Strong Consistency in Cache Augmented SQL Systems[*]

Shahram Ghandeharizadeh, Jason Yap, Hieu T. Nguyen

Database Laboratory Technical Report 2014-06

Computer Science Department, USC

Los Angeles, California 90089-0781

October 28, 2014

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.

A 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, e.g., interactive social networking actions. The key insight is that query result look up 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

[*]A shorter version appeared in the Proceedings of the ACM/IFIP/USENIX Middleware Conference, Bordeaux, France, December 2014.
[1]Insert, delete, and update commands.

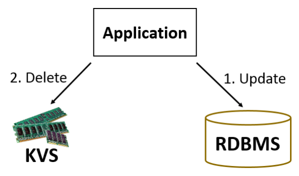


Figure 1a: Invalidate

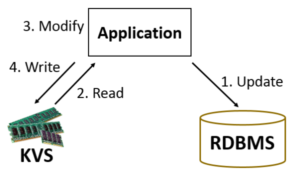


Figure 1a: Refresh

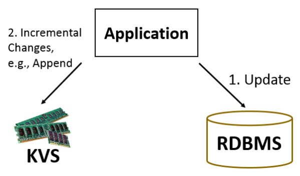


Figure 1a: Incremental Update (δ)

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

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 G 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. With this approach, once a session provides accurate results with an isolated execution, it is guaranteed to produce accurate results when executed concurrently with other sessions. 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 framework provides *strong consistency* and is desirable as it makes systems easier for a programmer to reason about [26]. 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 G. We show starvation can be avoided by performing the KVS operations as a part of the RDBMS transaction that constitutes a session.

In [14], Facebook describes the concept of a lease to avoid undesirable race conditions when invalidate is imple-

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.

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 7.
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.

mented 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 [28, 18, 35]. 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 H.

The rest of this paper is organized as follows. Section B introduces terms and their definitions as used in this paper. Sections C and D 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 F presents an implementation of the I/Q leases using Twemcache, Section G evaluates this implementation using a social networking benchmark named BG. We review related work in Section H. Brief conclusions and future research directions are presented in Section I.

B 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 [28, 18, 35]. 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. Assumptions of the IQ framework are shown in Figure 3.

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

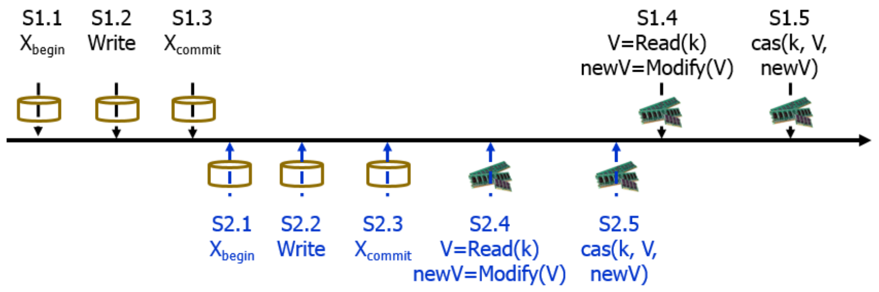


Figure 2: Use of compare-and-swap (cas) does not provide strong consistency. Two concurrent sessions may update the RDBMS and the KVS in different order, leaving the data in the KVS and the RDBMS in an inconsistent state.

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 D, The 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. (This paragraph is the basis of Assumptions 2 and 3 of Figure 3.) 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

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.

```
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, $V_{old}$, $V_{new}$)
```

5a. 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, $V_{old}$, $V_{new}$)
```

5b. Confirm Friend

Table 5: Pseudo-code of two interactive social networking actions implemented as sessions with no leases.

1. A session implements simple operations that read and write a small amount of data from big data, categorized into read and write sessions, see below. Examples include interactive social networking actions such as view a member's profile, confirm friendship, and others.
2. A read session may either read or read-and-write a single key-value pair from the KVS. The latter is due to a KVS read for a key that observes a miss and issues an RDBMS transaction (read operation, see Table 3) to compute a value for the referenced key, writing the resulting key-value pair to the KVS.
3. A write session issues one RDBMS transaction (either an RDBMS write, delete, or update operation) and may impact multiple key-value pairs, see Table 3. With invalidate, the impacted key-value pairs are deleted. With refresh, the value of the impacted key-value pairs are read, modified, and written back to the KVS. With incremental update, the value of the impacted key value pair is updated by the KVS using operations such as append and prepend, see Table 3.
4. Strong consistency is realized when N concurrent read and write sessions are serialized. When a session S_1 is ordered before another session S_2 then S_1 does not observe S_2's updates to either the RDBMS or the KVS. S_2 must observe S_1's updates regardless of whether it accesses the RDBMS or the KVS. There are $N!$ valid serial schedules and each guarantees strong consistency [19].

Figure 3: Assumptions of the IQ framework.

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.

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 Table 6 and discussions of Sections C and D.

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. Section F.4 describes how to decide the life time of leases.

Section F.1 describes an implementation of a KVS client that hides the concept of leases and their back off from the programmer, simplifying the application programs. Table 8 shows two different re-writes of the pseudo-code of the "Invite Friend" to use the I/Q leases with refresh.

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

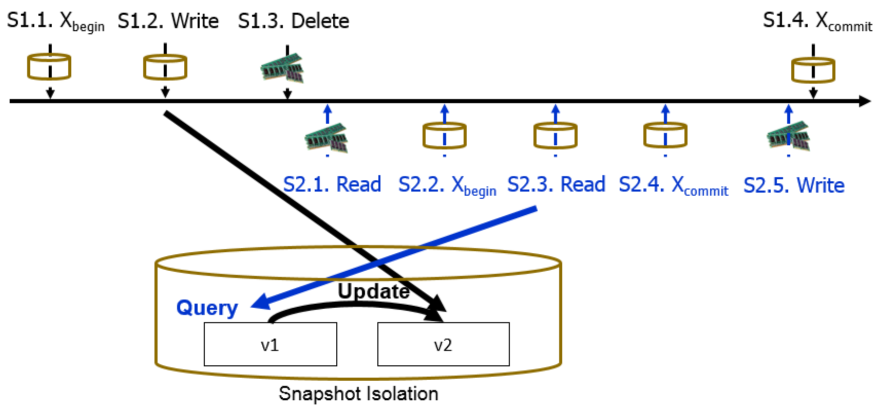


Figure 4: Snapshot isolation enables S2 to compute and insert a stale value in the KVS.

C Invalidate

This section describes how invalidate incurs an undesirable race condition with snapshot isolation to insert stale data in the KVS. Section C.2 presents how the I/Q framework prevents this race condition. Finally, Section C.4 presents correctness of the IQ framework formally.

C.1 Undesirable Race Condition

To improve performance, many RDBMSs offer snapshot isolation, a multi-version concurrency control [9] 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 [31] between a read and a write session, inserting stale data in the KVS. An example is shown in Figure 4 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.

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 4. To illustrate, consider moving Step 1.3 of Figure 4 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.

C.2 Solution

The IQ framework consists of two leases, I and Q, that prevent the race condition of Figure 4. 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 6.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 [14]. 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 4. The KVS grants the Q lease always, see Figure 6.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 Table 6.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.)

C.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 ses-

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

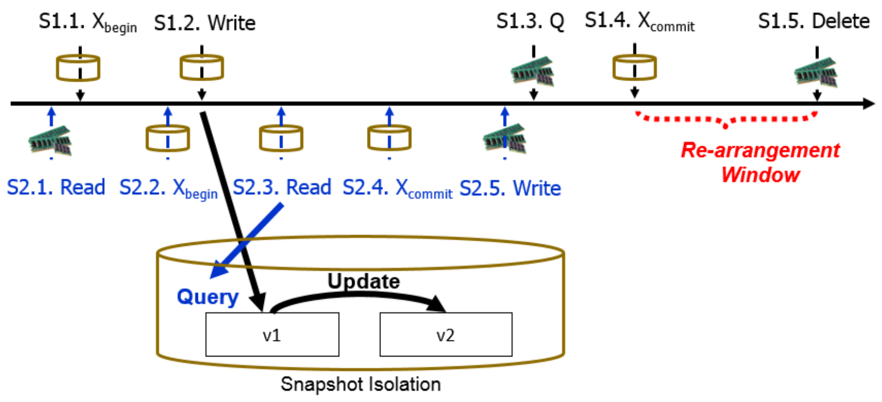


Figure 5: Re-arrangement window for read sessions that are serialized prior to the write session S1.

sions 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.

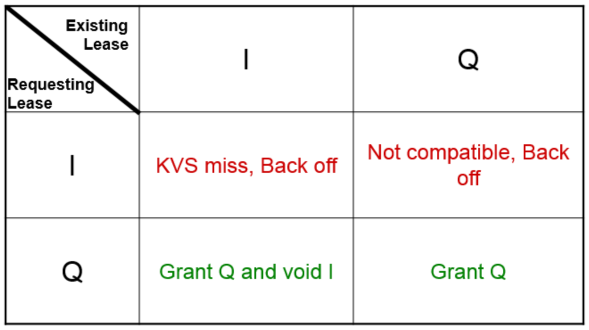
With this optimization, Step 1.3 of S1 in Figure 4 obtains a Q lease without deleting its referenced key. S1 deletes its key in Step 1.5 after Step 1.4 once it commits, see Figure 5. Assuming the referenced key is KVS resident, Step 2.1 of S2 observes a KVS hit, eliminating its remaining steps 2.2 to 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 F 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.

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.

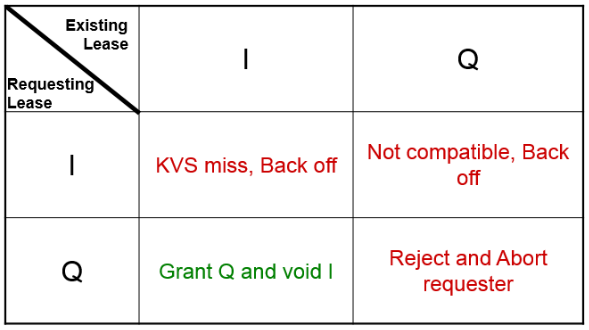
The proposed optimization results in a higher probability of concurrent read sessions being re-arranged to have occurred before a write session in a serial schedule. This is due to a longer re-arrangement window that is non-existent without the optimizations of this section. This is shown in Figure 5 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 F contains the optimizations detailed in this section.



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

6a. 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

6b. Refresh

Table 6: Compatibility matrices of I/Q leases.

C.4 Correctness

Intuitively, the IQ framework is correct because it (1) identifies the key-value pairs absent from the KVS with corresponding RDB data that is in the process of being updated in the RDBMS, and (2) requires the concurrent read sessions that reference the absent key-value pairs to wait until the in progress RDBMS updates either abort or commit. Together, they prevent a read session from computing and inserting a stale value in the KVS.

More formally, consider two concurrent sessions. A read and a write session, denoted S_R and S_W, respectively. S_W performs an RDBMS transaction (X_W) that updates some data in the RDB that is the basis of a key-value absent from the KVS. S_R references this key-value pair, observes a miss, and performs an RDBMS transaction (X_R) that queries the database to compute the key-value pair. To show correctness, we must show when S_R computes a value using a previous snapshot of the database (due to the use of snapshot isolation), its KVS write (KVS_W) operation is ignored by the KVS. Below, we provide correctness by assuming S_W impacts only one key-value pair. Subsequently, we eliminate this assumption to show correctness is preserved when S_W impacts multiple key-value pairs.

While S_R consists of X_R and KVS_W, S_W consists of X_W followed by KVS_{Delete}. KVS_{Delete} invalidates the key-value pair(s) impacted by S_W. There are six possible ways for operations of S_R and S_W to overlap:

1. $X_R, KVS_W, X_W, KVS_{Delete}$
2. $X_R, X_W, KVS_W, KVS_{Delete}$
3. $X_R, X_W, KVS_{Delete}, KVS_W$
4. $X_W, X_R, KVS_W, KVS_{Delete}$
5. $X_W, X_R, KVS_{Delete}, KVS_W$
6. $X_W, KVS_{Delete}, X_R, KVS_W$

In the first scenario, S_R obtains an I lease that it releases after its KVS_W. The X_W of S_W may overlap S_R's X_R, causing X_R to employ snapshot isolation to compute a stale value. using an older version of the RDB. However, X_W

must obtain a Q lease prior to its commit time, invalidating the I lease of S_R and causing the KVS to ignore the stale value, i.e., KVS_W operation of S_R. It is possible for X_W to overlap X_R and obtain its Q lease at its commit time and after KVS_W completes. All sessions that read the resulting key-value pair are serialized to have occurred prior to S_W. This means S_R is prior to S_W and does not observe the update performed by S_W because it completes its execution prior prior to X_W commit. Moreover, the key-value pairs in the KVS are consistent with RDB. This final state of the data in the RDBMS and KVS is consistent with a valid serial schedule.

The second and third scenarios are a special case of the first because it shows the X_W obtaining its Q lease prior to S_R's KVS_W. This invalidates S_R's I lease, causing the KVS to ignore the KVS_W operation of S_R. The sessions are now serialized by the RDBMS with no stale key-value pairs in the KVS, ensuring correctness.

With the sequence of Scenario 4, S_R observes a KVS miss and must obtain an I lease prior to its X_R. If it encounters the Q lease granted to S_W when it performed its X_W operation then it must back off until S_W releases its lease, serializing S_R to occur after S_W. It is possible for S_R and S_W to race with one other when requesting their leases. If S_R wins, the Q lease request of S_W invalidates S_R's I lease causing the KVS to ignore its subsequent KVS_W operation. If S_W wins then S_R must back off and wait until S_W commits X_W and release its lease, observing S_W's update. Both serial schedules are correct.

Scenarios 5 and 6 are similar to the discussion of Scenario 4 because, in each case, the I lease required by S_R must race with the Q lease of S_W. The resulting serial schedules are correct due to the discussion of Scenario 4.

In passing, note that the Q lease is required to provide strong consistency. With the I lease only and no Q lease, the discussions of Scenario 1 and 4 would violate the strong consistency guarantees. This explains why the *lease* of Facebook that is identical to our I lease is insufficient to provide strong consistency, see Section H.

When S_W invalidates multiple (k) key-value pairs, each key-value pair may impact k different concurrently executing S_R session: S_{R_1}, S_{R_2}, ..., S_{R_k}. (See the next paragraph for a generalized discussion of multiple concurrent read sessions referencing the same key with S_W impacting $n + m$ read sessions.) S_W may not impact two or more read sessions because each S_{R_i} reads at most one key-value pair. The interaction between S_W and each S_{R_i} is identical to the above discussion, serializing S_{R_i} and S_W. This order does not impact the ordering of the other read sessions as the read sessions are independent of one another when ordered in a serial schedule relative to S_W. Thus, the final serial schedule is ensures isolation of sessions and is correct.

It is possible for $n + m + p$ concurrent read sessions to race with S_W where n read sessions referencing unique keys ($n \leq k$), m read sessions referencing one of the n unique keys at least once, and p read sessions referencing keys different than S_W. The serial of the p read sessions is dictated by the RDBMS should they reference the same data in the RDB. With the read sessions referencing the same key and observing a miss, the KVS grants an I lease to one (S_i), causing the other concurrent read sessions to back off and try again. S_i may either succeed to produce a value or fail if S_W invalidates its I lease. With the former, those read sessions that observe a hit using S_i's value are serialized to have occurred at the same time as S_i. With the latter, one of the read sessions that observes a miss (a second time because

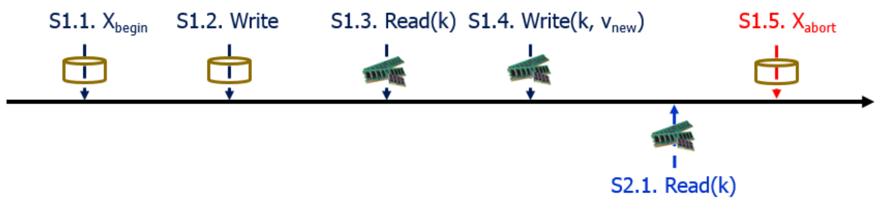


Figure 6: Without the IQ framework, CASQL results in dirty reads with refresh when an impacted key-value pair is updated prior to transaction commit.

S_i produced no value in the KVS) is granted the I lease and takes on the role of S_i while others back off and repeat the process. In both scenarios, the IQ framework guarantees a correct serial schedule consistent with Assumption 4, see Figure 3.

D Refresh and Incremental Update

In addition to the race conditions of Section C.1, the refresh and incremental update techniques suffers from undesirable race conditions attributed to R-M-W and δ operations. Section D.1 presents these race conditions. Subsequently, Section D.2 modifies the semantics of the I/Q leases and how they prevent these race conditions. This section concludes with a formal correctness of the IQ framework.

D.1 Problem definition

Section B 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 Session 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

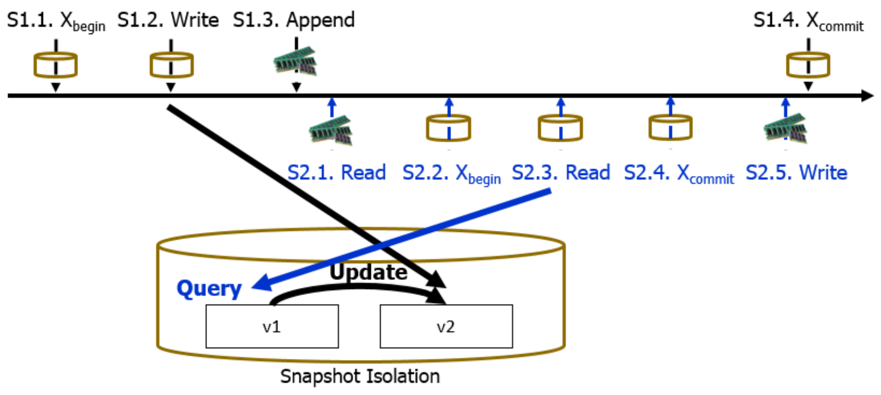


Figure 7: Use of RDBMSs with snap-shot isolation results in stale values being inserted in the KVS.

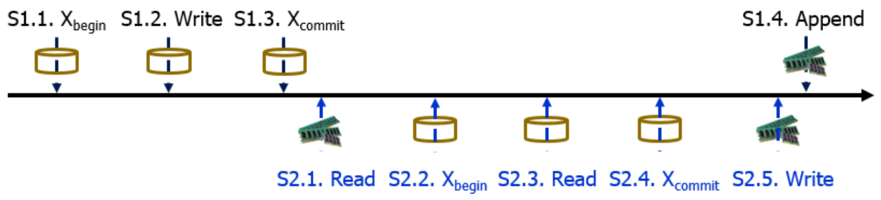


Figure 8: The key-value pair reflects two appends by S1 due to S2 observing its RDBMS update.

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.

D.2 Solution for refresh and δ

This section presents the solution for refresh first. Subsequently, Section D.2.1 describes its extensions to incremental update. Section D.2.2 describes optimizations to these solutions to enhance concurrency between read and write sessions.

Our solution processes read sessions using I leases in an identical manner to the discussion of Section C.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:

1. 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.
2. The session must write its updated key-value pairs to the KVS once its RDBMS transaction commits and release its Q leases.
3. 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 D.1 by implementing the cas command as two separate commands:

- 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 6.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 F, 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 Table 6.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. To illustrate, consider S1 acquiring a Q lease on data item D1 and observing a conflict with S2 when acquiring a Q lease on a different data item D2 causing S1 to back off and try. If S2 attempts to acquire a Q lease on D1 then it will conflict with S1 and back off, resulting in a deadlock. Instead of backing off (when Q leases of two sessions conflict), we require each session to release all its leases and try again after a random time out period, rendering the IQ framework free from deadlocks. Depending on how the leases are acquired, this design may or may not result in starvation during peek system load as described in Section G.

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.

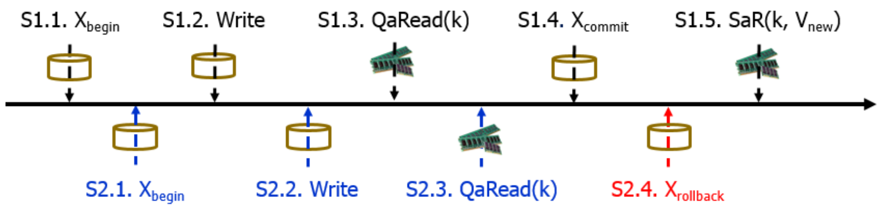


Figure 9: The RDBMS transaction of Session 2 is aborted, rolled back, and retried because its QaRead requests a Q lease that conflicts with the existing Q lease of Session 1.

Figure 9 shows how the sessions of Figure 2 are extended with the QaRead and SaR commands. In this figure, the sessions are implemented to issue their QaRead as a part of their RDBMS transaction. In Step 2.5, once S2 invokes QaRead, the KVS detects its conflict with that of S1 as they reference the same key. Since S1 issued its QaRead earlier and was granted the Q lease, the KVS returns an abort message to S2. In response, Session 2 aborts its RDBMS transaction (Step 2.6) and tries again.

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 G shows both approaches to provide a comparable performance with the former resulting in starvations, see Table 10.

D.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 6.b.

D.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 F describes an implementation of this optimizations for both refresh and incremental update.

D.3 Correctness

To show the correctness of the IQ framework with refresh, we must show the following two conditions are satisfied:

1. Read sessions that observe a miss and race with write sessions do not produce stale key-value pairs in the KVS, and

2. The serial order of multiple concurrent write sessions updating the KVS is identical to their serial order of updating the RDBMS.

The first condition is identical to the discussion of invalidation, see Section C.4. With the second condition, the main insight into the correctness of the IQ framework is the observation that a write session S1 must obtain a Q lease on its referenced key-value pair prior to updating the RDBMS. Once its RDBMS transaction commits, S1 may write its key-value pair to the KVS and release its Q lease. A concurrent session S2 that requests a Q lease on the same key-value pair is aborted and restarted until S1's releases its Q lease. In essence, S2 is serialized to occur after S1, performing its KVS write and RDBMS write after S1. Once S1 releases its Q lease, it is not allowed to acquire additional Q leases. Thus, there is no possibility of a circular dependency between S1 and S2, ensuring correctness of the IQ framework.

It is possible for the server hosting S1 to crash. S1's Q lease will time out and the KVS deletes its quarantined key-value pair because it is undecidable whether S1 committed its RDBMS transaction or not. Session S2 may now obtain the Q lease on the key-value pair and update the RDBMS. S2 may either generate a new key-value for the KVS and release its Q lease or simply release its Q lease without generating a value. With the latter, a read session S3 that references this key-value pair would observe a miss, query the RDBMS, and populate the KVS. If S1's RDBMS transaction commits prior to its server crashing then S3 is serialized after it. If S1's transaction is aborted then deletion of the key-value pair (due to time out) was not necessary. This has no impact on correctness. It may diminish system performance slightly to ensure correctness and strong consistency.

E Session and Lease Strength

The Q lease is stronger and more restrictive than the I lease. A session may include arbitrarily complex logic and request I and Q leases that reference the same key repeatedly during its life time. If it requests a new lease that is the same strength as its existing lease then it is provided with the same lease to proceed forward. If it requests a stronger lease then the IQ framework upgrades its lease to the stronger lease as long as there is no conflict. Below, we provide details.

There are several ways a session may repeat a lease request for the same key. It may either (1) acquire a Q lease and request either a Q or an I lease for the same key, or (2) acquire an I lease and request either an I or a Q lease for the same key. With each, a repeat request may reference a lease that has expired. This is discussed at the end of this section. In the following, we describe each scenario assuming the existing lease is valid.

With the first, once a session acquires a Q lease on a key and repeats its Q lease request, the KVS server grants the same lease to the session, enabling the session to proceed forward. Should the session request an I lease for the same key, the KVS server returns a null value and no lease. The Whalin Client detects this and grants a distinctive (-1) lease to the session to proceed to query the RDBMS for a value. When the session tries to insert its computed value, the client detects the distinctive lease and returns (false) without insering the key-value pair in the KVS server.

When the KVS grants an I lease to a session and the session repeats its I lease request for the same key then the session is provided with the same lease to proceed forward. Should the session request a Q lease (by performing a QaRead) on the key then its existing lease is upgraded to a Q lease. If there is an existing Q lease granted to a different session, the current session must abort.

The session must be the holder of the lease in order to repeat its lease request. For example, if a Session 1 obtains an I lease on a key and another Session 2 requests a Q lease on the same key, the IQ framework grants the Q lease to Session 2 by invalidating the I lease of Session 1. If Session 1 repeats its I lease on the same key then it must back off as it is no longer a lease holder of the key and its request is not compatible with the existing lease on the key.

F An Implementation

We implemented the IQ framework by extending the Twitter memcached version 2.5.3 [33] and the Whalin memcached client version 2.6.1 [41] 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. Below, we provide the details of the IQ-Client and the IQ-Server in turn.

F.1 IQ-Client

A session employs the connections of an IQ-Client to issue commands to the IQ-Server. The implementation of our IQ-Client extends the Whalin memcached client version 2.6.1 [41] with the following additional commands:

- IQget(key) returns either 1) success with a value for the specified key, 2) a miss with a token corresponding to an I lease, 3) a miss with no back off, or 4) a miss with back off. With the first, this method returns the retrieved value to the caller. With the second, the method maintains the provided token on behalf of the caller by associating it with the specified key in anticipation of the caller computing a value for the missing key and invoking IQset(key,value), see below. The third represents the scenario where a session with an existing I or Q lease on the key references the same key using IQget and observes a IQ-Server miss. The return value informs the IQ-Client that the lease already exists, see discussions of Section E.

With the fourth, the method sleeps for a pre-specified duration (that is configurable and is set to 500 millisecond) and tries again. It maintains sufficient history to detect when a session backs off repeatedly, increasing the duration of sleep (back off) exponentially.

- IQset(key,value) issues the set command to the IQ-Server. It is modified by the IQ-Client to include the token that identifies the I lease granted on the key. IQ-Client maintains the association between keys and tokens using its internal hash table. If there is no token for the identified key then this method raises an exception[3]. Otherwise, its return value informs the caller whether the set was performed successfully or was ignored by the IQ-Server because its lease had been invalidated (by either a delete for the same key, a Q lease request by another session, or a lease expiration).

- QaRead(key): Provides the QaRead interface of the refresh technique per specification of Section D.2, returning 1) a value and a token for the Q lease granted by the IQ-Server, 2) a token for the Q lease with no value, 3) no token with either a value or no value, and 4) quarantine unsuccessful. With the first and second, the provided value (or lack of a value) is returned to the caller. The second scenario occurs when the session issues QaRead for a key with either no value or an existing I lease. The IQ-Server returns with no value and grants the Q lease by either upgrading the existing I lease of this session or invalidating the I lease of another session. The third case occurs when a session performs QaRead two or more times after obtaining a Q lease. Finally, the fourth scenario is when there is an existing Q lease granted to a different session. In this case, the IQ-Server returns the INVALID message which causes this method to release all leases granted to this session and raise an exception to the caller, notifying it to either abort or re-start the execution of the session.

- SaR(key, v_{new}): Provides the SaR interface of the refresh technique per specification of Section D.2. Similar to IQset, this method looks up the Q token assigned for the referenced key and provides it to the IQ-Server along with the specified key and v_{new}.

- GenID(): Returns a unique Transaction Identifier(TID) to identify the IQ-Server delete or δ operations[4] that implement either invalidate or incremental update, see discussions of Section F.5. This unique identifier might be generated by either the IQ-Client or the IQ-Server (using Java UUID).

[3]Note that the standard set(key,value) is supported by IQ-Client and, an application that wants to force a value for a key, may use this command directly to bypass the IQ framework.

[4]Performed by either the application or the RDMBS triggers that execute as a part of the transaction that invokes them.

- Quarantine-and-Register, QaReg(TID, key): Acquires a Q lease on the specified key. The provided TID is obtained using GenID() method of the client. A session uses this command in preparation to invalidate one or more key-value pairs.
- Delete-and-Release, DaR(TID): Provides the IQ-Server with the identity of the RDBMS transaction that may have issued QaReg commands to delete one or more key-value pairs. The IQ-Server uses the TID to identify these key-value pairs, delete them from the KVS, and release their granted Q leases. See Section F.5 for an example use case.
- IQ-δ(TID, k_i, v_i): Provides the interface of an incremental change command where δ might be an incremental update such as append or prepend. It obtains a Q lease from the IQ-Server similar to the discussion of the QaRead command.
- Commit(TID): Provides the IQ-Server with the TID of the write session whose changes must be placed into effect and its Q leases released.
- Abort(TID): Provides the IQ-Server with the TID of the write session whose changes must be discarded and its Q leases released.

Typically, a session instantiates a Whalin connection with the IQ-Server and uses it during its life time. This connection is used with the different commands. It maintains the tokens that identify the I and Q leases granted by the IQ-Server. In combination with our design of the commands such as QaRead and SaR, the management of leases is transparent to the application and its software developer. The primary responsibility of a software developer is to catch the raised exceptions to restart a write session and either abort or restart the session. Table 8 shows two alternative ways that one may implement the “Invite Friend” session of Table 5.a, see Section G.2 for details.

F.2 IQ-Server

We implemented the IQ-Server by extending the Twitter memcached versions 2.5.3 [33]. It implements invalidate, refresh, and incremental update simultaneously using the regular memory of Twitter memcached extended with a reserved memory space. The regular memory is used to store application specified key-value pairs. The reserved space is fix-sized and stores two kinds of keys used to implement the IQ framework: 1) an application specified key associated with either an I or a Q lease, and 2) a TID associated with a set of impacted keys. The latter is used to identify a session that may either invalidate (see QaReg below) or incrementally update (see IQ-δ below) a set of application keys during its life time. With incremental update, both the old and the new version of a value of the impacted keys are stored in the regular memory space of the memcached managed using LRU[5]. The IQ-Server implements the following commands:

[5]Should LRU swap out the new version of a value produced by a session (using δ) that commits, the IQ-Server deletes its old version, see IQ-δ below.

1. IQget(key): The IQ-Server looks up the value of the key. If it does not exist then it has encountered a miss. In this case, if there is no pending I or Q lease on the key (by looking up the key in the reserved space) then the IQ-server issues an I lease on the key by generating a unique token, inserting the[6] key and its token in the reserved space, and returning a miss along with the token to the IQ-Client. If there is a pending I or Q lease granted to another session, it returns a miss with back off. If the pending I or Q lease belongs to the calling session, the IQ-Server returns a miss and no token with invalidate and refresh. (With incremental update, it looks up the hash table for the latest version of the value produced by the calling session and returns this value.) If there is a value and no pending I or Q lease, the IQ-Server returns the value.

2. IQset(key, v_{new}, token): The IQ-Server uses the key to look up the key in the reserved space for the issued I lease token. If this token matches the provided input token, then the caller is the holder of a valid I lease and the IQ-Server changes the value of the key to v_{new} atomically. Otherwise, the IQ-Server ignores the specified value and returns "not stored".

3. QaRead(key, token): When the input token is null, the IQ-Server grants a Q lease on the key as long as there is no pending I or Q lease on the specified key. If a value exists for the identified key then the value is also returned. If there is a pending I lease on the specified key then this method invalidates the I lease and grants the Q lease to the caller. Should there be a pending Q lease, the IQ-Server informs the requesting session to abort and release all its leases and restart per specifications of Section D.2.

   When the input token is not null, it identifies either an I or a Q lease held by the calling session. In this case, the IQ-Server uses the token to identify whether the caller owns an I or a Q lease on the referenced key. With the I lease, it upgrades the lease to a Q lease and provides the caller with a new token (and no value to return). With a Q lease, it simply returns the null value[7].

4. SaR(key, v_{new}, token) swaps the value of the key with v_{new} atomically only if its token identifies a valid Q lease on the key. The Q lease is valid if there is an existing token associated with the key in the reserved space that equals the provided input token. If the token is invalid, this command ignores v_{new} and returns without updating the key.

5. Quarantine-and-Register, QaReg(TID, keys): Acquires a Q lease on each of the specified keys and maintains a key=TID with its value set to the specified list of keys. Both occupy the memory of the reserved space. This command is used by the invalidate technique (see Section F.3) and implements the compatibility Table 6a. Should one of the acquired Q leases expire for TID, the IQ-Server deletes that key.

6. Release(key, token): Employs the key and token to identify a pending lease and removes it from the reserved

[6] The token is assigned to the `lease_token` meta-data property of the value `null`.

[7] This corresponds to a session performing multiple R-M-W operations on a single key by interleaving their read operations. The semantics of such an interleaved processing by one session using a single key is not defined.

space. If the input token is not valid (its value does not match the token associated with the key in the reserved spaced) then the release command is ignored.

7. Delete-and-Release, DaR(TID): Retrieves the set of keys associated with the specified TID in the reserve space. For each key, DaR deletes that key from the IQ-Server's regular memory, and releases its lease by deleting its entry from the reserved space. This command is used to implement the invalidate technique, see Section F.3.

8. IQ-δ(TID, k_i, v_i): The IQ-Server processes an incremental update (proposed by a write session identified using TID) by first looking up the reserve space using TID to identify all keys with a pending value proposed by this session. If k_i is in this set of keys, the IQ-Server retrieves its pending value from the regular in-memory store and applies the incremental change using v_i to this pending value. Otherwise, the IQ-Server tries to obtain a Q lease on the key. If an existing Q lease exists then it returns Abort to the IQ-Client. Otherwise, it generates a token to identify the lease and inserts it in the reserved space. Next, it associates k_i with TID using the reserved space. It retrieves the current value of k_i (it it exists), makes a copy of it to construct its pending version, applies the incremental update to this pending version using v_i, and stores it its memory by associating it with $f(k_i)$.

   In the above, the old value is the one prior to the write session TID. The pending value is the value proposed by the mid-flight write session TID. Both are stored as key-value pairs in the KVS. The key for the pending value is a deterministic function of the input key using special characters, $f(k_i)$.

   In contrast to QaRead, the IQ-Server does not return a token for the Q lease to the IQ-Client. It only returns a value indicating whether the lease was granted or not that may, in turn, cause the session to abort. When this command returns abort, it releases the Q leases of the session. (A future extension might be to also release the I leases of the session.)

9. Commit(TID): The IQ-Server looks up TID in the reserve space to identify the keys impacted by this write session that is committing. With invalidate, these keys are deleted from memory. With incremental update, the IQ-Server replaces the previous version of the values of each key with its pending version. Finally, the IQ-Server removes all the Q leases on the keys impacted by TID.

10. Abort(TID): The IQ-Server looks up TID in the reserve space to identify the keys impacted by this write session. It deletes all impacted keys and their pending values. Next, it deletes all the Q leases held by this session. (A future extension might be to also release the I leases of the session.) Note that this command is invoked by a session when it encounters an exception.

The server is able to support invalidate, refresh, and incremental update simultaneously because different commands are used to implement each technique. These commands implement the corresponding element of the compatibility table of Table 6.

Note that it is acceptable for the IQ-Server to grant a Q lease for a key to one session (S_2) that uses invalidate while another session (S_1) holds a Q lease on the same key and employs either the refresh or the incremental update technique. This is because S_2 will delete its referenced key (due to its use of invalidate), guaranteeing a serial schedule.

F.3 Trigger Implementation

As detailed in Section F.2, the server implements commands for invalidate, refresh, and δ. To implement invalidate and δ techniques using RDBMS triggers, we provide a dynamic link library that exposes the following IQ-Server commands: Quarantine-and-Register, QaReg(TID, keys) with invalidate, and the IQ version of the incremental commands of memcached with δ (IQ-Append and IQ-Prepend). Before executing an RDBMS write operation, the application first calls GenID() to obtain a unique identifier (TID).

With invalidate, a trigger computes the set of keys impacted by the proposed update to the RDB and invokes QaReg using the TID along with the set of keys. With δ, a trigger invokes a command such as IQ-Append using the provided TID, the impacted key, and its incremental change. The TID can be passed to the trigger using a session variable, e.g session context information with the Microsoft SQL Server [29] or a per-session package with Oracle 11g [22].

In our current implementation, the IQ-Server uses the KVS memory to maintain a key=TID whose value contains the list of keys identified by the invocation of the invalidate and δ IQ-Server commands. With invalidate, a session completes its execution by invoking DaR(TID) of the IQ-Server to delete the keys associated with the provided TID and release their Q leases. With δ, a session completes by invoking the Commit(TID) of the IQ-Server which identifies its impacted keys and updates their values to their pending version, releasing their Q leases.

F.4 Life Time of a Lease

When the Q lease on a key-value pair expires, the IQ-Twemcached server marks the key-value as expired. The next reference (by a get, set, IQget, IQset, and IQ-δ) for this key-value pair deletes it. Expiration of Q leases is undesirable as it results in stale data with invalidate, refresh, and δ. In the following, we show how expired leases allow a session to insert stale data with invalidate and δ. This section concludes with a discussion of the lease life time.

Figure 10 shows how an expired Q lease belonging to session S1 enables a read session S2 to insert a stale value in the KVS. In this schedule, S2 observes a KVS miss and is granted an I lease after the Q lease of S1 expires. S2's querying of the RDBMS races with S1's RDBMS transaction, employing snapshot isolation to compute a stale value that it inserts in the KVS in Step 2.5. This stale value remains in place after Step 1.5 as the IQ-Server ignores the value produced by Step 1.5. One may extend the current implementation to require the DaR command to delete its referenced key-value pairs with an expired lease.

Figure 11 shows how an expired Q lease causes refresh to produce stale data. Similar to the discussion with invalidate, Session 2 incurs a KVS means and races with the write session S1 with an expired Q lease to query the RDBMS using snap shot isolation. S2 inserts its computed value int he KVS at Step 2.5. Once S1 commits in Step 1.4,

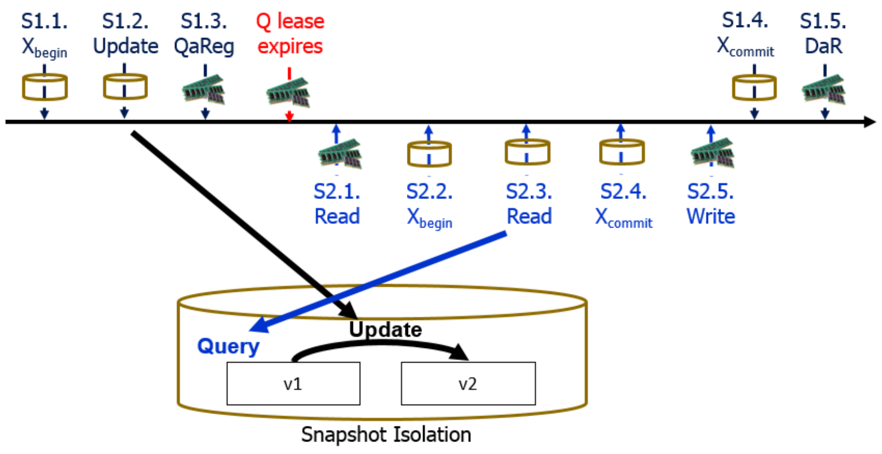


Figure 10: Expired leases cause invalidate to produce stale data.

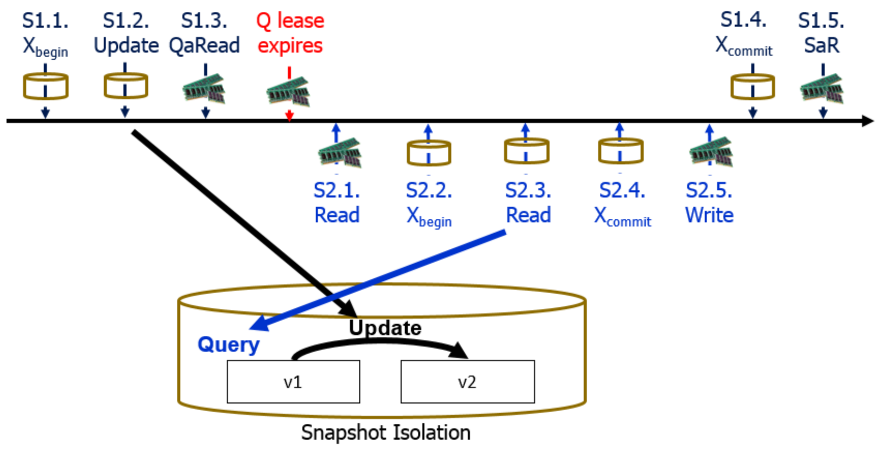


Figure 11: Expired leases cause refresh to produce stale data.

the KVS is left with a stale value. The SaR command of Step 1.5 is ignored because its Q lease is no longer valid. One may extend the current implementation of SaR to delete its referenced key when its Q lease is expired. This reduces the likelihood of another session computing a stale value.

One approach to solve the above is to set the life time of a lease to a high value. (Facebook suggests 10 seconds for its leases [14].) Moreover, the KVS may adjust the life time of leases by monitoring the delay from when it grants a lease to the time that a KVS write references the lease. One such a technique is detailed and evaluated in [15]. The basic idea is to maintain the maximum observed delay for a moving window of time, say 60 seconds, and multiply this by some inflation value (say 2) and use it as the life time of the lease. We refer the interested reader to [15] for details.

F.5 Repeated Lease Requests

Both the IQ-Client and the IQ-Server participate to enable a session, executing as a thread, to repeat the KVS operations of Table 4 for the same key-value pair multiple times sequentially. These repeated operations satisfy the following three properties. First, they either obtain the same lease or upgrade an existing I lease to a Q lease if possible. Second, the session observes its own updates. Third, they do not reduce system performance by penalizing read sessions that could benefit from an existing value (and serialized to have occurred prior to a write session).

In our implementation, the IQ-Client maintains the leases acquired by a session transparently. These are maintained in a hash table that associates each key with its lease tokens. This hash table is per IQ-Client (Whalin) connection to the IQ-Server and allocated to the session. Every time a session issues either an IQget or an IQset command for k_i, the IQ-Client looks up the hash table to determine the current lease token held on k_i. If one exists, it is provided the lease as a part of the command issued to the IQ-Server. The IQ-Server implements the four possibilities described in Section E.

A unique aspect of our implementation is how the IQ-Client and the IQ-Server facilitate invalidate when RDBMS triggers are used to delete key-value pairs impacted by an update to the RDBMS [20, 16]. As detailed in Section F.1, the IQ-Client provides a GenID() method that a session (say S_1) uses to generates a unique identifier for its RDBMS transaction, a TID. S_1 uses its TID when issuing its RDBMS transaction. The trigger uses this TID (e.g., provided to it by using the session object of the RDBMS) when invoking the QaReg() method of the KVS to invalidate a key-value pair. In response, the QaReg() method obtains a Q lease on the identified key. If this is successful, the IQ-Server maintains a hash table named *InProgressSessions* in the reserved space that associates the TID with the identified key. If this transaction invokes either the same or a different trigger to delete different keys then the IQ-Server associates these keys with the existing TID in the reserved space. Once the RDBMS returns, S_1 may perform an IQget on a key that was deleted by a trigger. The IQ-Client provides the TID used by S_1 as a part of this IQget command. The IQ-Server uses this TID to lookup the InProgressSessions hash table to retrieve a list of keys that were invalidated. If the key referenced by the IQget is found in this list then the IQget returns with a miss, enabling S_1 to query the RDBMS to observe its own update. Note that if a different session (say S_2) issues an IQget() for a key invalidated by

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 7: Four mixes of social networking actions with BG.

Members($\underline{userid}$, $username, pw, firstname, lastname, job, gender, jdate, ldate, address, email, tel, profileImage$, $thumbnailImage$, $\#PendingFriends$, $\#Friends$, $\#Resources$)

Friends($\underline{\widehat{inviterID}, \widehat{inviteeID}}, \widehat{status}$)

Resource($\underline{rid}$, $\widehat{creatorid}$, $\widehat{walluserid}$, $type, body, doc$)

Manipulation($\underline{mid}$, $modifierid$, $\widehat{rid}$, $\widehat{creatorid}$, $timestamp, type, content$)

Figure 12: Relational design of BG's database. The underlined attribute(s) denote the primary key of a table. Attributes with a hat denote the indexed attributes.

S_1, its TID would not match the TID entry of the InProgressSessions. The IQ-Server proceeds to look up the value of the referenced key. If no value is found then a miss is reported and the calling session must back off (due to the existing Q lease of S_1). If a value is found, the IQ-Server return it for use by S_2. In a serial schedule, S_2 would appear before S_1. This ensures enhanced performance by increasing the KVS hit rate and minimizing the number of queries to the RDBMS. In our experiments with a 10% update mix, this design provided almost a factor of two enhancement in the overall system throughput compared to a scheme that deletes the key invalidated by S_1 immediately.

Once S_1 issues its DaR command using its TID, the IQ-Server probes the InProgressSessions hash table with the TID to identify a list of key-value pairs. Next, it deletes these key-value pairs and releases their Q leases. It is possible to have a key with no value and a Q lease. In this case, the DaR (execution on the IQ-Server) simply releases the Q lease on this key. This enables read sessions to proceed to query the RDBMS, compute a value for the key, and insert the resulting key-value in the KVS.

G An Evaluation

This section employs the BG social networking benchmark [6] to evaluate the implementation of Section F. We start with a description of BG. Subsequently, Section G.3 presents performance results.

G.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 7. Each action is implemented as a session of the IQ framework using the physical data design of Figure 12. 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 G.3 reports on the amount of stale data using a CASQL system with and without I/Q leases.

G.2 Client Designs

As detailed in Section D.2, one may implement invalidate, refresh and δ clients by acquiring Q leases either prior to or during the RDBMS transaction. Table 8 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 10 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 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

```
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)
```

8a. 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)
```

8b. KVS operations during the RDBMS transaction

Table 8: Two alternative implementations of the Invite Friend session of Figure 5.a using QaRead and SaR commands.

Percentage of QaRead requests that back off computed across all write sessions						
System Load	Skewed, θ=0.1		70-20, θ=0.27		Uniform, θ=0.99	
	During	Prior	During	Prior	During	Prior
Low, 10 threads	0.75%	0.85%	0.18%	0.24%	0.01%	0.02%
Medium, 100 threads	2.54%	5.27%	1.21%	3.03%	0.18%	0.39%
High, 200 threads	3.63%	10.25%	2.2%	6.46%	0.45%	1.6%

Percentage of Aborted QaRead requests across all write sessions						
System Load	Skewed, θ=0.1		70-20, θ=0.27		Uniform, θ=0.99	
	During	Prior	During	Prior	During	Prior
Low, 10 threads	0.07%	0.71%	0.18%	0.21%	0.01%	0.02%
Medium, 100 threads	1.86%	1.77%	0.99%	0.98%	0.17%	0.17%
High, 200 threads	2.41%	2.39%	1.61%	1.52%	0.36%	0.38%

Table 9: Behavior of QaRead with alternative implementations as a function of system load with a different degree of skew in access pattern. BG is configured with 10K members, 100 friends and requests per member, using a cold KVS.

Table 5.a, 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.

Table 9 shows the percentage of QaRead requests that back off using the two alternatives with a different degree of skew (modeled using a Zipfian distribution with different exponent values [6]). A more uniform access pattern results in lower contention for data items, reducing the percentage of QaRead requests that back off. However, the percentage of requests that back off is higher when the KVS operations are performed prior to the RDBMS transaction. It is interesting to note that the percentage of aborted QaRead requests is approximately the same with both implementations, see Table 9.

Table 10 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

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 10: Average and maximum number of times an aborted session restarts (due to Q lease conflicts) when the KVS operations are performed either prior-to or during the RDBMS transaction that constitutes a session. BG is configured with 10K members, 100 socialites generating requests for 10 minutes, a Zipfian distribution with exponent 0.27 is used to select members who generate requests.

requests referencing 20% of data (Zipfian distribution with θ=0.27) [42]. 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 during 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.

G.3 Performance Results

This section compares the performance of two variants of Twemcache:

- Twemcache extended with read leases of [14], labeled Twemcache.
- Twemcache extended with I/Q leases using the implementation of Section F, labeled IQ-Twemcached.

Table 11 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 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

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 11: 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 12: SoAR of Twemcache and IQ-Twemcached with a 100% utilized CPU.

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 11 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 11. Table 12 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.

H 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 [28, 18, 35], optimistic or commit-time validation [5, 11, 23], and timestamps [34, 39]. See [8] for a survey of these algorithms and how one may combine them. IQ resembles locking and is least comparable to the timestamp protocols. The latter serialize transactions based on their order of arrival and not discussed further. 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 [28, 18, 35] 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, at most one session may acquire an I lease on a key-value pair while multiple transactions may acquire a S lease 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 Table 6.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, 11, 23] 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.

Leases are used to support cache consistency in distributed file systems [17], Web proxy caches [25, 43, 13], and content distribution networks [30]. The basic idea is as follows. The first access for a data item[8] by a client[9] causes the server[10] to grant the client a lease on the data item. There might be multiple lease holders on a data item. Once the lease expires, the client may send a lease renewal request along with other metadata (such as if-modified-since or a hash of the content of the data item) to the server. The server responds with a new lease (or a lease denial) along with either a not-modified or the updated object. Modifications to the data item during the validity of a lease cause the server to send invalidation messages to the clients holdings the lease. The server may not perform the update until it receives invalidate acknowledgments from all clients or the lease times out and expires. The IQ leases are similar in that they have a fixed life time and enhance availability of data once a KVS client holding a lease becomes unavailable. They are different as they have semantics such as Inhibit (I) and Quarantine (Q) that dictate their compatibility, see Figure 6. They prevent multiple threads from issuing the same query (key) to the database as it is a wasteful. Moreover, they prevent use of mechanisms such as snap-shot isolation and reader-writer race conditions from inserting stale data in the KVS. These concepts are absent with use of leases for distributed file systems, web proxy caches, and content

[8]It might be a file or a web page. It is comparable to a key-value pair.

[9]It might be a caching client of a distributed file server, a proxy cache server, a CDN node, or a CDN site. It is comparable to a KVS client.

[10]It might be a file server, a web server, or a CDN site. It is comparable to a KVS server.

distribution networks.

Two studies most relevant to our focus include TxCache [32] and the leases of [14]. We describe these in turn. TxCache [32] 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 [14], 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 4 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 4 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 C.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 G.3 and showed to produce stale data, see Table 11. Our proposed I lease is identical to leases of [14]. 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 [14] for an example. To maintain the replicated data consistent across data centers, one may use a technique such as parallel snapshot isolation [37], eventual consistency [40], per-record timeline consistency [12], causal+ [26] 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, 27, 10, 24, 1, 38]. 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 [27], computing changes and shipping them to the caches [2, 10], shipping log records [24], 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].

I 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 Table 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 [36, 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.

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

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

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

[11] 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.

[12] 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), August 2008.

[13] V. Duvvuri, P. J. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. *IEEE Trans. Knowl. Data Eng.*, 15(5):1266–1276, 2003.

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

[15] S. Ghandeharizadeh and J. Yap. Gumball: A Race Condition Prevention Technique for Cache Augmented SQL Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, 2012.

[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] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *SOSP*, pages 202–210, 1989.

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

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, pages 677–680. Morgan Kaufmann, 1993.

[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*, pages 981–992, 2008.

[22] Oracle Inc. Triggers, Packages, and Stored Procedures, http://docs.oracle.com/html/B16022_01/ch3.htm.

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

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

[25] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the 17th International Conference on Distributed Computing Systems, Baltimore, MD, USA, May 27-30, 1997*, pages 12–21, 1997.

[26] 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.

[27] 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.

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

[29] Microsoft Developer Network. Using Session Context Information, SQL Server 2008 R2, http://msdn.microsoft.com/en-us/library/ms189252.aspx.

[30] A. G. Ninan, P. Kulkarni, P. J. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *World Wide Web Conference*, 2002.

[31] 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.

[32] 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.

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

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

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

[36] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan. Feed Following: The Big Data Challenge in Social Applications. In *DBSocial*, pages 1–6, 2011.

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

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

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

[40] W. Vogels. Eventually Consistent. *Communications of the ACM, Vol. 52, No. 1*, pages 40–45, January 2009.

[41] 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.

[42] 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.

[43] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Trans. Knowl. Data Eng.*, 11(4):563–576, 1999.