 8. In this figure, S1 appends it change to a value that is produced by the read session S1 observing S2's modifications to the RDBMS, resulting in S2's append to be reflected twice in the KVS.

It is difficult (if not impossible) to assume the serial schedule imposed by the RDBMS on the concurrent read and write sessions will be realized by the KVS. This is because the concurrent RDBMS transactions may reference different rows of a table and then update the same key-value pair in the KVS. In this case, the RDBMS will not detect a conflict as it is unaware of the key-value pairs referenced by the sessions, making it difficult to produce a serial schedule.

4.2 Solution for refresh and δ

This section presents the solution for refresh first. Subsequently, Section 4.2.1 describes its extensions to incremental update. Section 4.2.2 describes optimizations to these solutions to enhance concurrency between read and write sessions.

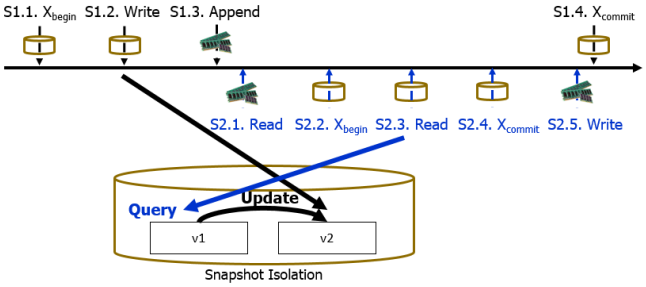


Figure 7: S2 inserts a stale value.

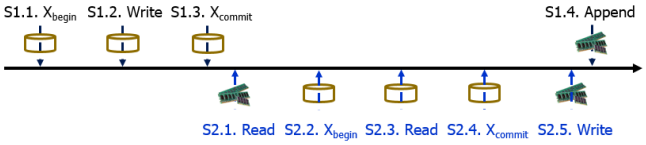


Figure 8: The key-value pair reflects two appends by S1.

Our solution processes read sessions using I leases in an identical manner to the discussion of Section 3.2. To provide strong consistency with the write sessions that perform KVS δ or R-M-W operations, a session must satisfy the following conditions. First, it must obtain Q leases on those key-value pairs that it intends to update prior to committing its RDBMS transaction. It may do so either prior to starting its RDBMS transaction or as a part of its RDBMS transaction. Second, the session must write its updated key-value pairs to the KVS once its RDBMS transaction commits and release its Q leases. Third, once the KVS grants a Q lease on a key-value pair and the lease expires, the KVS deletes the key-value pair.

We use the Q lease to prevent the race condition of Section 4.1 by implementing the cas command as two separate commands: Quarantine-and-Read, QaRead(key), and Swap-and-Release, SaR(key, vnew). We describe these in turn.

Quarantine-and-Read, QaRead(key), acquires a Q lease on the referenced key from the server and reads the value of the key (if any). If there is an existing Q lease on the referenced key granted to a different session then the server returns an abort message, see the matrix in Figure 5.b. In this case, the requesting session must release all its leases, roll back any RDBMS transactions that it may have initiated (see below), back off for some time, and re-try its execution. As detailed in Section 5, some of these functionalities such as maintenance of the leases is implemented in the client of the KVS and are transparent to both the application and its developer.

It is possible for QaRead to reference a key with no value in the KVS. In this case, the KVS grants a Q lease (if one does not exist) to the session and returns a KVS miss for the key, i.e., a null value. In this case, the application has a choice. It may either check and skip updating of the value or it may query the RDBMS and compute a value, modify

it, and insert it using the Swap-and-Release, SaR, command below.

When a QaRead lease encounters an I lease granted to a different session, it invalidates the I lease. This is to prevent undesirable race conditions attributed to snapshot isolation from inserting stale data in the KVS since the RDBMS ordering of the execution between the reader (with an I lease) and a writer (QaRead) is unknown to the KVS.

Swap-and-Release, SaR(key, v_{new}), changes the current value of the specified key with the new value, v_{new}, and releases the Q lease on the key. When v_{new} is null, the KVS simply releases the Q lease.

The QaRead implements the compatibility matrix of Figure 5.b which aborts a session requesting a Q lease for a key-value pair with an existing Q lease. This is because the serial order of two concurrent write sessions in the RDBMS is not known to the KVS. By aborting and restarting one of the write sessions competing for the same key-value pair, the KVS serializes this session after the one holding the Q lease because it may not proceed with its RDBMS transaction until the session owning the Q lease releases it.

A session must issue the QaRead command for each key that it intends to update using the R-M-W. Should the KVS respond with abort for a QaRead command, the session must release all its leases and acquire them again in order to avoid the possibility of deadlocks.

Once the RDBMS transaction of a session commits, the session must issue SaR for each impacted key with its new value. This updates the values in the KVS and releases the Q leases on the impacted keys.

Our proposed use of Q leases resembles two phase locking as it requires a session to issue all its QaRead calls prior to the RDBMS transaction commit and all its SaR invocations after transaction commit. A session may perform its QaRead calls either prior to the start of the RDBMS transaction or as a part of the RDBMS transaction. Our evaluation of Section 6 shows both approaches to provide a comparable performance with the former resulting in starvations, see Table 6.

4.2.1 Solution for δ

With incremental update, a write session employs the following commands:

1. IQ-δ(k_i,δ_i) where δ is an incremental update command such as append, k_i is the impacted key, and δ_i is the change to the key such as the value to be appended to the existing value of k_i. Similar to refresh, a session must issue this command either prior or as a part of its RDBMS transaction. And, similar to QaRead, if there is an existing Q lease on k_i granted to another write session then the KVS return an abort message, causing the calling write session to release all its leases and abort its in-progress RDBMS transaction (if any) and try again.

2. Commit causes the KVS to release the Q leases of the write session.

Similar to QaRead, IQ-δ implements the compatibility matrix of Table 5.b.

4.2.2 Optimizations

To enhance concurrency of read and write sessions, one may require the KVS to maintain an older version of the k_i-v_i pair that is in the process of being updated by a write session S1. Once S1 commits, the KVS server puts its modifications and changes into effect and releases its Q leases. This enables a read session referencing the same key-value to observe a KVS hit for the older version of k_i, preventing it from requesting an I lease. In a serial schedule this read session will be ordered to have occurred prior to S1.

The write session S1 must be able to observe its own modification. Hence, with this optimization, the KVS server must identify S1 referencing k_i and provide it with the version that it either refreshed or incrementally updated in the KVS. Section 5 describes an implementation of this optimizations for both refresh and incremental update.

5. AN IMPLEMENTATION

We implemented the IQ framework by extending the Twitter memcached version 2.5.3 [30] and the Whalin memcached client version 2.6.1 [38] named IQ-Server and IQ-Client, respectively. An application employs the IQ-Client to communicate with the IQ-Server. Both components participate to implement the IQ framework. We represent a lease as a unique token generated by the IQ-Server and maintained by the IQ-Client on behalf of a session. These tokens are transparent to the application and managed by the commands supported by the IQ-Client. Due to lack of space, we provide a brief overview of commands. We refer the interested reader to [18] for a detailed description of the IQ-Client and the IQ-Server and a separation of the functionality between the two.

1. IQget(key) returns either a value for the specified key, a miss with a token corresponding to an I lease, a miss with either no backoff or backoff.
2. IQset(key,value) is extended by the IQ-Client to include the token that identifies the I lease granted to the calling session. If the token matches the one granted by the IQ-Server, the value of the key is updated. Otherwise, the command is ignored.
3. QaRead(key) implements the R in R-M-W of refresh per Section 4.2. It returns either a value and a token for the Q lease granted by the IQ-Server, a token for the Q lease with no value, no token with either a value or no value, and quarantine unsuccessful. The latter causes the session to release all its leases and abort its execution by rolling back its RDBMS transaction.
4. SaR(key, v_new) implements the W in R-M-W of refresh per Section 4.2. This method succeeds in setting the value of key to v_new only when its Q token is valid.
5. GenID() returns a unique transaction identifier, TID, to identify the session that is either invalidating or δ keys.
6. Quarantine-and-Register, QaR(TID, key), acquires a Q lease on the specified key in support of invalidate.
7. Delete-and-Release, DaR(TID), deletes the keys invalidated by the session identified using TID and releases its Q leases.
8. IQ-δ(TID, k_i, v_i) implements an incremental change such as append or prepend. It obtains a Q lease similar to the discussions of QaRead. Its proposed update is maintained by the IQ-Server and performed once the session commits, see below.
9. Commit(TID) causes the IQ-Server to apply all the changes proposed by the session TID performing incremental updates on keys. Next, it releases the Q leases of this session.
10. Abort(TID) causes the IQ-Server to discard all changes proposed by the session TID performing incremental updates on keys. Next, it releases the Q leases of this session.

BG Action	Very Low 0.1% Write	Low 1% Write	High 10% Write
View Profile	40%	40%	35%
List Friends	5%	5%	5%
View Friends Requests	5%	5%	5%
Invite Friend	0.02%	0.2%	2%
Accept Friend Request	0.02%	0.2%	2%
Reject Friend Request	0.03%	0.3%	3%
Thaw Friendship	0.03%	0.3%	3%
View Top-K Resources	40%	40%	35%
View Comments on Resource	9.9%	9%	10%

Table 5: Four mixes of social networking actions with BG.

6. AN EVALUATION

This section employs the BG social networking benchmark [6] to evaluate the implementation of Section 5. We start with a description of BG. Subsequently, Section 6.3 presents performance results.

6.1 BG Benchmark

BG is a benchmark that rates [6] a data store for processing interactive social networking actions that either read or write a small amount of data from a social graph. Nine actions constitute the core of BG and are used to realize three workloads with a different mix of read and write actions, see Table 5. Each action is implemented as a session of the IQ framework using the physical data design of [6]. Details of those BG actions that write/delete/update rows of the RDB and manipulate KVS key-value pairs are as follows:

Invite Friend(InviterID, InviteeID): This action inserts a row (InviterID, InviteeID, 1) in the Friendship table of the RDB. The third value is for the status attribute with 1 denoting a pending friendship. In addition, it increments the number of pending friends for the row corresponding to InviteeID in the Users table.

This action invalidates/refreshes two keys: 1) the key-value pair corresponding to the profile of the InviteeID by incrementing its number of pending friends by one, 2) the key-value pair corresponding to the pending friend request of InviteeID by extending the value to include the profile of the InviterID. While the first key is read by the 'View Profile' action, the second is read by the 'View Friends Requests' action.

Reject Friend(InviterID, InviteeID): This action removes the row corresponding to (InviterID, InviteeID) from the Friendship table and decrements the number of pending friends for the member row corresponding to InviteeID. With refresh, it updates two keys as follows: (1) it decrements the number of pending friends of the key-value corresponding to the profile of the InviteeID and (2) it removes the profile of InviteeID from the key-value containing the pending friend invitations extended to InviteeID. These two keys are deleted with invalidate.

Accept Friend(InviterID, InviteeID): This action updates the row (InviterID, InviteeID, 1) in the Friendship table by changing its status from 1 to 2. It inserts a new row (InviteeID, InviterID, 2) in the Friendship table. It updates the attribute 'number of pending friends' of the row corresponding to InviteeID by decrementing its value in the User table. Finally, it updates the the attribute 'number of friends' of the rows corresponding to InviterID and InviteeID in the User table by incrementing their values.

It invalidates/refreshes five different key-value pairs: the profile of InviterID and InviteeID are updated to show an increase in their count of friends (2 keys), the list of friends of InviterID and InviteeID are updated to reflect their respective profiles as one another's friends (2 keys), the profile information of the InviterID is removed from the the pending friend invitations of InviteeID (1 key).

Thaw Friendship(InviterID, InviteeID): This action removes the following two rows from the Friendship table: (InviterID, InviteeID, 2) and (InviteeID, InviterID, 2).

It invalidates/refreshes four key-value pairs: the profile of InviterID and InviteeID are updated to show a decrease in their count of friends (2 keys), and the list of friends of InviterID and InviteeID are updated by removing their respective profiles from one another's friend profiles (2 keys).

Given a workload, BG computes the Social Action Rating (SoAR) of its target data store using a pre-specified Service Level Agreement, SLA. In this study, we assume the following SLA: 95% of actions to observe a response time faster than 100 milliseconds. The maximum number of simultaneous actions per second that satisfies this SLA is the SoAR of the system for a workload. The social graph used to compute the SoAR of a data store consists of M members, ϕ friends per member, and ρ resources (e.g., images) per member. In this study, we analyze a small and a large social graph consisting of 10K and 100K members, respectively. There are 100 resources and 100 friends per member in all experiments.

A unique feature of BG is its ability to quantify the amount of unpredictable data produced by a data store. This includes either stale, inconsistent, or simply invalid data produced in response to a read action. BG detects these by maintaining the initial state of a data item in the database (by creating them using a deterministic function) and the change of value applied by each write action. There is a finite number of ways for a BG read action that reads data, e.g., List Friend, to overlap with a concurrent BG action that writes data, e.g., Invite Friend. BG enumerates these to compute a range of acceptable values that should be observed by the read action. If a data store produces a different value then it has produced unpredictable data. This process is named validation and is detailed in [6]. Section 6.3 reports on the amount of stale data using a CASQL system with and without I/Q leases.

6.2 Client Designs

As detailed in Section 4.2, one may implement invalidate, refresh and δ clients by acquiring Q leases either prior to or as a part of the RDBMS transaction. Figure 9 shows these two alternatives with refresh for the Invite Friend action. This section quantifies their tradeoff.

The first implementation invokes the QaRead and SaR commands of KVS (in support of R-M-W) prior to starting the RDBMS transaction. Hence, if the QaRead fails then no RDBMS roll-back is required. The session simply backs off and re-tries until it succeeds. A limitation of this technique is that, with a high system load, a write session might be restarted more than its fair share due to Q lease conflicts, see discussion of Table 6 below. In particular, there is no queuing mechanism to prevent a write session from starving for its Q lease.

The second implementation applies the KVS QaRead and SaR commands during the RDBMS modify/write operation

Invite Friend (InviterID, InviteeID)
1. Key = "Profile"+InviteeID
2. V_{old} = QaRead (Key)
3. V_{new} = Increment V_{old}.#PendingFriends
4. Begin RDBMS Xact
   a. Insert (InviterID, InviteeID, 1) in PendingFriends
   b. Update PendingCount of invitee by 1 in Users table
5. Commit Xact
6. SaR (Key, V_{new})

(a) KVS operations prior to the RDBMS transaction

Invite Friend (InviterID, InviteeID)
1. Begin RDBMS Xact
   a. Insert (InviterID, InviteeID, 1) into Friendships table
   b. Update PendingCount of invitee by 1 in Users table
   c. Key = "Profile"+InviteeID
   d. V_{old} = QaRead (Key)
   e. V_{new} = Increment V_{old}.#PendingFriends
2. Commit Xact
3. SaR (Key, V_{new})

(b) KVS operations during the RDBMS transaction

Figure 9: Two alternative implementations of the Invite Friend session of Figure 10a using QaRead and SaR commands.

Invite Friend (InviterID, InviteeID)
1. Begin RDBMS Xact
   a. Insert (InviterID, InviteeID, 1) into Friendships table
   b. Update PendingCount of invitee by 1 in Users table
2. Commit Xact
3. Key1 = "Profile"+InviteeID
4. V_{old} = KVS Read (Key)
5. V_{new} = Increment V_{old}.#PendingFriends
6. KVS Compare-and-Swap (Key, V_{old}, V_{new})

(a) Invite Friend

Confirm Friend (InviterID, InviteeID)
1. Begin RDBMS Xact
   a. Update status of (InviterID, InviteeID) to 2 in Friendship table
   b. Insert (InviteeID, InviterID, 2) into Friendships table
   c. Update PendingCount of invitee by -1 in Users table
   d. Update FriendCount of inviter and invitee by 1 in Users table
2. Commit Xact
3. Key1 = "Profile"+InviteeID
4. V_{old} = KVS Read (Key1)
5. V_{new} = Decrement V_{old}.#PendingFriends and Increment V_{old}.#ConfirmedFriends
6. KVS Compare-and-Swap (Key1, V_{old}, V_{new})
7. Key2 = "Profile"+InviterID
8. V_{old} = KVS Read (Key2)
9. V_{new} = Increment V_{old}.#ConfirmedFriends
10. KVS Compare-and-Swap (Key2, V_{old}, V_{new})

(b) Confirm Friend

Figure 10: Pseudo-code of two interactive social networking actions implemented as sessions with no leases.

and prior to its transaction commit. This reduces the duration of time Q leases are held in the KVS by a session. However, it increases the complexity of the software for two reasons. First, when the QaRead command of the KVS fails then the RDBMS transaction must be aborted. Second, the developer must be aware of the transaction semantics and its interaction with the modification proposed for a key-value pair. In particular, a KVS read that observes a miss may query the RDBMS to observe the transactional changes and compute a value. If the modification to the value is idempotent (repeating it two or more times produces the same result) then applying it to the retrieved value is acceptable. However, if the modification is not idempotent, e.g., increments the number of pending friends as shown in Figure 10a, then the correctness of software might be compromised by applying the modification to the key twice. One solution is for the developer to author additional software to differentiate between a KVS read that observes a miss or a hit. Another possibility is to employ multiple RDBMS connections and use a different connection to handle KVS reads that observe a miss. This causes the query issued (read transaction) to not observe the updates proposed by the write transaction, avoiding the complexity associated with differentiating between a read that observes a KVS hit or a miss. We implemented this second approach.

Workload	QaRead calls prior to RDBMS transaction **start**		QaRead calls **during** RDBMS transaction	
	Avg	Max	Avg	Max
0.1%	2	4	0	0
1%	6.02	74	1.18	5
10%	4.61	77	1.33	9

Table 6: Average and maximum number of times an aborted session restarts (due to Q lease conflicts).

Table 6 shows the average and maximum number of times a restarted session attempts to obtain its Q lease with the two alternative implementations. The reported numbers are with a high system load, 200 threads, and 70% of requests referencing 20% of data (Zipfian distribution with θ=0.27) [39]. The first column shows the different mixes of write actions. The average number of times an aborted session restarts is lower when the KVS operations are performed as a part of a session's RDBMS transaction. Moreover, the maximum number of session restarts is significantly lower, demonstrating that this implementation does not cause a session to starve when obtaining its Q lease.

6.3 Performance Results

This section compares the performance of two variants of Twemcache: (1) Twemcache extended with read leases of [27], labeled Twemcache, and (2) Twemcache extended with I/Q leases using the implementation of Section 5, labeled IQ-Twemcached.

Table 7 shows the amount of stale data produced by these two alternatives for two different social graphs consisting of 10K and 100K members. These results are gathered with a cold RDBMS cache by restarting the RDBMS at the beginning of each experiment. The 10K social graph is small and fits in the memory of the RDBMS, enabling it to perform a few hundred actions per second. The 100K social graph is too large to fit in the memory of the RDBMS and the RDBMS performs 15-25 actions per second. In the following, we discuss obtained results with the 10K and 100K

System Load	Mix of Actions	10K members		100K members	
		Invalidate	Refresh	Invalidate	Refresh
Low, 10 Threads	0.1%	0.6%	0%	0%	3.3%
	1%	0.5%	0%	0%	3.5%
	10%	0.2%	0%	0%	3.1%
Moderate, 100 Threads	0.1%	1.7%	0.02%	0%	3.3%
	1%	1.1%	1.4%	0%	3.4%
	10%	0.9%	6.3%	0%	3%
High, 200 Threads	0.1%	2.0%	0%	0%	3.2%
	1%	1.3%	1.8%	0%	3.4%
	10%	1.3%	8.3%	0%	2.8%

Table 7: Percentage of unpredictable data using invalidate/refresh with Twemcache by itself. These percentages are reduced to zero with the IQ-Twemcached

	Invalidate		Refresh	
	Twemcache	IQ Twemcached	Twemcache	IQ Twemcached
0.1%	31,492	31,473	31,338	31,184
1%	31,144	31,246	30,615	30,352
10%	29,317	29,204	29,194	29,277

Table 8: SoAR of Twemcache and IQ-Twemcached with a 100% utilized CPU.

social graphs in turn.

With the 10K social graph, the percentage of read actions that observe stale data increases as a function of system load due to a higher number of concurrent threads. With invalidation, these threads increase the likelihood of cache misses that may compute stale key-value pairs by using the snapshot isolation and inserting these in the KVS. This holds true with refresh and the added possibility of these threads updating the same key-value pairs simultaneously, suffering from the write-write conflict shown in Figure 2.

With the 100K social graph, the percentage of unpredictable data with invalidate is negligible and close to zero. This is because the likelihood of concurrent threads referencing the same data is lower due to a social graph that is ten times larger. With refresh, approximately 3% of read sessions observe stale data because, once a stale key-value is inserted in the KVS, there is no mechanism to remove it from the KVS. This percentage is a constant with different number of threads because the RDBMS is the bottleneck resource and limits the number of concurrent threads to between 15-25. Moreover, the larger database size results in a longer response time for each session. This longer life time for each session increases the possibility of two or more session encountering a write-write conflict with refresh, resulting in a higher percentage of unpredictable reads. This probability does not change with a higher system load (200 threads) as the number of concurrently executing sessions is limited to 15-25 by the RDBMS.

With the IQ-Twemcached, the reported percentage of unpredictable reads in Table 7 is reduced to zero. This is because the leases prevent the undesirable race conditions that cause the cache to produce stale data.

With a warm cache, the performance observed with the IQ-Twemcached is comparable with the Twemcache. There are two reasons for this. First, the overhead of the IQ framework is negligible. Second, the percentage of session restarts due to Q leases conflicting with one another is low, see Table 7. Table 8 shows the SoAR (highest throughput) observed with the Twemcache and the IQ-Twemcached by limiting the core of the cache server to one, causing the CPU of the cache server to become fully utilized. Both Twemcache and IQ-Twemcached provide comparable performance.

7. RELATED WORK

The IQ framework guarantees serial schedule of sessions. There exists a vast number of concurrency control algorithms, most of which are based on either locking [26, 19, 32], optimistic or commit-time validation [5, 12, 22], and timestamps [31, 36]. See [9] for a survey of these algorithms and how one may combine them. IQ resembles locking and is least comparable to the timestamp protocols (not discussed further). The latter serialize transactions based on their order of arrival. Similar to locks, the IQ framework produces a serial schedule based on how sessions compete to acquire leases. Similar to two-phase locking (2PL), a session has a growing and a shrinking phase with IQ. Its growing phase is prior to the RDBMS transaction commit when it acquires its leases. Its shrinking phase is after the RDBMS transaction commits when the KVS server applies session's changes to the impacted key-value pairs and releases the session's Q leases.

A lease is different than a lock because it has a finite life time. When it expires, its key-value pair is released without coordination with the lease holder. Similar to the Shared (S) and eXclusive (X) locks [26, 19, 32] of a lock manager, the I and Q leases are not compatible with one another. However, the semantics of I and Q leases make them different than S and X locks. When one compares the I lease with the S lock, there may exist at most one I lease on a key-value pair while there may exist multiple S locks on a data item. This makes the I lease similar to an X lock. However, the Q lease preempts an existing I lease always, preventing the I lease holder from populating the KVS with its key-value pair. If one assumes the I lease is similar to an X lock, then the Q lease is stronger. Note that two or more Q leases are compatible with one another with invalidation, see Figure 5.a, making it similar to the S lock. This is not true with either refresh or incremental update. Thus, the I and Q leases are fundamentally different than S and X locks.

IQ is also similar to the Optimistic Concurrency Control (OCC) algorithm [5, 12, 22] as a session consists of a read and a write phase for the KVS. Its write phase occurs after the RDBMS transaction commit and, similar to the write phase of OCC, is always successful. During its read phase, IQ obtains leases as it validates the values read from the KVS. This concept is missing from OCC. Moreover, IQ lacks the explicit validation phase of OCC. Instead, it rolls a session back during its read phase once it detects a conflict using the I/Q leases.

Two studies most relevant to our focus include TxCache [29] and the leases of [27]. We describe these in turn. TxCache [29] is a transparent caching framework that extends an RDBMS with additional software to produce invalidation tags to the KVS. These tags are generated by the RDBMS updates and cause the KVS to generate versions of the key-value pairs to implement snapshot isolation with the KVS. Our proposed framework maintains a single version of a key-value pair and requires no software changes to the RDBMS. Moreover, TxCache's tags are designed for the invalidate technique. It does not consider the refresh and incremental

update techniques (see Figure 1) and does not propose use of leases to provide strong consistency.

In [27], Facebook describes how it uses a lease to avoid undesirable race conditions that cause the KVS to produce stale data with an invalidate technique. In addition, the same lease is used to prevent thundering herds; a burst of requests observing a KVS miss for the same key and querying the RDBMS for the same result. Facebook lease provide strong consistency with invalidate when it is implemented in the application. The IQ framework is different as it provides strong consistency with invalidate, refresh and incremental update independent of its implementation in either the application the RDBMS triggers. To elaborate, consider the race condition shown in Figure 3 as the KVS delete is performed by a trigger [16, 20]. Facebook lease does not prevent this undesirable race condition. To illustrate, assume S2 obtains its Facebook (I) lease as a part of Step 2.1. Since Step 1.3 occurs prior to Step 2.1, the lease provided as a part of Step 2.5 is valid and inserts its stale value in the KVS successfully. To enable Facebook lease to provide strong consistency, one may change the sessions of Figure 3 in two ways. First, the session must perform its KVS delete in the application. Second, it must issue the KVS delete after the RDBMS transaction commits, switching the order of Steps 1.3 and 1.4. Now, the Facebook lease prevents the undesirable race condition attributed to snapshot isolation. Similar to discussions of Section 3.3, the read session that observes stale values between Step 2.5 until the time S1 deletes the key, requiring KVS reads that observe these values to be serialized prior to S1.

Facebook lease is implemented in the Twitter memcached version that we evaluated in Section 6.3 and showed to produce stale data, see Table 7. Our proposed I lease is identical to leases of [27]. Our framework is different because it introduces the Q lease and defines its compatibility with the I lease to reduce the amount of stale data down to zero. Moreover, our framework supports the refresh technique to update the KVS. Our implementation of I/Q leases enables an application to use both invalidate and refresh simultaneously.

IQ is designed to provide strong consistency within a data center. One may deploy the CASQL solution in different data centers with replicated data, see [27] for an example. To maintain the replicated data consistent across data centers, one may use a technique such as parallel snapshot isolation [34], eventual consistency [37], per-record timeline consistency [13], causal+ [24] and others. While these techniques focus on network partitions, IQ focuses on normal mode of operation and use of leases to prevent undesirable race conditions in a data center. Its objective is to provide strong consistency with no modification to the RDBMS software.

There are mid-tier caches that process SQL queries [2, 25, 11, 23, 1, 35]. These caches maintain fragments of the RDB to distribute processing of queries across the caches and backend servers intelligently. The cached data is maintained consistent with the changes to a backend server using a variety of techniques such as use of materialized views with asynchronous data replication [25], computing changes and shipping them to the caches [2, 11], shipping log records [23], and invalidation of the impacted rows [1]. Our target CASQL system is different as the KVS maintains unstructured key-value pairs. It has no ability to process SQL queries and provides a simple interface that supports commands such as get and set, see second column of Table 3. Thus, the KVS does not incur the overhead of query processing estimated at a high percentage of useful work performed by today's RDBMSs [21].

8. CONCLUSION

This study demonstrates the feasibility of implementing strong consistency in CASQL systems using an off-the-shelf RDBMS. It is based on a simple programming model that acquires I/Q leases from the KVS either prior to the start of an RDBMS transaction or during the processing of the RDBMS transaction. In our implementation, the existence of tokens and the concept of back off is transparent to the developer of a session. Moreover, it enables a developer to use invalidate, refresh, and incremental update methods of KVS update simultaneously, see Figure 1. While the developer must apply the KVS updates after the RDBMS transaction commits which in turn releases the obtained leases, the framework is robust to node (application) failures as leases have a finite life time and may expire. The IQ framework provides for non-blocking execution and is free from deadlocks.

The current IQ framework limits a session to at most one RDBMS transaction. A key research question is whether the framework provides strong consistency guarantees for sessions consisting of multiple RDBMS transactions. We intend to investigate this by extending BG with more complex actions that require sessions with multiple transactions, e.g., streams [33, 7]. We also plan to analyze other benchmarks such LinkBench [4] and RUBiS [3] to evaluate this research question. Another short term effort is to publish our IQ implementation (Whalin client, Twemcache server), its documentation, and example code segments on the web. More long term, we are exploring an incremental approach [20] to update key-value pairs in the KVS and use of the IQ framework to provide strong consistency.

9. REFERENCES

[1] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*, 2003.

[2] K. Amiri, S. Park, and R. Tewari. DBProxy: A dynamic data cache for Web applications. In *ICDE*, 2003.

[3] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and Implementation of Dynamic Web Site Benchmarks. In *Workshop on Workload Characterization*, 2002.

[4] T. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. *ACM SIGMOD*, June 2013.

[5] D. Badal. Correctness of Concurrency Control and Implications in Distributed Databases. In *COMPSAC Conference*, November 1979.

[6] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR*, January 2013.

[7] S. Barahmand, S. Ghandeharizadeh, and D. Montauk. Extensions of BG for Testing and Benchmarking Alternative Implementatios of Feed Following. *ACM SIGMOD Workshop on Reliable Data Services and Systems (RDSS)*, 2014.

[8] S. Barahmand, S. Ghandeharizadeh, and J. Yap. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. *CIKM*, 2013.

[9] P. Bernstein and M. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2), June 1981.

[10] P. Bernstein and N. Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Transactions on Database Systems*, 8:465–483, February 1983.

[11] C. Bornhovdd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive Database Caching with DBCache. *IEEE Data Engineering Bull.*, pages 11–18, 2004.

[12] S. Ceri and S. Owicki. On the Use of Optimistic Methods for Concurrency Control in Distributed Databases. In *Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1982.

[13] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 1(2), Aug. 2008.

[14] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. *ACM/IFIP/USENIX Middleware*, December 2014.

[15] S. Ghandeharizadeh and J. Yap. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, June 2013.

[16] S. Ghandeharizadeh and J. Yap. SQL Query To Trigger Translation: A Novel Consistency Technique for Cache Augmented DBMSs. In *USC DBLAB Technical Report 2014-07, http://dblab.usc.edu/users/papers/sqltrig.pdf, Submitted for Publication*, 2014.

[17] S. Ghandeharizadeh, J. Yap, and S. Barahmand. COSAR-CQN: An Application Transparent Approach to Cache Consistency. In *International Conference On Software Engineering and Data Engineering*, 2012.

[18] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems, http://dblab.usc.edu/Users/papers/IQTechReport.pdf. In *USC Database Laboratory Technical Report Number 2014-06*, 2014.

[19] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*. Sprinter-Verlag, 1979.

[20] P. Gupta, N. Zeldovich, and S. Madden. A Trigger-Based Middleware Cache for ORMs. In *Middleware*, 2011.

[21] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, 2008.

[22] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6, June 1981.

[23] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE*, pages 177–189, 2004.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*, 2011.

[25] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-Tier Database Caching for e-Business. In *SIGMOD*, 2002.

[26] D. Menasce and R. Muntz. Locking and Deadlock Detection in Distributed Databases. In *Third Berkeley Workshop on Distributed Database Management and Computer Networks*, 1978.

[27] R. Nishtala et. al. Scaling Memcache at Facebook. *NSDI*, 2013.

[28] F. Perez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Elastic SI-Cache: Consistent and Scalable Caching in Multi-Tier Architectures. *VLDB Journal*, 2011.

[29] 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 *OSDI*. USENIX, October 2010.

[30] M. Rajashekhar and Y. Yue. Twitter memcached (Twemcache) is version 2.5.3, https://github.com/twitter/twemcache/releases/tag/v2.5.3.

[31] D. Reed. Naming and Synchronization in a Decentralized Computer System, Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, 1978.

[32] D. Rosenkrantz, R. Stearns, and P. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3, June 1978.

[33] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan. Feed Following: The Big Data Challenge in Social Applications. In *DBSocial*, 2011.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-Replicated Systems. In *SOSP*, 2011.

[35] T. T. Team. Mid-Tier Caching: The TimesTen Approach. In *Proceedings of the SIGMOD*, 2002.

[36] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4, June 1979.

[37] W. Vogels. Eventually Consistent. *Communications of the ACM, Vol. 52, No. 1*, January 2009.

[38] G. Whalin, X. Wang, and M. Li. Whalin memcached Client Version 2.6.1, http://github.com/gwhalin/Memcached-Java-Client/releases/tag/release_2.6.1.

[39] J. Yap, S. Ghandeharizadeh, and S. Barahmand. An Analysis of BG's Implementation of the Zipfian Distribution, USC Database Laboratory Technical Report Number 2013-02.