Gumball: A Race Condition Prevention Technique for Cache Augmented SQL Database Management Systems*

Shahram Ghandeharizadeh, Jason Yap

Database Laboratory Technical Report 2012-01

Computer Science Department, USC

Los Angeles, California 90089-0781

September 11, 2012

Abstract

Query intensive applications augment a Relational Database Management System (RDBMS) with a middle-tier cache to enhance performance. An example is memcached in use by very large well known sites such as Facebook. In the presence of updates to the normalized tables of the RDBMS, invalidation based consistency techniques delete the impacted key-value pairs residing in the cache. A subsequent reference for these key-value pairs observes a cache miss, re-computes the new values from the RDBMS, and inserts the new key-value pairs in the cache. These techniques suffer from race conditions that result in cache states that produce stale data. The Gumball Technique (GT) addresses this limitation by preventing race conditions. Experimental results show GT enhances the accuracy of an application hundreds of folds and, in some cases, may reduce system performance slightly.

A Introduction

The workload of certain application classes such as social networking is dominated by queries that read data [7]. An example is the user profile page. A user may update her profile page rarely, every few hours if not days and weeks. At the same time, these profile pages are referenced and displayed frequently: Every time the user logs in and navigates between pages. To enhance system performance, these applications augment a standard SQL based RDBMS, e.g., MySQL, with a cache manager. Typically, the cache manager is a Key-Value Store (KVS), materializing key-value pairs computed using the normalized relational data. A key-value pair might be finely tuned to the requirements of an application, e.g., dynamically generated

*A short version of this paper appeared in the proceedings of the Second ACM SIGMOD Workshop on Databases and Social Networks (DBSocial), May 20, 2012.

HTML formatted pages [12, 27, 5, 26, 10, 17], and the KVS may manage a large number (billions) of such highly optimized representation. A Cache Augmented SQL RDBMS, CASQL, enhances performance dramatically because a KVS look up is significantly faster than processing SQL queries. This explains the popularity of memcached, an in-memory distributed KVS deployed by sites such as YouTube [14] and Facebook [35].

With CASQLs, a consistency technique maintains the relationship between the normalized data and its key-value representation, detects changes to the normalized data, and invalidates[1] the corresponding key-value(s) stored in the KVS. Almost all techniques suffer from race conditions, see Section B. The significance of these race conditions is highlighted in [33] which describes how a web site may decide to not materialize failed key-value lookups because the KVS may become inconsistent with the database permanently.

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 site's terms of use. Section C shows how 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 paper presents the Gumball Technique (GT) to detect race conditions and prevent them from causing the key-value pairs to become inconsistent with the tabular data. GT is an application transparent, deadlock free (non-blocking) technique implemented by the KVS. Its key insight is that it is permissible to ignore cache put operations to prevent inconsistent states. Its advantages are several folds. First, it applies to all applications that use a CASQL, freeing each application from implementing its own race condition detection technique. This reduces the complexity of the application software, minimizing costs. Second, in our experiments, it reduced the number of observed inconsistencies dramatically (more than ten folds). Third, GT requires no specific feature from an RDBMS and can be used with all off-the-shelf RDBMSs. Fourth, GT adjusts to varying system loads and has no external knobs that require adjustments by an administrator. Fifth, while GT employs time stamps, it does not suffer from clock drift and requires no synchronized clocks because its time stamps are local to a (partitioned) cache server. The disadvantage of using GT is that it *may* slow down an application slightly when almost all (99%) requests are serviced using KVS. This is because GT re-directs requests that may retrieve stale cache data to process SQL queries using the RDBMS.

The primary contribution of this paper is the design of GT and how it detects and prevents race conditions. To the best of our knowledge, GT is novel and not presented elsewhere. A secondary contribution is an implementation and evaluation of GT using a social networking benchmark. Obtained results show GT imposes negligible overhead while reducing the percentage of inconsistencies dramatically.

[1] Other possibilities include refreshing [12, 23] or incrementally updating [24] the corresponding key-value.

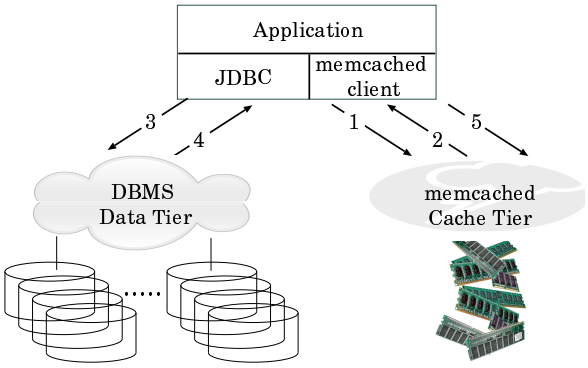


Figure 1: Architecture of a CASQL. Arrows show processing of a CS_{fuse} with a key reference whose value is not found in KVS.

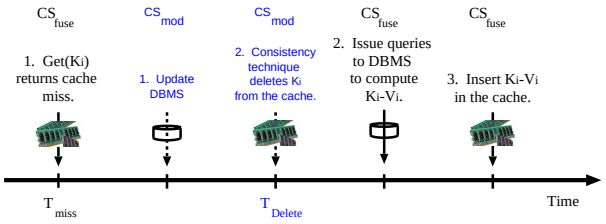
The rest of this paper is organized as follows. The race condition is formalized in Section B and Section C describes GT to detect and prevent it. We highlight the tradeoffs associated with GT in Section D. Section E presents related work. Brief conclusions are offered in Section F.

B Problem Statement

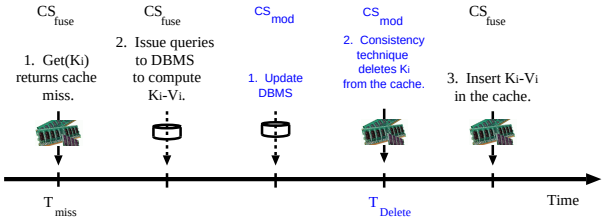
Figure 1 shows the architecture of a typical CASQL. Both the KVS and the RDBMS consist of a client and a server component that communicate using the TCP protocol. The application employs the RDBMS client (e.g., JDBC) to issue SQL queries to the RDBMS server. Similarly, it employs the KVS client to issue insert, get and delete commands to one or more instances of a KVS server deployed on several (potentially thousands of) PCs with a substantial amount of memory.

The key-value pairs in KVS might pertain to either the results of a query [23] or a semi structured data obtained by executing several queries and glueing their results together using application specific logic [12, 10, 33, 23]. With the former, the query string is the key and its result set is the value. The latter might be the output of either a developer designated read-only function [33] or code segment that consumes some input to produce an output [23]. 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.

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. This corresponds to Alice in the example of Section A 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_{mod} and CS_{fuse} employ the concept of transactions, their KVS and RDBMS operations are non-transactional and may leave the KVS inconsistent. One



2.a) Acceptable.



2.b) Undesirable.

Figure 2: Two interleaved processing of CS_{fuse} and CS_{mod} referencing the same key-value pair.

scenario is shown in Figure 2.b. CS_{fuse} looks up the KVS and observes a miss, Arrows 1 and 2 of Figure 1, and 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 1. Prior to CS_{fuse} executing Arrow 5, CS_{mod} issues both its transaction to update the RDBMS and delete command to update the KVS. Next, CS_{fuse} inserts k_j-v_j into the KVS. This schedule, see Figure 2.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_{mod} 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 2.b shows an interleaved processing that satisfies these conditions, resulting in an undesirable race condition. The race condition of Figure 2.a does not result in an inconsistent state and is acceptable.

C Gumball Technique

Gumball Technique (GT) is designed to prevent the race conditions of Section B 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 Figure 3. Its details are as follows. When the server

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 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 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-$T_{miss} > \Delta$) then ignore the put operation.
3.	If (g_i exists and T_{miss} 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.
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}.

Figure 3: GT enabled delete, get, and put pseudo-code. All time stamps are local to the server containing k_i-v_i.

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, see Section C.1.1.

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 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, see discussions of Figure 2 in Section B. For example, with the acceptable race condition of Figure 2.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.

C.1 Value of Δ

Ideally, Δ should be set to the elapsed time from when CS_{fuse} observes a KVS miss for k_i to the time it issues a put(k_i,v_i,T_{miss}) operation. Δ values greater than ideal are undesirable because they cause gumballs to occupy memory longer than necessary, reducing the KVS hit rate of the application. Δ values lower than ideal cause GT to reject KVS insert operations un-necessarily, see Step 2 of the put pseudo-code in Figure 3. They slow down a CASQL significantly because they prevent the server from caching key-value pairs. In one experiment, GT configured with a small Δ value slowed the system down ten folds by causing the KVS to sit empty and re-direct all requests to the RDBMS for processing. This section describes how GT computes the value of Δ dynamically.

C.1.1 Dynamic computation of Δ

GT adjusts the value of Δ dynamically in response to CASQL load to avoid values that render the KVS empty and idle. The dynamic technique is based on the observation that the KVS server may estimate the CS_{fuse} response time, RT, by subtracting the current time (T_C) from T_{miss}: $RT = T_C - T_{miss}$. When a put is rejected because its RT is higher than Δ, GT sets the value of Δ to this RT multiplied by an inflation (α) value, Δ=$RT \times \alpha$. For example, α might be set to 1.1 to inflate Δ to be 10% higher than the maximum observed response time. (See below for a discussion of α and its value.)

Increasing the value of Δ means requests that observed a miss prior to this change may now pass the race condition detection check. This is because GT may have rejected one or more of these put requests with the smaller Δ value when performing the check $T_C - T_{miss} > \Delta$. To prevent such requests from polluting the cache, GT maintains the time stamp of when it increased the value of Δ, T_{adjust}. It ignores all put operations with T_{miss} prior to T_{adjust}.

GT reduces the value of Δ when a k_i-g_i is replaced with a k_i-v_i. It maintains the maximum response time, RT_{max}, using a sliding window of 60 seconds (duration of sliding window is a configuration parameter of the KVS server). If this maximum multiplied by an inflation value (α) is lower than the current value of

Δ then it resets Δ to this lower value, $\Delta = RT_{max} \times \alpha$. Note that decreasing the value of Δ does not require setting T_{adjust} to the current time stamp: Those put requests that satisfy the condition $T_C - T_{miss} > \Delta$ will continue to satisfy it with the smaller Δ value.

The dynamic Δ computation technique uses α values greater than 1 to maintain Δ slightly higher than its ideal value. In essence, it trades memory to minimize the likelihood of Step 2 of the put pseudo-code (see Figure 3) from ignoring its cache insert unnecessarily and redirecting future references to the RDBMS. This prevents the possibility of an application observing degraded system performance due to a burst of requests that incur KVS misses and are slowed down by competing with one another for RDBMS processing. Moreover, gumballs have a small memory footprint. This in combination with the low probability of updates minimizes the likelihood of extra gumballs from impacting the KVS hit rate adversely.

D Evaluation

This section analyzes the performance of an application consistency technique with and without GT using a realistic social networking benchmark based on a web site named RAYS. We considered using other popular benchmarking tools such as RUBiS [3], YCSB [13] and YCSB++ [32]. We could not use RUBiS and YCSB [13] because neither quantifies the amount of stale data. The inconsistency window metric quantified by YCSB++ [32] measures the delay from when an update is issued until it is consistently reflected in the system. This metric is inadequate because it does not measure the amount of stale data produced due to race conditions by multiple threads. Below, we start with a description of our workload. Subsequently, we present performance results and characterize the performance of Gumball with different Δ values.

D.1 RAYS and a Social Networking Benchmark

Recall All You See (RAYS) [22] 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. 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 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 these friends to view their streams.

We use two popular navigation paths of RAYS to evaluate GT: Browse and Toggle streaming (Toggle for short). While Browse is a read-only workload, Toggle results in updates to the database requiring the key-value pairs to remain consistent with the tabular data. 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. It issues 38 SQL queries to the RDBMS. With a CASQL,

Database parameters	
ω	Number of users in the database.
ϕ	Number of friends per user.
Workload parameters	
$\mathcal{N}$	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.
μ	Probability of a user invoking the Toggle sequence.

Table 1: Workload parameters and their definitions.

Browsing issues 8 KVS get operations. For each get that observes a miss, it performs a put operation. With an empty KVS, the get operations observe no 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 SQL queries. With a CASQL, Toggle issues 7 get operations and, with an empty KVS, observes a miss for all 7. This causes Toggle to perform 7 put operations to populate the KVS. 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. With Trig, these updates invoke triggers that delete KVS entries corresponding to both the profile[2] and devices pages. With a populated KVS, the number of deletes is higher because each toggle invalidates the "Live Friends" section of those friends with a KVS 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 Browsing sequence with probability $(1 - u)$. There is a fixed think time ϵ between the user clicks that constitute a sequence.

The workload generator maintains both the structure of the synthetic database and the activities of different users to detect key-value pairs (HTML pages) that are not consistent with the state of the tabular data, termed *stale* data. The workload generator produces unique users accessing RAYS simultaneously. 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 a system with and without GT are comparable because the same workload is used with each alternative.

[2]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.

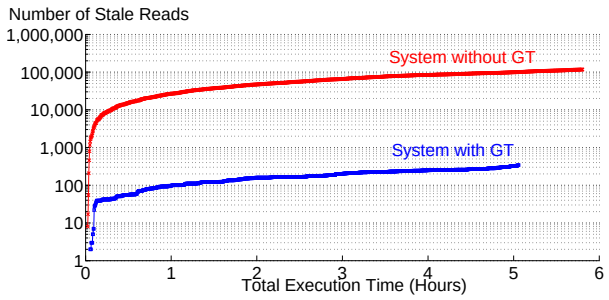


Figure 4: Number of stale reads in one experiment; ω=1000, ϕ=10, $\mathcal{N}$=100, n=10,000, ϵ=θ=0, μ=1%.

To measure the amount of stale data with 100% accurracy, the workload generator must maintain the status of different devices managed by RAYS and serialize simultaneous user requests and issue one to CASQL at a time. This is unrealistic and would eliminate all race conditions. Instead, we allowed the workload generator to issue requests simultaneously and used time stamps to detect its internal race conditions. This results in false positives where an identified stale data is due to an in-progress change to a time stamp. These false positives are observed when the workload generator is using RDBMS only.

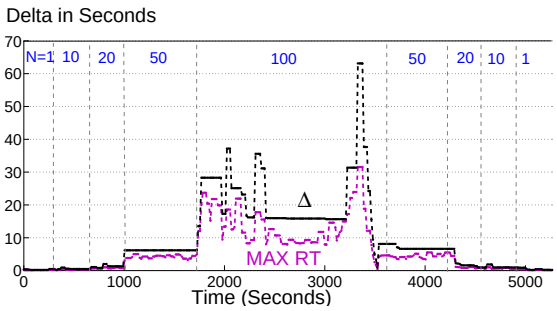
D.2 Performance Results

We conducted many experiments to quantify a) the amount of stale data eliminated by GT, b) the impact of GT on system performance, and c) how quickly GT adapts Δ to changing workload characteristics. In all experiments, GT reduced the amount of stale data 100 folds or more. Below, we present one experiment with 300 fold reduction in stale data and discuss the other two metrics in turn.

In this experiment, we focus on an invalidation based technique implemented in the application. We target a small database that fits in memory to quantify the overhead of GT. With larger data sets that result in cache misses, the application must issue queries to the RDBMS. This results in higher response times that hide the overhead of GT. If race conditions occur frequently then GT will slow down a CASQL by reducing its cache hit rate. In our experiments, race conditions occur less than 3% of the time[3]. Thus, GT's impact on system performance is negligible.

Figure 4 shows the number of requests that observe stale data when a system is configured to either use GT or not use GT. The x-axis of this figure is the execution time of the work-load. The y-axis is log scale and shows the number of requests that observe stale data. With GT, only 343 requests (less than 0.009% of the total number of requests) observe stale data. We attribute these to the false positives produced by our workload generator, see Section D.1. Without GT, more than 100,000 requests (2.4% of the total requests) observe stale data. The cache hit rate is approximately 85% with and without GT. Even though the database

[3] Approximated by the amount of stale data produced without GT.



2) Value of Δ.

Figure 5: Varying system load; ω=1000, ϕ=2, n=10,000, ϵ=100 msec, θ=0, μ=10%.

is small enough to fit in memory, the cache hit rate cannot approximate 100% because 10% of requests execute Toggle (μ=10%) which invalidate cache entries.

GT adapts to changing workloads by adjusting the value of Δ. We varied system load by varying the number of simultaneous users accessing the system, $\mathcal{N}$. We looked at different patterns ranging from those that change the load abruptly (switch from 1 to 100 simultaneous users) to those that change the load gracefully. In each case, GT adjusts the value of Δ quickly, minimizing the number of KVS inserts rejected due to a small value of Δ. Such rejections are typically a negligible percentage of the total number of requests processed. Below, we report on one experiment.

This experiment varied the number of simultaneous users ($\mathcal{N}$) from 1 to 10, 20, 50, 100 and back to 50, 20, 10 and 1. For each setting, a user issues 1000 requests. Figure 5 shows the value of Δ when compared with the maximum observed RT, see discussions of Section C.1.1. As we increase the load, GT increases the value of Δ to prevent rejection of KVS inserts unnecessarily. Similarly, when we decrease the load, GT reduces the value of Δ to free memory by preventing gumballs from occupying the cache longer than necessary. Δ is higher than the observed maximum response time because its inflation value is set to 2. This experiment issues more than two hundred thousand put requests and GT rejects fewer than 600 due to small Δ values.

E Related Work

Cache augmented RDBMSs have been an active area of research dating back to 1990s [27, 12, 41, 18, 15, 10, 31, 29, 1, 40, 30, 9, 17, 2, 4, 5, 33, 24]. The focus of this paper is on a subset with the following characteristic. First, the cache must support simple insert, get, and delete operations [6, 33]. There exist complex caches with the ability to process SQL queries, e.g., TimesTen [40], DBProxy [2], DBCache [31, 9], Cache Tables [1], MTCache [30], Ferdinand [21]. While these fall beyond the focus of GT, a KVS might be their building component which may render GT relevant. Second, the cache (KVS) must be at the same

abstraction as the RDBMS where security and privacy of content is guaranteed by the application and its infrastructure [27, 12, 41, 18, 15, 29, 4, 5, 33, 24, 23]. It does not apply to proxy caches [10, 16, 17] that are external to the application.

TxCache [33] presents the concept of race conditions and how they may leave the cache (KVS) inconsistent with the RDBMS. It proposes a slight modification to those RDBMSs that implement MVCC [8] to (a) maintain validity interval for read-only queries whose results are represented as key-value pairs, and (b) generate *invalidation* streams in the presence of update transactions. This stream contains transaction's time stamp and all invalidation tags corresponding to key-value pairs in the cache. This enables TxCache to implement transactions with snapshot isolation. GT is different in several ways. First, while both GT and TxCache employ time stamps to detect race conditions, the semantics of these time stamps are different. GT's time stamp pertains to when KVS fails to produce a value for a referenced key while TxCache's time stamp is the update transaction time stamp issued by the RDBMS. Second, GT is a building block of a consistency technique while TxCache is a consistency technique. Certain properties of a transaction (atomicity, consistency, isolation, durability) may impose significant overhead and not be applicable to an application [38, 11, 24, 32]. A consistency technique may abandon these properties while utilizing GT to prevent race conditions that leave the cache inconsistent permanently. The same cannot be realized with TxCache because it is a consistency technique that implements transactions with snapshot isolation. Third, GT can be used with all SQL RDBMSs because it does not either modify or require a prespecified functionality from the RDBMS.

One approach to prevent race conditions is to serialize all writes using the RDBMS [24]. If the RDBMS is the system bottleneck, GT is a better alternative because it imposes no additional load on the RDBMS. Its entire implementation is by the cache.

Tombstones [20] mark deleted objects in systems that allow replica content to diverge [37]. A key design challenge is when to delete them. Some systems employ a two-phase commit protocol to delete tombstones safely [25, 34, 36]. To continue operation in the presence of failures and intermittent network connectivity, some studies have proposed deletion of tombstones at each site after a fixed period, long enough for most updates to complete propagation, but short enough to keep the space overhead low [39, 28]. Clearinghouse [19] removes tombstones from most sites after the expiration period and retains them on a few designated sites indefinitely. When a stale operation arrives after the expiration period, some sites may incorrectly apply that operation. However, the designated sites will distribute an operation that re-installs tombstones on these sites.

A gumball is similar to a tombstone because it identifies a deleted key-value pair. However, GT's management of gumballs is novel. First, GT deletes a gumball after a fixed time interval Δ. GT controls the value of Δ dynamically using the response time of the key-value pair, i.e., the RDBMS service time and queuing delays attributed to system load. Second, GT ignores put operations for k_i-v_i that incur response

times greater than Δ (independent of gumball existence). GT is not a substitute for tombstone management algorithms and their assumed optimistic replication environment and vice versa.

F Conclusion

GT is a race condition detection and prevention technique for mid-tier in-memory caches that complement a RDBMS to enhance performance. This simple technique works with all RDBMSs and alternative invalidation-based approaches to cache consistency. Its evaluation highlights the need for a benchmark that can accurately measure the amount of stale data produced by a framework. Our social networking benchmark suffers from a few false positives (thousandth of one percent of issued request). These should be eliminated without slowing down the workload generator.

G Acknowledgments

We wish to thank the anonymous refrees of DBSocial 2012 for their valuable comments.

References

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

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

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

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *ICDE*, 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] Six Apart. Memcached Specification, http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt.

[7] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio A. F. Almeida. Characterizing user behavior in online social networks. In *Internet Measurement Conference*, 2009.

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

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

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *SIGMOD*, pages 97–108, 2002.

[17] 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.

[18] 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.

[19] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic Algorithms for Replicated Database Maintenance. In *Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[20] M. J. Fischer and A. Michael. Sacrificing Serializability to Attain Availability of Data in an Unreliable Network. In *Symposium on Principles of Database Systems (PODS) Conference*, pages 70–75, 1982.

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

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

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

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

[25] R. Guy, G. Popek, and T. Page. Consistency Algorithms for Optimistic Replication. In *IEEE International Conference on Network Protocols*, 1993.

[26] V. Holmedahl, B. Smith, and T. Yang. Cooperative Caching of Dynamic Content on a Distributed Web Server. In *HPDC*, pages 243–250, 1998.

[27] 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.

[28] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computing Systems*, 10:3–25, February 1992.

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

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

[31] 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.

[32] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Cloud Computing*, New York, NY, USA, 2011. ACM.

[33] 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.

[34] D. Ratner. Roam: A Scalable Replication System for Mobile and Distributed Computing, Ph.D. thesis, UC Los Angeles, Tech Report UCLA-CSD-970044, 1998.

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

[36] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *Conference on Distributed Computing (DISC)*, pages 297–314, 2000.

[37] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 5:12–27, 2005.

[38] M. I. Seltzer. Beyond Relational Databases. *Commun. ACM*, 51(7):52–58, 2008.

[39] H. Spencer and D. Lawrence. Managing Usent, O'Reilly & Associates, Sebastopol, CA, ISBN 1-56592-198-4, 1998.

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

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