COSAR-CQN: An Application Transparent Approach to Cache Consistency*

Shahram Ghandeharizadeh, Jason Yap, Sumita Barahmand

Database Laboratory Technical Report 2012-02

Computer Science Department, USC

Los Angeles, California 90089-0781

{shahram,jyap, barahman}@usc.edu

June 23, 2012

Abstract

Cache managers speed up the performance of data intensive applications whose workload is dominated by queries. An example is memcached which is in use by very large well-known sites such as Facebook. A key challenge of such systems is how to maintain the state of the cache consistent with that of the database in the presence of updates. With SQL based database management systems (DBMSs) that support query change notification mechanism, one possible approach would require the cache manager to a) register queries used to compute a key-value pair with the DBMS to subscribe for notifications when the query result sets change, and b) delete the cached key-value pair(s) once notified of a change by the DBMS. This approach is independent of how the application updates the database, eliminating application specific software for cache consistency. This reduces the complexity of the application software and minimizes the software development cycle associated with designing, developing, and maintaining software in support of cache consistency. This paper presents the design and implementation of this approach in a cache manager named COSt AwaRe, COSAR. This approach is named Continuous Query change Notification, CQN. It is one of several cache consistency techniques implemented in COSAR. We compare CQN with these alternatives, quantifying its strengths and limitations.

A Introduction

The workload of certain classes of applications is dominated by queries that read data. An example is the profile page of a user registered with one of the many social networking sites. A user may update his or her

*This research was supported by an unrestricted cash gift from Oracle Inc. A shorter version of this paper appeared in the *Twenty First International Conference On Software Engineering and Data Engineering*, Los Angeles, California, June 27-29, 2012.

profile rarely, every few hours if not days and weeks. At the same time, these profile pages are referenced and displayed frequently: Every time a user logs in and navigates between pages. A simple implementation, named *Classic*, would require the application to repeatedly invoke the logic to issue queries to the database management system (DBMS) and fuse their results together to compute the final desired output, i.e., the HTML snippet corresponding to the profile page of a user. An alternative, named Cache Augmented SQL DBMS (*CASQL*), extends the DBMS with a cache manager and saves the final results (the HTML snippet) in the cache to service future references [7, 3, 25]. A cache look up is faster than executing multiple queries and processing their results, providing users with fast response times. It also enhances system throughput by freeing both the DBMS and the application server to process other requests. As an example, the response time to compute a Browsing sequence in a social networking application named RAYS [13] (see Section C) is 6 milliseconds with CASQL compared to 61 milliseconds with Classic. This performance difference explains the popularity of memcached, a widely used cache manager that is deployed by sites such as YouTube [8], Facebook [26], Twitter, and Wikipedia/Wikimedia.

With a CASQL architecture, a developer identifies code fragments that manipulate infrequently updated data. Execution of a code fragment with an input produce an output. This output *value* is identified using a unique *key*. The developer extends the code to look up the key prior to executing the code fragment with its input. If the cache manager returns the value, then the value is used without executing the code fragment. Otherwise, the code fragment executes and the resulting key-value pair is inserted into the cache for use by future references, see the discussion of Figure 1.b for more details.

A key challenge of the CASQL architecture is how to maintain the cached key-value pairs consistent with both incremental and bulk updates to their tabular counterparts in the database. One may taxonomize today's approaches to cache consistency as follows:

1. Application (App) driven: 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. To illustrate, with our 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-value pair. With the latter, a subsequent cache look up observes a miss and invokes the application logic to compute the new key-value pair and inserts it into the cache.

2. Synthetic: The programmer extends the application to provide a Time To Live, TTL, for each key-value pair. The cache manager invalidates a key-value pair once its TTL expires. When TTL is a rough estimate, this approach may either purge cached key-value pairs un-necessarily or serve stale data for some time.

3. DBMS trigger (Trig) driven: The programmer extends the DBMS with triggers [24] 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.

This paper presents the design and implementation of an approach, named Continuous Query change Notification (CQN), that makes the cache manager aware of the DBMS by requiring it to register SQL queries used to compute a key-value pair with the query change notification mechanism of the DBMS. Once notified of a change, the cache manager deletes the impacted key-value pair(s).

This approach is advantageous for several reasons. First, it eliminates the need for a developer to design, develop, and maintain application specific software for cache consistency. This enables organizations to develop features faster with reduced software complexity. Second, it provides for physical data independence: The programmer is no longer burdened with whether the data resides in the cache or the DBMS. This enhances robustness of the deployed system by preventing assumptions that compromise availability of the data. An example comes from Facebook where, due to dependence of data on different forms of storage, a 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 [18]. Physical data independence property of CQN, see Section D for details, would have avoided both the flawed assumption and the resulting 2.5 hour down time with a price tag of millions of dollars.

CQN incurs two forms of overhead. First, the time to register different query instances with the DBMS. To elaborate, our target social network applications are different than financial applications because queries are not known in advance. (There is no fixed collection of stocks.) While the query templates are known, the value of variables employed by a query are available only at run time. And, these values evolve as new data is inserted, e.g., a new user joins a social network. Second, a DBMS processes updates slower as the number of registered queries increases because it must detect a change to the result sets of one or more registered queries and generate notifications to the cache server.

Contributions of this study are two folds. First, CQN and its design decisions. Second, a comparison of CQN with several alternative approaches to cache consistency. Our findings are as follows. CQN expedites software development cycle to enable organizations to quickly develop and deploy features. However, its performance is not as competitive as either App or Trig because the change notification mechanism of commercial DBMSs is relatively new. As this mechanism matures, we anticipate its performance to improve, closing the performance gap between CQN and App/Trig. Meanwhile, should an organization employ CQN for a feature, as that feature becomes more popular and attracts more traffic, the organization may switch to either App or Trig to meet its performance requirements.

The rest of this paper is organized as follows. Section B details related work. Section D presents

the conceptual and physical architectures of a CASQL and how CQN approximates them given today's implementation of change notification by commercial DBMSs and their limitations. The overhead of query registration and how CQN hides it are detailed in Section E. Section F compares CQN with the 3 alternative approaches to cache consistency. Brief future research directions are outlined in Section G.

To illustrate concepts and quantify the tradeoffs associated with CQN's design decisions, we present numbers from a benchmark that models a social networking application named RAYS [13]. This benchmark is detailed in Section C.

B Related Work

CASQL architecture has been an active area of research since the late 1990s [17, 7, 28, 11, 9, 6, 21, 19, 1, 27, 20, 5, 10, 2, 3, 4, 25, 16]. The focus of this study is on middle-tier caches [17, 7, 28, 11, 9, 19, 3, 4, 25, 16, 15] that augment a DBMS with a key-value store. Caches that process SQL queries [27, 2, 21, 5, 1, 20, 12] are beyond the focus of CQN. CQN is a transparent caching technique because it maintains key-value pairs consistent with updates to the tabular data seamlessly with no additional software from the developer. Similar transparent caching techniques include TxCache [25] and CacheGenie [16]. While CacheGenie employs triggers, TxCache extendes the DBMS with additional software to generate invalidation streams. CQN is novel because it uses query change notification mechanism of a DBMS. This mechanism is offered by several DBMS products such as Oracle 11g [23] and Microsoft SQL Server 2005 and its later releases [22]. It is relatively new and still evolving. We expect the architecture of CQN to change with them. We speculate the final architecture to utilize CQN as the intermediary between the application and the DBMS, rendering the details of the cache manager and its interaction with the DBMS transparent to the developer. See Section D.

C RAYS and a Social Networking Benchmark

Recall All You See (RAYS) [13] 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 user's 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).

We use two popular navigation paths of RAYS to both describe and evaluate CQN. They are named Browsing friends (Browse) and Toggle streaming (Toggle). While Browse is a read-only workload, Toggle

	Operation	Browse	Toggle
Classic	SQL Queries	38	23
	SQL Updates	0	3
CQN	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 1: Characteristics of two different sequences of page visits and clicks with RAYS using an empty cache.

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 the Classic architecture, Browse issues 38 SQL queries to the DBMS, see Table 1. With CQN, Browse registers 33 distinct queries and issues 8 get operations. For each get that observes a cache miss, it performs a put 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 Classic. CQN issues 7 get operations that observe a cache miss with an empty cache. CQN 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 1. With CQN, these updates invalidate cached entries corresponding to both the profile[1] 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, inter-arrival 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. $\mathcal{N}$ threads model $\mathcal{N}$ simultaneous users accessing the system. In the single user (1 thread, $\mathcal{N}$=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.

[1]The user's profile page displays the count of devices that are streaming and a toggle of a streaming device either increments or decrements this counter.

Term	Definition
$\mathcal{N}$	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.
μ	Probability of a user referencing a Toggle sequence.

Table 2: Workload parameters and their definitions.

We target a small database consisting of 1,000 unique users, ω=1,000, eliminating cache replacement as an experimental variable. A DBMS 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 CQN, quantifying the overhead of registered queries when the DBMS processes updates.

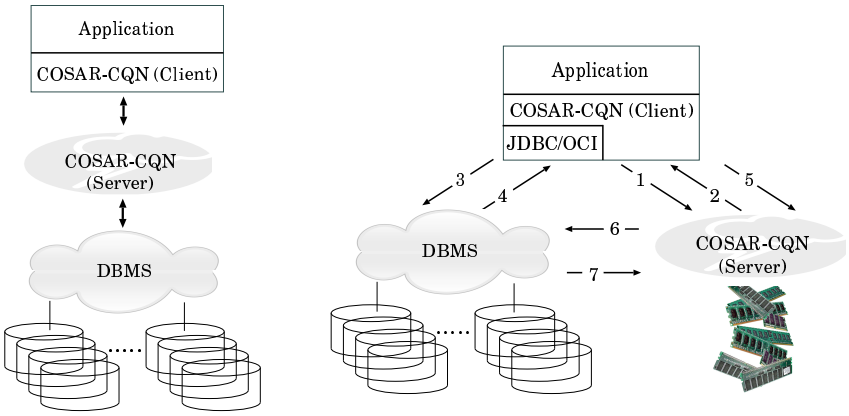
The workload generator maintains the structure of the synthetic database along with information about the activities of different users to detect cached data (HTML pages) that are not consistent with the state of the database, termed *stale* data. The workload generator produces unique simultaneous users 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 DBMS product used for our reported performance numbers. The term DBMS refers to an anonymous commercial product. 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 DBMS releases and avoided it by issuing updates one at a time using our infrastructure.

D Architecture and overview

COSAR implements CQN using its client and server components that communicate using the TCP protocol. In a typical deployment, the application links with the client component to issue put and get commands to the server deployed on a PC with a substantial amount of memory. In the following, we focus on one instance



1.a) Conceptual architecture 1.b) Physical architecture

Figure 1: CQN architectures.

of the server.

COSAR requires a developer to identify a key-value pair, k_i-d_i, for a code segment i, CS_i. The start of CS_i is identified with a cache look up, d_i=get(k_i). If d_i is found then the code segment following CS_i proceeds to use it. Otherwise, CS_i executes to compute d_i. CS_i might be complex consisting of arbitrary loop and branch programming constructs. Each branch may execute a different sequence of SQL statements and different conditional statements may invoke put(k_i,d_i) operation independently.

The conceptual architecture calls for CQN as the intermediary between the application and the DBMS, see Figure 1.a. CQN would execute the SQL commands issued by a branch of CS_i that computes k_i-d_i and register queries with the continuous query change notification mechanism of the DBMS. The put(k_i,d_i) would cause CQN to insert the k_i-d_i pair into the cache and associate it with the registered queries. An update to the DBMS (either by the application or an off-line program for bulk update) would generate notifications to the CQN server to delete the impacted k_i-d_i pairs.

This conceptual architecture is desirable because it provides for physical data independence: The programmer is no longer burdened with whether the data resides in the cache or the DBMS. The application is authored to identify CS_i, k_i, and d_i. It is the responsibility of COSAR-CQN to decide which k_i-d_i occupies the cache and how the cached entries are maintained up to date.

While we speculate the conceptual architecture will be realized in the near future as the query change notification mechanism of commercial DBMSs evolves and matures, it is not fully viable at the time of this writing. This is because the query change notification mechanism of DBMSs may either not register certain class of SQL commands or register them at the granularity of table change notification. Examples include nested SQL queries, aggregates such as count, complex qualification lists ("where" clause) with disjuncts using multiple join clauses, queries that reference BLOBS, and others. Registration of these queries at the granularity of a table (instead of a query) is undesirable because a change (insert/delete/update) to a

SELECT u.userid, u.picture, u.uname
FROM users u, user_cameras uc, friends f
WHERE ((f.frdID = 14511 AND f.userid = u.userid) **OR** (f.userid = 14511 AND f.frdID = u.userid)) AND uc.IsStreaming = 1 AND u.userid = uc.userid AND f.status = 2

a. Original query

SELECT u.userid, u.picture, u.uname
FROM users u, friends f, user_cameras uc
WHERE (f.userid = 14511) AND f.status = 2 AND u.userid = f.frdID AND uc.userid = u.userid AND uc.IsStreaming=1

b. Query fragment 1

SELECT u.userid, u.picture, u.uname
FROM users u, friends f, user_cameras uc
WHERE (f.frdID = 14511) AND f.status = 2 AND u.userid = f.userid AND uc.userid = u.userid AND uc.IsStreaming=1

c. Query fragment 2

Figure 2: The left most query computes "Live Stream" portion of the user profile with RAYS. Its disjunctive predicate, "OR" clause, prevents it from being registered with a DBMS at the granularity of Query Change, QC, notification. This query can be represented as two queries, see the center and right most columns. CQN registers each with a DBMS at the granularity of QC notification and associates their call back function with the cached k_i-d_i entry.

table may invalidate many, potentially millions of, cached k_i-d_i pairs that were not impacted by the update (because the result of the queries used to compute them did not change). As an example, with RAYS, the "Live Friends" section of a user's profile is constructed by issuing the query shown in Figure 2.a. The qualification list of the query is complex and the DBMS would register this query at the granularity of table change notification. With millions of users, millions of key-value pairs corresponding to the "Live Friends" fragment of the user profile might be cached. An update by one user would invalidate millions of cached entries for different users, rendering the cache ineffective.

Today, CQN addresses this limitation in two ways. First, it extends the client component of COSAR to enable the developer to re-write queries that are trivial to fix, see Section D.1. Second, it empowers the programmer to identify queries registered at the granularity of a table (along with other potentially problematic queries). For each such query Q_E that should be registered at the granularity of query change, the programmer employs the client interface of CQN to replace Q_E with a manually written set of one or more queries, $\{\text{Q}\}$. Queries in set $\{\text{Q}\}$ must compute either the same or a superset of results produced by Q_E. One or more of the queries in set $\{\text{Q}\}$ may still fail to register at the granularity of query change notification. Using the CQN monitoring tool, the programmer identifies these, re-writes them, and incorporates these re-writes into the software. The programmer may repeat this process several times until all queries in set $\{\text{Q}\}$ are registered at the granularity of query change notification.

Ideally, $\{\text{Q}\}$ should consist of only one query representing Q_E. Moreover, it should compute the same (a tight bound of the) result set produced by the original query. The automatic re-writes produced by CQN strive to achieve this goal, see Section D.1.

In our example with "Live Friends", the programmer registers the two queries shown in 2.b and 2.c (set $\{\text{Q}\}$) for the executed query shown in 2.a (Q_E).

For certain applications, there may exist queries that might not be re-writable to facilitate query change notification. A good example includes queries that employ the "LIKE" clause. If table notification is appropriate for such queries then it should be employed. Otherwise, the developer must utilize one of the techniques described in Section A (and supported by COSAR) to maintain the cached data consistent with

the database. Switching to one of these alternatives is as follows: The developer specifies an empty {Q} to turn off CQN and provides additional software to implement either App or Trig.

Figure 1.b shows the physical architecture of COSAR-CQN. CQN extends the client component of COSAR to be a wrapper for a DBMS interface such as JDBC. It performs the cache look up of k_i by contacting the server, Arrow 1, to either retrieve data or observe a cache miss, Arrow 2. With a cache miss, the application executes CS_i which in turn issues SQL commands. The client component intercepts these SQL commands and executes them using JDBC to produce results, Arrows 3 and 4. Moreover, it maintains a list of these queries, set {E}. If a query in set {E} does not register at the desired granularity, the application substitutes that element in {E} with a programmer specified set of queries, {Q}. CS_i processes the results of the original executed queries to compute (k_i,d_i) and issue put(k_i,d_i). CQN client extends this with the set of queries {E} to be registered and issues a put(k_i,d_i,{E}) to the CQN server. The server registers the final set of queries in {E} with the query change notification component of the DBMS and associates the identifier of each query registration, a query-id, with the cached k_i-d_i pair, Arrow 6. When the server receives a notification for one of these query-ids, Arrow 7, it deletes the corresponding k_i-d_i from the cache. In the case where {E} is empty, the server inserts k_i-d_i into the cache and returns.

Architecture of Figure 1.b suffers from write-write conflicts (Arrow 5 conflicting with with Arrow 7) that may leave the cache inconsistent with the tabular data. CQN employs the gumball technique [14] to detect such race conditions and prevent them from causing the cache to produce stale data.

D.1 SQL Query Re-Write

CQN performs trivial SQL query re-writes to utilize the query notification mechanism of commercial DBMSs for a larger number of queries seamelessly. We anticipate commercial DBMSs to implement similar re-writes in their future releases. These re-writes are as follows. First, some DBMSs do not allow registering queries with a "*" in their target list, e.g., "SELECT * FROM ... WHERE ...". Moreover, they may limit the number of attributes that appear in the target list of a query. To address these limitations, CQN replaces the "*" with the list of attributes of the referenced table. It does so by referencing the metadata of each referenced table to extract its column names. When the number of referenced columns exceeds the limit imposed by a DBMS, CQN generates multiple queries, each referencing different column names. For example, CQN re-writes a "SELECT *" query of RAYS by instantiating the 45 columns of its referenced tables (using metadata provided by the DBMS). With a DBMS that limits the registered queries to reference 10 columns only, CQN generates five queries and registers each with the DBMS. With all queries, their tuple variable list and qualification lists are identical. Their only difference is their target list. CQN client caches database metadata, avoiding the overhead of repeated JDBC and DBMS invocations.

Second, with aggregates such as "SELECT count(*)", CQN replaces "count(*)" with the primary key of the referenced tables. With all aggregates written as "SELECT count/max/min/avg(attr)", CQN replaces

"count/max/min/avg(attr)" with the referenced attribute "attr". These re-writes may result in false positives. As an example, consider a "SELECT count(*)" when the primary key of a row is modified. The result of the query is unchanged and there should be no notifications. However, our rewrite, "SELECT primary key", results in a notification because its result has changed. Such false positives are undesirable because they delete key-value pairs un-necessarily.

Finally, some DBMSs do not register SQL queries with an "Order by" clause. CQN removes this clause (along with "Top X" clauses, if any) for all SQL statements. This re-write may result in false positivies. For example, with a query that retrieves "Top 4" rows using the order by clause, a database update may not change the identify of these first 4 rows. However, out re-write generates a notification.

E Query Registeration Overhead

Registering queries with a DBMS is a time consuming operation. Repeat registrations of the same query is significantly (10 times) faster than its very first registration. A registered query persists in the presence of system restarts and is considered a repeat registration. CQN maintains a list of queries that it registers with the DBMS and does not re-register the same query twice. This is several orders of magnitude faster than repeat registration of the same query. Below, we focus on the overhead of registering a query the first time and techniques to hide this overhead.

The exact amount of time to register a query may vary from a few milliseconds to seconds. The exact amount of time depends on the complexity of the qualification list ("where" clause) of the query, the load imposed on the DBMS, and the number of registered queries. With the later, query registration time may increase as a function of the number of queries registered with the DBMS.

Given a put(k_i,d_i,$\{\text{E}\}$), a simple implementation of CQN may registers queries in set $\{\text{E}\}$ and then inserts k_i-d_i in the cache synchronously. This technique would process requests at a slower rate than Classic due to the overhead of query registration. An alternative, named gradual, would register queries associated with the inserted key-value pairs as a background activity. A key-value pair is associated with an application specific TTL, the maximum duration of inconsistency window tolerated by the application. Gradual checks to see if queries in $\{\text{E}\}$ are registered with the DBMS. If this is the case, it inserts k_i-d_i in the cache and associates them with the query-id of the registered queries, ignoring the specified TTL. Otherwise, it proceeds to register queries in set $\{\text{E}\}$ asynchronously and inserts k_i-d_i in the cache for the specified TTL. When TTL=0, gradual does not insert k_i-d_i in the cache unless queries in set $\{\text{E}\}$ are registered. With TTL values greater than zero, gradual caches key-value pairs and services user requests from the cache, freeing DBMS resources to register queries sooner and populating the cache faster.

Gradual is advantageous for several reasons. First, it enables CQN to approximate Classic for the first few hundred user requests when the system starts for the very first time. Second, as the value of TTL

increases, CQN outperforms Classic by a wider margin.

F A Comparison

This section compares CQN with Application, Trig and Synthetic along three dimensions: 1) man hours required to design, implement and debug an approach, 2) average processing time, 3) served stale data. Consider each dimension in turn.

F.1 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. CQN 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 Application 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.

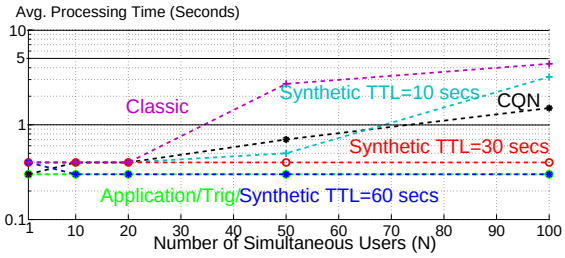
While the benchmarks we conducted were focused on users toggling their devices to start or stop streaming, in the real RAYS system, the user is able to perform other operations, such as modify their profile information or friend relationships with others. Such modifications may cause some cached data 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 CQN, all of these cases are automatically covered by registering queries. CQN 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. CQN used them to associate with a cached key-value pair.

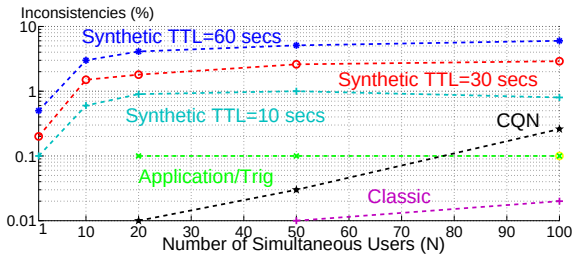
In order to fairly compare the different techniques, we had to develop Application/Trig invalidation schemes that would cover all the potential modifications by users. Most of the 90 man hours was spent on identifying these scenarios and implementing the appropriate invalidation schemes in the form of 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. CQN eliminates this analysis and its associated software.

F.2 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 3. These results pertain to a warm cache, one



3.a) Average processing time



3.b) Percentage inconsistent data

Figure 3: Comparison of alternative approaches. ϵ=100 msec, θ=0, ω=1,000, μ=1%, n=10,000.

with a cache hit rate close to 100%. As a reference point, we also present numbers from Classic. With this approach, the most up-to-date data should be retrieved directly from the DBMS 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 3.b.

Overall, both Application and Trig approaches provide the best processing time and request rates. Moreover, they produce least amount of stale data[2]. Synthetic is sensitive to the value of TTL. A high TTL value (60 seconds) enables Synthetic to approximate the performance of the Application and Trig approaches, see Figure 3.a. However, it produces the highest amount of stale data, see Figure 3.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 $\mathcal{N}$=50 and 100, CQN results in a higher average processing time when compared with Application and Trig approaches. This is because its tens of thousands of registered queries cause the DBMS to processes SQL update commands slower. As shown in Table 3, the average processing time for a Toggle sequence is significantly higher with CQN. 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 C. On the other hand, the average processing time of Browse with CQN remains low in all configurations.

With 1 user, $\mathcal{N}$=1, the response time of updates with CQN is slower than Classic. This appears to be a limitation in the latest release of the DBMS and we anticipate it to be resolved in future releases. We are almost certain that the performance of updates with CQN will improve as query notification feature of commercial DBMSs matures, rendering CQN viable for a larger number of simultaneous users and registered queries (data set sizes).

In sum, CQN expedites software development cycle to enable organizations to quickly develop and deploy features. As the application becomes popular (higher number of simultaneous users), the organization

[2]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.

$\mathcal{N}$	Application		CQN	
	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 3: Processing time (Seconds) of Browse and Toggle Sequences. ϵ=100 msec, θ=0, ω=1,000, μ=1%, n=10,000.

may switch to either the Application or Trig approach to enhance average processing time.

G Future research

Query change notification is a recent addition to SQL based DBMSs and we anticipate it to evolve in two ways. First, it will process SQL update commands, query registrations, and change notifications faster. Second, it will support more complex queries such as aggregates using the re-write rules of Section D. Both are in synergy with CQN and its approach to cache consistency: While the first enables the cache manager to support workloads with higher frequency of updates, the second eliminates the need for the developer and CQN to re-write queries that should be registered with the DBMS.

More longer term, we envision *CASQL* to make the cache fully transparent, realizing the architecture of Figure 1.a. It will be a distributed system that implements physical data independence and provide applications with a host of cache consistency techniques.

H Acknowledgments

We wish to acknowledge and thank D. Gawlick, S. Vemuri, and L. Chidambaran for participating in our technical discussions and influencing our design decisions. We also wish to thank D. Cohen for participating in discussions that facilitated COSAR-CQN. This research was supported by an unrestricted cash gift from Oracle Inc.

References

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

[2] K. Amiri, S. Park, and R. Tewari. DBProxy: A Dynamic Data Cache for Web applications. In *ICDE*, 2003.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *ICDE*, 2005.

[4] 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.

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

[6] 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.

[7] 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.

[8] C. D. Cuong. YouTube Scalability. Google Seattle Conference on Scalability, June 2007.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] S. Ghandeharizadeh and J. Yap. SQL Query To Trigger Translation: A Novel Consistency Technique for Cache Augmented DBMSs. In *Submitted for Publication*, 2012.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] Microsoft. Working with Query Notifications, SQL Server 2008 R2 Native Client, Developer's Guide, http://msdn.microsoft.com/en-us/library/ms130764.aspx, 2008.

[23] Oracle. Using Continuous Query Notification, Chapter 12, Oracle Database Advanced Application Developer's Guide 11g Release 1 (11.1), http://download.oracle.com/docs/cd/b28359_01/ appdev.111/b28424/adfns_cqn.htm, 2008.

[24] Oracle. Using the MySQL memcached UDFs, MySQL 5.0 Manual, Chapter 14, Section 14.5.3.7, http://dev.mysql.com/doc/refman/5.0/en/ha-memcached-interfaces-mysqludf.html, 2008.

[25] 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.

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

[27] The TimesTen Team. Mid-Tier Caching: The TimesTen Approach. In *SIGMOD*, 2002.

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