A Comparison of Two Cache Augmented SQL Architectures*

Shahram Ghandeharizadeh and Hieu Nguyen
Database Laboratory Technical Report 2018-04
Computer Science Department, USC
Los Angeles, California 90089-0781
{shahram,hieun}@usc.edu

Abstract

Cloud service providers augment a SQL database management system with a cache to enhance system performance for workloads that exhibit a high read to write ratio. These in-memory caches provide a simple programming interface such as get, put, and delete. Using their software architecture, different caching frameworks can be categorized into Client-Server (CS) and Shared Address Space (SAS) systems. Example CS caches are memcached and Redis. Example SAS caches are Java Cache standard and its Google Guava implementation, Terracotta BigMemory and KOSAR. How do CS and SAS architectures compare with one another and what are their tradeoffs? This study quantifies an answer using BG, a benchmark for interactive social networking actions. In general, obtained results show SAS provides a higher performance with write policies playing an important role.

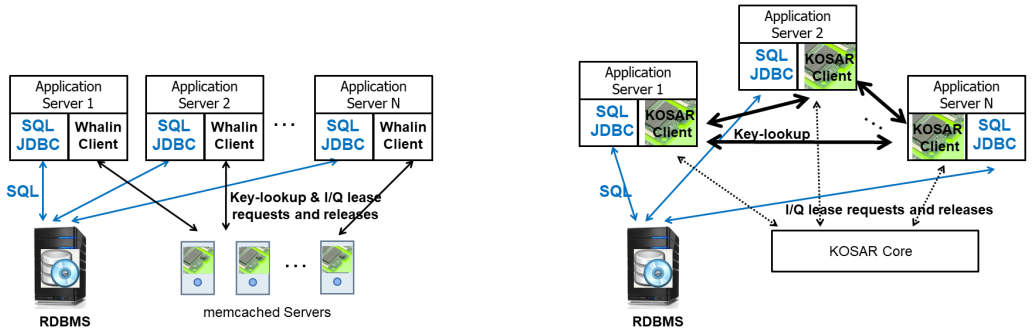
1 Introduction

Cache Augmented Database Management Systems (CADBMSs) are a proven technology deployed widely by popular social networking sites such as Facebook [36], Tinder, and Wikipedia. These caches require an application developer to identify a code path or a method (function) with a unique input as a key and its results as a value [38, 26, 21, 36]. Next, the code path is extended to look up the key. If the cache returns a value then the application consumes the value without executing the code path. Otherwise, it executes the code path to generate the corresponding key-value pair and inserts this key-value in the cache for future look up. In the presence of updates to the database, the application may either delete (termed *write-around* or invalidate) or update the impacted key-value pairs (termed *write-through* or refill). These are similar to the write-around and write-through techniques of host-side[1] caches [31, 9, 27, 30].

*A shorter version of this paper appeared in the *Tenth TPC Technology Conference on Performance Evaluation and Benchmarking*, TPCTC 2018, Rio De Janerio, Brazil, August 2018.

[1]A key difference is that the write is manipulating a key-value pair instead of a disk/SSD block.

There is a spectrum of software architectures for CADBMSs. We term the two extreme ends of this spectrum as the Client-Server (CS) and the Shared-Address Space (SAS) architectures. Both provide a simple interface such as get, insert/put, delete, increment, decrement, and others. Moreover, both may use leases such as those proposed by [23] to prevent undesirable race conditions that cause the cache to generate stale data.



1.a Client Server (CS)

1.b Shared Address Space (SAS)

Figure 1: Two alternative architectures for cache augmented database management systems.

With CS, there are stand-alone cache manager processes per server. Examples include memcached [35, 44] and Redis [39]. Figure 1.a shows this architecture for memcached. An application server uses memcached's client component, Whalin Client [44], to issue requests to an instance of memcached process executing on a memcached server[2]. With multi-core servers, a cache server may host multiple[3] memcached processes, each process is termed a *cache manager instance*. The memory of the cache server is partitioned across these instances. Cache entries would be represented as key-value pairs and assigned to different instances of memcached. This might be realized by partitioning key-value pairs using either a hash function, by assigning ranges of keys, or a hybrid of the two [1].

With SAS, the cache manager is a library that runs in the address space of an application, see KOSAR Client of Figure 1.b. This library manages memory and implements a replacement technique such as LRU. We use the term *cache manager instance* to refer to an instance of the library running as a part of an application node. A coordinator implements leases, see KOSAR Core of Figure 1.b. As depicted in Figure 1.b, the cache manager instances may communicate and collaborate with one another to facilitate key-lookup. Different cache manager instances may reference and compute the same key-value pair independently. A coordinator, KOSAR Core, maintains consistency between different cache manager instances. Examples of SAS architecture include Java Cache standard [28] along with its Apache Ignite [4] and Google Guava [25] implementations, Terracotta BigMemory [42], JBoss [10] and

[2]A physical server may host both the Application and memcached processes. We use such a deployment to evaluate CS architecture in Section 5.

[3]While memcached is multithreaded, launching multiple instances reduces contention for its synchronization primitive on its LRU queue. In our experiments with a 16 core processor, we have observed a 20% enhancement in throughput when launching eight instances instead of one.

KOSAR [19]. Both Java Cache standard and Google Guava are single node implementations and lack the concept of a coordinator.

While hybrids of CS and SAS are possible, this study focuses on CS and SAS only[4].

Software architecture of SAS and CS are fundamentally different. With CS, there is a cache manager client and a cache manager server that communicate via message passing. SAS combines these two into one with a published interface for use by the application server. Moreover, it has a coordinator that grants leases to prevent undesirable race conditions that insert stale data in the cache.

The architectural differences cause CS and SAS to process read and write requests differently. With CS, a read that looks up the value of a key incurs the network overhead to issue the request to a cache manager instance. This is in the form of serializing the command and transmitting it across the network back to the client. If the cache manager instance finds the value then it serializes the value and transmit it across the network. SAS avoids this network overhead every time an application node finds its referenced key in the memory of its local cache manager. With SAS, writes manipulate the in-memory representation of a key-value pair. With CS, these writes incur the network overhead of transmitting the key-value pair similar to reads.

A key question is how CS and SAS architectures compare with one another? And, do they scale horizontally? The **primary** contribution of this study is an answer to these questions using KOSAR [19] as the representative of the SAS architecture[5] and IQ-Twemcached [22] as the representative of the CS architecture[6]. While we acknowledge the availability of other candidate systems, we selected KOSAR and IQ-Twemcached because they are the only systems that implement leases of [23] to provide strong consistency. To elaborate, both CS and SAS architectures suffer from undesirable race conditions that insert stale data in the cache [23]. Leases of [23] detect these race conditions and prevent them from causing the state of the cache and the database to diverge.

While both KOSAR and IQ-Twemcached may be configured to control the number of key-value replicas across the cache, we configured IQ-Twemcached in its simplest form where clients hash partition key-value pairs with no replication. We configured KOSAR such that each client may construct a replica of its referenced key-value pair. This means that with N cache manager instances, KOSAR may construct N replicas of a popular key-value pair while IQ-Twemcached has at most one replica of the same key-value pair in one cache manager instance. KOSAR does not guarantee N replicas of a popular key-value pair because cache replacement policy of each cache manager instance operates independent of the others.

CADBMSs are designed for workloads that exhibit a high read to write ratio. Examples include social networking applications with Facebook reporting 500 reads for every 1 write [8] and enterprise systems trending towards read-dominated workloads [32, 16]. For this evaluation, we use a macro benchmark that emulates interactive social networking actions [6] and generate read-heavy workloads. We do not use TPC-C [11] because its workload is write-heavy with 92% of transactions performing writes [15]. Several studies have shown the CS architecture using write-around does not enhance the performance of TPC-C work-

[4]Evaluation and comparison of a hybrid with CS and SAS is deferred to future work, see Section 6.

[5]At the time of this writing, Google Guava is not distributed and could not be used for this evaluation. Apache Ignite is a candidate for future analysis, see Section 6.

[6]Redis is the other obvious alternative that we intend to include in our future studies, see Section 6.

load [12, 2]. This is due to frequent write operations that invalidate key-value pairs and cause subsequent reads to populate the cache only to be invalidated again. See Section 6 for additional details.

Our evaluation highlights the following main lessons:

1. SAS provides a higher performance than CS. However, CS scales better than SAS for some workloads.

2. Choice of a write policy (write-around versus write-through) impacts both the performance observed with each architecture and their scalability characteristics. The write-through policy provides superior performance and scalability characteristics when compared with write-around. However, its resulting software is more complex, see Section 3.2.

3. One may configure client components of a SAS architecture in either a greedy or a cooperative mode. In the greedy mode, each client manages its key-value pairs independent of the other clients. In the cooperative mode, a client with a replica of a key-value pair (k_i-v_i) services another client's cache miss for k_i. Our evaluation shows cooperative is superior to greedy with write-around, enhancing both the performance and scalability of the SAS architecture. With write-through, there is little difference between greedy and cooperative. See Section 4.

4. Both architectures scale sub-linearly even with a 1000:1 read to write ratio and a handful of nodes. While scalability of SAS is limited by the processing capability of the RDBMS server, scalability of CS is limited due to load imbalance across cache manager servers.

All performance numbers presented in this paper are gathered using a cluster of servers. Each server is a 4 (8 hyper-threaded) core Intel i7-3770 3.40 GHz CPU with 16 GB of memory, a 1 Gbps networking card, and a 1 Terabyte disk.

The rest of this study is organized as follows. We describe related work in Section 2. Section 3 provides an overview of the BG benchmark and its implementation using write-around and write-through techniques. While the write-around implementation is similar with CS and SAS, we present a fine-tuned implementation of write-through for each of CS and SAS architectures. Section 4 quantifies the performance benefits of using a cooperating technique to manage the content of caches with SAS, showing it is key to scaling SAS. Section 5 compares the CS and SAS architectures using their performance and scalability characteristics. Our conclusions and future research directions are detailed in Section 6.

2 Related Work

In a data center deployment, caches may be deployed either inside or outside the RDBMS. Caches outside the RDBMS include host-side caches [9, 27, 30] and application-side caches [38, 26, 21, 36]. Caches inside the RDBMS include the buffer pool manager of a RDBMS server [40, 37, 29, 34], client-side [14, 17, 43] and mid-tier [33, 7] caches.

Host-side, RDBMS server, client-side and mid-tier caches are transparent to an application. Host-side caches are deployed seamlessly using a storage stack middleware or the operating system (termed the caching software) [9, 27, 30]. They stage disk pages from disk to NAND Flash to expedite their processing. The RDBMS server cache may employ algorithms such as LRU [13], LRU-K [37] and its queue based implementation [29], and ARC [34] to manage buffer frames occupied by disk pages [40]. Client-side caches [14, 17, 43] distribute the processing of the algebraic operators that constitute a query across both the client and server component of a RDBMS. They ship data to a client for caching and processing. The book by Franklin [17] provides a survey of these caches. Mid-tier caches offload part of a workload to intermediate database servers that partially replicate data from the back-end server [33, 7]. These may cache entire tables, materialized views, or query fragments, providing distributed query execution.

This study considers application-side caches that are external to the RDBMS and managed by the application. These caches provide a simple get, insert, delete, increment, decrement interface and have no query processing ability. We evaluate a non-transparent version of these caches. Our initial attempt to evaluate a transparent version of one of these caches produced results that we could not explain. A future research direction is to revisit this cache and others similar to it in light of the lessons learned from this study to explain their performance and scalability characteristics.

In [2], we compare a host-side cache named Flashcache with an application-side cache based on the CS architecture (IQ-Twemcached [23]). The same IQ-Twemcached is used in this study. This study is different because we compare two application-side caches based on different software architecture with one another, namely, KOSAR and IQ-Twemcached.

One finds studies and white papers on the web comparing the CS and SAS architectures with one another. However, to the best of our knowledge, this study is the first to compare the CS and SAS architectures scientifically. We show the application's use of write-around and write-through policies impact the final conclusions. This level of analysis is lacking from the studies available on the web.

3 A Social Networking Benchmark

We use BG [3, 6], a benchmark that produces interactive social networking actions, to evaluate the alternative caching architectures. Rows of Table 1 show the seven BG actions that constitute our focus. Three of these actions read data while the other four write data. The read actions are View Profile, List Friends, and View Pending Friend requests. The write actions are Invite Friend, Reject Friend Request, Accept Friend Request, and Thaw Friendship. We configure BG with three different mixes of these actions that emulate a different ratio of read to write actions, varying from 10:1 to 1000:1, see Table 1. According to [8], the ratio of read to write actions at Facebook is 500:1.

We used the Social Action Rating (SoAR) metric of BG to compare the SAS and CS architectures with one another. SoAR is the highest throughput observed from the system while satisfying a service level agreement, SLA. In all our experiments, the SLA was set at 95% of actions observing a response time faster than 100 milliseconds with no anomalies [45]. An anomaly refers to either stale, inconsistent, or simply erroneous data produced by the

system. This metric is quantified during the validation phase of BG.

BG Social Actions	Read to Write Ratio		
	1000:1	100:1	10:1
View Profile	33.3%	33%	30%
List Friends	33.3%	33%	30%
View Friend Req	33.3%	33%	30%
Invite Friend	0.04%	0.4%	4%
Accept Friend Req	0.02%	0.2%	2%
Reject Friend Req	0.02%	0.2%	2%
Thaw Friendship	0.02%	0.2%	2%

Table 1: Three interactive social networking workloads.

The three read actions of BG emulate a socialite either viewing the profile of a member, listing her friends, or listing her pending friend requests. The referenced member may be the same as the socialite, e.g., viewing her own profile or list of friends. View Profile retrieves the profile of the referenced member and includes the member's number of friends. If this is a self reference then the number of pending friend invitations of the member is also retrieved. List Friends shows the profile of ten friends of a member. Similarly, List Pending Friends shows the profile of ten members who have extended a friendship invitation to the referenced member.

The four write actions pertain to social activities that members may perform on one another. These actions are also invoked by a socialite on another member. They are self explanatory based on their names and we refer the interested reader to [6] for the details.

BG is a stateful benchmark in that it only generates valid actions. For example, it does not emulate a socialite to extend a friend invitation (using Invite Friend Request) to another member if they are already friends. Similarly, a socialite performs Thaw Friendship on a member who is a friend of that socialite. To prevent the social graph from either running out of friendships (for the Thaw Friendship action) or members to invite (for the Invite Friend Request), we ensured the mix of write actions is symmetric so that friendships are thawed and created with the same probability, see Table 1.

BG uses a closed emulation model to generate request where a thread emulates a socialite picked using a Zipfian distribution of access. We used 0.27 as the exponent of the distribution to generate a skewed pattern of reference where 30% of members serve either as socialites or referenced members by 70% of generated actions.

The version of BG used in this study is different than [6] in that all BGClients share[7] one social graph to generate requests, see Integrated DataBase (IDB) of [3]. This ensures different application servers incur read-write, write-read, and write-write conflicts. We used the feature of BG to detect anomalies (stale, erroneous, or simply wrong) to verify the two architectures implement actions correctly.

[7]The version of BG described in [6] partitions a social graph into N logical subgraphs and assigns each to a BGClient for request generation. It fails to evaluate the SAS architecture objectively because its request generation results in no read-write and write-write conflicts between different BGClients, i.e., emulated application servers of Figure 1.

With write-around, both the representation of key-value types and their management is identical with the alternative architectures. However, an implementation of write-through with CS is different than with SAS. We fine-tune the representation of key-value pairs with SAS because its implementation of read-modify-write (RMW) incurs a lower overhead than CS, see Section 3.2 for details.

Below, we provide an overview of BG actions with write-around and write-through in turn. Section 3.3 presents how the Inhibit (I) and Quarantine (Q) leases are applied to prevent insertion of stale values in the IQ-Twemcached.

Read Action	jProfile	iFrdList	iPndList
View Profile(j)	$V_{jProfile}$		
List Friends(i)	$\{V_{jProfile}\}$	$V_{iFrdList}$	
View Friend Req(i)	$\{V_{jProfile}\}$		$V_{iPndList}$

Table 2: Last three columns identify the keys used to implement BG's read actions shown as rows with a member id as their input argument.

Write Action	Inviter i		Invitee j		
	iProfile	iFrdList	jProfile	jFrdList	jPndList
Invite Friend			✖		✖
Accept Friend Request	✖	✖	✖	✖	✖
Reject Friend Request			✖		✖
Thaw Friendship	✖	✖	✖	✖	

Table 3: Keys invalidated by BG's write actions.

3.1 Write-Around

With write-around, BG's write actions identify keys whose values are no longer valid and delete (invalidate) them from the cache. A subsequent reference for these keys populates the cache with valid values by querying the RDBMS.

An implementation of BG's actions maintains three types of key-value pairs: Member profile, friend list, and pending list of friend invitations. Each is specific to a member and its key is constructed by concatenating the referenced member id (i) with a corresponding string token: "Profile", "FrdList" and "PndList". These keys are shown as the columns of Table 2. Rows of this table are the read action.

The View Profile action looks up the value $V_{jProfile}$ of the jProfile key. This value contains the member attributes (firstname, lastname, date of birth, join date, etc.,) of the member with userid j. It includes value of simple analytics such as the member's number of friends and pending friend invitations as required by BG. (A write action may invalidate a jProfile key to maintain its simple analytics value consistent with the database, see discussions of Table 3.)

The List Friends action fetches the profile of ten friends of Member i. It looks up iFrdList to obtain $V_{iFrdList}$, a set containing the f profile of Member i's friends: {jProfile$_1$, jProfile$_2$,

..., jProfile$_f$}. Each jProfile key identifies the profile of a friend. The action retrieves the value of 10 such keys. Assuming Member i has 10 or more friends, this action retrieves the value of 11 keys.

View Friend Request is similar to List Friends and maintains a iPndList key associated with a $V_{iPndList}$ value. $V_{iPndList}$ contains a list of keys identifying profile of those j members who invited Member i to be friend. This action also fetches ten profiles from the list $V_{iPndList}$.

Table 3 shows the keys invalidated by write actions. While Invite Friend and Reject Friend Request invalidate two keys, Accept Friend Request invalidates 5 keys in the cache. These deleted key-value pairs correspond to the two members i and j that are impacted by the write actions. To illustrate, consider the Invite Friend action. It consists of an inviter i (the socialite) extending a friend invitation to an invitee j. The jProfile of invitee must be invalidated because the value of simple analytic pertaining to its number of pending friend invitations has changed. Similarly, the jPndList of the invitee j must be invalidated because the inviter's profile key (iProfile) must be added to this list.

Once the invitee j accepts the friend invitation of the inviter i (see the Accept Friend Request action of Table 3), both member's profile keys (iProfile and jProfile) must be invalidated because this list is now obsolete: the jProfile should be added to iFrdList and iProfile should be added to jFrdList. Morover, the keys iFrdList and jFrdList must be invalidated because each should be added to the list of the other. Finally, the jPndList of the invitee j is void as the iProfile must be removed from this list since the inviter is now a friend (and no longer an inviter). A similar rationalization describes the list of keys voided by the Reject Friend Request and Thaw Friendship actions.

3.2 Write-Through

Write Action	Inviter i		Invitee j		
	iProfile	iFrdList	jProfile	jFrdList	jPndList
Invite Friend			Invites^{++}		PUSH(iProfile)
Accept Friend	Friends^{++}	PUSH(jProfile)	Friends^{++}	PUSH(iProfile)	POP(iProfile)
Reject Friend			Invites^{--}		POP(iProfile)
Thaw Friendship	Friends^{--}	POP(jProfile)	Friends^{--}	POP(iProfile)	

Table 4: With SAS, write actions modify the key-value pairs directly. PUSH and POP operations employ the invitee/inviter key of MProfile.

Read Action	jProfile	iFrdList	jFrdCnt	iPndList	jPndCnt
View Profile(j)	$V_{jProfile}$		$V_{jFrdCnt}$		$V_{jPndCnt}$
List Friends(i)	$\{V_{jProfile}\}$	$V_{iFrdList}$			
View Friend Requests(i)	$\{V_{jProfile}\}$			$V_{iPndList}$	

Table 5: Last five columns identify the keys looked up by BG's read actions (rows) with write-through. With a miss, the system computes the value and inserts it in the cache.

Write Action	Inviter i		Invitee j			
	iFrdList	iFrdCnt	jFrdList	jPndList	jFrdCnt	jPndCnt
Invite Friend				Append		+1
Accept Friend Request	Append	+1	Append	RMW	+1	-1
Reject Friend Request				RMW		-1
Thaw Friendship	RMW	-1	RMW		-1	

Table 6: With CS, BG's write actions are implemented using Read-Modify-Write (RMW) and incremental updates.

With write-through, the implementation of BG actions and their referenced key-value pairs can be fine tuned for the software architecture of SAS and CS. Below, we describe each implementation in turn, starting with SAS.

With the **SAS architecture**, we implemented the read actions using the same type of key-value pairs as write-around, see Table 2. Key difference is that the write actions update the values instead of deleting them. For example, when an inviter i extends a friend invitation to invitee j, we increment the number of invites of jProfile by one to reflect an increase in the number of friend invitations. We also push the iProfile (key identifying the inviter's profile) in the list of pending friend invitations of invitee, jPndList. Once the invitee accepts the inviter's invitation (Accept Friend Request), we increment the number of friends of the iProfile and jProfile. Next, we push the inviter's iProfile key in the jFrdList of invitee and vice versa. Finally, we pop the inviter's iProfile key from jPndList of invitee since they are friends now. Table 4 shows the impact of Reject Friend Request and Thaw Friendship on the values of different keys.

With the **CS architecture**, Table 5 shows a total of 5 key-value pairs are used to implement BG's read action. The primary change is the separation of the number of friends (jFrdCnt) and pending friend invitations (jPndCnt) from member profile (jProfile). This separation enables us to use the increment and decrement methods of IQ-Twemcached to implement the impact of write actions on simple analytics, see Table 6. It is advantageous because it implements the analytics with one message instead of two required by RMW. A draw back of this design is that a read action such as View Profile must now retrieve multiple key-value pairs: 1) jProfile, and 2) jFrdCnt. In case of a self reference, jPndCnt, must also be retrieved to show the number of pending friend invitations.

Table 6 shows the FrdList and PndList are maintained using either IQ-Twemcached's Append functionality or RMW. They are used to insert and delete profile keys from a list, respectively. To elaborate, a write action such as Invite Friend appends the inviter i's profile key to the jPndList of invitee j. Once the invitee j accepts the friendship invitation, the inviter i's key must be removed from the jPndList using RMW: the action reads the value (a list) of jPndList, modifies it by removing the profile of the inviter i's profile key from the list, and writes the value (list) back as the value of jPndList.

3.3 Strong Consistency

Both write-around and write-through use the IQ framework [23] to prevent undesirable race conditions that cause the cache to produce stale data. Below, we provide an overview

of this framework and the compatibility of its leases with write-around and write-through.

The IQ framework defines a *session* to consist of a single RDBMS transaction and multiple key-value look ups or invalidations. A session is assigned a unique identifier and is atomic. When a session observes a cache miss, it must obtain an I lease in order to query the RDBMS. Multiple I leases on the same key are not compatible with one another, causing one to backoff and try again. Hence, given a key-value pair, at most one thread may compute it by querying the RDBMS. All other concurrent threads consume the key-value computed and inserted by this thread.

A common cause of stale data is a RDBMS's use of snap-shot isolation. This mechanism causes a RDBMS query (due to a cache miss) to race with a write action (a RDBMS transaction) to compute an obsolete value for insertion in the cache. To invalidate a key-value pair, a session must obtain a Q lease on that key-value pair. A Q lease voids an I lease on the same key. In order for a cache insertion to succeed, the thread must provide a valid I lease. An I lease voided by a Q lease prevents insertion of the key-value pair, preventing insertion of stale data due to use of snap-shot isolation.

With write-around, Q leases are compatible with one another and allowed to proceed in parallel because deletion of a key by multiple threads does not suffer a write-write race condition. However, with write-through's use of RMW and incremental changes (e.g., increment and decrement), the Q leases are no longer compatible due to potential write-write race conditions. Once a Q lease for a key arrives, if there is an existing Q lease on the referenced key, then the requester is aborted. This causes the session to abort its RDBMS transaction, release all its leases, and try again.

BG's write actions are simple and a session knows all impacted keys in advance, see Table 6. Both the IQ-Twemcached and KOSAR implementations use the following optimization. A session sorts its referenced keys and requests a Q lease on each in turn. Should the session encounter an abort message for one of the keys, it backs-off and tries to acquire the Q lease on that key again. After several attempts it abort the session. This optimization is effective because all sessions acquire Q leases on sorted keys.

With write-through, when a session updates the value of a key, the new value is not visible to other concurrently executing sessions. Once a session commits, with the IQ-Twemcached server, the value of a key is updated and becomes visible. With KOSAR, the KOSAR Client processing the session propagates the change to all other KOSAR Clients with a copy of the impacted key and updates its local value of the keys.

With write-around, once a session commits, it releases all its leases by sending a message to the IQ-Twemcached server and KOSAR Core. The IQ-Twemcached server simply deletes those keys with a Q lease granted to this session. With KOSAR, the Core sends a message to all KOSAR Clients with a replica of the deleted keys to delete their key-value replicas.

In sum, with write-around, I leases are not compatible with one another causing one to back-off and try again, a Q lease voids an I lease to prevent possible insertion of stale data, and multiple Q leases are compatible. With write-through, a key difference is that Q leases are no longer compatible with one another, causing a conflicting session to abort and re-start. See [23] for additional details.

4 Cooperative Cache Management

With SAS, a cooperating cache management technique has a significant impact on system performance and its scalability. This is specially true with write-around even when there is enough memory to materialize all key-value pairs in the cache.

Two cache management techniques supported by KOSAR are Greedy and Ingest. Greedy is non-cooperative and requires each cache to manage its content using its local replacement policy independent of the other caches. This means a cache that observes a miss for a key-value pair must compute this key-value pair using the RDBMS even though as many as N-1 replicas may exist in the other caches. With the RDBMS as the slowest component, once an update impacts a popular key-value pair and invalidates it, this key-value pair is potentially re-computed N times. Once by each cache that observes a miss for it. This causes the cache servers to wait for the RDBMS, limiting the scalability of the system.

Ingest, on the other hands, is a cooperative cache management technique. It ensures a key-value pair is computed by at most one cache. Other caches that observe a miss for this key are directed to fetch it from the cache that has a replica of it. In essence, each cache may serve as a producer of a key-value pair missed by another cache as long as the producer has a copy of the referenced key.

KOSAR implements Ingest as follows. Its Core maintains a list of caches (KOSAR Clients of Figure 1.b) with a copy of a key-value pair. When a KOSAR Client requests an I lease (due to a cache miss) on a key, the Core provides it with the list of KOSAR Clients that have a replica of this key. The KOSAR Client selects one of these randomly, fetches a replica of its required key-value pair, and releases its I lease. The Core decides whether the client may maintain this copy or not based on the allowed number of replicas[8].

Figures 2a-c show SoAR of Greedy and Ingest with write-around and write-through techniques. The x-axis shows the number of clients. The scale of the y-axis changes in these figures, highlighting a significant difference in SoAR for different workloads. Greedy and Ingest provide a comparable performance with write-through because each experiment has a warmup phase and write-through updates key-value pairs instead of deleting them. With write-around, Ingest is superior to Greedy. With a 10:1 read to write ratio, Greedy provides a lower SoAR as we increase the number of BG clients beyond 1 to 2 and higher, see Figure 2.a. This results in a system with a poor scalability characteristics, see Figure 2.d. In particular, the observed SoAR with 8 nodes is less than half that observed with 1 node. With 1 node, the CPU of the application server is 100% utilized. With 8 nodes, the CPU of the RDBMS server is 100% utilized. Every time a write action invalidates a popular key-value pair, Greedy requires each cache to compute it independently. This imposes a higher load onto the RDBMS as a function of additional caches. Processing of requests using RDBMS is slower than looking up results in the cache, causing the SoAR with 8 nodes to be lower than that with one node.

With a 1000:1 workload, Greedy supports a higher SoAR from 1 to 4 clients. Its SoAR drops sharply from 4 to 8 clients. The explanation for this is similar to the above with the system switching from the application server being 100% utilized with 1 node to the RDBMS becoming 100% utilized with 8 nodes. A key observation is that a lower frequency of writes

[8]Release and notify are performed with 1 message.

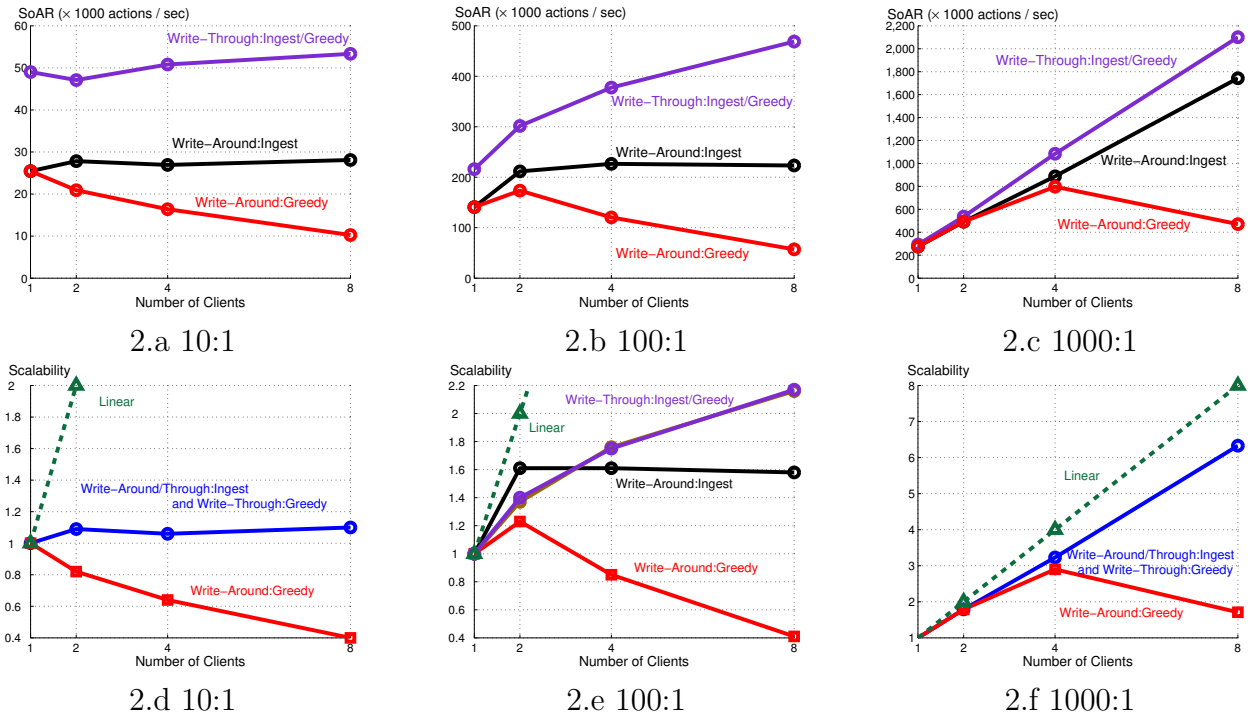


Figure 2: Performance and scalability of SAS architecture (KOSAR), 100K member social graph, 10 friends and 0 resources per member. When performance of several techniques is so close that the curves are hard to distinguish, we represent them as one curve to make the figures legible. The scalability with different number of clients is computed relative to the SoAR observed with 1 client, scalability with x clients = $\frac{SoAR(x)}{SoAR(1)}$. With one client, scalability is 1.

enhances the scalability of Greedy.

By computing a key-value pair only once independent of the number of cache servers, Ingest enables write-around to provide similar performance to write-through. However, the scalability of Ingest is limited with read to write ratios of 10:1 and 100:1 as the RDBMS must process transactions that use SQL DML commands to implement the write actions. The RDBMS becomes the bottleneck with all configurations to limit system scalability. For example, with 10:1, both the RDBMS disk and CPU are heavily used. Disk shows a sustained queue of 1.25 elements (a 100% sustained utilization) and the CPU is more than 80% utilized. The cache servers are idle most of the time waiting for the RDBMS to finish processing the write actions. Hence, increasing the number of cache servers does not enhance system SoAR, see Figure 2.d.

5 A Comparison of CS with SAS

This section compares IQ-Twemcached (CS architecture) with KOSAR configured using Ingest (SAS architecture). We report on both system SoAR and its scalability characteristics.

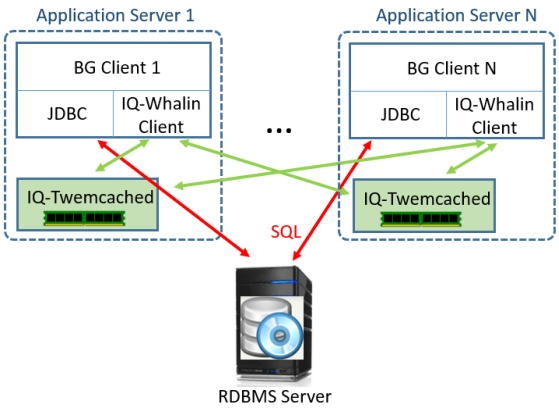


Figure 3: Physical organization of CS architecture.

As shown in Figure 3, with the CS architecture, there is a pairing of BGClients with the IQ-Twemcached. Hence, with a single BGClient, there is no physical network transmission because the IQ-Twemcached and the BGClient issuing requests are on the same server. However, with 2 and more nodes, the BGClients hash partitions the key-value pairs across the IQ-Twemcached instances. As detailed in Section 5.2, the network overhead incurred with two or more nodes impacts the scalability of the CS architecture when compared with a single node. This is particularly true with 100:1 and 1000:1 workloads.

We start with a single node comparison of the two architectures. Subsequently, we analyze the scalability of each architecture.

5.1 Single-Node Comparison

Read to write ratio	No Cache SQL-X	Write-Around		Write-Through	
		CS IQ-Twemcached	SAS KOSAR	CS IQ-Twemcached	SAS KOSAR
10:1	12,227 (RDBMS CPU)	20,300 (RDBMS CPU/Disk)	25,436 (RDBMS Disk)	29,898 (App Server CPU)	49,004 (RDBMS Disk)
100:1	18,516 (RDBMS CPU)	26,916 (App Server CPU)	140,857 (App Server CPU)	41,509 (App Server CPU)	215,850 (App Server CPU)
1000:1	21,969 (RDBMS CPU)	28,430 (App Server CPU)	275,317 (App Server CPU)	43,348 (App Server CPU)	292,899 (App Server CPU)

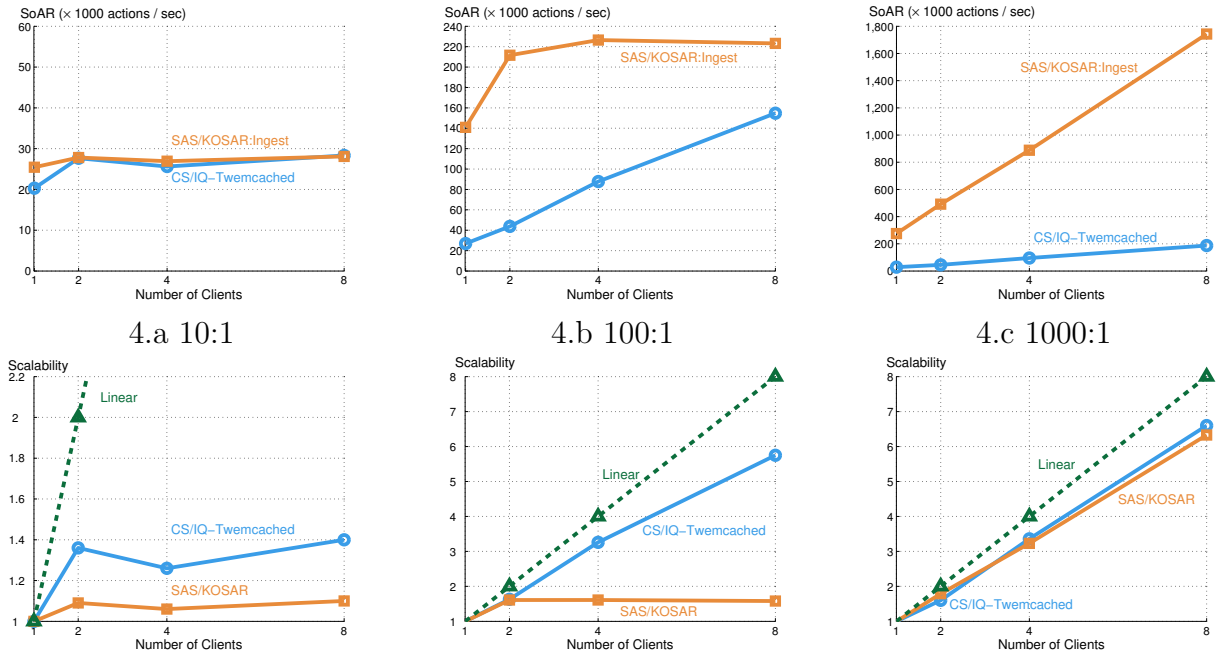
Table 7: SoAR with a single cache node as a function of read to write ratio of BG actions. The resource that becomes the bottleneck is identified in parentheses below the SoAR rating.

Table 7 shows SAS is superior to CS by providing a higher SoAR with different workloads. Both architectures enhance the performance of a RDBMS by itself, compare column 2 with the other columns. A higher read to write ratio enhances the performance of both caching solutions. Moreover, write-through outperforms write-around by re-filling the impacted key-value pairs instead of deleting them. The SoAR of SAS/KOSAR is 5 to 9 folds higher than

CS/IQ-Twemcached with 100:1 and 1000:1 read to write ratios. This difference drops to 25%-60% with a 10:1 read to write ratio because the RDBMS disk becomes 100% utilized with SAS and limits its performance.

Table 7 shows the SAS architecture is generally more efficient than the CS architecture. In particular, it does not incur the repeated overhead of fetching a key-value pair using the network stack of the operating system and deserializing it to obtain the required value. Hence, SAS outperforms CS by a wide margin even though the CPU of the application server is the bottleneck resource with both architectures.

5.2 Scalability Comparison



4.a 10:1 4.b 100:1 4.c 1000:1

4.d 10:1 4.e 100:1 4.f 1000:1

Figure 4: A comparison of CS/IQ-Twemcached and SAS/KOSAR with write-around.

Figures 4 and 5 show the SoAR and scalability of the different architectures with write-around and write-through, respectively. SAS is more scalable than CS only with the 1000:1 workload. While CS is more scalable than SAS with the 10:1 and 100:1 workloads, SAS provides either the same or a significantly higher performance (SoAR) than CS with all workloads. With the 10:1 workload, both CS and SAS architectures fails to scale. This is true with both the write-around and write-through policies, see Figures 4.d and 5.d. The frequent writes cause the RDBMS to become the bottleneck with one cache manager instance and remain the bottleneck with additional instances. The RDBMS is busy processing the SQL DML commands (insert, delete, update) issued by the write actions. These transactions result in a sustained disk queue at the RDBMS server that limits scalability and dictates overall performance. Switching to SSD increases throughput. However, it does not change

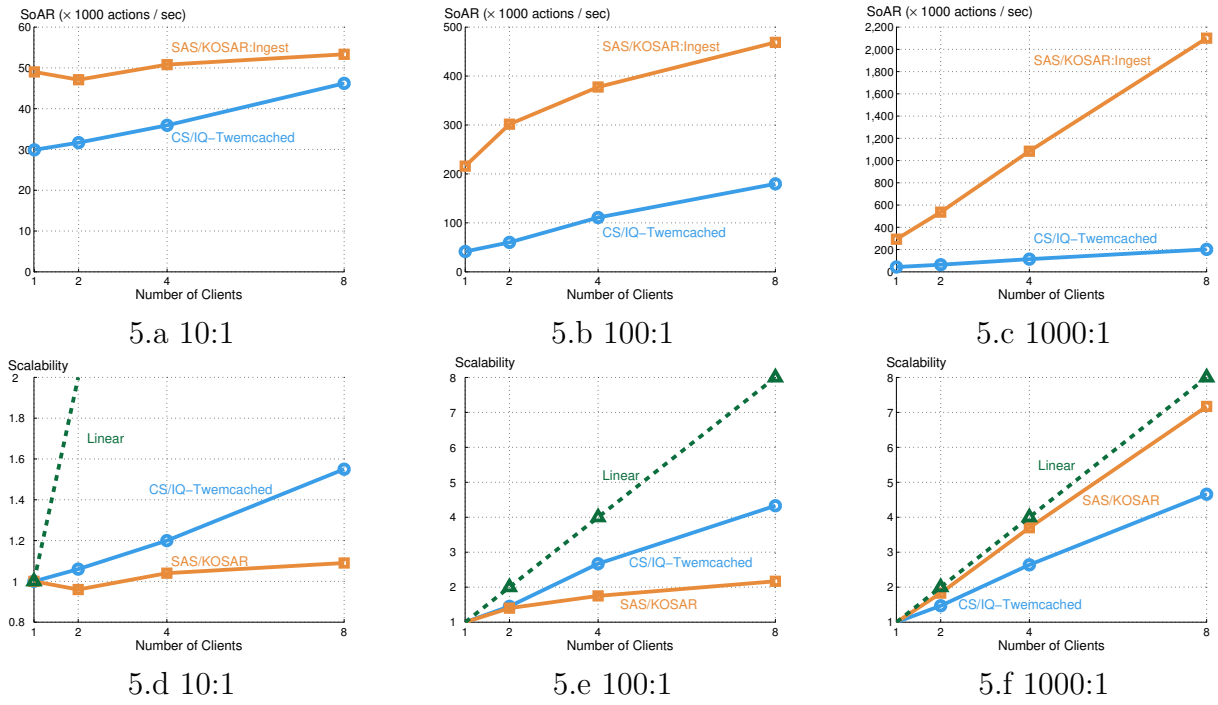


Figure 5: A comparison of CS/IQ-Twemcached and SAS/KOSAR with write-through.

the scalability results because the throughput with one cache server would be higher (due to use of SSD). Below, we compare the scalability of different architectures with write-around and write-through in turn.

With write-around and a 100:1 workload, the CPU of the RDBMS server becomes 100% utilizes and limits the scalability of the SAS architecture[9]. The RDBMS is busy servicing queries generated by cache misses. These cache misses are attributed to write actions that delete cached key-value pairs. An increase to 1000:1 read to write ratio reduces the imposed load on the RDBMS to enable SoAR to scale. However, the RDBMS continues to remain the bottleneck, preventing SAS from scaling linearly.

With the CS architecture, the 1 Gbps network bandwidth of the individual application servers has a high utilization with 100:1 and 1000:1 workloads. The CPU utilization of the application servers is also high ($> 80\%$). The BGClients generate approximately the same amount of load on the IQ-Twemcached servers. This load is not evenly distributed across the IQ-Twemcached instances due to partitioning of the key-value pairs. This imbalance explains the sublinear scalability with the CS architecture. (The SAS architecture does not observe the same imbalance due to replication of key-value pairs.)

Write-through reduces the dependence of an architecture on the RDBMS by requiring the write actions to compute new key-value pairs. This reduces the number of queries issued to the RDBMS, enhancing the observed SoAR. With a 1000:1 read to write ratio, the SAS architecture scales almost linearly because the CPU of the different application servers is almost 100% utilized. With the CS architecture the network card of each node

[9]With 8 nodes, the average application server utilization is lower than 30%.

remains fully utilized to dictate both the SoAR of the system and its scalability. Similar to the discussion of write-around, CS does not scale due to load imbalance across the IQ-Twemcached instances[10].

6 Conclusions and Future Research

Both SAS and CS architectures enhance the performance of a RDBMS dramatically for workloads with a high read to write ratio. The SAS architecture is higher performant than the CS architecture. SAS fails to scale linearly due to the RDBMS becoming the bottleneck. One approach to resolve this limitation is to deploy multiple RDBMS instances and shard a database across them [5]. Challenges of this design include additional design considerations to perform cross-shard filter or join, processing of transactions that update data in different fragments, and others as detailed in [41]. An alternative is to use a write-back policy that applies writes to the RDBMS asynchronously. An implementation of this policy may buffer writes in the caching layer. A challenge is how to process key-value references that observe a cache miss and issue a query to the RDBMS with pending buffered writes. The cache manager requires novel algorithms to apply the relevant buffered writes to the cache prior to processing the RDBMS query to compute the missing cache entry.

The evaluation presented in this study can be extended in several ways. First, this study considered only one sample system for SAS and one sample system for CS. A future effort is to expand this evaluation to include other sample systems such as Apache Ignite for SAS and Redis for CS. A key question is whether strong consistency is a requirement for an apple-to-apple comparison. Use of leases introduces delays that slows down performance. At the same time, strong consistency may not be required for applications such as social networking. We intend to investigate this tradeoff in greater detail.

Second, this study did not consider a hybrid of the CS and SAS architectures. Such an architecture may deploy the cache of CS with the coordinator of SAS, see KOSAR Core of Figure 1.a. This may enhance availability of data when application servers fail frequently. In general, analyzing failure of caches and its impact on the performance of alternative architectures is an open research topic.

Finally, we are aware of no study that has evaluated a write-heavy workload such as TPC-C with alternative write policies including write-back [20]. With these policies, a SAS architecture may provide a superior performance while preserving the ACID property of transactions.

References

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-Sharding for Datacenter Applications. In *OSDI*, pages 739–753, 2016.

[2] Y. Alabdulkarim, M. Almaymoni, Z. Cao, S. Ghandeharizadeh, H. Nguyen, and L. Song. A Comparison of Flashcache with IQ-Twemcached. In *IEEE CloudDM*, 2016.

[10]Similar results is reported by other systems that partition data [18, 24].

[3] Y. Alabdulkarim, S. Barahmand, and S. Ghandeharizadeh. BG: A Scalable Benchmark for Interactive Social Networking Actions. *Future Generation Computer Systems*, 85:pp. 29–38, August 2018.

[4] Apache. Ignite - In-Memory Data Fabric, https://ignite.apache.org/, 2016.

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

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

[7] 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 *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA*, 2003.

[8] N. Bronson, T. Lento, and J. L. Wiener. Open Data Challenges at Facebook. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1516–1519, 2015.

[9] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side Flash Caching for the Data Center. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

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

[11] T. P. P. Council. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp.

[12] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB*, 6(14):1942–1953, 2013.

[13] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM, Vol. 11, No. 5*, pages 323–333, 1968.

[14] D. J. DeWitt, P. Futtersack, D. Maier, and F. Vélez. A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, 1990.

[15] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.

[16] M. Faust, M. Boissier, M. Keller, D. Schwalb, H. Bischoff, K. Eisenreich, F. Färber, and H. Plattner. Footprint Reduction and Uniqueness Enforcement with Hash Indices in SAP HANA. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II.*

[17] M. J. Franklin. *Client Data Caching: A Foundation for High Performance.* Kluwer Academic Publishers, AH Dordrecht, The Netherlands, 1996.

[18] S. Ghandeharizadeh and D. J. DeWitt. A Multiuser Performance Analysis of Alternative Declustering Strategies. In *Proceedings of the Sixth International Conference on Data Engineering, Los Angeles, California, USA*, pages 466–475, 1990.

[19] S. Ghandeharizadeh and et. al. A Demonstration of KOSAR: An Elastic, Scalable, Highly Available SQL Middleware. In *ACM Middleware*, 2014.

[20] S. Ghandeharizadeh and H. Ngueyn. Design, Implementation, and Evaluation of Write-Back Policy with Cache Augmented Data Stores. Technical Report 2018-06, USC Database Laboratory, 2018.

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

[22] S. Ghandeharizadeh, J. Yap, and H. Nguyen. IQ-Twemcached. http://dblab.usc.edu/users/IQ/.

[23] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. *Middleware*, December 2014.

[24] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *SOSP '03: Proceedings of nineteenth ACM SIGOPS symposium on Operating systems principles*. ACM Press, 2003.

[25] Google. Guava: Core Libraries for Java, https://github.com/google/guava, 2015.

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

[27] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash Caching on the Storage Client. In *USENIXATC*, 2013.

[28] Java Community Process. JCACHE - Java Temporary Caching API, https://jcp.org/en/jsr/detail?id=107, 2014.

[29] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, pages 439–450, 1994.

[30] H. Kim, I. Koltsidas, N. Ioannou, S. Seshadri, P. Muench, C. Dickey, and L. Chiu. Flash-Conscious Cache Population for Enterprise Database Workloads. In *Fifth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2014.

[31] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write Policies for Host-side Flash Caches. In *FAST 13*, 2013.

[32] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[33] P. Larson, J. J. Goldstein, H. Guo, and J. Zhou. MTCache: Transparent Mid-tier Database Caching in SQL Server. In *ICDE*, 2004.

[34] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*. USENIX, 2003.

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

[36] 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 *NSDI*, pages 385–398, Berkeley, CA, 2013. USENIX.

[37] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *ACM SIGMOD*, 1993.

[38] 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.

[39] RedisLabs. Redis, https://redis.io/.

[40] M. Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 24(7):412–418, 1981.

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

[42] Terracotta. BigMemory, http://terracotta.org/products/bigmemory.

[43] K. Voruganti, M. T. Özsu, and R. C. Unrau. An Adaptive Data-Shipping Architecture for Client Caching Data Management Systems. *Distrib. Parallel Databases*, 15(2), March 2004.

[44] 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.

[45] S. G. Yazeed Alabdulkarim, Marwan Almaymoni. Polygraph: A Plug-n-Play Framework to Quantify Anomalies. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.