TRANSPARENT CONSISTENCY IN CACHE AUGMENTED DATABASE MANAGEMENT SYSTEMS

by

Jason Yap

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)
May 2014

Abstract

Cache Augmented Database Management Systems (CADBMSs) enhance the performance of simple operations that exhibit a high read to write ratio, e.g., interactive social networking actions. They are realized by extending a data store such as a Relational Database Management Systems (RDBMS) with a Key Value Store (KVS). At the time of writing, memcached is a popular in-memory KVS in use by a number of Internet service providers such as Facebook, YouTube, Wikipedia and others.

A key insight of CADBMSs is that query result lookup using the KVS is significantly faster than query processing using the RDBMS. A challenge is how to maintain these query results consistent in the presence of updates to the RDBMS. Today's CADBMS solutions require a developer to design, implement, debug, and maintain software to address this challenge. This dissertation presents novel design decisions to realize physical data independence that hides the details of the storage structure (KVS or RDBMS) from applications and their developers. These designs simplify the complexity of application software to expedite their development life cycle.

The proposed designs can be categorized into two groups. The first group prevents race conditions that cause the KVS to produce stale data. Our primary contribution here is the IQ framework and its simple programming model that employs Inhibit (I) and Quarantine (Q) leases to provide strong consistency. We describe the compatibility of the leases when the KVS is either invalidated or refreshed in the presence of updates to the RDBMS.

The second group includes transparent techniques that invalidate the key-value pairs of the KVS in the presence of updates to the RDBMS. Our primary contribution is the SQL Query to Trigger translation (SQLTrig) technique. It provides the application developers with the SQL query language and observes the performance enhancements of a KVS without requiring additional software. It intercepts the queries issued by an application and authors software in the form of triggers that describes the template of the query. It registers these triggers with the RDBMS prior to inserting the query and its result set as a key-value pair in the KVS. An insert, delete, update command to the RDBMS invokes the trigger to compute the query (key) whose result set (value) has changed. The trigger invalidates this key-value pair from the KVS in a transactional manner.

We describe a software prototype that embodies both the SQLTrig technique and the IQ framework. We use a social networking benchmark to compare this prototype with a non-transparent consistency technique where the developer extends the application

software to maintain key-value pairs consistent with the relational data. Obtained results demonstrate that both provide comparable performance.

Acknowledgements

I would like to extend my deepest gratitude to the many people who have helped and supported me over the years in the road towards completing my Ph.D.

First and foremost, I would to thank my advisor, Shahram Ghandeharizadeh, for his wisdom and guidance. I have learned a great deal throughout this whole process and I thank him for sharing his time, experience, and advice so freely.

I would also like to thank the members of my guidance committee, Nenad Medvidović, Leana Golubchik, William G. J. Halfond, and François Bar. Their feedback and advice was greatly beneficial in improving my research. My time at the Computer Science department of the University of Southern California has been a pleasure and has provided me with a great environment that fostered my research and introduced me to some very brilliant people.

I offer my gratitude to Oracle Inc. for supporting my research with an un-restricted cash gift. Our collaboration produced many fruitful ideas and helped spur and shape my research direction. I would particularly like to thank Dieter Gawlick for spearheading the effort as well as Srinivas Vemuri and Lakshminarayanan Chidambaran for their technical insight into Oracle.

I am greatly appreciative of Sumita Barahmand from the Database Laboratory for her help and friendship over the years. Always willing to help with everything from practicing presentations to discussing ideas, her help was a great boon during my time at the lab. I would also like to thank other past and present researchers of the Database Lab, Nasser Alrayes, Reihane Boghrati, Litao Deng, Jorge Gonzalez, Connor Gorman, Showvick Kalra, Lakshmy Mohanan, Neeraj Narang. All of whom have collaborated with me or helped me in many ways and made the lab a better place.

Last but certainly not least, I would like to thank my family. I would like to thank my parents for their love and support and always encouraging me to pursue my ambitions. My late brother, Jeffrey, will forever be missed. He was one of my greatest sources of awe and inspiration while I was growing up and I am thankful that he encouraged me to enter the field of computing in the first place. I would also like to extend my deepest gratitude to my uncle and aunt, David and Veronica, for very graciously providing a place to stay and treating me so well during my time here.

Thank you to all my friends, family, collaborators, colleagues and professors. I am forever indebted to you for all the help I have received and know that this dissertation would not have been possible were it not for all of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the era of no "one-size-fits-all", organizations extend a database management system (DBMS) with a key-value store (KVS) to enhance the velocity of simple operations that either read or update a very small amount of big data. The resulting cache augmented database management system, CADBMS [38], targets applications that perform simple operations and exhibit a high read to write ratio. An example application is social networking with interactive actions such as browse a profile, view a friend's resource (say a picture) and post a comment on it, generate a friend request and accept one, and others [12]. According to [14], 92% of user activities in social networking applications are read-only browse operations. A popular in-memory KVS is memcached [55], in use by well known Internet destinations such as YouTube and Wikipedia. Its simple interface provides put, get, and delete of key-value pairs computed using data in the DBMS.

A CADBMS deployment assumes query result look up is both faster and more efficient[1] than executing the query. A developer utilizes a CADBMS by identifying code segments in an application that manipulate read intensive data, e.g., the code to compute the profile page of a user. Execution of this code segment with an input, user-id, produces an output, the HTML fragment pertaining to the user profile. This output is termed the *value* and identified using a unique *key*. This key is typically constructed using the input to the code segment, e.g., "Profile"+user-id. Next, the developer extends the code to look up the key prior to executing the code segment with its input. If the KVS returns the value then the value is used without executing the code segment. Otherwise, the code segment executes and the resulting key-value pair is inserted in the KVS for use by future references. A code segment

[1] Performs less wasteful work [45] .

may execute several queries and perform arbitrarily complex application logic. As longs as it is deterministic, a key-value pair identifies a unique input to this code segment and its unique output.

To demonstrate performance enhancements obtained using CADBMS, Figure 1.1 shows the throughput[2] of several different systems with a social networking benchmark named BG [12]. The workload consists of a mix of aforementioned social networking actions with 1% of actions updating the database. Target systems include:

1. SQL-X: A commercial[3] relational database management system (RDBMS).

2. A Client-Server (CS) CADBMS consisting of SQL-X extended with memcached [55] server version 1.4.2 (64 bit). In the presence of updates to the RDBMS, two different approaches to maintain the cached key-value pairs consistent were considered: Either using application software or RDBMS triggers, see Section 5.1 for details.

3. A Shared Address Space (SAS)[4] CADBMS with application consistency: SQL-X extended with Ehcache [81] and application software to maintain key-value pairs consistent.

4. MongoDB, a document store, representing a NoSQL solution. See [24] for a taxonomy of NoSQL systems.

While SQL-X struggles to process 224 actions per second, once extended with memcached, it can process almost 6,000 actions per second. Ehcache enhances the performance of SQL-X 70 folds to provide a throughput of more than 15,800 actions per second. Though MongoDB outperforms SQL-X, its single node performance is lower than all CADBMSs. One may incorporate the principles of a CADBMS in MongoDB (or any NoSQL solution) to enhance its single node performance. This complements the ability of these systems to scale out.

A challenge of using a CADBMS is how to maintain the key-value pairs consistent with both incremental and bulk updates to the database. One may taxonomize today's approaches

[2]All performance numbers reported here employ a BG social networking database consisting of 10,000 users with 100 friends per user and 100 resources per user. The number of threads used to generate the workload is 100 and no service level agreements, i.e., no values for tolerable response time (β) and the percentage of requests that must observe this response time (α). See [12] for details.

[3]Due to licensing restrictions, the identity of this commercial DBMS is not disclosed.

[4]Chapter 3 details the client-server and shared address space CADBMS architectures.

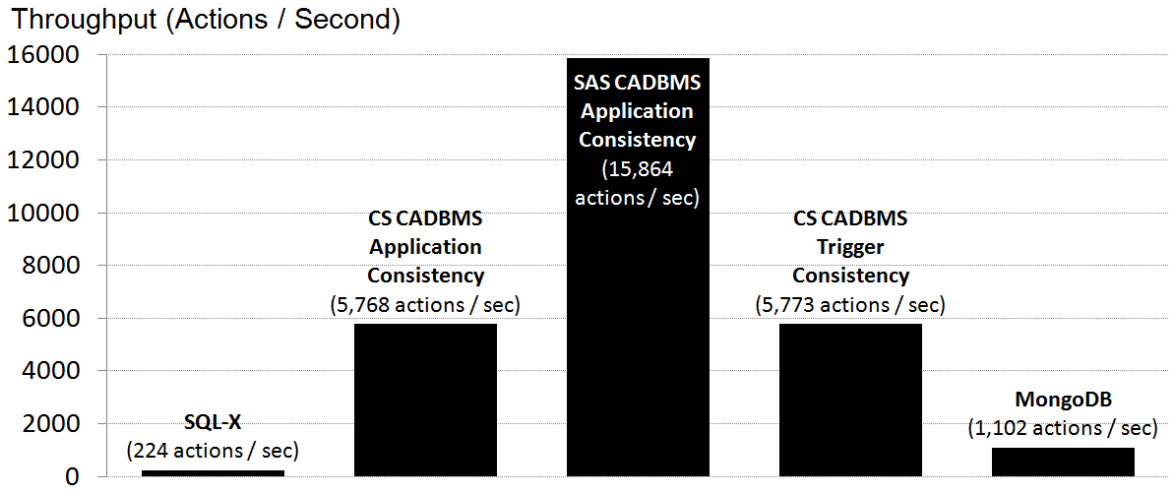


Figure 1.1: Throughput of 5 different systems with BG benchmark.

into time to live and invalidation techniques. With the former, the developer extends the application to provide a Time To Live, TTL, for each key-value pair inserted in KVS. The KVS invalidates a key-value pair once its TTL expires, causing a subsequent reference for it to observe a miss, re-compute the value and insert it in the KVS. Figure 1.2 shows the throughput and amount of stale data observed with a client-server CADBMS (SQL-X using memcached with Whalin client) with different TTL values. Its experimental setting is identical to the one shown in Figure 1.1 except for the use of TTL. The x-axis of Figure 1.2 shows different TTL values, ranging from 30 seconds to 5 minutes. As TTL increases, the throughput of the CADBMS is enhanced due to a higher KVS hit rate. It also causes a larger percentage of reads to observe stale data because the key-value pairs are stale and inconsistent with their tabular representation in SQL-X. As a comparison, the figure also shows the throughput and the amount of stale data produced by the alternative techniques that invalidate data.

One may implement an invalidation based technique in either the application or the DBMS. With the former, the developer identifies code segments of the application that update the database and extends them to either invalidate, refresh, or propagate the change to the key-value pairs in the KVS. With the latter, the database administrator authors triggers (notification mechanisms) to update the KVS. These two techniques are implemented using SQL-X and Figure 1.2 shows they provide throughput comparable to TTL of 5 minutes with a significantly (several orders of magnitude) lower amount of stale data. The two techniques provide comparable performance, see the two CS CADBMS bars in Figure 1.1. Applica-

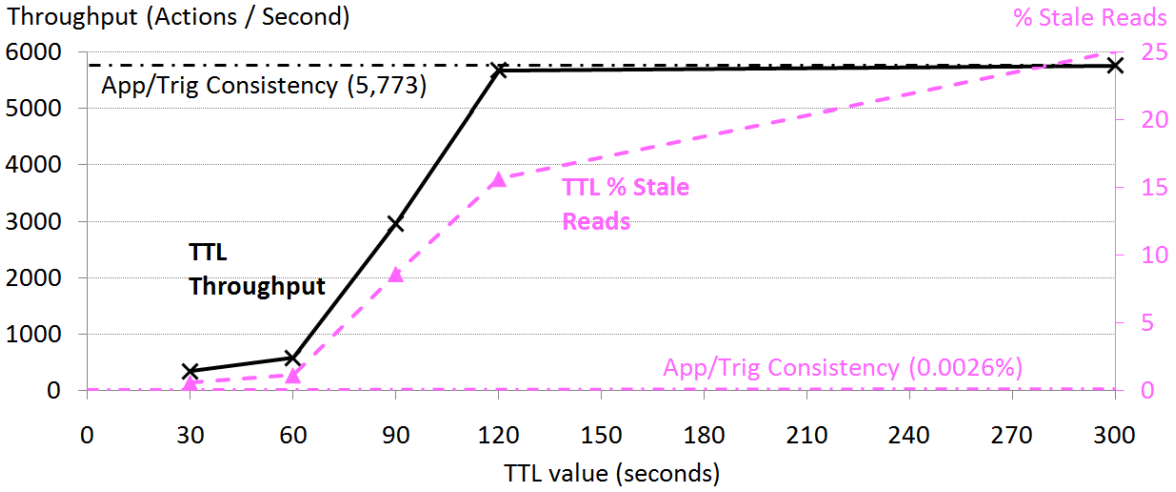
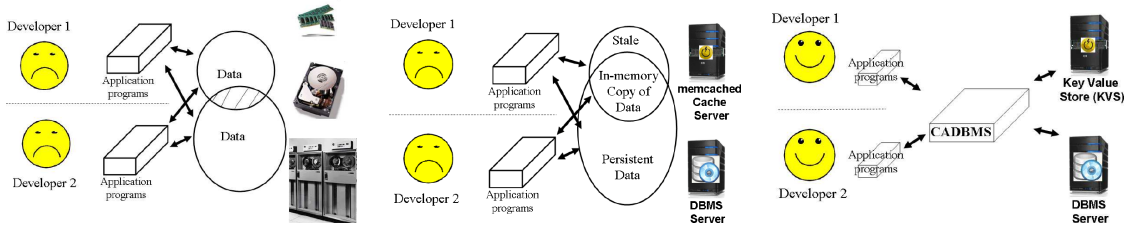


Figure 1.2: Throughput and % stale reads observed with Time-To-Live(TTL) based consistency as a function of the TTL value. Application and Trigger based consistency are also shown for comparison.

tion and trigger consistency techniques produce some stale data because they suffer from race conditions between writes to SQL-X and memcached. These race conditions are further elaborated on in Chapter 4.

1.1 Extending CADBMS Technology

Today's CADBMSs are in their infancy and resemble the data intensive applications of 1970s that existed at the dawn of DBMSs, see Figure 1.3. They lack *physical data independence* that hides the details of the storage structure from user applications. In essence, when a CADBMS employs a transactional DBMS, the application developer authors software to maintain the normalized tables of the DBMS (magnetic disk of 1970s) consistent with the un-normalized key-value pairs stored in a KVS (main memory of 1970s). Physical data independence is desirable because it enables a CADBMS to hide details of DBMS and KVS from the application developer to provide functionalities such as transparent cache consistency (maintaining the content of KVS and DBMS consistent with one another seamlessly), dynamically adjust the content of KVS to enhance overall system performance, and support different forms of consistency ranging from weak to strong. This is beneficial and superior to today's state of the art for two reasons. First, it reduces the complexity of application



1.3.a) Prior to RDBMSs. 1.3.b) CADBMSs today. 1.3.c) Future CADBMSs.

Figure 1.3: Decades of database technology.

software and expedites software development life cycle, empowering application developers to introduce features more rapidly at reduced costs.

Second, it enhances robustness of the deployed system by preventing software assumptions that compromise availability of the data. An example comes from Facebook where, due to dependence of data on different physical forms of storage, a software component was authored with the assumption that configuration data from the cache is obsolete and erroneous while its counterpart in the database is correct. Every time this component observed erroneous data from the cache, it would query the database to refresh the cache with correct data. On September 23, 2010, erroneous configuration data was inserted into the database, causing this component to overwhelm the DBMS with repeated queries for the correct data [48]. Physical data independence would have avoided both the flawed assumption and the resulting 2.5 hour down time with a price tag of millions of dollars.

1.2 Reader's Guide

The primary contribution of this dissertation are two frameworks to realize physical data independence in CADBMSs, see Figure 1.3.c. The first, named SQLTrig, is a transparent technique that utilizes the structure of the SQL language to author triggers on the fly to maintain the KVS consistent in the presence of updates to the RDBMS. The second, named the IQ framework, prevents race conditions between the KVS and the RDBMS that cause them to reflect a different value for a data item, i.e., insert stale data in the KVS.

This dissertation is organized as follows. Chapter 2 presents the current state of the art in transparent caches and alternative ways to maintain consistency between the KVS and the RDBMS. Chapter 3 describes two different architectures to realize a CADBMS. Chapter 4

describes the Gumball technique and the IQ framework as two alternatives to prevent race conditions that insert stale data in the KVS. The IQ framework is a successor to the Gumball technique and handles a wider variety of race conditions including those attributed to the use of MVCC [16] and snapshot isolation [77]. Chapter 5 presents alternative techniques to maintain the KVS consistent in the presence of updates to the RDBMS, introducing SQLTrig as the final decision. Chapter 6 presents the correctness of a system that utilizes SQLTrig in combination with the IQ framework to cache queries and their result sets. We present our conclusions and future research directions in Chapter 7.

Chapter 2

Related Work

Cache augmented RDBMSs have been an active area of research since 1990s [2, 3, 5, 4, 17, 27, 32, 33, 50, 52, 43, 47, 67, 87]. The term CADBMS is used to refer to a subset with the following characteristic. First, the cache must support simple insert, get and delete operations [8, 67]. There exist complex caches with the ability to process SQL queries, e.g., TimesTen [80], DBProxy [3], DBCache [17, 18], Cache Tables [2], MTCache [52], Ferdinand [34]. While these fall beyond the focus of SQLTrig, these systems may use the principles outlined by SQLTrig to minimize the amount of software required to maintain their cache consistent with a RDBMS.

Second, SQLTrig is designed for middle-tier [47, 27, 87, 33, 32, 51, 5, 4, 67, 43] caches at the same abstraction as the RDBMS where security and privacy of content is guaranteed by the application and its infrastructure. It does not apply to proxy caches [43, 23, 32] that are external to the application.

Early transparent cache consistency techniques invalidated cached entries at the granularity of either table change or combination of table and column change [4]. These are suitable with web sites that disseminate information (e.g., stock market ticker prices [51], results of Olympic evens [27]) where a table is the basis of a handful of cache entries. They become inefficient with applications such as social networking where each row of a table is the basis of a different cached entry and there are many (billions of) rows and corresponding cache entries. With these techniques, an update to a row would invalidate many (billions of) cached key-value pairs even though only a single key-value pair should be invalidated.

TxCache [67] is a transparent caching framework that supports transactions with snap shot isolation. It is designed for RDBMSs that support multi-version concurrency con-

trol [16], e.g., PostgreSQL, and extends them to produce invalidation tags in the presence of updates. A generated tag is based on a query whose results is used to generate a cached key-value pair. The tag is for one attribute value of a table (TABLEKEY). This works when the workload of an application consists of simple exact-match selection predicates. Details of how this technique works for queries with join predicates are not clear. SQLTrig can be adapted to support such queries in TxCache. Moreover, SQLTrig can be used with all SQL RDBMSs that support triggers because it does not either modify or require pre-specified concurrency control technique from the RDBMS.

CacheGenie [43] employs an Object-Relational Mapping(ORM) framework such as Django to generate the SQL queries, object instances stored in the cache, and DBMS triggers to invalidate cached objects. It can perform this for a subset of query patterns generated by ORM. The difference between SQLTrig and CacheGenie are as follows. First, SQLTrig generates triggers based on the issued SQL queries and not an ORM description. Thus, SQLTrig is applicable for use with both ORM and non-ORM frameworks. Second, while CacheGenie caches the results of a query, SQLTrig supports both query result and semi-structured data caching.

2.1 Consistency

The IQ framework embodies a concurrency control algorithm that guarantees serial schedule of sessions. There exists a vast number of concurrency control algorithms, most of which are based on either locking [56, 39, 71], optimistic or commit-time validation [11, 25, 49], and timestamps [70, 82]. See [15] for a survey of these algorithms and how one may combine them. IQ most closely resembles locking and least similar to the timestamp protocols (not discussed further) because it produces a serial schedule based on how sessions compete to acquire leases instead of the order in which they are issued to the system. 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 transaction commit point when it applies its changes to the KVS and releases its leases. IQ is different than lock based protocols because it is non-blocking and deadlock free.

IQ is also similar to the optimistic concurrency control (OCC) algorithm [11, 25, 49] technique as it has a read and a write phase. Its write phase occurs after the RDBMS transac-

tion commit and, similar to the write phase of OCC, succeeds always. 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 its IQ leases.

Two studies most relevant to our focus include TxCache [67] and the leases of [62]. We describe these in turn. TxCache [67] 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 technique and does not propose use of leases to provide strong consistency.

In [62], 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. We named Facebook's lease as the read lease and detailed it in Section 4.2.2. It is implemented in the Twitter memcached version that we evaluated in Section 4.2.5 and showed to produce stale data, see Table 4.10. Our proposed I lease is identical to leases of [62]. 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 IQ 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 CADBMS solution in different data centers with replicated data, see [62] for an example. To maintain the replicated data consistent, one may use a technique such as parallel snapshot isolation [77], eventual consistency [85], per-record timeline consistency [28], causal+ [53] 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. Our objective is to satisfy the freshness property and provide strong consistency with no modification to the RDBMS software.

There are mid-tier caches that process SQL queries [3, 54, 18, 52, 2, 80]. 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 [54], computing changes and shipping them to the caches [3, 18], shipping log records [52], and invalidation of the impacted rows [2]. Our target CADBMS architecture 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 4.4. Thus, the KVS does not incur the overhead of query processing estimated at a high percentage of useful work performed by today's RDBMSs [45].

2.2 Materialized Views and Key-Value Pairs

A key-value pair used in CADBMSs shares similarities with a materialized view, MV, of a RDBMS. Both involve maintaining a separate physical copy of the data in order to enhance the velocity of data intensive applications. This section is presented as a series of questions to help describe each approach and distinguish between the two.

What is a view?

A view is a virtual table defined using an expression that references other tables in a relational database management system (RDBMS). It is re-computed every time a query references the view. A view might be authored using SQL, relational algebra [41], datalog and others [21]. This writing assumes SQL.

What is a materialized view?

A materialized view (MV) stores the tuples of the view in the database. One may construct index structures on the materialized view. Hence, accesses to the view are much faster than re-computing it. Typically, a database administrator (DBA) analyzes the workload of an application to authors MVs and their indexes. For an example with data warehousing queries see [74]. This study shows MVs enhance the performance of row-stores significantly. Selecting which virtual views to materialize, the view selection problem, has been studied extensively [73, 41, 42].

It is time consuming for a RDBMS to materialize a view and its indexes. Hence, in the presence of updates to the base tables referenced by a MV, it is not efficient to drop the MV. Instead, MVs are maintained up to date incrementally [41, 72, 59]. This approach computes changes to the MV and applies them to the MV to bring it up to date.

A query optimizer may employ a MV to process SQL queries that do not reference it explicitly [75, 59]. Moreover, a physical database design adviser may recommend index structures on a MV as it is a table [1, 20].

What is a Key-Value Store (KVS)?

A KVS maintains key-value pairs consisting of a unique identifier (key) associated with some arbitrary data (value). It provides a simple interface such as put, get, and delete to store, retrieve, and delete key-value pairs. It provides little or no ability to interpret its value with no query mechanism for the content of the values [24, 79]. A popular KVS is memcached in use by many popular Internet destinations such as YouTube and Wikipedia.

What is a Cache Augmented DBMS (CADBMS)?

Cache Augmented Database Management System, CADBMS, systems are an important class of distributed systems, targeting applications with a high read to write ratios. These systems augment a RDBMS with a KVS to enhance overall velocity of operations that retrieve and process a very small amount of the entire data set [2, 3, 5, 4, 17, 27, 32, 33, 50, 52, 43, 47, 67, 87]. They may materialize either the results of a query (key=query string, value=result set computed by the RDBMS) or a code segment (key=unique identifier for the code segment constructed using its input, value=output of the code segment) as key-value pairs in the KVS. This enhances performance because a cache look up is much faster than executing either a SQL query or a code segment that issues several queries. In the presence of updates to the tabular data, a CADBMS solution may maintain the cached key-value pairs consistent transparently [5, 4, 27, 32, 33, 43, 67].

How are materialized views similar to cached key-value pairs?

Both MV and key-value pairs store a separate physical copy of the tabular data. This copy must be maintained consistent with the base tables. The RDBMS automatically maintains

MVs consistent and serialize transactions to provide ACID properties, e.g., by using the REFRESH ON COMMIT. Similarly, transparent caching techniques maintain key-value pairs consistent in the presence of updates to the RDBMS. For example, TxCache [67] implements snap-shot isolation and one may configure SQLTrig to implement serial schedules.

Both MVs and key-value pairs might be used to enhance velocity of data retrieval by approximating the final answers of a posed query. For example, an application may utilize REFRESH ON DEMAND option when authoring a MV and update it periodically. Queries processed using such views may observe stale data. Similarly, a CADBMS system may incrementally update key-value pairs and cause the application to observe either stale data [51] or suffer from dirty reads [43].

How are materialized views different than cached key-value pairs?

MVs and key-value pairs are suitable for different application classes. MVs enhance performance of decision support applications and their On-Line Analytical Processing (OLAP). KVS and key-value pairs enhance performance of queries that read a very small amount of entire dataset repeatedly. Thus, one is not a substitute for the other. This is elaborated on below.

SQL queries used to compute a MV typically retrieve many rows. It is not uncommon to find index structures on a MV to expedite processing of SQL queries that reference it. In contrast, a key-value pair corresponds to an SQL query (or a code segment) that is very selective (outputs a few values), e.g., retrieve the profile information of a member of a social networking site given the users login and password. A CADBMS enhances the performance of interactive operations when its key-value pairs are accessed far more frequently than they are updated. This is because a key-value look up is faster than processing SQL queries [37].

With a CADBMS, there may exist millions (if not billions) of key-value pairs pertaining to different instances of a simple SQL query whose results are cached as key-value pairs in the KVS. For example, with a social networking application, each SQL query issued on behalf of a member to retrieve her profile might be a key-value pair in the KVS. In contrast, there exists a few (in the order of tens of) MVs authored by a database designer to enhance overall system performance based on a known or expected workload.

SQL queries that are the basis of a key-value pair with a relational CADBMS are much faster to execute than those that are the basis of a MV with a RDBMS. This explains why RDBMSs maintain MVs instead by incrementally updating them while CADBMS systems

invalidate key-value pairs by deleting and re-computing them.

Finally, a MV is typically created by a human and crafted to specifically meet the expected needs for an OLAP workload. The workload should be known in advance in order to use MVs effectively. On the other hand, a CADBMS with a transparent cache generates key-value pairs dynamically as a workload executes. It does not require advanced knowledge of the workload. (When a CADBMS in employed non-transparently, a human participates to identify code segments whose results should be cached.)

Can MVs and key-value pairs co-exist?

MVs and key-value pairs are implemented by different components and may co-exist. While a RDBMS implements MVs, a KVS implements key-value pairs. Thus, one may use the RDBMS of a CADBMS to author MVs to enhance processing of OLAP queries. And use its KVS to cache the result of OLAP queries in order to expedite the processing of those issued repeatedly. A query optimizer extended with a cache manager for data warehouses and data marts was explored in [75]. Extensions of these ideas to a CADBMS is a future research direction.

Is it possible to use MVs as a substitute for key-value pairs?

MVs are not a substitute for key-value pairs. This is shown using the view profile action of the BG benchmark [12]. This action is a simple SQL query that retrieves profile information (a row) of a Member table with 10,000 rows (members). BG issues 100,000 view profile requests in turn. Each request references a member. Using a commercial RDBMS, the average execution time of this query is 2.5 milliseconds. If one defines 10,000 MVs (one for each query) and executes the same workload, the average execution time of each query increases to 6 milliseconds. Using a CADBMS with a cold cache, the average execution of the query is 1.5 milliseconds. A warm cache with 10,000 key-value pairs (one per query) reduces this time to 0.3 milliseconds.

Chapter 3

System Architectures

3.1 Client Server Architecture

Figure 3.1 shows the architecture of a typical cache augmented Relational Database Management System (RDBMS). The application communicates with the RDBMS through a client (eg. JDBC) and similarly to the cache through its own client, typically through a TCP or UDP connection. The cache layer can be comprised of multiple cache nodes, potentially hundreds of nodes, representing a large memory space.

One example application that utilizes this architecture is a social networking site, such as Facebook [76]. The RDBMS serves as a persistent data repository that can be queried or modified under the relational model. When a user makes a request for their profile page, the application issues one or more SQL queries to the database and processes their results in order to generate a HTML document that is returned to the user.

This on-the-fly generation of data to satisfy requests is referred to as *dynamic* web as opposed to *static* web, where data does not change. While static web can easily be cached to improve performance, dynamic web requires a more careful approach. With dynamic web, the data can change between every request producing a different result every time. In such cases, caching will yield no benefit and could actually slow the system down due to overhead.

However, if the underlying data changes infrequently (i.e. updates are rare compared to reads), then the application will produce the same final HTML document every time the same request is made. Instead of requiring the application to issue multiple queries to the database every time, a cache can be used to store this final HTML document and serve the

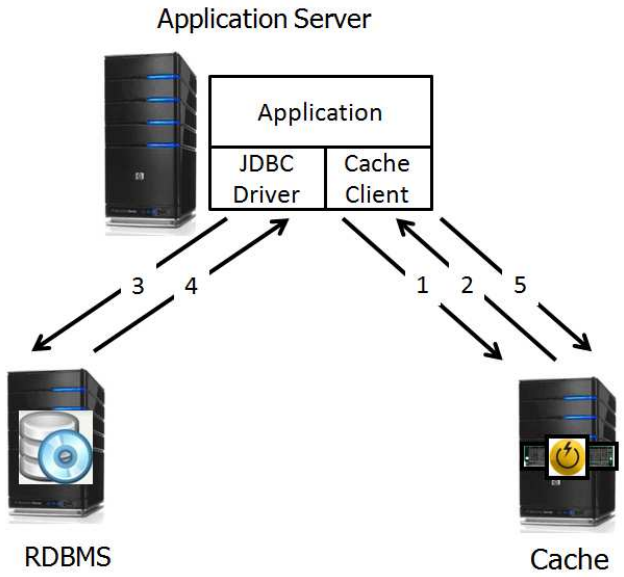


Figure 3.1: Cache augmented RDBMS architecture.

request.

In order to do this, the application is modified to be aware of the cache. A developer identifies a fusion code segment CS_{fuse} in the application that consumes some input to produce an output. CS_{fuse} might be complex, consisting of arbitrary loop and branch programming constructs. Each branch may execute a different sequence of SQL queries depending on the results of an earlier query in the sequence. The final output of CS_{fuse} is a value. This is associated with a developer specified logical key that might be constructed using the value of one or more of the input parameters. The developer extends CS_{fuse} as follows (Steps 1 to 5 correspond with arrows 1 to 5 in Figure 3.1):

1. Look up the cache using the key corresponding to the code segment, k_i.
2. If the data, d_i, is found in the cache, skip to Step 6. Otherwise, continue with Steps 3 - 5.
3. Issue SQL queries based on the application logic in CS_{fuse}.
4. RDBMS returns the query results. The application constructs the HTML page using the results.
5. Store the page as d_i into the cache under the key k_i.

6. Return the data d_i as the resulting HTML page.

By utilizing a cache, the system can skip the multiple queries and application processing in steps 3 - 4 every time the required HTML page is found in the cache. This helps reduce the network traffic and the number of round trips required as well as the amount of work that needs to be done on both the application server and the RDBMS. However, when the cache does not contain the required k_i- d_i pair, the system incurs the additional communication overhead of checking the cache and populating it with the constructed result. This scenario occurs when: (i) the cache is brought on-line and starts in an empty state, (ii) a previously stored k_i-d_i pair was evicted to free up memory on a heavily utilized cache, or (iii) a previously stored k_i-d_i pair was invalidated because the cached copy no longer matched the data on the RDBMS.

Since the cache holds an independent copy of the data, whenever a change occurs to the data in the RDBMS, the copy in the cache becomes out-of-sync. In order to ensure that the cache produces data that is consistent with the database, the application developer is required to author code which explicitly maintains the consistency of the data in the cache. Every time an update occurs to the RDBMS, the copy of the data in the cache which was affected by this change has to be invalidated. The problem with this approach is that the developer is required to have sufficient understanding of the application logic in order to correctly author the invalidation code. In larger code bases, this approach is error-prone and can lead to bugs which affect the entire system. An example of this was Facebook's outage in 2010 which was caused by an error in their consistency checking software [48]. When the application logic changes, the invalidation code has to be re-authored, thus exposing the software to more possible bugs. Furthermore, changes to the data made outside of the expected code path (eg. updates made directly to the database) might not be captured by the invalidation logic and result in a cache serving stale data.

The transparent caching techniques, described in Chapter 5, rely on two mechanisms, (i) cues from the application to indicate the data dependencies of cached k_i-d_i pairs and (ii) notification from the RDBMS when a relevant change is detected. The cues that can be used are readily available: the SQL queries which are being issued by the application to the RDBMS. By intercepting these queries through a RDBMS client wrapper, the system automatically identifies these queries without additional changes required to the application software. The framework keeps track of these dependencies in order to determine which cache entries are affected whenever a change is detected in the database. There are two

different mechanisms examined by this study that can be used to detect these changes, Query Change Notification and Triggers. Further detail on the workings and differences between the two mechanisms are provided in Chapter 5. In both cases, a change is detected at the RDBMS and a notification is issued to the cache. The cache then processes the notification to determine which cache entries were affected. One thing to note is that it is important that the notifications and corresponding invalidations are as fine-grained as possible, in order to avoid unnecessary invalidations of unaffected cached data. Excessive invalidations will lower the cache hit rate and thus, lead to poor system performance.

The following describes two approaches to realize physical data independence in CADBMSs:

1. Migrate CADBMS into a mature database technology such as a RDBMS (or a NoSQL such as Couchbase [30]), implementing a KVS transparently beneath a high level language such as SQL (or a programmable interface using JSON-like representation of data).

2. Use an Object-Relational Mapping (ORM) framework such as Hibernate or Django to embody a CADBMSs as a middleware.

One may migrate a CADBMS into an RDBMS at different abstraction levels. It might be implemented by the query optimizer and execution engine of a RDBMS [67]. Alternatively, it might be implemented by the client component of a RDBMS such as its JDBC driver [37]. Below is a description of the latter.

One may extend the JDBC driver of a RDBMS to intercept queries and looks up their (serialized) result set in the cache. If a value is found then it is deserialized and returned to the application. Otherwise, the query is executed using the RDBMS, returning the result set to the application. To maintain the KVS and the RDBMS consistent *transparently*, the extended JDBC driver may utilize query change notification mechanism of a RDBMSs by registering queries that are the basis of a cached query result set [37], (key=query, value=result set). Query change mechanism is a recent feature supported by Oracle 11g and Microsoft SQL Server 2005 and 2008 editions. When an update changes the state of the database, the RDBMS notifies the KVS of those queries whose results have changed. The KVS maintains a mapping of queries to key-value pairs and either invalidates or refreshes the impacted key-value pairs. This technique is not viable today because the change notification mechanism of RDBMSs is in its infancy and suffers from the following limitations. First, they support a limited class of queries. None support simple aggregate queries (e.g., count Joe's number

of friends) that are central to diverse applications. Second, the time to register a query is significant and slows down updates so dramatically (tens of seconds) that it is difficult to argue the action is interactive [37]. To address this limitation for those applications that tolerate stale data, the CADBMS may perform RDBMS updates asynchronously. This may cause a transaction to not observe its own update and be unacceptable for those applications that demand consistent reads. For change notification to be a building block of physical data independence, it must evolve to support a larger class of SQL queries while registering queries (at bursts of thousands per second) and processing RDBMS updates quickly.

With the second approach, the CADBMS will be a pass through entity that directs SQL queries either for execution to its RDBMS component or look up in the KVS. As an example, CacheGenie [43] consumes high-level description of Django (an ORM framework) objects to generate SQL queries, object instances stored the KVS (memcached), and RDBMS triggers to either propagate RDBMS updates to key-value pairs or invalidate them. CacheGenie frees the developer from managing the KVS or maintaining it consistent with the RDBMS. It observes a factor of 2.5 improvement in throughput for read-mostly workloads in Pinax (when compared with the RDBMS). While it is not clear whether this approach is feasible with all object descriptions, it is a promising approach toward realizing physical data independence.

Once an approach to realize physical data independence is identified, one may formalize the interaction between the RDBMS and the KVS to identify functionalities of a CADBMS, consistency and availability of data, and administrative tools to maintain a deployment. To elaborate, consider the first approach to physical data independence assuming industrial strength RDBMSs mature to support query change notification efficiently. The high level language of this approach might be SQL. The resulting CADBMS may use its RDBMS component to support materialized views to improve performance of certain queries. It may store the result of queries that reference these views as key-value pairs in the KVS, expediting their subsequent reference.

The SQLTrig framework solves these problems by modifying the architecture to abstract away the cache as a distinct entity that needs to be separately maintained. The developer can assume that they are interacting with a unified data repository as shown in Figure 3.2. No custom code needs to be authored to maintain the consistency of the cache as it will be handled automatically by the framework. Underneath, SQLTrig realizes the physical architecture by utilizing transparent caching techniques, Figure 3.3.

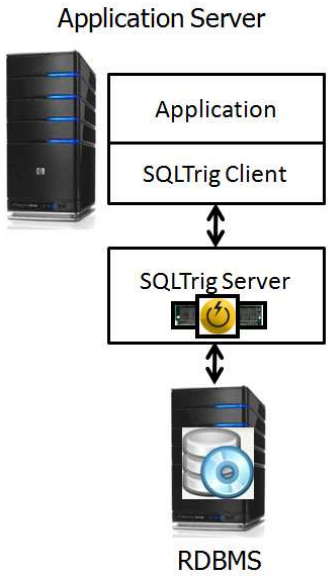


Figure 3.2: Conceptual Architecture as seen by the developer.

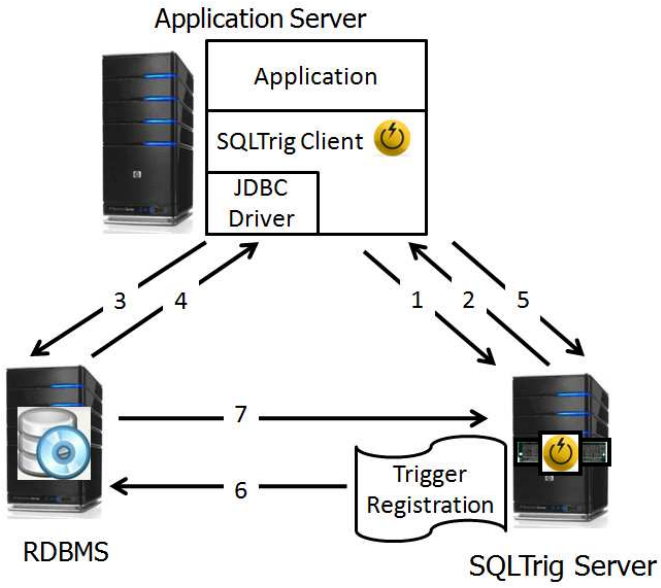


Figure 3.3: Physical Architecture as implemented by the SQLTrig framework.

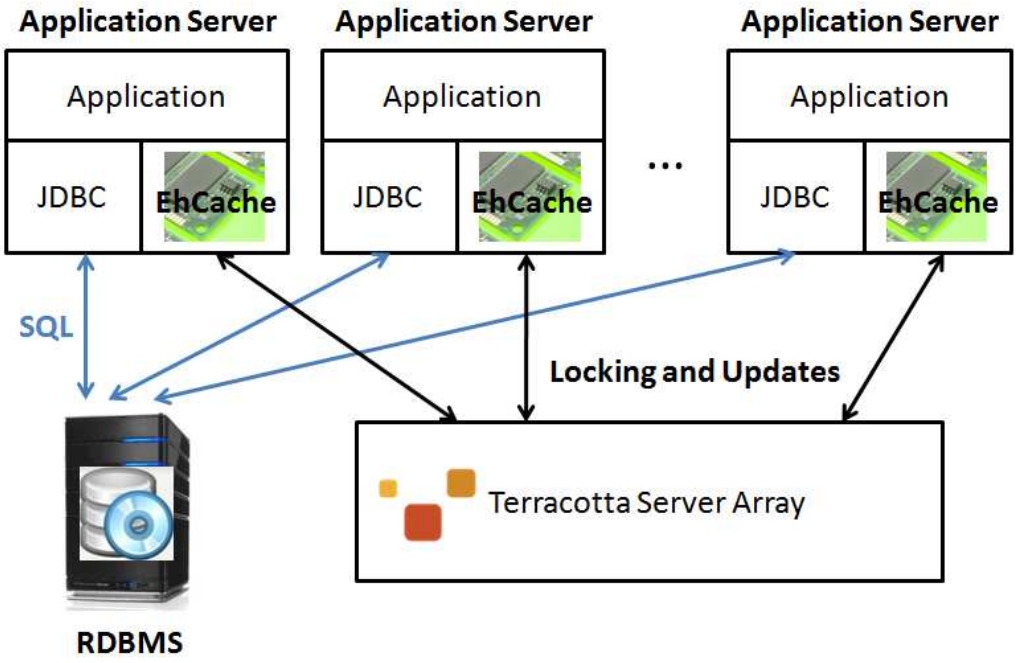


Figure 3.4: Physical Architecture as implemented by the SQLTrig framework.

3.2 Shared Address Space Architecture

An alternative architecture, named *shared address space* (SAS), requires the KVS to run in the address space of the application, see Figure 3.4. Example KVSs include Terracotta Ehcache [81] and JBoss Cache [22]. They operate in either stand-alone or in a distributed mode. With the latter, a key-value pair might be replicated either across a subset or all application+KVS instances. The KVS may implement the concept of a transaction to atomically update all replicas of a key-value in different instances.

When compared with the Client Server architecture, SAS may slow down writes in order to improve the performance of reads. Performance of reads is enhanced by eliminating the overhead of retrieving a value across the network, uncompressing and deserializing it. Writes might be slowed down because they must propagate to all replicas of a key-value pair in different KVS instances, see Figure 3.4. When writes are rare, SAS may outperform the Client Server architecture dramatically (order of magnitude or more). While the prototype transparent cache CADBMS was implemented in a Client Server architecture with SQLTrig, the same concepts can be applied to a SAS architecture to extend it with a transparent caching layer.

Chapter 4

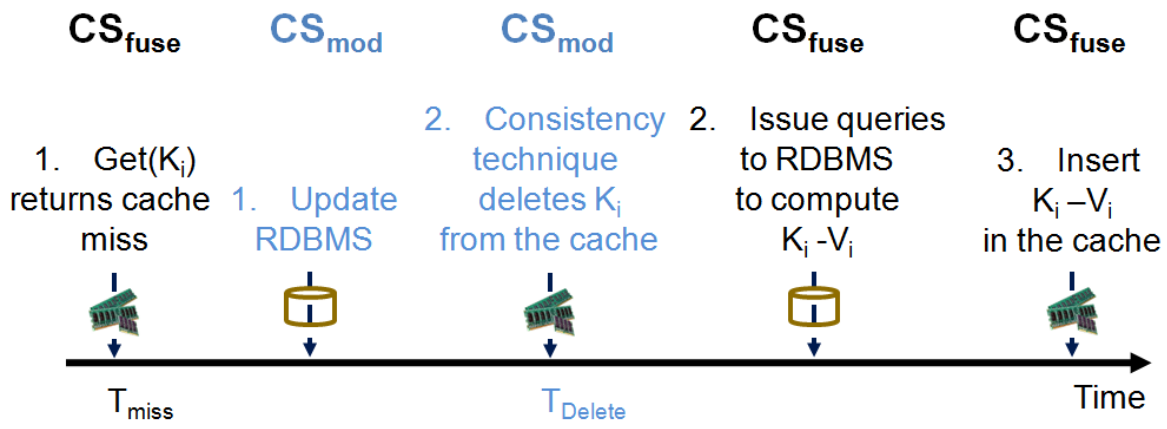
Consistency

4.1 Gumball

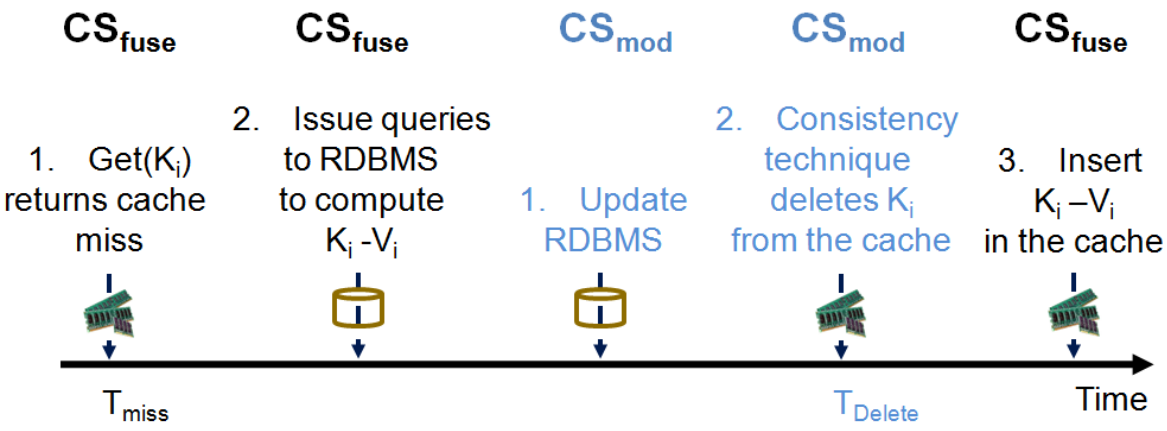
In the presence of updates to the RDBMS, a consistency technique deployed either at the application or the RDBMS may delete the impacted cached key-value pairs. This delete operation may race with a look up that observes a cache miss, resulting in stale cached data.

As an example, consider Alice who is trying to retrieve her profile page while the web site's administrator is trying to delete her profile page due to her violation of the site's terms of use. It is possible for an interleaved execution of these two logical operations leave the KVS inconsistent with the database such that the KVS reflects the existence of Alice's profile page while the database is left with no records pertaining to Alice. A subsequent reference for the key-value pair corresponding to Alice's profile page succeeds, reflecting Alice's existence in the system. This inconsistent state is the result of race conditions that occur in CADBMS and the inconsistent cache object can remain indefinitely if it is never updated with the latest value.

To illustrate a race condition, assume the user issues a request that invokes a segment of code (CS_{fuse}) that references a $k_j - v_j$ pair that is not KVS resident because it was just deleted by an update issued to the RDBMS (i.e. Alice referencing her profile page after updating her profile information). The administrator who is trying to delete Alice from the system invokes a different code segment (CS_{mod}) to delete $k_j - v_j$. Even though both CS_{fuse} and CS_{mod} employ the concept of transactions, their KVS and RDBMS operations are non-transactional and may leave the KVS inconsistent. One scenario is shown in Figure 4.1.a where CS_{fuse} looks up the KVS and observes a miss, Arrows 1 and 2 of Figure 3.1, and



4.1.a) Acceptable.



4.1.b) Undesirable.

Figure 4.1: Two interleaved processing of CS_{fuse} and CS_{mod} referencing the same key-value pair.

computes $k_j - v_j$ by processing its body of code that issues SQL queries (a transaction) to the RDBMS to computes v_j, Arrows 3 and 4 of Figure 3.1. Prior to CS_{fuse} executing Arrow 5, CS_{mod} issues its transaction to update the RDBMS and deletes k_j from the KVS. Next, CS_{fuse} inserts $k_j - v_j$ in the KVS. This schedule, see Figure 4.1.b, renders the KVS inconsistent with the RDBMS. A subsequent look up of k_j from KVS produces a stale value v_j with no corresponding tabular data in the RDBMS.

In sum, a race condition is an interleaved execution of CS_{fuse} and CS_{mod} with both referencing the same key-value pair. Not all race conditions are undesirable; only those that cause the key-value pairs to become inconsistent with the tabular data. An undesirable race condition is an interleaved execution of one or more threads executing CS_{mod} with one or more threads executing CS_{fuse} that satisfy the following criteria. First, the thread(s) executing CS_{fuse} must construct a key-value pair prior to those threads that execute CS_{fuse} that update the RDBMS. And, CS_{mod} threads must delete their impacted key-value pair from KVS prior to CS_{fuse} threads inserting their computed key-value pairs in the KVS. Figure 4.1.b shows an interleaved processing that satisfies these conditions, resulting in an undesirable race condition. The race condition of Figure 4.1.a does not result in an inconsistent state and is acceptable.

4.1.1 Gumball Implementation

Gumball Technique (GT) is designed to prevent the race conditions of Section 4.1 from causing the key-value pairs to become inconsistent with tabular data. It is implemented within the KVS by extending its simple operations (delete, get and put) to manage gumballs, see Table 4.1. Its details are as follows. When the server receives a delete(k_i) request, and there is no value for k_i in the KVS, GT stores the arrival time of the delete (T_{delete}) in a gumball g_i and inserts it in the KVS with key k_i. With several delete(k_i) requests issued back to back, GT maintains only one g_i denoting the time stamp of the latest delete(k_i). GT assigns a fixed time to live, Δ , to each $k_i - g_i$ to prevent them from occupying KVS memory longer than necessary. The value of Δ is computed dynamically.

When the server processes a get(k_i) request and observes a KVS miss, GT provides the KVS client component (client for short) with the miss time stamp, T_{miss}. The client maintains k_i and its T_{miss} time stamp. Once CS_{fuse} computes a value for k_i and performs a put operation, the client extends this call with T_{miss}. With this put(k_i,v_i,T_{miss}), a GT

delete(k_i)
1) If k_i-v_i exists then delete k_i-v_i and generate gumball g_i, i.e., k_i-g_i, with T_{g_i} set to the current time. 2) If k_i-g_i exists then change T_{g_i} to the current time. 3) If no entry exists for k_i then generate g_i, i.e., k_i-g_i, with T_{g_i} set to the current time.
get(k_i)
1) If k_i-v_i exists then return v_i. 2) If either k_i-g_i exists or no entry exists for k_i then report a cache miss with current time as T_{miss} time stamp.
put(k_i, v_i, T_{miss})
1) Let T_C be the server system time. 2) If (T_C - T_{miss}) then ignore the put operation. 3) If (g_i exists and $T_m iss$ is before T_{g_i}) then ignore the put operation. 4) If (v_i exists and its time stamp is after T_{miss}) then ignore the put operation. 5) If ($T_{miss} < T_{adjust}$) then ignore the put operation. 6) Otherwise, insert k_i-v_i with its time stamp set to T_{miss}.

Table 4.1: GT enabled delete, get, and put pseudo-code. All time stamps are local to the server containing $k_i - v_i$.

enabled KVS server compares T_{miss} with the current time (T_C). If their difference exceeds Δ , $T_C - T_{miss} > \Delta$, then it ignores the put operation. This is because a gumball might have existed and it is no longer in the KVS as it timed out. Otherwise, there are three possibilities: Either (1) there exists a gumball for k_i, $k_i - g_i$, (2) the KVS server has no entry for k_i, or (3) there is an existing value for k_i, $k_i - v_i$. Consider each case in turn. With the first, the server compares T_{miss} with the time stamp of the gumball. If the miss happened before the g_i time stamp, $T_{miss} < T_{gumball}$, then there is a race condition and the put operation is ignored. Otherwise, the put operation succeeds. This means g_i (i.e., the gumball) is overwritten with v_i. Moreover, the server maintains T_{miss} as metadata for this $k_i - v_i$ (this T_{miss} is used in the third scenario to detect stale put operations, see discussions of the third scenario).

In the second scenario, the server inserts $k_i - v_i$ in the KVS and maintains T_{miss} as metadata of this key-value pair.

In the third scenario, a KVS server may implement two possible solutions. With the first, the server compares T_{miss} of the put operation with the metadata of the existing $k_i - v_i$ pair. The former must be greater in order for the put operation to over-write the existing value. Otherwise, there might be a race condition and the put operation is ignored. A more expensive alternative is for the KVS to perform a byte-wise comparison of the existing value

with the incoming value. If they differ then it may delete $k_i - v_i$ to force the application to produce a consistent value.

GT ignores the put operation with both acceptable and undesirable race conditions. For example, with the acceptable race condition of Figure 4.1.a, GT rejects the put operation of CS_{fuse} because its T_{miss} is before $T_{gumball}$. These reduce the number of requests serviced using the KVS. Instead, they execute the fusion code that issues SQL queries to the RDBMS. This is significantly slower than a KVS look up, degrading system performance. Since the occurrence of this race condition is typically rare, the impact on overall system performance is negligible.

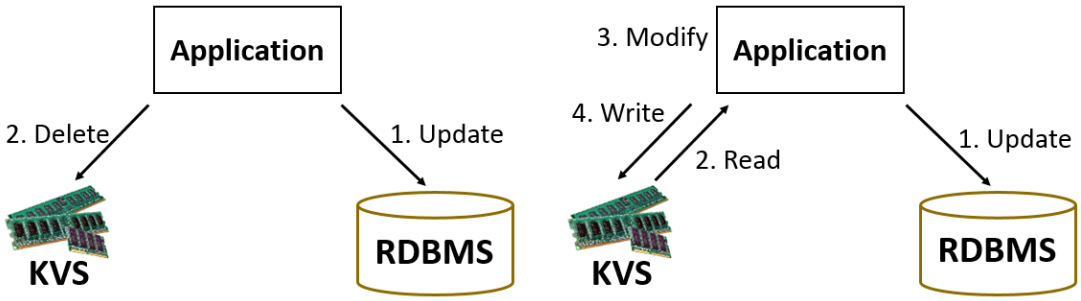
One limitation of GT is that it may allow the KVS to store stale data if the RDBMS is configured with snapshot isolation[1] [77]. This race condition is elaborated further in Section 4.2.2. GT does not capture information on the order in which transactions are issued to the RDBMS, which prevents it from being able to resolve race conditions due to snapshot isolation. This motivates the need for the IQ framework described in Section 4.2 which supports snapshot isolation as well.

4.2 IQ Leases

A challenge of CADBMSs is how to maintain key-value pairs of the KVS consistent in the presence of updates to the RDB. Key-value pairs impacted by an RDBMS update can either be invalidated [47, 26, 27], refreshed [26], or incrementally updated [43], see Chapter 5 for further details. The focus of this section is on the first two techniques, see Figure 4.2. (Support for incremental update is a future research direction, see Chapter 7.) With invalidate, the application is authored to delete the impacted key-value pairs. A subsequent reference for these keys observes a KVS miss, executes the computation that queries the RDBMS to compute a new value, and inserts the resulting key-value pair in the KVS. One may implement this technique by authoring RDBMS triggers on a table. These are invoked when a row is inserted/deleted/updated. They compute the impacted keys and delete them from the KVS. With SQLTrig, these triggers are generated dynamically.

With the refresh technique, the application identifies the impacted keys, reads their values

[1]With snapshot isolation, the RDBMS allows read transactions to proceed simultaneously with a write transaction by maintaining multiple versions of the data. The read transactions are serialized to occur before the write transaction, thus ensuring that transactions are consistent for a given snapshot in time.



4.2.a. Invalidate 4.2.b. Refresh

Table 4.2: Two techniques to maintain the key-value pairs of the KVS consistent with updates to the tabular data in the RDBMS.

and modifies them to obtain new values, and writes the new key-value pairs back to the KVS, see Figure 4.2.b. Note that refresh is more complex than invalidate because it must go one step further and compute a new value for each impacted key. Hence, it is rare to find this technique implemented as triggers because the RDBMS is typically the slowest component and complex triggers render it slower.

We define a session as a sequence of at most one RDBMS transaction and multiple KVS operations, see Table 4.3 and Section 4.2.1 for a formal definition of these terms. When concurrent sessions use invalidate and refresh, they may incur a variety of undesirable race conditions that cause a session to observe stale data from the KVS. We propose a novel framework named IQ that serializes all concurrent sessions regardless of whether they read/write/read-modify-write RDB data using the RDBMS or key-value pairs using KVS. This is termed *strong consistency* and it is desirable as it makes systems easier for a programmer to reason about [53]. The framework is designed for a networked CADBMS consisting of multiple sharded (or replicated) RDBMSs and KVSs. We assume the RDBMS instances and KVS instances implement strong consistency of their own independently. Strong consistency during normal model of operation is a challenging topic and constitutes the focus of this section.

A key ingredient of strong consistency is the *freshness* property of the KVS read operations. This property requires each KVS read to observe a key-value pair that reflects the most up to date version of the RDB, see Section 4.2.1 for a formal definition. We implement the freshness property using two leases, Inhibit (I) and Quarantine (Q). The KVS grants an I lease to a session when its referenced key observes a miss. When a session intends to

Term	Definition
BG Action	An interactive social networking activity such as invite friend, see Table 5.5.
Command	An atomic implementation of an operation using either a KVS or an RDBMS, see last two columns of Table 4.4.
KVS	A key value store such as memcached.
Operation	Read (R), Write (W), Delete, Read-Modify-Write (R-M-W) using either the KVS or the RDBMS, see Table 4.4.
Transaction	A logical sequence of one or more RDBMS operations executed atomically.
RDB	A relational database.
RDBMS	A relational database management system such as MySQL.
Session	A sequence of operations consisting of at most one RDBMS transaction and one or more KVS operations.

Table 4.3: List of terms and their definitions.

write/delete a key, it must obtain a Q lease on the key from the KVS. Sections 4.2.2 and 4.2.3 detail how IQ leases handle race conditions, including those discussed for Gumball in Section 4.1, in the context of invalidate and refresh. With simple sessions that implement social networking actions, we present benchmarking results in Section 4.2.5 that show these leases reduce the amount of stale data to zero with minimal impact on system performance.

4.2.1 Overview

Our proposed IQ framework targets CADBMS systems realized using an off-the-shelf RDBMS and a key-value store that supports simple operations such as get, set, compare-and-swap, and delete. No changes to the RDBMS software are necessary to implement the framework. Instead, we extend the KVS with new commands that implement the I/Q leases. In addition, we introduce a simple programming model for how these commands must be used in combination with the RDBMS transactions to implement sessions (see below for a formal definition). This model requires a session to acquire and release leases in a manner similar to the two phase locking protocol [56, 39, 71]. The framework is non-blocking and deadlock free. It may delete key-value pairs and abort and re-start sessions to realize strong consistency.

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

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

Table 4.4: Alternative actions and their implementation with memcached and a SQL system.

The focus of this study is on simple read (R), write (W), delete, and read-modify-write (R-M-W) operations that manipulate a small amount of data. While these operations are well defined with SQL (see the third column of Table 4.4), their implementation with a KVS may vary from one system to another. We focus on a variant of memcached [55, 68] in Section 4.2.4 to describe an implementation of the freshness property. The second column of Table 4.4 shows the different memcached commands that implement the alternative operations.

One may implement the R-M-W operation of the RDBMS as a transaction that provides Atomicity, Consistency, Isolation, and Durability (ACID) properties [40]. With memcached, one may use compare-and-swap (cas instead of set for W) to realize an atomic implementation of R-M-W. The idea is to maintain the old value (v_{old}) retrieved by the R operation, apply the M to compute a new value (v_{new}), and implement the W operation with cas using v_{old} and v_{new}. When the cas fails, the application may re-try the operation starting with the R.

We define a *session* as a sequence of operations consisting of at most one RDBMS transaction and several KVS operations. Each RDBMS operation is a transaction with ACID properties. A session starts when it executes its first operation. With no leases, the end of a session is when it performs its last operation. When configured with leases, a session ends once it has released its last acquired lease.

Table 4.5 shows the existence of the different KVS operations with invalidate and refresh. Invalidate does not use the R-M-W operation with the KVS 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.

The *freshness* property applies to a KVS read operation. It requires every key-value pair in the KVS to reflect the latest state of the relational database (RDB) in the RDBMS. Formally, for each key k_i in the KVS, each corresponding value v_i must correspond to a

Operation	Invalidate	Refresh
Read	✓	✓
Write	✓	✓
Delete	✓	✓
R-M-W	X	✓

Table 4.5: Presence of KVS operations with invalidate and refresh.

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})

4.6a. 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})

4.6b. Confirm Friend

Table 4.6: Pseudo-code of two interactive social networking actions implemented as sessions with no leases.

function f that performs its computation using the RDB produced by the latest session (S_{last}) that completed its changes to the RDB:

$$\{\forall(k_i, v_i), k_i \in KVS, v_i \equiv f(RDB, S_{last})\} \tag{4.1}$$

Multiple sessions may execute concurrently and overlap in arbitrarily complex ways. The freshness property ensures strong consistency for the simple operations of Table 4.4 by requiring the CADBMS to serialize concurrent sessions as if they executed in isolation one after another.

To realize the freshness property, we introduce two leases named Inhibit (I) and Quarantine (Q). The KVS grants these leases on a key. The I lease is issued on a key when the KVS observes a miss for the key referenced by the KVS read operation. The Q lease is issued when the application intends to either delete or write a value for a key. Leases collide when they reference the same key. Refresh and invalidate handle collisions in different ways, see Table 4.7 and discussions of Sections 4.2.2 and 4.2.3.

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 due to how it uses the Q leases: If the KVS holds Q leases indefinitely (similar to locks) then node failures may degrade system performance severely. With time outs, the KVS recovers from node failures that prevent an application from releasing it lease. Section 4.2.4 describes how to decide the life time of leases.

A contribution of this study is to ensure that the serial schedule of concurrent sessions performing R-M-W operations (using refresh) is identical with both the RDBMS and the KVS. This is realized using the I/Q leases and a programming framework for their usage. Section 4.2.4 describes an implementation of a KVS client that hides the concept of leases and their back off from the programmer, simplifying their usage. Table 4.8 shows two different re-writes of the pseudo-code of the "Invite Friend" to use the I/Q leases.

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

4.2.2 Invalidate

This section describes the race conditions that cause invalidate to violate the freshness property. Subsequently, we present how the Inhibit (I) and Quarantine (Q) leases are used to prevent these race conditions.

Problem Definition

This section starts with an overview of the read lease of [62] and how it prevents undesirable race conditions between (1) sessions that update the RDBMS and delete key-value pairs from the KVS and (2) sessions that observe a KVS miss to compute a value and insert it in the KVS. Next, this solution is shown to violate the freshness property when the RDBMS is configured[2] with snapshot isolation and the session employs RDBMS triggers to invalidate key-value pairs. Finally, it is observed that even when the session deletes its impacted keys after the transaction commits, there may exist a window of time when the KVS does not satisfy the freshness property and produces stale data. Section 4.2.2 employs the I/Q leases to resolve these limitations.

The read lease of [62] is identical to the I lease presented in Section 4.2.1. The KVS grants this lease to a session that encounters a miss for a referenced key that has no pending read lease, providing the session with a token. The session may query the RDBMS, compute a value for the key and insert the key-value using its token. The KVS inserts the provided key-value pair only if the token identifies a valid lease. The KVS invalidates a lease for a key if it receives a delete for the key. It ignores all inserts with tokens that reference an invalid lease. This enables the KVS to prevent potential race conditions where a KVS miss computes and inserts a stale value.

Multiple KVS misses may reference the same key. In this case, the KVS grants a read lease to one caller and requires others to back off for a pre-specified duration of time and repeat their KVS read. This back off time may increase exponentially as a requester collides with other requesters repeatedly for the same key [62].

The read lease by itself does not prevent a CADBMS system from producing stale data when the RDBMS employs snapshot isolation. Snapshot isolation guarantees (1) all reads made in a transaction observe a consistent snapshot of the RDB and (2) the transaction

[2]Numerous industrial strength RDBMSs provide multi-version concurrency control [16] which offers snapshot isolation to enhance concurrency of transactions and improve application performance.

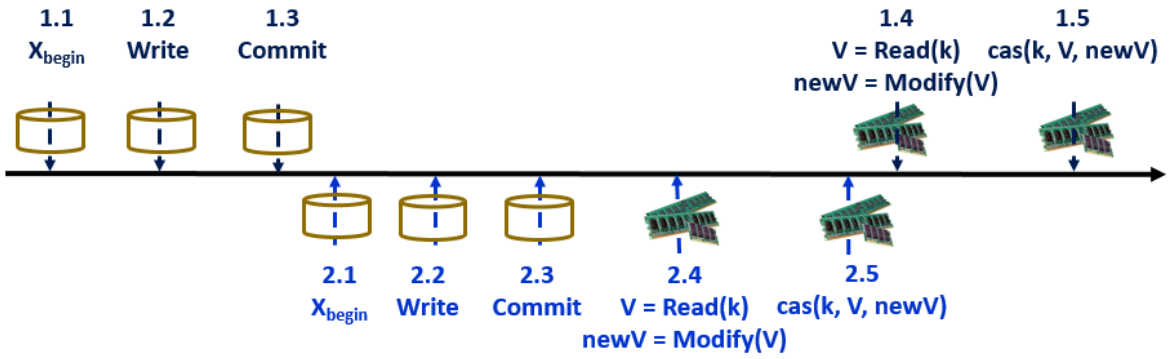


Figure 4.2: Snapshot isolation enables Session 2 to compute and insert a stale value in the KVS.

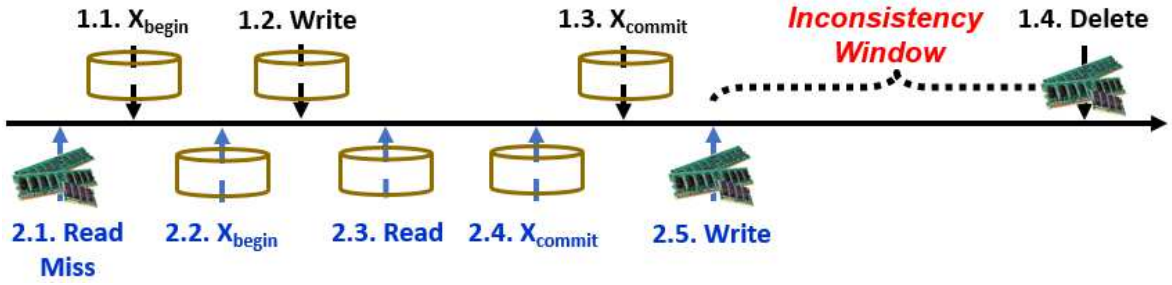


Figure 4.3: Deletion of key-value pairs after transaction commit point results in an inconsistency window.

will commit only if none of its updates conflict with any concurrent updates made since that snapshot. With invalidate, snapshot isolation results in two different undesirable race conditions that violate the freshness property. We describe these in turn.

The first race condition is when the sessions use RDBMS triggers to invalidate key-value pairs. These triggers execute as a part of the transaction that updates the RDBMS, say T_1. After the trigger deletes the impacted key-value pair and prior to the commit point of T_1, another session (Session 2, S_2) may perform a KVS look up for the impacted key. S_2 observes a miss and queries the RDB to compute a value using the RDB state prior to T_1 committing, see Figure 4.2. S_2 inserts this stale key-value pair in the KVS. After T_1 commits, the key-value pair inserted by T_2 is no longer valid. A subsequent read for this key-value pair violates the freshness property.

One may try to solve the limitation shown in Figure 4.2 by requiring the session to delete the impacted key-value pairs after its RDBMS transactions commit. This approach may

results in an inconsistency window during which the system violates the freshness property. This is illustrated in Figure 4.3 that shows a Session 2 (S_2) performing a KVS look up for the same key-value pair as Session 1 (S_1). S_2 observes a miss and queries the RDBMS concurrently with the transaction that performs the write of S_1. Snapshot isolation enables the RDBMS read of S_2 (Step 2.3) to compute its result using an old RDB state. The window of time between Step 2.5 to when S_1 deletes the key enables a KVS read to violate the freshness property.

A possible solution is to require the KVS delete to occur as a part of the transaction commit. However, we are not aware of an RDBMS that enables one to implement this solution.

Solution

We resolve the race conditions described in the problem definition by requiring a transaction to obtain a Q lease on a key that it intends to delete. After the transaction commits, the application issues a KVS delete for the key, purging the key and releasing its Q lease. While there is a Q lease on a key, the KVS ignores all write operations for the key. However, all reads for the key are satisfied as long as they observe[3] a KVS hit. Those reads that observe a KVS miss must obtain an I lease for their referenced key. When this collides with an existing Q lease, the read must back off and try again. See Table 4.7.a for compatibility of I and Q leases.

To illustrate the use of I and Q leases, consider the two sessions shown in Figure 4.2. Session 1 is modified in two ways. First, Step 1.3 is replaced with a request for a Q lease. Second, a new step, Step 1.5, is added to delete the impacted key and release the Q lease. With these changes and the compatibility matrix of Table 4.7.a, Step 2.1 of Session 2 that observes a KVS miss must obtain an I lease on the quarantined key. The KVS notifies it to back off and try again, pushing this step to succeed once Session 1 deletes its referenced key and releases its Q lease. This prevents Session 2 from computing and inserting a stale value.

If Step 2.1 of Session 2 observes a KVS hit then it proceeds to consume the produced value. This satisfies the freshness property because Session 1 has not finished as yet. This highlights the fact that the freshness property is at the granularity of sessions (and not RDBMS transactions). For example, there is a window of time between when a transac-

[3]In a serial schedule, the sessions performing the reads appear before the one holding the Q lease.

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

4.7.a. Invalidate

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

4.7.b. Refresh

Table 4.7: Compatibility matrices of I/Q leases.

tion commits to the time it deletes its impacted key from the KVS and releases its Q lease. During this time, another session may read the value of this key. This session is re-ordered to have occurred prior to the one that updates the RDBMS. Hence, the serial schedule is at the granularity of sessions and satisfies the freshness property.

Acquiring Q leases as a part of a transaction and its subsequent release after the transaction commits is similar to two phase locking [56, 39, 71] and provides strong consistency.

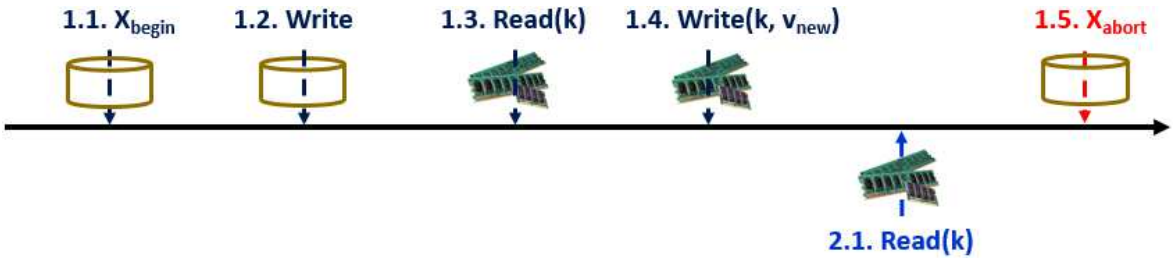


Figure 4.4: CADBMS results in dirty reads with refresh when an impacted key-value pair is updated prior to transaction commit.

4.2.3 Refresh

In addition to the race conditions of Section 4.2.2, the refresh technique suffers from undesirable race conditions attributed to its R-M-W operation. This section presents these race conditions and how they violate the freshness property. Subsequently, Section 4.2.3 describes the use of I and Q leases to prevent these race conditions.

Problem definition

An atomic implementation of R-M-W must maintain the value (v_{old}) observed by the R operation for the referenced key, modify v_{old} in its memory to compute a new value (v_{new}), and implement W using an atomic KVS compare-and-swap (cas). When used in combination with a transaction processing RDBMS, the writing of v_{new} must happen after the transaction commits. Otherwise, the CADBMS solution may suffer from dirty reads. This is shown in Figure 4.4 with Session 1 writing a v_{new} in the RDBMS that is consumed by Session 2. Subsequently, the RDBMS aborts the transaction that constitutes Session 1 (due to a deadlock), causing Session 2 to observe a value that should not have existed. This violates the freshness property as Session 2 did not observe the latest state that completed successfully.

Another challenge of R-M-W is how to produce the same serial schedule with both the RDBMS and the KVS. In their simplest form, race conditions between two concurrent sessions, S_1 and S_2, update the RDBMS in a manner that realizes one serial order (S_1 followed by S_2) and observe a different serial order from the KVS (S_2 followed by S_1). This may violate the freshness property because a subsequent KVS read may no longer be a function of the RDB.

It is possible for two sessions to update two different rows of the RDB and conflict

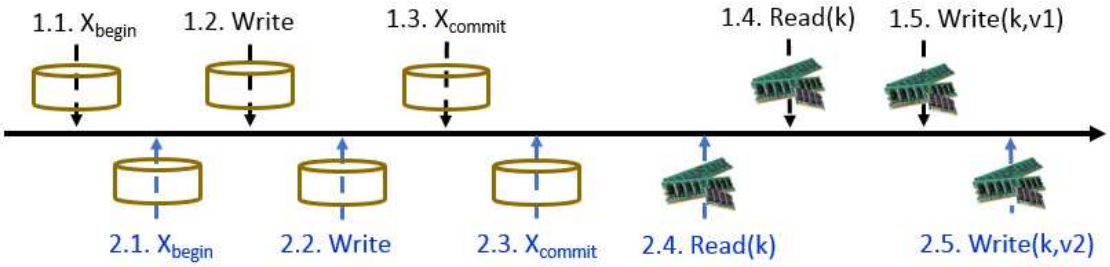


Figure 4.5: Two logical operations race with one another to update the RDBMS correctly only to result in a KVS state that violates the freshness property.

by referencing the same key-value pair in the KVS. This is because the KVS values are not normalized and may glue data from different rows together. Hence, two sessions may simultaneously update two different rows of two different tables that impact the same key. Use of cas to implement R-M-W accommodates associative operations such as increment and decrement of a field. Otherwise, it is possible to violate the freshness property.

To illustrate, consider the concurrent executions of Sessions 1 and 2, S_1 and S_2, in Figure 4.5. The cas of Step 2.5 fails when the value written by Step 1.5 is different than the one S_2 read in Step 2.4. This causes S_2 to repeat its R-M-W to update the KVS. This results in an inconsistency window that enables another session to consume a value that violates the freshness property.

With Figure 4.5, it is possible to re-arrange the R-M-W of S_1 to occur after S_2, i.e., 2.4 and 2.5 to occurs prior to 1.4 and 1.5. In this case, both cas of S_1 and S_2 succeeds. However, their serial order as reflected in the RDBMS is not the same. This produces the correct key-value if S_1 and S_2 either manipulate different fields of a value (e.g., S_1 increments the number of friends while S_2 decrements the number of pending friends) or are associative (e.g., or S_1 increments the number of friends while S_2 decrements it, see Invite Friend and Confirm Friend implementation of Table 4.6). Otherwise, the next KVS read that references the produced key-value pair violates the freshness property as it does not reflect the latest RDB state.

Solution

We use the Q lease to prevent the race condition described in the problem definition by implementing the cas command as two separate commands:

1. Quarantine-and-Compare, QaC(key, v_{old}), acquires a Q lease on the referenced key from the server. In addition, the server must verify that the current value of the key equals v_{old}. If both conditions are satisfied then the server grants the Q lease and returns a token to the requester. Otherwise, the server returns an abort message. In this case, the requesting session must release all its leases, roll back any RDBMS transaction that it might have initiated (see below), back off for some time, and re-try its execution.

2. 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.

The QaC implements the compatibility matrix of Table 4.7b 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 these two sessions in the RDBMS is not known to the KVS. By aborting and restarting the requesting session, the KVS serializes this session after the one holding the Q lease.

A session must issue the QaC command for each key that its intends to R-M-W. Should the KVS respond with abort for a QaC command, the session must release all its leases in order to avoid the possibility of deadlocks. To illustrate, consider Session 1 (S_1) acquiring a Q lease on data item D1 and observing a conflict with Session 2 (S_2) when acquiring a Q lease on data item D2. If S_2 attempts to acquire a Q lease on D1 then it will conflict with S_1. If S_1 (S_2) retries acquiring a lease on D2 (D1) repeatedly, it will encounter a conflict indefinitely, resulting in a deadlock. By requiring each session to release all its leases and try again after a random time out period, the IQ framework becomes deadlock free.

One may perform the QaC calls either prior to the start of the RDBMS transaction or as a part of the RDBMS transaction. Once the transaction commits, the session must issue SaR for each impacted key with its new value. This updates the value in the KVS and releases the Q lease on the key.

Consider the two alternative possibilities to issue the QaC command. When QaC is issued prior to the start of the transaction and the KVS returns an abort message, then the session must release all its leases, back off for some time and re-try its execution. When QaC is issued as a part of the RDBMS transaction and the KVS returns an abort message, the session must abort the in-progress transaction, back off, re-start the transaction, read the impacted key, modify its value, and issue QaC for the key. Section 4.2.5 provides a quantitative comparison of these two alternatives. Surprisingly, they provide comparable

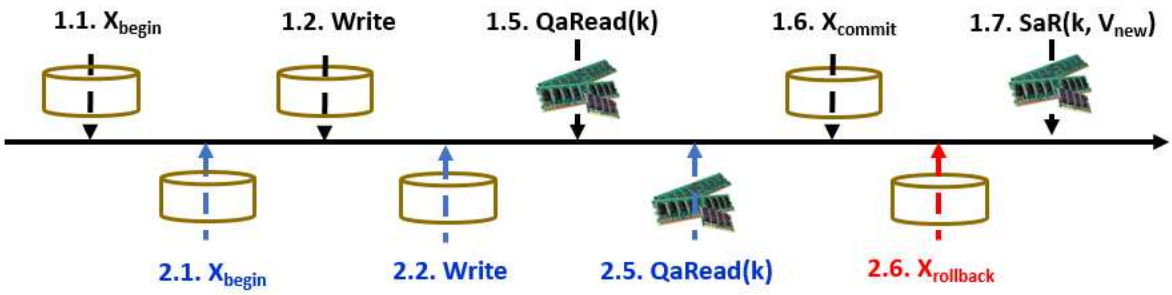


Figure 4.6: The RDBMS transaction of Session 2 is aborted, rolled back, and retried because its QaC requests a Q lease that conflicts with the existing Q lease of Session 1.

performance because the one that performs more work results in fewer aborts.

The IQ framework avoids the dirty read of Figure 4.4 by requiring Session 1 to update a key-value pair (using SaR) after its RDBMS transaction commits.

Figure 4.6 shows how the sessions of Figure 4.5 are extended with the QaC and SaR commands. In this figure, the sessions are implemented to issue their QaC as a part of their RDBMS transaction. In Step 2.5, once Session 2 issues its QaC call, the KVS detects its conflict with that of Session 1 as they reference the same key. Since Session 1 issued its QaC earlier and was granted the Q lease, the KVS returns an abort message to Session 2. In response, Session 2 aborts its RDBMS transaction (Step 2.6) and tries again.

Note that our proposed use of Q leases resembles two phase locking as it requires a session to issue all its QaC calls prior to the RDBMS transaction commit and all its SaR invocations after transaction commit.

4.2.4 An Implementation

This section details an implementation of the server and the client components of a KVS that realizes the I/Q leases of Section 4.2.3. These are an extended version of the Twitter memcached versions 2.5.3 [68] and Whalin memcached client version 2.6.1 [86], respectively. We conclude with a description of changes to implement the invalidate technique. A unique feature of our implementation is that it supports those applications that employ a hybrid of invalidate and refresh techniques for different keys simultaneously.

Client

We modified the Whalin memcached client version 2.6.1 [86] to support the I/Q leases of Section 4.2.3 by providing the following four new commands for use by the programmer:

- QaC(key, v_{old}): Provides the QaC interface of the refresh technique per specification of Section 4.2.3. This interface issues messages to the KVS server that implements its functionality, see the Server description of Section 4.2.4.

- SaR(key, v_{new}): Provides the SaR interface of the refresh technique per specification of Section 4.2.3. Similar to QaC, this interface issues messages to the KVS server that implements its functionality, see the Server description of Section 4.2.4.

- Q(key): sends a message to the KVS server to obtain a Q lease on its referenced key. This command is used to implement strong consistency with invalidate, see Section 4.2.2. (The delete command purges a key and releases its Q lease, see the Server description of Section 4.2.4.)

- GenID(): Returns a unique Transaction Identifier(TID) to identify the KVS delete operations performed by RDBMS triggers[4] that implement invalidate. This unique identifier might be generated using either a Java UUID or by a call to the KVS.

We assume a session instantiates a Whalin connection with the KVS and uses it during its life time. This connection is used with both QaC and SaR invocations. It maintains the KVS provided tokens[5] corresponding to a lease on a key. Once the SaR is issued for the key, the client identifies the token for the key and provides it to the server to release the corresponding Q lease. Thus, tokens are transparent to the application software developer. Table 4.8 shows two alternative ways that one may implement the "Invite Friend" session of Table 4.6.a, see Section 4.2.5 for details.

Server

The server is designed to support both invalidate and refresh simultaneously. We implemented this design by extending the Twitter memcached versions 2.5.3 [6] to implement the following commands:

[4]Triggers execute as a part of the transaction that invokes them.

[5]This is also true with I leases (where the client implements back off seamlessly).

1. Q(key, value): Returns a token pertaining to the Q lease acquired by the server on the specified key. The implementation checks to ensure the provided value matches the existing value for the key. Otherwise, no lease is granted and the returned token instructs the requester to roll back its RDBMS transaction and restart its session per specifications of Section 4.2.3. This command is used to implement the QaC(key, v_{old}) command of the client.

2. 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. This command is used by the invalidate technique (see Section 4.2.4) and implements the compatibility Table 4.7.a. Should one of the acquired Q leases expire for TID, the KVS deletes that key.

3. Release(key, token): Employs the key and token to identify a pending lease and removes it. If the token is not valid then the release command is ignored.

4. Set(key, v_{new}, token, v_{old}): Employs the compare-and-swap feature of the KVS to swap the value of the referenced key with the new one as long as the current value of the key equals v_{old} and the provided token is valid. If the token is not valid then the lease has expired and the server (1) deletes the existing key-value pair, (2) adjusts the time to live of the leases based on a 60 second sliding window of response times.

5. Get(key): Returns the value for the referenced key with a hit. Otherwise, the server acquires an I lease on the referenced key and returns a token for this key.

6. Delete-and-Release, DaR(TID): Retrieves the value v where key=TID. For each string token k in v, DaR deletes the corresponding key=k from the KVS, and releases the lease on k. This command is used to implement the invalidate technique, see Section 4.2.4.

The server is able to support both invalidate and refresh mode of key-value maintenance because different commands are used to implement how the Q leases is used with each. The software for each implements the corresponding element of the compatibility table of Table 4.7.

Note that it is acceptable for the KVS 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

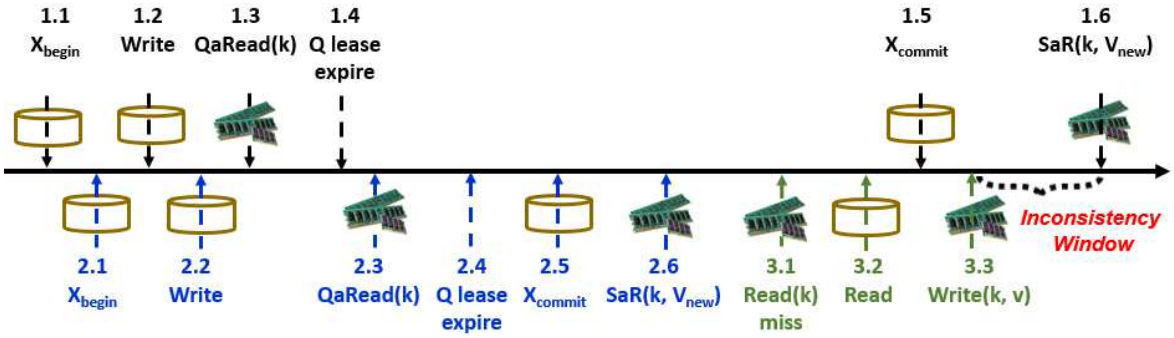


Figure 4.7: Expired leases cause refresh to produce stale data.

the refresh technique. This is because S_2 will delete its referenced key (due to its use of invalidate), preventing a violation of the freshness property.

KVS using Invalidation

As detailed in the Server description of this section, the server implements commands for both invalidate and refresh. To implement the invalidate technique using RDBMS triggers, we provide a dynamic link library that exposes the Quarantine-and-Register, QaReg(TID, keys), KVS command. Before executing an RDBMS write operation, the application first calls GenID() to obtain a unique identifier (TID). When a trigger computes the set of keys impacted by the proposed update to the RDB, it invokes QaReg using the TID along with the set of keys. The TID can be passed to the trigger through a session variable (e.g session context information in the Microsoft SQL Server [61] or a per-session package in Oracle 11g [46]). The server maintains a key=TID whose value contains the list of keys identified by the trigger. When a session commits a transaction, it issues a DaR(TID) to the KVS to delete the keys associated with TID from the KVS and release their Q leases.

Life Time of a Lease

With refresh, the life time of a lease is important because it impacts the strong consistency guarantee of the IQ framework. In particular, when a KVS write references a Q lease token that is no longer valid, the current implementation assumes the lease has expired and delete the key-value pair. Even with this in place, an expired lease may produce stale data for some time. One such a possibility is shown in Figure 4.7. Session 1 (S_1) is delayed in a manner

that it writes its value after a significant delay from when its Q lease expires in Step 1.4. During this time, Session 2 (S_2) is able to acquire its Q lease and observe a time out as well. When S_2 issues its SaR, the KVS deletes its referenced key-value pair because its token fails to identify a valid lease. Now, a third session (S_3) references the same key, observes a KVS miss, and computes a new value that it inserts in the KVS after S_1 commits its transaction. This provides for an inconsistency window where a KVS read violates the freshness property.

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 [62].) 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 [36]. 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 [36] for details.

4.2.5 Evaluation

This section employs the BG social networking benchmark [12] to evaluate the implementation of Section 4.2.4. The description of BG was presented in Section 5.7.

Client Designs

As detailed in Section 4.2.3, one may implement clients in two alternative ways. Table 4.8 shows these two alternatives for the Invite Friend action. This section quantifies their trade-off.

The first implementation invokes the read, modify and QaC commands of KVS (in support of R-M-W) prior to starting the RDBMS transaction. Hence, if the QaC fails then no RDBMS roll-back is required. The session simply backs off, re-tries the read, modification and QaC until it succeeds. A draw back of this technique is that, once QaC succeeds, leases are held for the duration of time the RDBMS performs its modify/write operation.

The second implementation applies the read, modify and QaC actions after the RDBMS modify/write operation and prior to the commit point of the transaction. This reduces the duration of Q leases in the KVS. However, it increases the complexity of the software for two reasons. First, when the QaC command of the KVS fails then the RDBMS transaction must be aborted. Second, the developer must be aware of the transaction semantics and its

Invite Friend (InviterID, InviteeID)

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

4.8a. KVS operations prior to 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} = KVS Read (Key)
   e. V_{new} = Increment V_{old}.#PendingFriends
   f. QaC (Key, V_{old})
2. Commit Xact
3. SaR (Key, V_{new})

4.8b. KVS operations as a part of transaction

Table 4.8: Two alternative implementations of the Invite Friend session of Figure 4.6.a using QaC and SaR commands.

	QaC calls prior to transaction **start**	QaC calls prior to transaction **commit**
0.1%	2	2
1%	672	197
10%	2,542	1,437

Table 4.9: Number of rejected write leases (QaC calls) with two client implementations of Section 4.2.4.

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 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 Table 4.6.a, then the correctness of software might be compromised by applying the modification to the key twice.

One approach to support the above 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 to not observe the updates proposed by the transaction, avoiding the complexity associated with differentiate between a read that observes a KVS hit or a miss. We implemented this second approach.

Table 4.9 shows the number of failed QaC invocations with the alternative client implementations. Performing KVS actions prior to transaction start results in more failures due

	10K members		100K members	
	Twemcache	IQ	Twemcache	IQ
0.1%	36%/2.8%	0%/0%	15%/0.7%	0%/0%
1%	37%/1.1%	0%/0%	15%/0.01%	0%/0%
10%	10%/0.5%	0%/0%	5%/0%	0%/0%

Table 4.10: Percentage of unpredictable data using refresh/invalidate with Twemcache by itself and Twemcache extended with the I/Q leases.

to Q lease collisions for the same key. However, it provides a performance identical to its alternative because it does not roll-back transactions and the number of rejected write leases is a very small percentage of the total number of performed operations.

Performance Results

This section compares the performance of two variants of Twemcache:

- Twemcache extended with read leases of [62], labeled Twemcache.
- Twemcache extended with I/Q leases using the implementation of Section 4.2.4, labeled IQ.

Table 4.10 shows the amount of stale data produced by these two alternatives for two different social graphs consisting of 10K and 100K members. With both social graphs, the amount of stale data decreases as a function of the percentage of write actions. This is because, once stale data is inserted in the KVS, it is removed only by another write action. A higher frequency of write action increases the likelihood of stale key-value pairs being restored to their correct value.

The amount of stale data observed with Twemcache is significantly lower with invalidate when compared with refresh. There are two reasons for this. First, invalidate does not use the R-M-W operation. Second, Twemcache is configured with read leases of [62].

It is interesting to note that the percentage of stale reads is lower with the large social graph when compared with the small social graph. This is because the assumed Zipfian mean (0.27) directs 80% of the actions to 20% of the members. The number of members is ten times higher with the large social graph, reducing the likelihood of read and write actions competing for the same data items.

	10K members		100K members	
	Twemcache	IQ	Twemcache	IQ
0.1%	56,792	56,783	57,032	57,019
1%	45,186	44,844	13,898	12,603
10%	39,912	37,518	1,907	1,911

Table 4.11: SoAR using refresh with Twemcache by itself and Twemcache extended with the I/Q leases.

The SoAR of the invalidation technique is identical for Twemcache and IQ (not shown) because both implementation require the RDBMS triggers to issue the same keys. Hence, both issue the same number of calls to the KVS. The only incurred overhead with IQ is the final call by the session to delete the impacted keys, DaR. This has no impact on the observed SoAR because the CPU of the KVS server is less than 50% utilized.

Table 4.11 shows the SoAR of the refresh technique with Twemcache by itself and once extended with the I/Q leases. With the small social graph, the network bandwidth (3 Gbps) of the Twemcache becomes fully utilized to dictate the observed SoAR. This explains why the two alternative deployments provide a comparable SoAR rating.

With the large social graph and the 0.1% mix of write actions, the network bandwidth of Twemcache remains fully utilized, causing the two variants of Twemcache to provide a comparable performance. With the 1% and 10% mix of write actions, the disk of the server hosting the RDBMS becomes fully utilized with a sustained queue of requests, dictating the observed SoAR. This is due to the large size of the 100K social graph that no longer fits in the memory of the RDBMS server. The difference in SoAR of IQ and Twemcache is lower than 10% and we attribute this to experimental noise.

Chapter 5

Cache Consistency Techniques

When a data object, such as a HTML page, is generated and stored in the cache, it exists as a separate copy of the data outside of the origin server or RDBMS. If the data on the RDBMS were to change, a system will serve incorrect data if it retrieves the old cached copy rather than the up-to-date version from the database. In order to prevent the cache from becoming inconsistent with the RDBMS, various cache consistency mechanisms are used to ensure that a RDBMS change occurs on the RDBMS is propagated the cache. These mechanisms differ in their implementation, resource requirements, runtime performance as well as their consistency guarantees.

A cache entry is considered stale or inconsistent with the RDBMS if a read operation from the cache produces different results than from querying the RDBMS. When an update transaction to the RDBMS changes the state of the data, if the cache continues to hold the obsolete copy of the data as a valid cache entry, any subsequent reads of the cached data will produce stale results. The amount of time elapsed from the update to the moment when the infrastructure is guaranteed to produce the updated value is termed the inconsistency window [84].

The various consistency mechanisms can be classified into two broad categories: non-transparent consistency techniques and transparent consistency techniques. Non-transparent consistency techniques require explicit implementation by a human application developer or database administrator with knowledge of specific application logic. The developer may be required to identify the relationship between the cached data and possible update operations which change the state of that data. On the other hand, transparent consistency techniques maintain the cache consistent without requiring additional input from the developer. These

techniques require no prior knowledge of the application workload and can be used with no change to application software. The characteristics of each class of techniques are detailed in the following sections.

5.1 Non-transparent Consistency Techniques

The non-transparent class of consistency techniques refers to techniques that require the application developer to manually author software to maintain the cache consistent with the database. An application developer performs global reasoning between the cache entries and their database counterparts to author custom invalidation software that accompanies update operations and functions to maintain the cache consistent with the database.

The benefit of these approaches is that the solution is custom tailored for the application. However, these approaches tend to be time-consuming to implement and are error-prone. Software bugs can occur in the consistency maintenance code due to programmer error or as a result of misinterpreting application logic or requirements. As the application evolves, the application developer must maintain the consistency logic up-to-date which opens the possibility for errors and software bugs once again.

The following sections describe each technique and their requirements.

5.1.1 Application Developer Consistency (ADC)

A human programmer (or database administrator) authors application specific software to update the cache. Typically, this software is an extension of the code where the application updates the database. With the example of the user profile page, the developer may extend the application software to update the user's profile with the appropriate commands to either update the cache or delete the cached key-data pair. With the latter, a subsequent cache look up observes a miss and invokes the application logic to compute the new key-data pair and inserts it into the cache.

5.1.2 RDBMS Trigger (Trig) Driven

The programmer extends the RDBMS with triggers [11] that detect changes to the underlying tables, compute the impacted key(s) and issue delete command(s) to the cache manager to purge these keys. For example, a key for a user's profile page might be stored in the cache

as the word “Profile” combined with a unique identification number assigned to the user, eg. *Profile-172*. This knowledge of the key naming convention is used when authoring the triggers as the triggers have to know how to generate the appropriate keys before issuing the delete to the cache. An update that occurs to the RDBMS will cause the trigger to fire and execute its body. The trigger has access to information about the row or rows that were changed and can extract information such as the value of a column in the changed row in order to compute the key.

For example, a user with ID *172* changes their profile which the application translates into an update on the *Users* table. The update causes the trigger on the *Users* table to fire and extract the value of the ID column from changed row, *172*. The trigger then concatenates this value with the word “Profile” to generate the key *Profile-172* and issue the delete for that key to the cache.

5.1.3 Synthetic

Alternatively, a Time-To-Live(TTL) for cache entries can be used to provide weaker consistency guarantees. Some applications do not have strict consistency requirements, where it is acceptable for users to witness stale data some of the time. This technique differs from the previous two approaches in that updates to the RDBMS do not actively trigger invalidations or updates of the corresponding cache entry. Instead, the cache entries are oblivious of changes made to the underlying data and depend on the TTL to be independently invalidated.

The TTL dictates the duration a cache entry is valid once it has been inserted into the cache. The developer extends the application to provide a Time-To-Live(TTL) for each key-data pair. The cache manager invalidates a key-data pair once its TTL expires. Larger values of TTL mean that the system holds cache entries longer, increasing the cache hit rate but also increasing the rate of stale reads. Smaller values of TTL reduce the rate of stale reads but as a result, reduce the cache hit rate and force the application to continually repopulate the cache for frequently read entries. The developer has to understand the nature of the application and its access patterns in order to estimate an optimum value of TTL for each key. However, this solution tends to be inflexible in the face of changing access patterns and unpredictable workloads.

5.2 Transparent Consistency Techniques

Transparent consistency techniques differ from non-transparent techniques in that the consistency of the cache is maintained automatically without special input by the developer. These techniques rely on two general mechanisms, (i) cues from the application to indicate the data dependencies of cached entries and (ii) notification from the RDBMS when a relevant change is detected. The cues that can be used are readily available: the SQL queries which are being issued by the application to the RDBMS. By intercepting these queries through a RDBMS client wrapper, the system automatically identifies these queries without additional changes required to the application software. The framework keeps track of these dependencies in order to determine which cache entries are affected whenever a change is detected in the database.

Two approaches were considered for the implementation of a transparent caching layer. First, the Query Change Notification(QCN) approach utilizes technology recently introduced in commercial RDBMS, described in Section 5.3. Evaluation of the system with QCN revealed limitations in the it's ability to scale. The time to register a query with the RDBMS was prohibitively expensive and there was a limit to the number of queries that could be registered with the RDBMS (hundreds). With time, optimization of the QCN mechanisms and improvements to the interfaces may overcome these limitations and make the QCN approach more viable. However, its present implementation does not make it suitable for supporting scalable transparent caching in a CADBMS.

The second approach is to dynamically author triggers that detect the changes to the data and send notifications to the cache. It mitigates the problems observed by the QCN approach by registering queries as templates rather than individual queries. Queries with the same structure but different values are considered as different instances of a parameterized template and in a typical application, there are only dozens of such query templates. This approach is described in Section 5.4 and was determined to be the most suitable approach to enabling transparent caching.

5.3 Query Change Notification (QCN)

Query Change Notification (QCN) is a mechanism where queries can be registered by an application with the RDBMS. The RDBMS provides a callback interface where the appli-

cation can receive notifications when the RDBMS detects that the results of the registered query has changed. The application can then process the notification to extract information regarding what has changed and react accordingly. In the context of a cache, this feature becomes very useful in maintaining the consistency of the cache with the RDBMS. Cache entries which were generated based on queries to the RDBMS can be invalidated if the system detects that the underlying data has changed. In order to do this, the cache has to be aware of and maintain the dependencies between queries and cache entries. The process to determine these dependencies is described in further detail in Section 5.5.1, as part of the RDBMS client wrapper.

For example, say the cache contains a key value pair, k_i-d_i, where the key, k_i, is *"Profile-172"* and the value is the HTML profile page for user with the ID *172*. This profile page was generated by issuing queries to the RDBMS, one of which is:

```
SELECT occupation
FROM users
WHERE user_id = 172;
```

The cache has registered the query with the database QCN system and mapped the dependency between the query and the key *"Profile-172"*.

When the following update occurs:

```
UPDATE users
SET occupation = 'student'
WHERE user_id = 172;
```

the RDBMS will detect that the result set for the registered query is affected. A notification is generated as a part of the update transaction and is sent to the cache as soon as it occurs, though possibly with some queuing delay. When the notification is received by the cache, it looks up the mapping between the affected query and cache entries and finds that *"Profile-172"* is dependent on the affected query. The cache entry is invalidated, causing a subsequent lookup for *"Profile-172"* to observe a cache miss.

Thus, when data is modified in the RDBMS, the change is immediately reflected in the cache as a result of the invalidation of dependent cache entries. The following read that

observes the cache miss winds up reading the latest value from the RDBMS and repopulates the cache with the up-to-date value of the entry.

QCN was first seen in Oracle under version 10g R2 as Database Change Notification [63]. It was later enhanced and re-branded as Continuous Query Notification (CQN) in Oracle 11g [64], the latest release. Similarly, Microsoft first introduced Change Notification in SQL Server in the 2005 version [57] and continues to support it in SQL Server 2012 [58], the latest release.

5.3.1 Query Registration And Notification

Queries are registered with the RDBMS by the cache server. The exact interface for this operation differs between the two RDBMS implementations of QCN but they share the fundamental concepts. The RDBMS takes a query string from an application and registers it for change notification. Multiple queries can be registered this way. When a change occurs, the RDBMS notifies the application and provides information that identifies which of the registered queries was actually affected. Thus, the same notification handling software can be used for all the different query registrations.

The rest of this section describes in detail how this is done with Oracle's CQN system. Oracle provides a C/C++ client library to communicate with the RDBMS. Through this interface, the cache server application first obtains a subscription handle and associates with it a callback function in the application. This callback function is called when a change occurs and is used to process the notification. Using the subscription handle, the cache application specifies a query to the RDBMS for registration with CQN. If this operation is successful, the RDBMS will return a unique identifier for the registered query, an 8 byte integer called a *QueryID*. If the application attempts to register the same exact query multiple times, the RDBMS will realize that it is a duplicate and return the same *QueryID*.

As mentioned earlier, the COSAR-CQN framework keeps track the possibly multiple queries used to generate a key-value pair. Each of these queries have to be registered using the mechanism described above. Each unique registration generates a new *QueryID*, which the framework associates with the key-value pair.

When a change occurs in the RDBMS that affects one of these registered queries, a notification is sent by calling the callback function. The notification contains one or more

```
SELECT m.userid, m.email, m.profileImage
FROM   Members m, Frds f
WHERE  f.frdID1=1 and m.userid=f.frdID2
```

5.1.a) Query instance

```
SELECT m.userid, m.email, m.profileImage
FROM   Members m, Frds f
WHERE  f.frdID1=? and m.userid=f.frdID2
```

5.1.b) Query template

Figure 5.1: A query instance to retrieve the friends of Member with userid=1 and its corresponding query template.

QueryIDs [1] and information about the type of change, the tables that were affected, and the rows that were affected. Using these *QueryIDs*, the cache application can look up and invalidate all associated key-value pairs. Thus, at this point the cache no longer holds the stale key-value pair and will force a subsequent lookup for that key to retrieve the up-to-date copy from the RDBMS and repopulate the cache.

When a query notification is received, one possible action is also to unregister the registration of the affected query. A reason to do this might be to reduce the number of registrations that exist in the RDBMS. CQN registrations impose some overhead on RDBMS update, insert and delete transactions because the system has to check these registrations every time while executing the transaction. However, the registration and unregistration operations themselves are expensive and should be avoided while the system is under heavy load. The decision of when to register or unregister a query depends on the frequency of access of the cache entry and modification of the data, as well as the system load.

5.4 Dynamically Generated Triggers (SQLTrig)

A disadvantage of the Trig approach described in Section 5.1.2 is that the application developer or database administrator has to manually author and maintain the trigger code used to invalidate the cache. This section describes a novel transparent KVS consistency technique named SQL Query To Trigger translation, *SQLTrig* for short, that overcomes the limitation of that model.

The input to the SQLTrig's query to translator is a query *instance* issued by an application. The translator consists of the following two components:

1. A query template generator, QTGen: The input to this component is the query instance and its output is a query template, and

[1] In some cases, multiple notifications are bundled into one call.

2. A trigger generator, TrigGen: Its input is a query template and its output is a set of triggers.

In the following, we start by describing what is a trigger. Next, we describe QTGen and TrigGen in turn.

A trigger is a procedure registered for execution with the RDBMS. It is specified on a table, say R, to execute when a row is either inserted in R, deleted from R, or updated. In essence, TrigGen authors software on the fly per query template. The execution of these triggers uses the inserted/deleted/updated row to compute those query instances whose result sets have changed and to invalidate (delete) them from the KVS. A trigger defined on a Table R may not query Table R because this table is in the process of being updated. TrigGen respects this constraint for all its authored triggers. It assumes triggers execute synchronously, returning an error code when it fails to delete a key-value pair (due to intermittent network connectivity). At run time, such failures cause the RDBMS transaction to abort, leaving the KVS and the RDBMS consistent with one another.

To compute the query template of a query instance, QTGen parses the SQL query to identify its selection predicates. These predicates appears in the qualification list (where clause) of the query and might be connected using Boolean logic (and, or, not). They compare an attribute of a table (e.g. Member.id) with a constant (e.g. 47645) using a comparison operator ($=$, $\leq$, $\geq$, $<$, $>$, $\neq$). QTGen replaces the constants with a wild card to compute the query template. Figure 5.1 shows a query instance and its corresponding query template produced by QTGen.

To translate a parsed query templates into triggers, TrigGen identifies the following five types of SQL queries with a "where" clause consisting of:

1. One exact-match selection[2] predicate: TrigGen authors triggers that produce the query instance whose result has changed. See Section 5.4.1 for details.

2. Several exact-match selection predicates connected using the logical *and*: Identical to the discussion of queries with one exact-match selection predicate, see Section 5.4.1 for details.

[2]An exact match is a comparison of an indexed tuple variable with a constant using an equality predicate, e.g., userid="654".

3. One or more join predicates and one or more exact-match selection[3] predicates connected using the logical *and*: TrigGen authors triggers that generate the query instance (key) whose result (value) has changed. The authored trigger may query one or more of the tables in the "from" clause of the query. A trigger does not query its own table that is in the process of being modified. See Section 5.4.2 for details.

4. One or more selection and join predicates connected using the logical *or*: TrigGen uses Boolean logic to break the original query instance into query fragments, each with a distinct where clause resembling one of the previous four cases. Conceptually, the union of the result of the query fragments computes the result of the original query. TrigGen requires the KVS to maintain a hash table that maps each query fragment to the original query (key). Next, it authors triggers to generate a query fragment based on the provided four classifications. When an RDBMS update invokes a trigger, it invokes a KVS method that consumes the query fragment to probe the hash table to identify the key (query instance with the logical *or*) whose value (result set) must be invalidated.

5. One of the previous four cases with the "select" clause of the query using an aggregate such as "count" or "sum": TrigGen employs the translation process of the above classification with one difference. Triggers are authored intelligently based on the aggregate. For example, a query that counts the number of rows should not be invalidated if one record of its referenced table is updated.

Below, we provide details of how TrigGen supports each class of queries in turn.

5.4.1 Exact match selection predicates

Consider the following query with a qualification list consisting of one exact-match selection predicate:

SELECT attr_1, attr_2, ..., attr_n
FROM R
WHERE attr_{n+1}=C_1

[3]Recall that queries purely with join predicates are not appropriate for use with SQLTrig because they are decision support style queries. SQLTrig targets query instances that are selective and large in number.

Its relational algebra equivalent is: $\pi_{attr_1,...,attr_n}(\sigma_{attr_{n+1}=C_1}(R))$. The translation process to generate triggers is as follows. With either an insertion or deletion of a row r, the trigger is authored to embody the query template and replace the wild card with the value of attr$_{n+1}$ of the impacted row r, i.e., r.attr$_{n+1}$. The resulting query instance is the key whose value has changed. The KVS deletes this key.

With an update, TrigGen authors the trigger to execute before update to a row r of Table R. Thus, the trigger may access the attribute value of the old and new version of row r. If r.attr$_{n+1}$ is being modified from C_1 to C_{new} then the result of two different query instances have changed. TrigGen authors an "If(C_1 != C_{new})/Else" statement to detect this by comparing the old (C_1) with the new value (C_{new}). When these two values are not equal, additional code is provided to generate two query instances by replacing the wild card of the query template with two different values: old and new values of r.attr$_{n+1}$, i.e., C_1 and C_{new}. The KVS deletes both keys.

When r.attr$_{n+1}$ is not modified, at least one of the attributes in the projection list, i.e., r.attr$_1$, r.attr$_2$, ..., r.attr$_n$, must be modified in order for the trigger to identify an impacted query (key). This is constructed by replacing the wild card of the query template with the value of r.attr$_{n+1}$.

In its most general form, the query's qualification list (where clause) may consist of k predicates connected using the logical *and*, attr$_{n+1}$=C_1 AND attr$_{n+2}$=C_2 AND ... AND attr$_{n+k}$ = C_k. Extension of the insert and delete trigger authoring process is trivial: k wild cards of the query template are replaced with the respective attribute values (attr$_{n+1}$ to attr$_{n+k}$) of the impacted row r. With an update, every time the value of one or more of the k attributes (attr$_{n+1}$ to attr$_{n+k}$) of a row r changes then two query instances are identified for invalidation. They are constructed by replacing the wild card of the query template with the old and new value of the k attributes of the impacted row r. KVS deletes these keys.

With a query instance converted into its algebraic equivalent, TrigGen performs simple string manipulations (change to uppercase and removal of extra spaces) and sorts the attribute names referenced by its project (and select operator) prior to constructing a query template and authoring triggers. This ensures the same query that is slightly different and issued by different methods of an application produce identical templates and triggers.

5.4.2 Equi-join predicates with one or more exact-match selection predicates

To describe SQLTrig's authoring of triggers for queries with a join predicate, consider the following query:

SELECT R.attr$_1$, R.attr$_2$, ..., R.attr$_n$
FROM R, S
WHERE S.$attr_j$=C1 and R.$attr_{n+1}$=S.$attr_i$

where tables R and S might be Members and Frds tables and C1 is the value 1 in Figure 5.1.a. SQLTrig constructs the algebraic representation of this query: $\pi_{attr_1,...,attr_n}(\sigma_{attr_j=C_1}(S) \bowtie_{R.attr_{n+1}=S.attr_i} R)$. Next, SQLTrig authors two sets of triggers, one for Table R and the other for Table S. The set of triggers is different for each table due the presence of the exact-match selection predicate referencing Table S. Both compute the query instance (key) whose result (value) has changed and should be invalidated. Below, we describe authoring of triggers for each table in turn. Subsequently, we generalize the discussion for complex "where" clauses consisting of an arbitrary number of join and exact-match selection predicates.

SQLTrig authors triggers that handle insert and delete of a row s from the table referenced by a selection predicate (S) as follows. It uses the query template and replaces its wild card with the value of the attribute referenced by the selection predicate, $s._{attr_j}$. With an update of row s, it authors the trigger to replace the wild card with the old and the new value of $s._{attr_j}$, identifying two query instances (keys) whose results (values) have changed. The KVS deletes these keys.

With the table that participates in the join clause and is not referenced by the selection predicate, Table R, SQLTrig authors triggers that handle insert and deletion of a row r as follows. It authors code to perform an exact-match look up of table S by transforming the equi-join predicate to an exact-match lookup: S.$attr_i$= r.$attr_{n+1}$. Note that r.$attr_{n+1}$ is a constant as r is a specific row of Table R (in the process of being inserted or deleted). For each matching record s, the authored trigger employs the value of $s.attr_i$ to replace as the value of the wild card in the query template. This query instance (key) should be invalidated because its result (value) has changed.

Figure 5.2 shows the pseudo-code for how SQLTrig processes a "where" clause consisting of an arbitrary number of equi-join and exact-match selection predicates. It groups predicates based on whether they are equi-join or exact-match selection predicates. Next,

1. Let T = Query template of the query instance
2. Let {P} = Selection predicates
3. Let {J} = Join predicates
4. Combine those selection predicates in {P} referencing the same table into one.
5. For each table R referenced by a selection predicate p in {P} do
    (a) let A = the attribute referenced by p
    (b) Author code to lookup the value of A from the row of R that is being inserted/deleted/changed and substitute for the wildcard in p
    (c) Let {Q} = {P} - p
    (d) For each q in Q
        i. Author code to use all elements of {J} to lookup the value of attribute referenced by q, q.attr
    (e) Author code to use the values computed in the for loop with {Q} to substitute for the wildcards in T
    (f) Author code to invalidate the resulting query instance
6. {S} = table referenced by the join predicate only
7. For each table s in {S} do
    (a) Let {SJ} = All join predicate that reference Table s
    (b) Author code to put value for s.attr in all elements of {SJ}
    (c) For each sj in {SJ} do
        i. Author code to use other elements of {J} to lookup the values of attributes
    (d) Author code to use values computed in Step 7(c)i to substitute for the wildcards
    (e) Author code to invalidate the resulting query instance

Figure 5.2: Pseudo-code for processing join predicates.

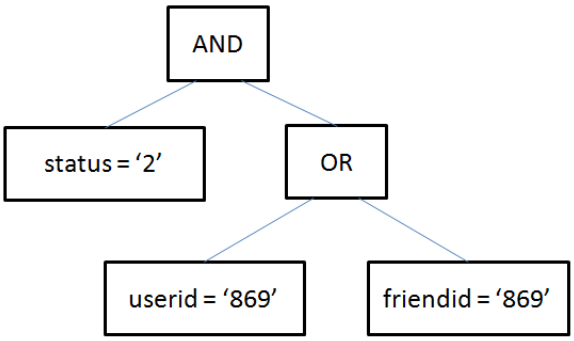


Figure 5.3: Parse tree for a query containing an "or" predicate.

it merges the exact match predicate that reference the same table into one. Subsequently, it generates triggers for those tables referenced by the exact match predicates, Step 5. Finally, it generates triggers for each table referenced by the join predicate, Step 7.

5.4.3 Logical "or" Connectivity

With queries whose "where" clause uses the logical *or* connectivity, TrigGen employs the distributivity property[4] of propositional logic to construct several sub-queries. In order to do so, TrigGen first parses the query and generates a binary tree where the internal nodes consist of either "and" or "or" nodes and the leaf nodes are the selection or join predicates (see Figure 5.3 for an example). The algorithm to parse the tree and construct the sub-queries is described in Algorithm 1. The "where" clause of each sub-query uses the logical *and* connectivity and is different for each sub-query. Logically, the union of the results of these sub-queries computes the same result as the original query.

TrigGen requires the KVS to maintain a hash table that associates each sub-query instance (an intermediate key, IntKey) with the original query instance (key). This hash table is identified by a unique name and is specific to this query template, i.e., it is popuated by the large number of instances of this query template. Subsequently, TrigGen employs the discussions of the previous sections to translate each query into a set of triggers with one difference. The trigger generates both a sub-query instance (IntKey) and the name of the hash table for its query template. The KVS uses the hash table name along with the sub-query instance (IntKey) to identify the query instance (key) whose result set (value) must be

[4]Distribution of conjunction (and) over disjunction (or).

Algorithm 1 Expands the parse tree and returns a list of all possible combinations of disjunct predicates.

```
1:  function EXPANDQUERIES(node)
2:      if node = Leaf Node then
3:          return {node.value}
4:      end if
5:      list ← {}
6:      if node = "AND" then
7:          for left in ExpandQueries(node.LeftChild) do
8:              for right in ExpandQueries(node.RightChild) do
9:                  list ← list.append(left+ " AND " +right)
10:             end for
11:         end for
12:     else if node = "OR" then
13:         for left in ExpandQueries(node.LeftChild) do
14:             list ← list.append(left)
15:         end for
16:         for right in ExpandQueries(node.RightChild) do
17:             list ← list.append(right)
18:         end for
19:     end if
20:     return list
21: end function
```

invalidated.

As an example, consider the following query:

```
SELECT userid
FROM   Friends
WHERE status='2' AND ( userid='869' OR friendid='869' )
```

Using the distributivity property of propositional logic, TrigGen generates the parse tree shown in Figure 5.3 and constructs the following two sub-queries:

1. $\pi_{userid}(\sigma_{(status='2') \ and \ (userid=869)}(Friends))$, and
2. $\pi_{userid}(\sigma_{(status='2') \ and \ (friendid=869)}(Friends))$.

TrigGen directs the KVS to maintain a hash table with a unique name, say X, that maps these two query instances (IntKeys) to the original query (key). Next, it uses the discussions of Section 5.4.1 to author triggers for each sub-query. Once activated, these triggers identify a sub-query string along with the mapping table X. The KVS probes mapping table X with the sub-query string (IntKey) to identify the original query (key) whose result set (value) has been invalidated. In a final step, the KVS deletes this key.

5.4.4 Simple Aggregates

Aggregates such as count are a common query with social networking applications. An example query is one that counts the number of friends for a given user:

```
SELECT count(f.friendid)
FROM   Friends f
WHERE f.userid='869'
```

TrigGen authors triggers by re-writing their target list to eliminate the aggregate. Subsequently, it uses the discussions of the previous 3 sections to author triggers. For example, with the example query, "count(f.friendid)" is replaced with "f.friendid". With "count(*)", the "*" is replaced with the primary key of the referenced table. TrigGen does recognize the presence of an aggregate and, once the triggers are generated, restores the target list of the query (key) generated to its original aggregate. This ensures the trigger produces the correct key for invalidation.

With aggregates that have no where clause, e.g., the sum of all values in a column, TrigGen associates KVS key-value pairs with the name of the reference table and the columns of interest. It authors triggers to generate the table name concatenated with the referenced columns as its output. This invalidates key-value pairs with any change involving those column values on record inserts, deletes and updates. The count aggregate with no qualification list is a special case where the key-value pair is associated with the table name and is invalidated at the granularity of a table change. However, only inserts and deletes generate query instances (keys) as updates do not affect the number of rows.

5.5 SQLTrig Implementation

SQLTrig utilizes the standard JDBC interface to provide the benefits of query result look up using a Key Value Store (KVS) without requiring either an application rewrite or a re-design of the database. The design presented here is in the context of a Client-Server architecture, CS, where the cache manager consists of a client and a server component that communicate via message passing [4, 5, 31] as described in Chapter 3. Typically, key-value pairs are partitioned across the KVS server instances. Hence, a key-value invalidation impacts one server instance. An example KVS system is the widely used memcached [55, 62].

This implementation uses a simple single node CS architecture (see Figure 5.4) and consists of the following:

1. An industrial strength RDBMS named[5] SQL-X.

2. A *SQLTrig server* realized by extending the implementation of the IQ framework of Section 4.2.4 which uses Twitter memcached (Twemcache) Version 2.5.3 [6]. It registers the SQLTrig client provided triggers with the RDBMS using its Trigger Registration component, see Item 4. It caches a key-value pair only when its corresponding triggers have been registered with SQL-X.

3. A *SQLTrig client* with a JDBC interface. It embodies the JDBC driver of SQL-X and Whalin memcached client version 2.6.1 [86]. This component intercepts SQL queries, identifies those that can be translated into triggers, and looks up their result set in the SQLTrig server (described below). With SQLTrig server misses, this component issues

[5]Due to licensing restrictions, the identity of the RDBMS cannot be disclosed and it is named SQL-X.

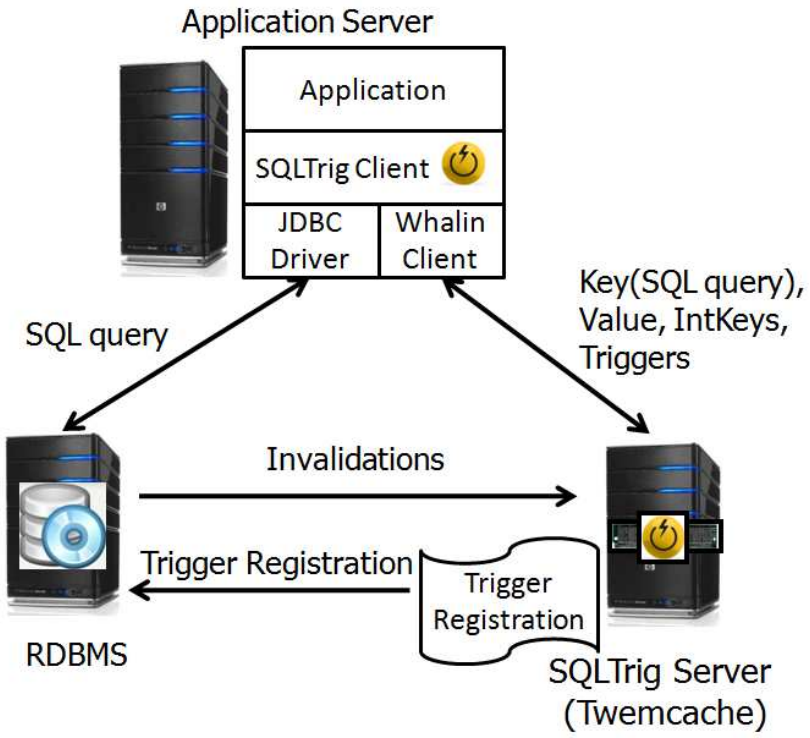


Figure 5.4: The components comprising the SQLTrig architecture.

the query to the RDBMS to obtain its result set, generate triggers for the query, provides the query instance and its result set along with the set of triggers to the SQLTrig Server.

4. A *Trigger Registration* (TR) module deployed with the SQLTrig server. It uses the JDBC driver of SQL-X to register triggers with the SQL-X server. The SQLTrig server uses this module to register triggers provided by the SQLTrig client. SQLTrig server communicates with TR using synchronous message passing.

The following sections describe SQLTrig client and server components in turn.

5.5.1 SQLTrig Client

The SQLTrig client is a wrapper that provides the JDBC interface of SQL-X for use by the application. It employs the JDBC driver of SQL-X to issue queries to SQL-X and the Whalin memcached client version 2.6.1 [86] to issue commands to the SQLTrig Server (extended Twitter Memcached). The client is previewed to all RDBMS queries and update commands

issued by the application including those commands that define transaction boundaries. To respect the consistency guarantees implemented by the developer, SQLTrig does not materialize key-value pairs pertaining to queries issued as a part of a multi-statement transaction. With single-statement queries that fall into one of the categories described in Section 5.4, the client looks up the result set (value) of the query (key) in the SQLTrig server (using its Whalin client). If the server provides a value, the client deserializes it into an instance of result set, and provides it to the application for further processing.

When the server reports a cache miss, the client issues the query to the RDBMS to obtain its result set. Next, it converts the query instance to a query template using QTGen, see Section 5.4. It then proceeds to use TrigGen to author triggers for the query template, {Trigs}. Since authoring triggers is a potentially expensive operation, the client avoids repeating this process for the same query template by maintaining authored triggers locally in a hash table for future lookup. In order to insert the result set in the SQLTrig server, it invokes SQLT-SETIK(k_i, v_i, {Trigs}, {IntKeys}) where k_i is a unique identifier for this key-value pair (the SQL query string itself), v_i is the serialized result set obtained from SQL-X, {Trigs} refers to the set of authored triggers for this query template, and {IntKeys} are the intermediate keys that the triggers will generate to match this instance of the query template.

Additionally, the SQLTrig client implements the IQ framework of Section 4.2 to provide strong consistency. In response to the Get (k_i), the SQLTrig server returns either the value, v_i, for a hit or grants an I lease for k_i in the event of a miss. As described in Section 4.2.4, the token associated with the I lease is maintained seamlessly by the SQLTrig client. The SQLT-SETIK command above passes this token to the server when attempting to insert k_i-v_i. The client wrapper also intercepts DML statements[6] to implement the IQ framework. Before executing the DML statement, the client first assigns a Transaction Identifier(TID) that can be accessed by the trigger body[7].

When executing the DML statement, the appropriate trigger is invoked and passes the TID along with its generated set of IntKeys to the SQLTrig server using Quarantine-and-Register-IntKey, SQLT-QAREGIK (TID, IntKey). This is a new command that resembles the Quarantine-and-Register command of the IQ framework but includes additional handling for IntKeys. This handling is described in Section 5.5.2. The server may fail to grant the Q lease

[6]Data Manipulation Language statements are SQL statements used to insert, delete, or update data in the RDBMS.

[7]The mechanism for passing the TID to the trigger varies with different RDBMS implementations. For example, with Oracle, packages [65] can be used to store values visible within the scope of a transaction.

if it is unable to allocate memory to store the Q lease, in which case the entire transaction rolls back and the client can either try to execute the transaction again from the beginning or abort the transaction altogether. If all Q leases are successfully obtained, the transaction is committed. The client now issues a Delete-and-Release, DaR (TID), to invalidate all associated key-value pairs and release any acquired leases.

The technique used to serialize and deserialize (marshall and unmarshall) the result of SQL queries impacts system performance significantly. Ideally, a marshalling technique must be fast, efficient and produce the most compact serialized representation. With the Java programming language, this can be done by marshalling a serializable version of the JDBC ResultSet class. Since the general Java SQL ResultSet(java.sql.ResultSet) class is not serializable, it has to be converted into an object that does support serialization.

One such method is to employ the CachedRowSet implementation[8] (by Sun, now Oracle) to generate a serializable instance of the query ResultSet class. This instance is populated with a ResultSet obtained by executing a query. Next, this instance is serialized into an array of bytes using the Java writeObject call. The resulting array of bytes is stored as the value portion of a key-value pair in the KVS. It might be compressed to minimize the memory footprint and network transmission time. When unmarshalling this array of bytes after reading it from the SQLTrig server, a corresponding Java readObject call is used to rebuild the original CachedRowSet instance. The Java marshalling and unmarshalling of objects are expensive because they are designed to handle arbitrarily complex classes.

To avoid this overhead, we implemented a custom marshalling of the ResultSet. It outperforms the Java marshalling technique because it is aware of the specific structure of the ResultSet object. It retrieves its number of columns and rows and stores them as the first eight bytes of an array. Subsequently, it stores the meta-data information for a column (name, length, table name, type) and its values for every row, producing a column store representation. Today, with variable length columns such as varchar, its data is stored as a series of {length, value} pair. An alternative representation would be to store all {length} values followed by {value} of the columns. This may produce a more compact representation when compressing our serialized representation.

We used the YCSB benchmark [29] (Workload C) to compare the generic Java marshalling technique with my implementation. YCSB is configured with one table consist-

[8] A commercial RDBMS software vendor may provide its own implementation of CachedRowSet as a part of its JDBC driver, e.g., OracleCachedRowSet. One may use this instead of the generic implementation.

	SQLTrig Marshalling		Generic Java Marshalling	
	No Compression	With Compression	No Compression	With Compression
Average Size (bytes)	1,536	972	7,671	3,787
Avg Latency (μs)	102	117	317	875

Table 5.1: Marshalling of YCSB Workload C ResultSet with SQLTrig and Java.

ing of ten string columns. Each column is 100 bytes long. The target query retrieves all columns of a single row. First row of Table 5.1 shows the average size of the resulting object with both SQLTrig's marshalling technique and the generic Java marshalling technique. The SQLTrig marshalling technique results in representations that are 3 to 4 times smaller in both compressed and uncompressed format. Moreover, the service time[9] to both generate and compress[10] the value is faster with my implementation, see the second row of Table 5.1.

5.5.2 SQLTrig Server

The SQLTrig server is implemented using the C language and extends the implementation of the IQ framework (see Section 4.2.4) using Twitter's memcached version 2.5.3, Twemcache [6]. The extensions implement the indexing in support of two new commands: SQLT-SETIK(k_i, v_i, {Trigs}, {IntKeys}) and SQLT-QAREGIK(TID, IntKey). Both are per SQLTrig client specification, see Section 5.5.1.

The SQLTrig server maintains a hash table of the triggers that have been registered with the RDBMS successfully. When the client issues the SQLT-SETIK command, the SQLTrig server determines if each trigger in the set {Trigs} is found in the hash table of the registered triggers. Next, it checks if all intermediate keys in the set {IntKeys} were already associated with the provided key k_i. If not, it associates each $IntKey_i$ with k_i by storing it as a key-value pair in the KVS, where the key is $IntKey_i$ and the value is a list of one or more associated keys (such as k_i). If this key-value pair is evicted from the KVS, all its associated keys must be invalidated as well[11]. The value v_i is only stored if three conditions are satisfied

[9]In this experiment, the RDBMS, cache server, and the client are hosted on the same PC. While there are inter-process communications, there are no inter-processor communications.

[10]Compression enables a more scalable infrastructure because it frees shared resources such as the cache space and the network bandwidth.

[11]A key is invalidated if it contains a value or an I lease. If it contains a Q lease, the key must be kept.

at the time of the SQLT-SETIK call (a) all triggers were registered, (b) all intermediate keys were already associated with k_i prior to this call, and (c) there exists an I lease for k_i with a matching token. If all conditions are met, k_i-v_i is inserted into the KVS.

If a trigger in the set {Trigs} is not found in this hash table, SQLTrig places the trigger in a registration queue and returns without inserting k_i-v_i in the KVS, i.e., discards k_i-v_i. A background trigger registration thread consumes elements of the trigger queue and issues commands to a Trigger Registration (TR) process to register each trigger with the RDBMS. TR maintains a list of triggers it has registered with the RDBMS and does not register the same trigger more than once. TR is written using Java and uses the JDBC driver of SQL-X to register triggers. It runs continually as a service and detects if it loses connection to the RDBMS (for example, due to the RDBMS restarting). In the event of connection loss, TR will re-connect to the RDBMS and rebuild its list of known triggers by querying the RDBMS. Once it registers a trigger successfully, it returns control to the background thread of the SQLTrig server. This thread inserts the trigger in the hash table of registered triggers and proceeds to the next trigger in its queue.

Quarantine-and-Register-IntKey, SQLT-QAREGIK(TID, IntKey), resembles the Quarantine-and-Register command of Section 4.2.4 and includes handling of IntKeys. When a SQLT-QAREGIK command is received, it looks up all the keys associated with the $IntKey_i$. For each key, k_i, a Q lease is acquired and k_i is associated with the TID. Additionally, $IntKey_i$ is also associated with the TID. When Delete-and-Release, DaR (TID), is called, the server invalidates and releases its lease on all keys associated with the TID. All keys associated with IntKeys that are tied to the TID are also invalidated. This step is necessary because the mapping of IntKey to a key may not have existed when SQLT-QAREGIK was called, meaning that no Q lease was acquired for that key. In the event that another session successfully stores a key-value pair into the KVS for that key, the key must also be invalidated by DaR. Finally, DaR removes the TID from the KVS.

5.6 Evaluation of QCN

This section compares an implementation of Query Change Notification (QCN)[12] with Application Developer Consistency (ADC)[13], RDBMS Trigger Driven (Trig)[14] and Synthetic[15] along three dimensions: 1) man hours required to design, implement and debug an approach, 2) average processing time, 3) served stale data. We first begin with a description of the benchmark used for this evaluation.

5.6.1 RAYS and a Social Networking Benchmark

Recall All You See (RAYS) [35] envisions a social networking system that empowers its users to store, retrieve, and share data produced by devices that stream continuous media, audio and video data. Example devices include the popular Apple iPhone and inexpensive cameras from Panasonic and Linksys. It is deployed on an Amazon EC2 instance with an active community of users. Similar to other social networking sites, a user registers a profile with RAYS and proceeds to invite others as friends. A user may register streaming devices with RAYS and invite others to view and record from them. Moreover, the users profile consists of a Live Friends" section that displays those friends with a device that is actively streaming. The user may contact one or more of these friends to view their stream(s).

For the purpose of evaluation the RAYS system is deployed in 2 different configurations. The first configuration is termed SQL-X, where the system utilizes only the RDBMS to serve all requests. The second configuration is named QCN, where a cache is utilized in a CADBMS architecture using the QCN approach described in Section 5.3. We use two popular navigation paths of RAYS to both describe and evaluate QCN. They are named Browsing friends (Browse) and Toggle streaming (Toggle). While Browse is a read-only workload, Toggle results in updates to the database requiring the cache to remain consistent with the database. We describe each in turn.

Browse emulates four clicks to model a user viewing her profile, her invitations to view streams, and her list of friends followed with the profile of a friend. With SQL-X, Browse issues 38 SQL queries to the RDBMS, see Table 5.2. With QCN, Browse registers 33 distinct queries and issues 8 get operations. For each get that observes a cache miss, it performs a put

[12] See Section 5.3 for a description of QCN.
[13] See Section 5.1.1 for a description of ADC.
[14] See Section 5.1.2 for a description of Trig.
[15] See Section 5.1.3 for a description of Synthetic.

	Operation	Browse	Toggle
SQL-X	SQL Queries	38	23
	SQL Updates	0	3
QCN	put	8	7
	get	8	7
	hits	0	0
	Registered queries	33	23
	Cached key-value pairs	8	7
	SQL Queries	38	23
	SQL Updates	0	3

Table 5.2: Characteristics of two different sequences of page visits and clicks with RAYS using an empty cache.

Term	Definition
N	Number of simultaneous users/threads.
n	Number of users emulated by a thread.
ε	Think time between user clicks executing a sequence.
θ	Inter-arrival time between users emulated by a thread.
ω	Number of users in the database.
u	Probability of a user referencing a Toggle sequence.

Table 5.3: Workload of parameters and their definitions

operation. With an empty cache, the get operations observe no cache hits and this sequence performs 8 put operations.

Toggle corresponds to a sequence of three clicks where a user views her profile, her list of registered devices and toggles the state of a device. The first two result in a total of 23 queries with SQL-X. QCN issues 7 get operations that observe a cache miss with an empty cache. QCN executes 23 queries and perform 7 put operations to populate the cache. With the last user click, if the device is streaming then the user stops this stream. Otherwise, the user initiates a stream from the device. This results in 3 update commands to the database, see Table 5.2. With QCN, these updates invalidate cached entries corresponding to both the profile1 and devices pages. With a populated cache, the number of deletes is higher because each toggle invalidates the Live Friends" section of those friends with a cached entry.

Our multi-threaded workload generator targets a database with a fixed number of users, ω. A thread simulates sequential arrival of n users performing one sequence at a time.

There is a fixed delay, interarrival time, θ, between two users issued by the thread. A thread selects the identity of a user by employing a random number generator conditioned using a Zipfian distribution with a mean of 0.27. N threads model N simultaneous users accessing the system. In the single user (1 thread, N=1) experiments, this means 20% of users have 80% likelihood of being selected. Once a user arrives and her identity is selected, she picks a Toggle sequence with probability of u and a Browse sequence with probability $(1 - u)$. There is a fixed think time ε between the user clicks that constitute a sequence. Table 5.3 contains a summary of the parameters used.

We target a small database consisting of 1,000 unique users, eliminating cache replacement as an experimental variable. A RDBMS update invalidates cached key-value pairs, resulting in a cache hit rate lower than 100%. We measure the time to perform updates with and without QCN, quantifying the overhead of registered queries when the RDBMS processes updates.

The workload generator maintains the structure of the synthetic database along with information about the activities of different users to detect cached data (HTML pages) that are not consistent with the state of the database, termed stale data. The workload generator produces unique simultaneous users accessing RAYS. This means a more uniform distribution of access to data with a larger number of threads. While this is no longer a true Zipfian distribution, obtained results from different alternatives are comparable because the same workload is used with each alternative.

In addition, we present average processing time of a sequence. Processing time consists of the service time to process the pages that constitute a sequence, think time between the clicks, and queuing delays (if any). To illustrate, with a think time of 100 msec, zero service time, and no queuing delay, the minimum processing time for Browse and Toggle sequences is 300 and 200 milliseconds, respectively. This is because Browse emulates 4 user clicks while Toggle emulates 3 user clicks. The first click is the arrival of the first page visit by the user, i.e., incurs no think time.

Due to licensing restrictions, we cannot disclose the identity of the commercial RDBMS product used for our reported performance numbers. The term RDBMS refers to an anonymous commercial product, referred to as SQL-X. This product has the following limitation when multiple threads update the database with tens of thousands of registered queries. The response time of an update increases dramatically from a few milliseconds with one thread to minutes with multiple threads. We anticipate this limitation to be resolved in subsequent

RDBMS releases and avoided it by issuing updates one at a time using our infrastructure.

5.6.2 Software development effort

Synthetic is trivial to implement and we spent less than an hour to implement it with RAYS by employing a global Time-To-Live(TTL) value. QCN is the next simplest technique and we spent approximately 5 hours to fine tune SQL queries used by Browse and Toggle sequences to register at the granularity of query level notification. With ADC and Trig approaches, there was significant overlap in the design of the cache consistency. Moreover, debugging one helps debugging of the other. We spent approximately 90 hours to implement both approaches. Below, we describe the details of this implementation.

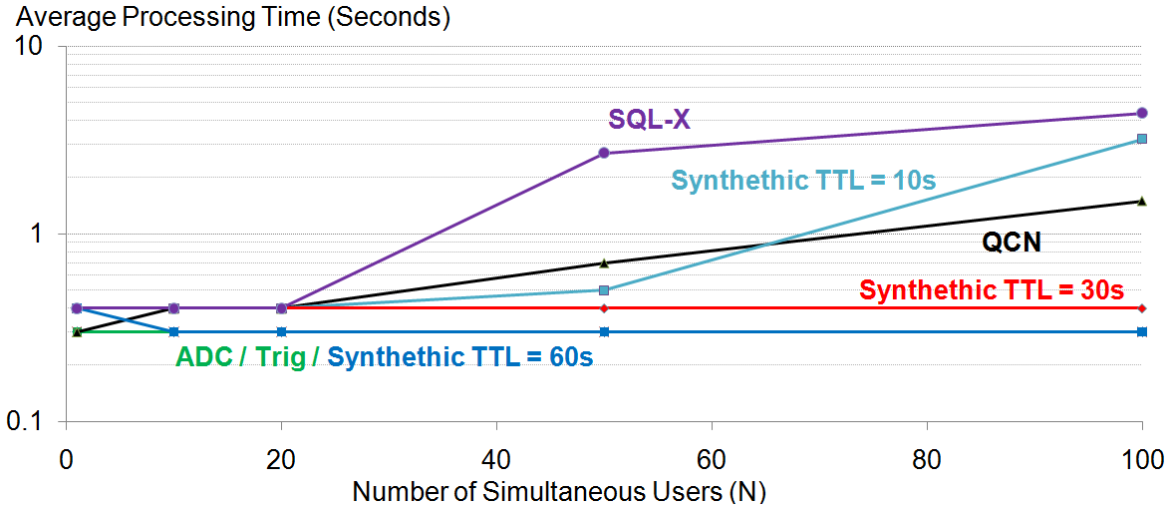
While the benchmarks we conducted were focused on users toggling their devices to start or stop streaming, in the real RAYS system, the user is able to perform other operations, such as modify their profile information or friend relationships with others Such modifications may cause some cached data to no longer be up-to-date and require invalidation to avoid serving stale data. Capturing all possible interactions becomes a tedious process of examining each possible modification and how it impacts the cache entries.

With QCN, all of these cases are automatically covered by registering queries. QCN development requires a programmer to use the monitoring tool to determine which queries are either not registered or registered at the granularity of table notification, and to re-write these queries. Note that the re-written queries do not impact the application logic. QCN used them to associate with a cached key-value pair.

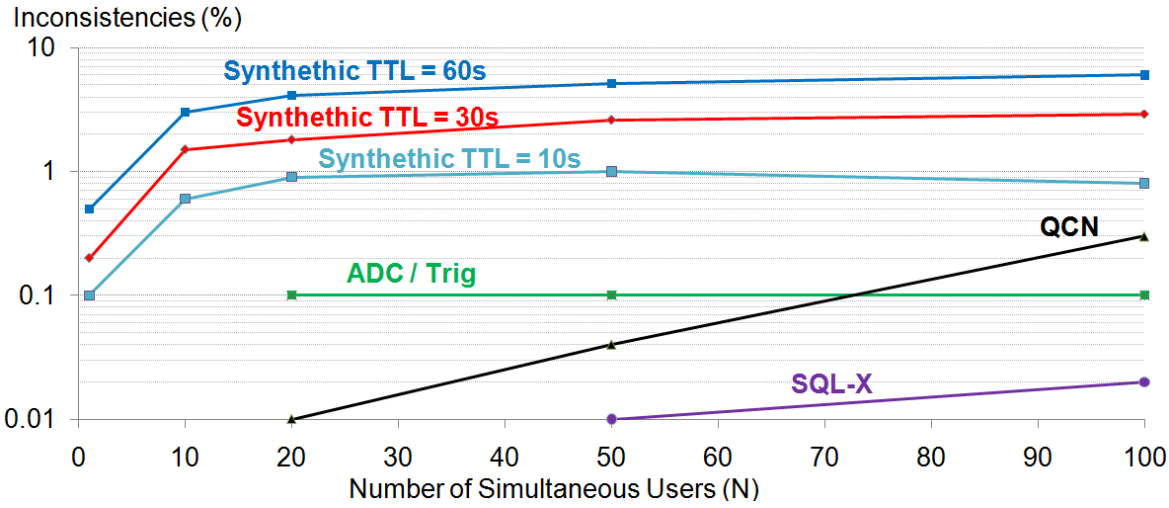
In order to fairly compare the different techniques, we had to develop ADC/Trig invalidation schemes that would cover all the potential modifications by users. Most of the 90 man hours was spent on identifying these scenarios and implementing the appropriate invalidation schemes in the form of database triggers or cache invalidation logic in the application. Issues with the transitive nature of friendship and how it was employed in our environment further complicated the implementation. QCN eliminates this analysis and its associated software.

5.6.3 Processing time and stale data

We analyzed the average processing time of each technique and its percentage of served stale data as a function of the number of simultaneous users, see Figure 5.5. These results pertain to a warm cache, one with a cache hit rate close to 100%. As a reference point,



5.5.a) Average processing time



5.5.b) Percentage inconsistent data

Figure 5.5: Comparison of alternative approaches. ε=100 msec, θ=0, ω=1,000, u=1%, n=10,000.

N	ADC		QCN	
	Toggle	Browse	Toggle	Browse
1	0.2	0.3	2.1	0.3
10	0.2	0.3	4.4	0.3
20	0.2	0.3	9.3	0.3
50	0.3	0.3	35.2	0.3
100	0.4	0.3	98.9	0.3

Table 5.4: Processing time (Seconds) of Browse and Toggle Sequences. ε=100 msec, θ=0, ω=1,000, u=1%, n=10,000.

we also present numbers from SQL-X. With this approach, the most up-to-date data should be retrieved directly from the RDBMS during every sequence. However, there are race conditions in the workload generator between the point of retrieval and verification. They results in a small amount of inconsistency with a high number of simultaneous users, see Figure 5.5.b.

Overall, both ADC and Trig approaches provide the best processing time and request rates. Moreover, they produce least amount of stale data[16]. Synthetic is sensitive to the value of TTL. A high TTL value (60 seconds) enables Synthetic to approximate the performance of the ADC and Trig approaches, see Figure 5.5.a. However, it produces the highest amount of stale data, see Figure 5.5.b. A lower TTL value minimizes the amount of stale data and increases the processing time. In general, it is challenging to decide a value for TTL.

With the number of simultaneous users, N, as 50 and 100, QCN results in a higher average processing time when compared with ADC and Trig approaches. This is because its tens of thousands of registered queries cause the RDBMS to processes SQL update commands slower. As shown in Table 5.4, the average processing time for a Toggle sequence is significantly higher with QCN. Furthermore, the time required to perform a Toggle sequence increases dramatically with a higher number of simultaneous users. This increase is largely due to queuing delays as the workload generator was restricted to issue one update at a time, see the last paragraph of Section 5.6.1. On the other hand, the average processing time of Browse with QCN remains low in all configurations.

With 1 user, N =1, the response time of updates with QCN is slower than SQL-X. This appears to be a limitation in the latest release of the RDBMS and we anticipate it to be

[16]With no consistency technique, the percentage inconsistency observed is 15%, 26%, 32%, 34%, and 42% with 1, 10, 20, 50, and 100 simultaneous users, respectively

BG Action	Read Only	Very Low (0.1%) Write	Low (1%) Write	High (10%) Write
View Profile	40%	40%	40%	35%
List Friends	5%	5%	5%	5%
View Friends Requests	5%	5%	5%	5%
Invite Friend	0	0.04%	0.4%	4%
Accept Friend Request	0	0.02%	0.2%	2%
Reject Friend Request	0	0.02%	0.2%	2%
Thaw Friendship	0	0.02%	0.2%	2%
View Top-K Resources	40%	40%	40%	35%
View Comments on Resource	10%	9.9%	9%	10%

Table 5.5: Four mixes of social networking actions with BG.

resolved in future releases. We are almost certain that the performance of updates with QCN will improve as query notification feature of commercial RDBMSs matures, rendering QCN viable for a larger number of simultaneous users and registered queries (data set sizes).

In sum, QCN expedites software development cycle to enable organizations to quickly develop and deploy features. The limitation of QCN with updates in current RDBMSs motivates the need for a different approach to transparent consistency, which is studied in SQLTrig.

5.7 Evaluation of SQLTrig

This section employs the BG [12] benchmark to compare the performance of an industrial strength relational database management system named SQL-X with itself and when extended with SQLTrig, see Section 5.5. As a comparison yard stick, we present performance of SQL-X with Twemcache and developer provided software to maintain the key-value pairs consistent with the tabular data. Below, we provide an overview of the BG benchmark. Subsequently, Section 5.7.2 characterizes the queries (keys) and result sets (values) generated by this benchmark. Section 5.7.3 presents the social action rating (SoAR) of the alternative configurations using BG. These results demonstrate that SQLTrig enhances the performance of SQL-X by more than two folds while providing physical data independence.

5.7.1 BG Social Networking Benchmark

BG [12] is a benchmark to quantify the performance of a data store for processing interactive social networking actions and sessions. These actions and sessions either read or write a very small amount of the entire data set. In addition to response time and throughput, BG quantifies the amount of unpredictable data produced by a data store. This metric refers to either stale, inconsistent, or invalid data produced by a data store. This is particularly useful because it enabled us to experimentally verify SQLTrig produces no stale data.

BG computes a Social Action Rating (SoAR) of a data store based on a pre-specified service level agreement (SLA) by manipulating the number of threads (i.e., emulated members) that perform actions simultaneously. SoAR is the maximum system throughput (actions per second) that satisfies the SLA. All SoAR ratings in Section 5.7.3 are established with the following SLA: 95% of requests observe a response time of 100 milliseconds or faster with no unpredictable (stale) data.

In the reported experiments, BG constructs a database consisting of either 10,000 or 100,000 members with 100 friends per member[17] and 100 resources per member. BG emulates members as socialites using a Zipfian distribution with exponent 0.27. This means roughly 20% of the members perform actions of Table 5.5 as socialites.

Table 5.5 shows the interactive *actions* of BG which are common to many social networking sites [12]. This table shows the four different workloads that we explore in this study. A read-only workload that performs no write actions and three different mix of read and write actions with the percentage of write actions varying from 0.1% to 10%. The workload of social networking applications is dominated ($>$ 99%) by read actions [9, 62].

All results reported below were obtained using eight nodes with the following specifications: Windows Server 2003 R2 Enterprise x64 bit Edition Service Pack 1, Intel® Core™ i7-2600 CPU 3.4 GHz, 16 GB RAM, Seagate 7200 RPM 1.5TB disk. The BG clients execute on six nodes preventing the benchmarking infrastructure from becoming the bottleneck. Two different nodes host the RDBMS and the KVS (either Twemcache server or SQLTrig server). These nodes communicate using a 1 Gigabit Ethernet switch.

BG Action	Size of key-value pairs in bytes			
	Custom Marshalling		Java Marshalling	
	Uncompressed	Compressed	Uncompressed	Compressed
View Profile	1,197	703	7,420	3,486
List 100 Friends	82,836	50,740	107,883	55,744
View Friends Requests (Empty)	338	111	5,778	2,411
View Top-5 Resources	1,865	1,198	8,351	4,110
View Comments on Resource (Empty)	185	91	5,359	2,365

Table 5.6: Size of key-value pairs produced by different BG actions.

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

Friends($\underline{\widehat{frdID1}, \widehat{frdID2}}$)

PdgFrds($\underline{\widehat{inviterID}, \widehat{inviteeID}}$)

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

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

Figure 5.6: SQL-X database design with no images. Two records in the Friends table represents the friendship between two members. The underlined attribute(s) denote the primary key of a table. Attributes with a hat denote the indexed attributes.

5.7.2 Size of key-value pairs

Table 5.6 shows the different BG actions and the characteristic of their produced key-value pairs. The *View Profile* action of BG consists of a SQL query that retrieves (1) the profile attributes of a member such as her first name, last name, picture, etc., (2) her number of friends, (3) her number of pending friend invitations, and (4) her number of resources. Per the schema of Figure 5.6, the "where" clause of the SQL query is an exact-match selection predicate referencing *userid* attribute of the Members table. As shown in Figure 5.6, aggregated information for items (2), (3), and (4) are represented as attributes. These counts are kept up-to-date as part of the stored procedures that modify the database state in response to actions such as Thaw Friendship and Invite Friend.

The SQL query that implements *List Friends* requires an equi-join between Friends and Members tables with an exact-match look up using the userid of the member whose friends is being listed. The SQL query for View Friend Request is similar and uses the PdgFrds table.

View Top-5 Resources is implemented using a SQL query that employs a range predicate on the priority attribute of the Resource table. The SQL query that implements *View Comments on Resource* consists of an exact-match selection predicate using the rid attribute of the Resource table.

Table 5.6 shows that the custom marshalling technique of Section 5.5.1 results in a more compact representation than the generic Java marshalling technique. This is consistent with the YCSB results shown in Table 5.1.

Actions that write to the RDBMS (such as *Invite Friend*, and *Thaw Friendship*, see Table 5.5) invoke SQLTrig's authored triggers to invalidate the impacted key-value pairs. A subsequent reference for these key-value pairs observes a KVS miss, is redirected to the RDBMS for processing, and a new key-value pair is inserted in the KVS. The keys invalidated by each write action are enumerated in Table 5.7.

5.7.3 Social Action Rating

This section reports on the Social Action Rating (SoAR) of the following three different configurations:

1. An industrial strength RDBMS named SQL-X by itself,

[17] 100 is the median number of friends for a Facebook member [83, 10].

BG Write Action	Number of Keys Invalidated	Keys Invalidated
Invite Friend (Inviter, Invitee)	2	View Profile for Invitee View Friends Requests for Invitee
Reject Friend Request (Inviter, Invitee)	2	View Profile for Invitee View Friends Requests for Invitee
Accept Friend Request (Inviter, Invitee)	5	View Profile for Invitee View Profile for Inviter View Friends Requests for Invitee List Friends for Invitee List Friends for Inviter
Thaw Friendship (Inviter, Invitee)	4	View Profile for Invitee View Profile for Inviter List Friends for Invitee List Friends for Inviter

Table 5.7: Keys invalidated by SQLTrig's authored triggers when processing a BG write action.

2. SQL-X configured with SQLTrig per discussions of Section 5.5, and

3. SQL-X configured with Twemcache and maintained consistent using developer provided software. This deployment is named application developer consistency, ADC, and is implemented as follows. It represents query instances and their results as key-value pairs. It extends the write actions of BG by identifying impacted query instances (keys) whose results (values) have changed and issuing Twemcache delete calls for these keys.

SQLTrig's authored triggers compute the same set of keys as those deleted by ADC. Hence, ADC is comparable to SQLTrig with one key difference: the BGClients issue the delete calls directly to the KVS and there are no authored/registered triggers. One may use the performance observed with ADC as a measuring yard stick to quantify the overhead of the triggers authored by SQLTrig and executed by the RDBMS in the presence of updates.

Below, we compare the SoAR of the three alternatives using both a small and a large social graph consisting of 10,000 and 100,000 members, respectively. With both graphs, each member has 100 friends and 100 resources. The small database is two Gigabytes in

Workload	10,000 Members			100,000 Members		
	SQL-X	ADC	SQLTrig	SQL-X	ADC	SQLTrig
Read Only	27,856	62,025	62,103	25,411	63,292	62,449
0.1% Write	23,144	62,479	62,051	21,584	61,032	62,594
1% Write	16,292	61,333	61,612	13,227	22,418	21,763
10% Write	-	-	-	10,055	14,404	12,004

Table 5.8: SoAR, actions per second, of SQL-X by itself, extended with Twemcache that is maintained consistent using developer provided software, and using SQLTrig. Results are shown for two different social graphs consisting of 10,000 members and 100,000 members. Each social graph consists of 100 friends per member and 100 resources per member.

size and fits in the memory of the server hosting SQL-X configured to be 6 Gigabytes in size. The large graph is 14 Gigabytes in size and does not fit in the memory of the RDBMS server.

Table 5.8 shows the SoAR of the alternative configurations with the small social graph (10,000 members). With the alternative workloads of Table 5.5, the cache augmented architecture enhances the performance of SQL-X more than two folds. SQLTrig and ADC provide comparable performance with the read-only, 0.1% and 1% update workloads. This is because the network bandwidth of the server hosting the KVS (3 Gbps) is fully utilized, dictating the observed SoAR. The CPU cores of the cache server are less than 50% utilized in these experiments. This means configuring the cache server with additional networking cards would enable it to support a higher SoAR rating. The workload with a 10% mix of write actions was not run with the small social graph because the data sharded across 6 BG Clients could not accurately satisfy that ratio of write actions. With a skewed workload (Zipfian exponent of 0.27) and a high throughput, friendship relationships between members in a small shard would be exhausted, leading to a write action ratio lower than 10%. This limitation was not observed with the larger social graph (100,000 members).

The 100,000 members column of Table 5.8 shows the SoAR of alternative configurations with the large social graph. With the alternative workloads of Table 5.5, the cache augmented architecture enhances the performance of SQL-X more than two folds. SQLTrig and ADC provide comparable performance with the read-only and 0.1% update workloads. This is because the network bandwidth of the server hosting the KVS (3 Gbps) is fully utilized, dictating the observed SoAR.

With a higher mix of write actions (1% and 10%), the server hosting SQL-X becomes disk-bound, forming a read and write queue to the disk. With SQLTrig, execution of some triggers (e.g. triggers generated from a query containing an equi-join predicate) require the trigger to query the RDBMS to identify impacted queries whose results are no longer valid. This causes SQLTrig to provide a lower SoAR than the developer provided solution (see last 2 rows of Table 5.8 for 100,000 members) when the RDBMS becomes disk-bound since the additional querying further exacerbates the bottleneck on the disk. This overhead translates to SQLTrig performing slower than ADC by 3% with the 1% update workload and 16% with the 10% update workload. ADC invalidates key-value pairs in the application (each client of BG), imposing a lower load on SQL-X to outperform SQLTrig.

In sum, a developer provided solution such as ADC outperforms SQLTrig when the server hosting SQL-X becomes disk-bound. This is due to the overhead of SQLTrig's use of triggers. With a lower ratio of write actions, this overhead is negligible (3%) but it becomes significant with a larger ratio of write actions when the disk becomes a much greater bottleneck to performance. Other experiments that fully utilize the network bandwidth of the KVS server show that ADC and SQLTrig provide comparable performance. This is because their percentage difference is less than 2% and we attribute this to experimental noise.

Chapter 6

Correctness of SQLTrig

SQLTrig supports consistent reads and produces a serial schedule of executed transactions due to 3 invariants presented in this section. These differentiate between read/write read/write operations of the RDBMS and the KVS. With the RDBMS, these operations pertain to transactions. With the KVS, these operations are at the granularity of get, put, and delete key-value pairs. In the case of query result caching with SQLTrig, a KVS get is equivalent to the execution of one read transaction. A serial schedule is at the granularity of transactions.

6.1 Properties

The invariants are realized based on an implementation of SQLTrig that satisfies the following properties:

1. RDBMS is configured to ensure ACID properties of transactions with no dirty reads, dirty writes, or un-repeatable reads.

2. Prior to populating the KVS with a key-value pair, SQLTrig registers triggers associated with the key-value pair and establishes the mapping between ITs and the key.

3. SQLTrig does not cache the result of queries that are a part of a multi-statement transaction.

4. RDBMS synchronously executes (SQLTrig authored) triggers as a part of a transaction that updates the database. During the execution of the trigger, readers of the affected rows are blocked and have to wait for the completion of the write transaction invoking

the trigger. Once a trigger invokes the KVS server to delete an IT, the KVS server must delete the corresponding key and return success. If this fails then the trigger fails and the transaction aborts. In order for a transaction to commit, all its invoked triggers must execute successfully. This is the invalidation technique of Section 4.2. (Refresh technique produces stale reads.)

5. SQLTrig employs the IQ framework of Section 4.2 to detect and resolve write-write conflicts that occur due to the coupling of RDBMS and KVS that impact the correctness of a subsequent read transaction that observes a KVS hit. When the application observes a KVS miss for a query, it executes a read transaction against the RDBMS and stores its resulting key-value pair in the KVS with a put operation. This read transaction may race with a DML transaction that invokes a trigger to delete the same key-value pair. The trigger delete may occur prior to the read transaction inserting its stale key-value pair in the KVS, causing the KVS to contain stale key-value pairs. The IQ framework enables the KVS to detect this race condition and ignore the put operation. This ensures the application will observe either a key-value pair that is consistent with the tabular data or a KVS miss that redirects it to issue a transaction to the RDBMS.

6. With RDBMSs configured to use Snapshot Isolation [77], a get that observes a miss and races with an RDBMS update may compute a stale value. The IQ framework prevents the stale value from this get operation from being inserted in the KVS.

6.2 Invariants

Invariant 1: All key-value pairs produced by the KVS at time T_1 are consistent with the state of the database at time T_1, reflecting all committed transactions up to T_1.

Three properties guarantee the correctness of this invariant. First, Property 2 ensures a transaction that updates the RDBMS invalidates the corresponding cached key-value pair. Second, Property 4 ensures a transaction does not commit until the invalidation is complete. If the body of the trigger fails then the RDBMS aborts the transaction, leaving the state of the database consistent with the cached key-value pairs. This guarantees a thread observes its own updates to the database because, once it issues a transaction, it cannot proceed until its RDBMS update is reflected in the KVS. Thus, for all committed transactions, triggers would

have invalidated all impacted key-value pairs. One or more of these invalidated key-value pairs may become KVS resident soon after an invalidation because a subsequent reference for them observed a KVS miss, issued transactions to the RDBMS, computes these key-value pairs, and inserted their most up-to-date version in the KVS. These entries are consistent with the state of the database and reflect all committed transactions.

Third, Property 5 detects and resolves KVS put-delete (i.e., write-write) race conditions that cause the cached key-value pairs to become inconsistent with the tabular database.

Invariant 2: No key-value pair in the KVS reflects uncommitted transactions (both mid-flight and aborted transactions).

Property 1 prevents data from a mid-flight DML transaction to be visible to other concurrently executing transactions. This prevents both dirty reads or un-repeatable reads, guaranteeing computed key-value pairs reflect result of queries computed using a consistent database state.

A mid-flight DML transaction may abort and result in one of two possible scenarios. First, the transaction aborts before causing the trigger to fire. In this case, the contents of the KVS and the state of data in the RDBMS will be unchanged and consistent with one another. Second, the transaction aborts after the trigger fires and executes its invalidation code, purging the cached key-value pair. In this case, the invalidation is unnecessary because the state of the database is unchanged (aborted transaction is rolled back). However, while the unnecessary invalidation may degrade the performance of the system, it will not violate the consistency of the framework because subsequent requests will result in a KVS miss and be directed to the RDBMS.

Invariant 3: Read-write conflicts due to concurrent transactions manipulating the same data item are serializable.

Consider two transactions that access the same data item D_i. One transaction reads D_i while the second updates D_i. Their concurrent execution results in two possible scenarios. In the first scenario, the reader observes a KVS miss (because the writer deleted D_i from the KVS) and is re-directed to the RDBMS which guarantees the serial schedule between the reader and the writer. Property 6 ensures the reader does not insert a stale value in the KVS due to the use of MVCC techniques such as Snapshot Isolation. In the second scenario, the reader consumes D_i from the KVS and the writer deletes it subsequently. In this case, the reader is ordered to occur prior to the updating transaction to produce a serial schedule.

In summary, these invariants guarantee that SQLTrig produces a serial schedule of trans-

actions. Due to Invariant 1 and Invariant 2, a request issued against the KVS will produce the same result as if it were issued against the RDBMS, at any point in time. With Invariant 3, race conditions that occur during read-write conflicts are resolved in a manner that yields a serial schedule. Along with Property 1, SQLTrig, provides for ACID properties.

Chapter 7

Future Research

While the initial implementation of SQLTrig has been shown to enable transparent caching in Cache Augmented Database Management Systems (CADBMS), there remain many interesting future extensions. These include the design and implementation of a scalable, highly available and elastic SQLTrig system using a multi-node deployment. In addition, one may extend both the queries supported by SQLTrig and the IQ framework. The following sections discuss these research questions in turn.

7.1 Scalability of the Cache Layer

The current implementation of SQLTrig functions with a single KVS node as its cache (a modified version of memcached [8]). The node is main-memory based and can be accessed significantly faster than the RDBMS, but if the load is high enough, even a single main-memory node can be overwhelmed. The scalability of a system is important when it comes to handling an increasing workload. An interesting research direction is to design and implement a scalable, highly available, and elastic SQLTrig. A preliminary design is described below.

A scalable SQLTrig utilizes multiple KVS nodes arranged in a cluster. Ideally, the load should be distributed evenly across the nodes such that any one node is not overwhelmed. To realize elasticity, the addition and removal of nodes should not require a shut-down of the entire system. Here, we assume the system is deployed in a data center-like topology, where all KVS nodes can be accessed by any client as well as any other KVS node.

As shown in Figure 7.1, the cluster features a horizontally hash partitioned key-space

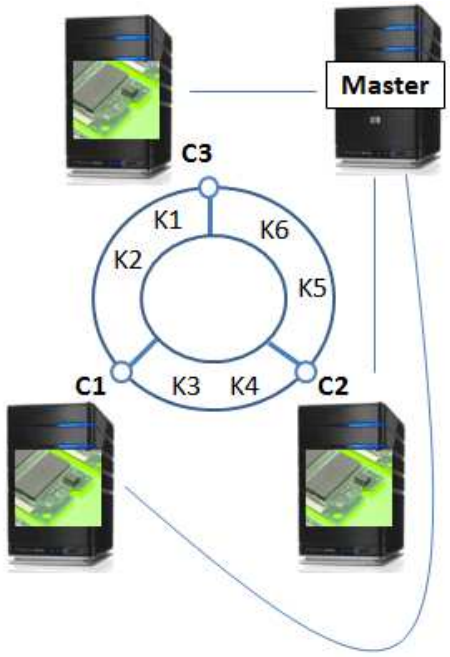


Figure 7.1: Distribution of the key space across 3 KVS nodes (C1, C2, and C3) in a cluster. The master node keeps track of all KVS nodes.

with no replication, similar to distributed hash table designs such as Chord [78] and CAN [69]. While these designs are decentralized, one may simplify the design using a centralized master node. Below we describe one such design.

Each KVS node is responsible for a portion of the hash space. The hash space is partitioned into P fragments. For example, consider the example shown in Figure 7.1 where P = 3. Partition 1 contains keys K1 and K2. The KVS node C1 is responsible for this partition. In practice, with S as the number of nodes, P should be much larger than S to allow for a better distribution of load across the nodes. This simplifies operations to invalidate all keys in a partition or disable access to them.

There is a centralized master node (e.g.,a ZooKeeper [7]) that is in charge of node additions and removals. The master node is aware of all KVS nodes and the portion of the hash space that each node is responsible for. When a node is added, it contacts the master node for a portion of the hash space and a list of all active nodes with their partition assignments. The master node re-assigns partitions across the nodes, taking the new node into account. When re-assigning partitions, 2 different approaches can be taken for existing data in any re-assigned partitions:

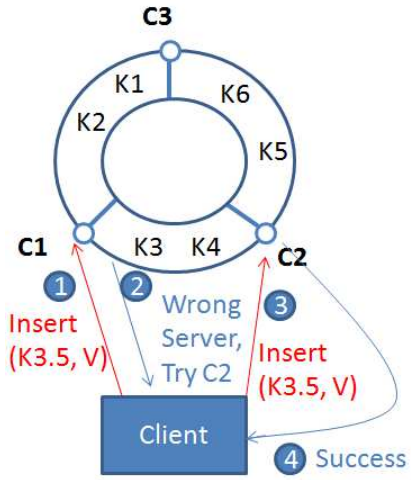


Figure 7.2: Insert procedure.

1. Invalidate all keys in the affected partitions. Since the KVS functions as a secondary store of the data, invalidating all keys in the partition would maintain the consistency of the data. Clients accessing the KVS will re-populate the key-value pairs over time. However, this course of action comes at the cost of reduced throughput when the keys are initially invalidated.

2. Migrate all keys in affected partitions to their new node. To improve the cache hit rate, migrating the key-value pairs to their new destination will allow the system to continue to serve requests for those keys. This might become an expensive operation if the system is incurring a heavy load. Internal key (IntKey) to Key mapping either has to be migrated as well, or the IntKey to Key mapping should always be shared among all KVS nodes.

Each KVS node is aware of other active nodes. When the master node changes the cluster, it notifies each KVS node of the new partition assignments. If a request arrives at the wrong node, it is handled differently depending on the type of request. If a delete request arrives at the wrong node, it is forwarded to the responsible node. The response is returned through the node first accessed. When a list of IntKeys is sent to a KVS node from the RDBMS, if that node responds with success, the RDBMS can assume that if a get/insert request initially arrives at the wrong node, the requester is redirected to the correct node. This has the effect of simplifying RDBMS invalidations.

ColumnName	*Type*	*Description*
ServerId	INTEGER	Unique Identifier for KVS server
HostName	VARCHAR(256)	KVS server hostname.
Port	INTEGER	KVS server port.
Active	BOOLEAN	Current status of the KVS server.

Table 7.1: CacheServers table.

As an example, say a client is trying to insert a key into the system, see Figure 7.2. The client first attempts to insert the key, K3.5 which maps to partition 2, to a KVS node, C1 (if no nodes are known to the client, it may contact the Master node). C1 is the wrong node and will respond with the node address it believes is responsible for that key using its partitioning information about its neighbors. In this example, C1 responds with "Wrong Server, Try C2". The client sends its insert command to C2. If this request succeeds, the value is stored in the KVS. The client library keeps track of C2 being in charge of that partition so that in the future, requests for partition 2 can be directed to this node first. If the request failed again, the client now contacts the Master node for the responsible node for the intended partition. The get operation goes through a similar process as well. If Gumball is enabled but the insert is against a different server from when the initial Get miss occurred, the value should not be inserted. This is because Gumball only works with local server timestamps.

The RDBMS maintains a table with a list of active KVS nodes. This table is named CacheServers and may contain the columns of Table 7.1. When a trigger is invoked, the deletes are issued against any of the active nodes. If a node fails to respond, it will be marked as inactive. The Master node will maintain the table up-to-date by adding/removing entries that identify different nodes. When a trigger fires, it first checks the CacheServers table. If the table is empty, the trigger returns. Otherwise, it select the first ACTIVE entry from the CacheServers table and sends a list of IntKeys to the KVS server. The list of IntKeys corresponds to the data modified by that write transaction. If the response of the KVS node is a success, the trigger was invoked successfully and the transaction may proceed with its execution. Otherwise, if the response was a failure, the trigger should restart and try again with either the same CacheServers table entry or a different entry. The trigger may re-try up to N times with N different entries in the CacheServers table. If it still fails, the RDBMS transaction is aborted.

When a KVS node receives a list of IntKeys to invalidate, it broadcasts the list to all active KVS nodes. Each KVS node then invalidates any entries in its KVS based on its local

IntKey to Key mapping. The benefit of this approach is that each KVS node only needs its local IntKey to Key mapping to determine the affected keys. However, the downside is that every invalidation is broadcast to all nodes always. Alternatively, the IntKey to Key mapping can be shared among all active nodes. Invalidations can be selectively sent to nodes that are responsible. This requires that the mapping is shared among all the nodes when it is generated. Furthermore, new nodes added to the system have to be aware of existing mappings as well.

7.2 Data Availability

Middle-tier caches are designed to utilize commodity off-the-shelf servers to lower the cost of deployment. However, these inexpensive servers are subject to failures.

Robust software is required to anticipate and handle these failures effectively. The availability of a system is an important aspect of data intensive applications and any amount of down time can cost businesses losses in the order of hundreds of thousands of dollars an hour [66]. This means that the system should continue operation in the presence of failures by keeping the data alive.

One key insight when dealing with failures in a CADBMS is that if a KVS server fails, the data still exists on the backing store, which could be a RDBMS that supports ACID properties. Data is never lost since the RBMDS always contains the authoritative copy of the data and requests can always be re-directed to the RDBMS. Since additional load is placed on the RDBMS, the system will likely experience diminished performance when recovering from such failures. Nevertheless, the system will still be able to function and service requests.

In order to address this issue of fault tolerance, three types of commonly occurring failures need to be handled.

1. Transient and permanent node failure.

   A node failure occurs when one of the servers enters an error state and does not respond to requests. A *permanent node failure* is when this error state persists and cannot be resolved without input or interference from the system administrator. The cause of the error state could be software (e.g. logical error causing an infinite loop) or in the hardware (e.g. hard disk or network card failure).

A *transient node failure* exhibits symptoms of a permanent node failure, but only temporarily. When the cause of the transient node failure ceases, operation of the server resumes as normal. For example, a hard disk to lock up temporarily due to malfunction and cause the node to fail to service requests. After some time, the malfunction could cease, causing the node to return to normal operation. Transient node failures are different from permanent node failures because the server will still contain cached objects after it recovers from the transient failure. Because of this, care must be taken to ensure that invalidations are handled properly and transient failures do not result in an inconsistent state of data.

2. Connection failure (due to congestion or packet loss).

   A client may also fail to communicate with the server when the underlying connection used for communication cannot successfully transmit a message. This may occur due to network congestion, where multiple simultaneous connections compete for the limited network bandwidth. Packets might be dropped, resulting in a timeout being observed by a request. When the request times out, it can be re-transmitted but under certain network conditions, the request may not make it to the server due to prolonged congestion.

   When a request fails to be transmitted, failure handling is performed differently depending on the type of request that failed. A failed Get is treated as a Get miss [62]. This results in a slower service time for the application's request since it couldn't access the cached value and has to query the RDBMS. Furthermore, after recomputing the value, the client will attempt to store it into the KVS, even though the value already exists. However, while the performance of the system is reduced, the correctness of the system is not affected. This failure scenarios is addressed using client-level retransmissions (separate from TCP retransmissions). When a failure is detected, the client attempts to re-transmit its request a certain number of times until the request succeeds or the number of re-transmissions reaches a threshold. More stringent methods of dealing with the failure are possible, but the cost of such techniques would outweigh their benefit.

   Failure of a Delete or invalidation requests however may result in stale data if the system does not handle them properly, thus impacting the correctness of the system [62]. If a Delete fails to remove a key-value pair, k_i and the KVS is allowed to continue serv-

ing that entry, subsequent Get requests for k_i may result in stale reads. In the SQLTrig framework, a key-value pair is only re-evaluated when a client observes a cache miss for that entry. Thus, if data is modified in the RDBMS but the corresponding invalidation message did not make it to the KVS server, the KVS will continue to serve that stale value until another modification is made which invalidates that key-value pair. The system must either ensure that the invalidation propagates to the affected KVS server or reconfigure itself to avoid clients accessing the KVS server with stale data.

3. Network partition.

   A network partition results in a grouping of nodes such that nodes in one group may not communicate with nodes in another group. These separate groups are termed to as *partition islands* where all the nodes within an island may communicate with one another, but none of the nodes can communicate with or is even aware of a separate island.

   Network partitions have to be considered and handled differently from connection or node failures. Within each island, it will appear as though the missing nodes have failed and the system will mend itself individually within those islands. In the presence of updates, the different islands may start to report a different and inconsistent view of the system and its data. The possibility of network partitions affects the consistency and availability guarantees that a system provides. As stated by the CAP theorem [19], in the presence of network partitions, a system may not provide both consistency and availability. Thus, a solution to handling network partitions must decide whether to sacrifice consistency or availability of data.

7.2.1 Proposed Solutions

One possible design utilizes the CacheServers table of Section 7.1. See Table 7.1 for a description of the table schema. The table is added to the RDBMS that keeps track of the available KVS servers and their status. It is created and maintained by the Master node and can be looked up by the triggers to decide where to send an invalidation.

This table offers multiple options for triggers to route their invalidation messages to the cache layer. The system is able to tolerate node failures by allowing the trigger to re-send its invalidation to other nodes if an earlier attempt was sent to a failed node. Heartbeat

messages are used by the Master node to keep track of the status of KVS nodes and update the CacheServers table.

In the event of network connectivity issues between the RDBMS and a KVS node, the KVS node may continue to serve requests to clients, providing data availability at the expense of sacrificing consistency. In order to ensure consistency, the KVS node should instead stop serving client requests as soon as it detects that it no longer has connectivity to the RDBMS. Additionally, the data contained in the node has to be purged as it is unaware of any missed invalidations from the RDBMS. This diminishes system performance during the time the KVS instance is re-populated with key-value pairs. This recovery period might be unreasonably long given today's memory capacities, i.e., tens and hundreds of Gigabytes, motivating the following two alternatives.

Deferred Invalidations Variant

A second possible approach is for the RDBMS to detect network dis-connectivity with a KVS instance and store the notifications for this KVS in a table. Once connectivity is re-established, the KVS contacts the RDBMS and reads the table to update its key-value pairs (while processing in-progress notifications due to on-going RDBMS updates). It starts to process request once it completes applying all the notifications buffered in the RDBMS table. A key question with this technique is how long should the RDBMS accumulate notifications? Given the inexpensive price of disks, an answer might be as long as the time required to replay these notifications during recovery time does not exceed the time for the KVS to reconstruct its entire contents, i.e., the first approach.

Client Proxy Variant

Yet another possibility might be to use other KVS instances (and CADBMS client components of the participating applications that are a component of the physical data independence solution) to route notifications from the RDBMS to the KVS instance that cannot communicate with the RDBMS. This requires the participants to employ a collaborative routing protocol similar to those used by peer-to-peer networks, e.g., CAN [69], Chord [78], etc. It is interesting to note that such a protocol facilitates elasticity of a CADBMS to accommodate addition (removal) of KVS instances incrementally with no down time.

7.3 IQ Framework Extensions

Chapter 4 demonstrates the feasibility of implementing strong consistency in CADBMSs using an off-the-shelf RDBMS. It is based on a simple programming model that acquires IQ leases from the KVS either prior to the start of an RDBMS transaction or during the processing of the RDBMS transaction. The current implementation of IQ supports both the invalidate and refresh methods of maintaining the KVS consistency with the RDBMS. With today's implementation, a session is limited to at most one RDBMS transaction. A key question is whether the framework provides strong consistency guarantees for sessions consisting of multiple RDBMS transactions.

In contrast to the invalidate and refresh methods, an incremental approach [43] can be used to update key-value pairs in the KVS. With an incremental approach, only a small portion of the value is changed when data is modified in the RDBMS. As an example, a key-value pair may contain the result set of a query selecting multiple rows from the RDBMS. An update to the RDBMS causes the result set for that query to yield an additional row. In response to the update, the refresh method recomputes the entire query in order to populate the cache with the latest value. On the other hand, an incremental update method would compute the particular row that should be added to the result set and modify the key-value pair by updating it with the additional row. The latter approach avoids re-executing the original query, which may be beneficial in the case of expensive queries that select many rows. The incremental update method provides another interesting avenue for researching the feasibility of applying the IQ framework to provide strong consistency.

7.4 Supporting Additional Query Types With SQLTrig

As described in Section 5.4, SQLTrig supports queries containing exact-match selection predicates, equi-join predicates, conjunctive (logical *and*) and disjunctive(logical *or*) combinations of those predicates as well as select clauses using aggregates. For unsupported queries, SQLTrig avoids caching the result set and defaults to routing those queries directly to the RDBMS for execution. Expanding on the types of supported queries would allow SQLTrig to cache a larger portion of the data in workloads that use those more complex queries.

Queries containing range predicates are a candidate for support. One may support these

using R-Trees [44]. TrigGen constructs one R-Tree for each query template whose clause references a range selection predicate. A dimension of the R-Tree corresponds to a referenced attribute. A query instance is a k dimensional polygon in the R-Tree (corresponding to its query template) and whose results are used to compute a key-value pair. The R-Tree is maintained by the KVS and TrigGen authors triggers to generate a k dimensional value. These probe the R-Tree to identify matching polygons. Each polygon identifies queries (keys) whose result sets (values) have changed. The KVS deletes these keys.

Other types of queries include outer-join queries, nested sub-queries and queries containing non-equijoin predicates, set operations or comparisons. To elaborate, consider nested sub-queries. These nested sub-queries consist of select statements nested in another SQL query. While arbitrarily complex queries can be constructed in this manner, SQLTrig might support such queries by breaking down the nested query and supporting a superset covering the selected rows. For example, consider the following query:

SELECT $attr_1$, (SELECT $attr_2$ FROM R_2 WHERE $attr_3$=C_1)
FROM R_1
WHERE $attr_4$=C_1

Since the nested sub-query does not interact with the rest of the query (uncorrelated sub-query), the query can be supported by authoring triggers for the nested sub-query separately and also for the parent query while excluding the nested sub-query. Similar to how queries containing logical *or* predicates are handled (see Section 5.4.3), the query above can be supported as though it were 2 separate queries. For correlated sub-queries, simple equi-joins between attributes selected in nested sub-queries can be translated to separate queries containing the join attribute and supported similar to how equi-join predicates are currently handled as described in Section 5.4.2.

7.5 SQLTrig In Other Environments

While SQLTrig is designed to work with the SQL relational model, the concept of a seamless caching approach with transparent cache consistency applies to many other data models as well. Experimental results [13] show that query result lookup enhances the performance of a NoSQL [24] data store solution such as MongoDB. Such extensions may require different

mechanisms for detecting updates to the data store, such as MongoDB's Oplog [60] which maintains a rolling record of all operations that modify data in the data store.

Additionally, SQLTrig is currently implemented based on the Client Server (CS) architecture, as described in Section 3.1. The Shared Address Space (SAS) architecture described in Section 3.2 presents an attractive area of future research for extending the SQLTrig implementation. Characteristics of the SAS architecture such as replication of key-value pairs across nodes present different challenges in realizing SQLTrig and maintaining strong consistency. This effort should include an analysis of how the internal keys, IntKeys, should be partitioned to enable an architecture to scale to a large number of nodes. The discussions of Section 7.1 to extend the implementation of SQLTrig to multiple SQLTrig servers with partitioned keys offer relevant mechanisms for working with distributed cache nodes in the SAS architecture.

Bibliography

[1] S. Agrawal, N. Bruno, S. Chaudhuri, and V. R. Narasayya. AutoAdmin: Self-Tuning Database Systems Technology. *IEEE Data Eng. Bull.*, 29(3):7–15, 2006.

[2] 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.

[3] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *ICDE*, pages 821–831, 2003.

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *ICDE*, pages 230–241, 2005.

[5] C. Amza, G. Soundararajan, and E. Cecchet. Transparent Caching with Strong Consistency in Dynamic Content Web Sites. In *Supercomputing*, ICS '05, pages 264–273, New York, NY, USA, 2005. ACM.

[6] C. Aniszczyk. Caching with Twemcache, http://engineering.twitter.com/2012/07/caching-with-twemcache.html.

[7] Apache. Apache ZooKeeper,http://zookeeper.apache.org/.

[8] Six Apart. Memcached Specification, http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt.

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

[10] L. Backstrom. Anatomy of Facebook, http://www.facebook.com/note.php?note_id=10150388519243859, 2011.

[11] D. Z. Badal. Correctness of Concurrency Control and Implications for Distributed Databases. In *COMPSAC*, 1979.

[12] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. 2013.

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

[14] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing User Behavior in Online Social Networks. In *Internet Measurement Conference*, 2009.

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

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

[17] C. Bornhövd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures. In *SIGMOD Conference*, 2003.

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

[19] Eric A. Brewer. Towards Robust Distributed Systems (Abstract). In *PODC*, page 7, 2000.

[20] N. Bruno and S. Chaudhuri. Physical Design Refinement: The ”Merge-Reduce” Approach. In *EDBT*, pages 386–404, 2006.

[21] N. Bruno and S. Chaudhuri. Constrained Physical Design Tuning. *VLDB J.*, 19(1):21–44, 2010.

[22] JBoss Cache. JBoss Cache, http://www.jboss.org/jbosscache.

[23] K. S. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. In *SIGMOD Conference*, pages 532–543, 2001.

[24] R. Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[25] 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.

[26] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *ACM/IEEE SC*, November 1998.

[27] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[28] B. F. 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.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[30] Couchbase. Couchbase 2.0 Beta, http://www.couchbase.com/.

[31] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, D. VanderMeer, K. Ramamritham, and D. Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *VLDB*, pages 667–670, 2001.

[32] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, and K. Ramamritham. Proxy-based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. *ACM Transactions on Database Systems*, pages 403–443, 2004.

[33] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[34] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable Query Result Caching for Web Applications. August 2008.

[35] S. Ghandeharizadeh, S. Barahmand, A. Ojha, and J. Yap. Recall All You See, http://rays.shorturl.com, 2010.

[36] S. Ghandeharizadeh and J. Yap. Gumball: A Race Condition Prevention Technique for Cache Augmented SQL Database Management Systems. In *Second ACM SIGMOD Workshop on Databases and Social Networks*, 2012.

[37] S. Ghandeharizadeh, J. Yap, and S. Barahmand. COSAR-CQN: An Application Transparent Approach to Cache Consistency. In *Twenty First International Conference On Software Engineering and Data Engineering*, Los Angeles, CA, Best Paper Award, 2012.

[38] Shahram Ghandeharizadeh and Jason Yap. Cache Augmented Database Management Systems. In *Proceedings of the ACM SIGMOD Workshop on Databases and Social Networks*, DBSocial '13, pages 31–36, New York, NY, USA, 2013. ACM.

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

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

[41] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[42] H. Gupta and I. S. Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

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

[44] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, pages 47–57, 1984.

[45] 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.

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

[47] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 49–60, 1997.

[48] R. Johnson. More Details on Facebook Outage of Thursday, Sept. 23, 2010, http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919, 2010.

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

[50] A. Labrinidis and N. Roussopoulos. WebView Materialization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 367–378. ACM, 2000.

[51] A. Labrinidis and N. Roussopoulos. Exploring the Tradeoff Between Performance and Data Freshness in Database-Driven Web Servers. *The VLDB Journal*, 2004.

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

[53] 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.

[54] 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.

[55] memcached. Memcached, http://www.memcached.org/.

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

[57] Microsoft. Using Query Notifications. `http://msdn.microsoft.com/en-us/library/ms175110(v=sql.90).aspx`.

[58] Microsoft. Working with Query Notifications, http://technet.microsoft.com/en-us/library/ms130764(v=sql.110).aspx, 2014.

[59] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialize View Selection and Maintenance Using Multi-Query Optimization. In *Proceedings of ACM SIGMOD*, May 2001.

[60] MongoDB. Replica Set Oplog, http://docs.mongodb.org/manual/core/replica-set-oplog/.

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

[62] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 385–398, Berkeley, CA, 2013. USENIX.

[63] Oracle. Database Change Notification. `http://docs.oracle.com/cd/E14072_01/java.112/e10589/dbchgnf.htm`.

[64] Oracle. Using Continuous Query Notification. `http://docs.oracle.com/cd/B28359_01/appdev.111/b28424/adfns_cqn.htm`.

[65] Oracle. PL/SQL Packages, http://docs.oracle.com/cd/a97630_01/ appdev.920/a96624/09_packs.htm, 1996.

[66] D. A. Patterson. A Simple Way to Estimate the Cost of Downtime. In *LISA*, volume 2, pages 185–188, 2002.

[67] 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.

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

[69] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, August 2001.

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

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

[72] K. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proceedings of ACM SIGMOD*, May 1996.

[73] N. Roussopoulos. View Indexing in Relational Databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[74] N. Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD Record*, 27(1):21–26, 1998.

[75] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't Trash your Intermediate Results, Cache 'em. *CoRR*, cs.DB/0003005, 2000.

[76] P. Saab. Scaling memcached at Facebook, http://www.facebook.com/note.php?note_id=39391378919, Dec. 2008.

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

[78] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, San Diego, California, August 2001.

[79] M. Stonebraker and R. Cattell. 10 Rules for Scalable Performance in Simple Operation Datastores. *Communications of the ACM*, 54, June 2011.

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

[81] Terracotta. Ehcache, http://ehcache.org/documentation/overview.html.

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

[83] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The Anatomy of the Facebook Social Graph. *CoRR*, abs/1111.4503, 2011.

[84] W. Vogels. Eventually Consistent. *ACM Queue*, 6(6):14–19, 2008.

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

[86] 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.

[87] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *VLDB*, pages 188–199, 2000.