A Comparison of Flashcache with IQ-Twemcached*

Yazeed Alabdulkarim, Marwan Almaymoni, Ziwen Cao
Shahram Ghandeharizadeh, Hieu Nguyen, Lingnan Song

Database Laboratory Technical Report 2016-01

Computer Science Department, USC

Los Angeles, California 90089-0781

February 22, 2016

Abstract

Person-to-person cloud service providers such as Facebook use Host-side (HsC) and Application-side (AsC) caches to enhance performance. Using Facebook's Flashcache as the representative of HsC and IQ-Twemcached as the representative of AsC, this study quantifies their tradeoffs using both a read-heavy and a write-heavy workload. Obtained results show Flashcache provides significant benefit for I/O intensive workloads, e.g., read-heavy workloads when the application working set does not fit in memory. IQ-Twemcached is most suitable for those workloads that fully utilize the CPU of the server hosting the database management system. For some workloads, the performance enhancement observed with the two caches deployed together is several folds higher than the sum of the performance benefit realized by each cache individually.

1 Introduction

Caches enable a large class of applications including cloud services. They stage data on a faster storage medium to expedite its processing. Two popular caches include Host-side and Application-side caches. They are either developed, further extended, or in-use by person-to-person cloud service providers. For example, Flashcache is a Host-side cache developed by Facebook [1]. memcached is a popular in-memory Application-side cache adapted for use by Facebook [33] and refined by Twitter with its Twemcache open source release [5].

Host-side caches (HsC) stage data blocks from persistent store, typically disk, onto NAND Flash [26, 11, 25, 23]. *Application-side* caches (AsC) typically use an in-memory (DRAM) key-value store to look up the result of functions and code segments instead of computing them repeatedly [35, 24, 21, 33]. In this study, we focus on HsC and AsC caches external to a database management system (DBMS) that require no software modifications to the DBMS. Hence, a DBMS driven application may deploy both at the same time.

There are two different architectures for AsC, client-server and shared-address-space [21]. Each may employ a different technique to maintain the DBMS and the cache consistent in the

*In the *ICDE Workshop on Cloud Data Management* (CloudDM), Helsinki, Finland, May 2016.

presence of updates. This study focuses on the client-server architecture using invalidation to maintain the cache and the DBMS consistent with one another. A comprehensive analysis of alternative AsC architectures and consistency techniques (such as refill) is a future research direction. We identify the Write-back policy of HsC in the context of AsC as an open research topic, see Section 6.

HsC and AsC have distinct characteristics that make them suitable for different workloads. It is important to understand their performance tradeoff in order to select one or both intelligently for a given workload. Our **primary contribution** is to quantify this tradeoff using IQ-Twemcached [2] as the representative of AsC and Facebook's Flashcache [1] as the representative of HsC. IQ-Twemcached is our extended version of Twitter's variant of memcached [5] with leases [22] to provide strong consistency. It is different than Flashcache in several ways. First, IQ-Twemcached uses DRAM while Flashcache uses NAND Flash. DRAM is several orders of magnitude faster than NAND Flash. Second, IQ-Twemcached requires additional software to be integrated with the application while Flashcache has no such requirement and is seamlessly integrated using the operating system. Third, while IQ-Twemcached requires networked communication between the application and the cache, Flashcache incurs no such overhead. Fourth, IQ-Twemcached and Flashcache are deployed across different software layers. Flashcache is an intermediary between a DBMS issuing read and write of blocks to devices managed by the operating system (OS). In contrast, IQ-Twemcached manages key-value pairs computed by an application using the query language of the DBMS, e.g., SQL.

We use two different benchmarks for the purposes of this evaluation, TPC-C [4] and BG [6]. TPC-C emulates on-line transaction processing (OLTP) applications and is write-heavy. It requires 92% of the transactions in the mix to include writes [15]. BG emulates the interactive actions issued by a person-to-person cloud service provider such as Facebook. It has the flexibility to generate read-heavy workloads. (Facebook reports a read to write ratio of 500:1 [10].)

Our evaluation provides insights into the benefits and overheads of each cache to empower a system designer to select each effectively. The key lessons of our study are as follows:

1. While the DRAM used by IQ-Twemcached is significantly faster than the NAND Flash used by Flashcache, IQ-Twemcached cannot provide more than 2-4x improvement. The DBMS server (either CPU or disk) becomes the bottleneck to limit IQ-Twemcached's (DRAM's) performance benefit.
2. NAND Flash provides storage capacities larger than DRAM. Twelve times larger in our experiments. With read-heavy workloads referencing databases and working sets larger than DRAM and smaller than Flash, Flashcache outperforms IQ-Twemcached by a wide margin [20].
3. IQ-Twemcached is most beneficial for those workloads that render their working set (or the entire database) main memory resident and utilize the CPU cores of the DBMS server fully. Query result look up (instead of query processing) reduces the imposed CPU load to enhance performance.
4. Flashcache is most beneficial for write-heavy workloads, such as OLTP. IQ-Twemcached by itself is not effective due to repeated cache invalidations.
5. For some workloads, the two caches together are much more beneficial than the sum of benefits provided by each cache alone. In some instances, three times more beneficial.

The rest of this paper is organized as follows. In Section 2, we survey the related research. Sections 3 and 4 provide an overview of AsC and HsC , respectively. Section 5 provides an evaluation of these two caches. Our future research directions are detailed in Section 6.

2 Related Work

In a data center, caches are applied at different software layers that implement components of a data intensive DBMS application: the operating system device mapper [40] (HsC), in the execution engine of a DBMS (server-side, mid-tier and client-side caches), and in the application itself (AsC). AsC and HsC deployed outside the DBMS constitute our focus. HsC may be managed by the application [23, 34] or deployed seamlessly using a storage stack middleware or the operating system (termed the caching software) [36, 13, 37, 32, 38, 11, 25, 30, 26]. Both AsC and HsC complement the buffer pool of a DBMS which is a server-side[1] cache.

Briefly, both client-side [14, 16, 41] and mid-tier [28, 9] caches are transparent to an application. They distribute the processing of the algebraic operators that constitute a query across the client and server components of the DBMS. With a client-side cache, the server ships disk pages for caching and processing to the client. The book by Franklin [16] provides a survey of these caches. Mid-tier caches offload part of a workload to intermediate database servers that partially replicate data from the backend sever [28, 9]. These may cache entire tables, materialized views, or query fragments, providing distributed query execution.

We are aware of caches inside the DBMS that resemble HsC and AsC. For example, Cassandra has a cache that stages rows on NAND Flash and Alibaba has a patch for MySQL Innodb that does the same [3]. These are similar to HsC. In the spirit of AsC, MySQL has a query result cache that can be either turned on or off. In contrast, our assumed caches are external with no software modification to the DBMS.

To the best of our knowledge, this is the first study to quantify the tradeoffs associated with HsC and AsC by comparing a representative of each with one another individually and together. Our evaluation quantifies their tradeoffs for both read-heavy and write-heavy workloads. A comparison and a combination of these two caches with a client-side/mid-tier cache is a future research direction, see Section 6.

3 Application-Side Cache: IQ-Twemcached

With the AsC, a software developer defines a key that identifies a function or a segment of software with a specific input. This function (or code segment) is extended to construct a key using its input and look up its value in the cache. If the value is found then it is returned without executing the function. Otherwise, the function is executed to obtain the value. This key-value pair is inserted in the cache for future reference. In the presence of SQL DML commands that impact one or more keys, the developer provides additional software to invalidate (delete) these keys. A subsequent reference for one of these keys results in a

[1] Also known as query-side or backend cache.

miss. This causes the system to execute the function to compute a value using its input, and insert the resulting key-value pair in the cache.

We use the IQ-Twemcached [2] as the representative of AsC. It is an extended version of Twemcache [5] that provides strong consistency using leases [22]. IQ-Twemcached supports both the I/Q and C/O leases. I/Q leases are designed to prevent undesirable read-write race conditions that insert stale data in the cache. Details of the I/Q framework are provided in [22]. The C/O leases have not been described to date and are detailed below.

An application uses the concept of sessions to implement strong consistency. A *session* consists of a sequence of SQL statements that constitute a transaction, a collection of key look ups, along with key-value invalidations from the cache due to updates to the DBMS. A session has a unique identifier with an explicit start and commit point. Sessions in combination with C/O leases implement serial schedule similar to the optimistic concurrency control [27] technique. This is realized as follows. IQ-Twemcached grants a C lease to a session that

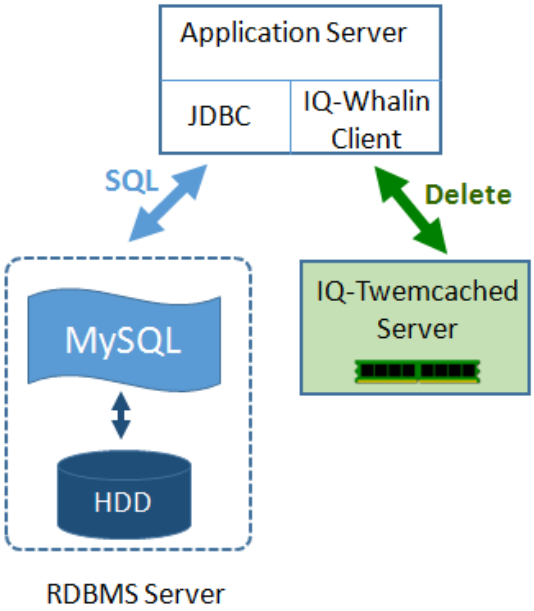

Figure 1: Application-side cache (AsC) using invalidation.

reads a key. Multiple C leases are allowed to co-exist on the same key-value pair to enable concurrent reads. IQ-Twemcached grants an O lease on a key that the application invalidates. Multiple O leases are allowed to co-exist on the same key. An O lease on a key voids all C leases on that key. A session that requests a C lease on a key with an O lease granted to another session is aborted.

During its life time, a session may acquire a handful of C and O leases on different keys. Prior to committing, a session requests the IQ-Twemcached server to *validate* the session. During this phase, the server checks to see if a C lease obtained by the session was voided. If so, the session aborts by rolling back its DBMS transactions. Otherwise, the session commits causing its DBMS transaction to commit.

With an aborted session, the IQ-Twemcached client may restart the session prior to

reporting a failed session. It is possible for a session to pass its validation phase and have the DBMS abort its transaction. This is acceptable because multiple key invalidation (deletions) are side-effect free. They may only degrade system performance without compromising consistency of the database with the cache.

IQ-Twemcached server marks a session as `ABORTED` when one of its C leases on a key is voided by an O lease granted to another session. It releases all leases of this aborted session. When this session issues a request (*ciget*, *oqreg* or *validate*) to the cache server, the server notifies it to abort without further actions.

A session consisting of a single DBMS transaction does not require the use of C/O leases. I/Q leases are sufficient to provide strong consistency for this session. A session with a multi-statement DBMS transaction that performs cache look ups in-lieu of the execution of one or more statements must use C/O leases. In essence, the C/O leases require the use of I/Q leases while the reverse is not true. A C lease acquires an I lease when it observes a cache miss. An O lease acquires a Q lease always.

Details of the IQ-Twemcached implementation is as follows. A session has a unique identifier (*Session ID* or *SID*) generated by the client prior to the start of the DBMS transaction. The client creates a SID using the MAC address of its networking card in combination with the system clock. The C lease for a key is issued by the *ciget* command. Similarly, the O lease for a key is issued by the *oqreg* command. The IQ-Twemcached server recognizes the start of a session when the (modified) Whalin client issues the first *ciget* or *oqreg* request with a SID that the server has not seen before. The server uses the SID to maintain meta-data on behalf of a session for the validate phase. Prior to the client committing its DBMS transaction, the session issues a *validate* request to the server. The validation succeeds if all C leases granted to this session are valid at the server. Next, the session transitions into pre-commit phase and proceeds to commit its DBMS transaction. If the DBMS transaction succeeds to commit, the session issues a commit request. Otherwise, it issues an abort request. In both cases, this causes the IQ-Twemcached to remove all session meta data and release all leases hold. With commit, the IQ-Twemcached deletes all keys with an O lease granted to this session.

Similar to I/Q leases, there is also expiration time for C/O leases. A session is aborted if one of its C leases expires. Expiration time allows other sessions to proceed in case the machine executing a session either fails or becomes unreachable unexpectedly.

4 Host-Side Cache: Flashcache

We used Facebook's Flashcache [1] as our HsC. Flashcache is a Linux kernel device mapper that implements a disk block cache. It uses NAND Flash to cache disk blocks, enhancing the performance of the Linux kernel to service disk I/Os.

Figure 2 shows the three cache write policies supported by Flashcache [1]. These policies have a profound impact on system performance based on how they process a disk block write. They are as follows:

1. **Write-around:** Writes the block to the disk only. Once the DBMS reads this disk block, then it is cached.
2. **Write-through:** Writes the block to both the disk and Flash device synchronously.

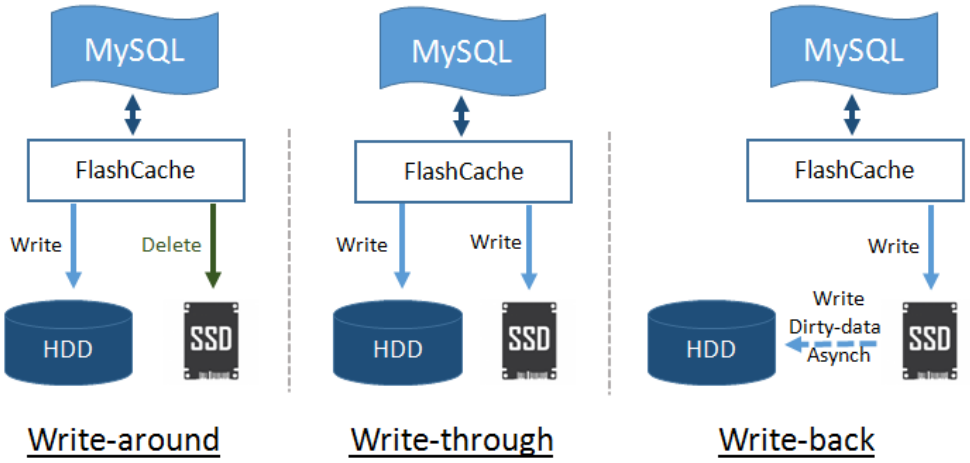

Figure 2: Alternative write policies of Host-side cache (HsC).

3. **Write-back:** Writes the block to the Flash device synchronously. A background thread writes dirty blocks to the disk asynchronously. The cache eviction mechanism of Flashcache may also flush dirty blocks to the disk.

5 A Comparison

We used MySQL (v5.6.27) with the InnoDB storage engine as our relational database management system (RDBMS). We configured MySQL with a 6 GB buffer pool, 4 GB InnoDB log file size, no query caching, 16 InnoDB buffer pool instances, 16 InnoDB threads for concurrency, and 64 MB InnoDB log buffer size.

We evaluated four alternative system configurations: MySQL, MySQL with IQ-Twemcached using the invalidation consistency technique (see Figure 1), MySQL with Flashcache using both the Write-around and Write-through policies (see Figure 2), and MySQL with both IQ-Twemcached and Flashcache (see Figure 3). When comparing AsC with HsC, HsC using the Write-around policy is comparable with AsC as both use invalidation. While we report on the Write-through policy, we do not report on the Write-back policy because there is no comparable AsC configuration. See Section 6 for how we intend to extend this evaluation to include such a comparison.

Using the Linux flexible[2] I/O tester, fio [29] benchmark, the number of sequential reads using 4 KB block sizes (I/O operations per second, IOPS) with Flash is more than 16x higher than disk. Random I/Os widen this gap to more than 250x folds due to disk seeks. With Flash, the IOPS for reads (98,100) is only 10% higher than writes (88,880).

We configured IQ-Twemcached with 10 GB of memory and 8 threads. Flashcache uses a 90 GB partition of a 250 GB SSD. We construct the database after creating the cache. This renders the database cache resident with the Write-through policy of Flashcache. However,

[2]Configured with a queue length of 16, 4 KB block size, and the appropriate database size, e.g., `fio -size=2100mb -direct=1 -name=1 -filename=/dev/sdb1 -rw=randread -ioengine=libaio -iodepth=16 -bs=4k` with the 20 warehouse TPC-C database of Section 5.2.

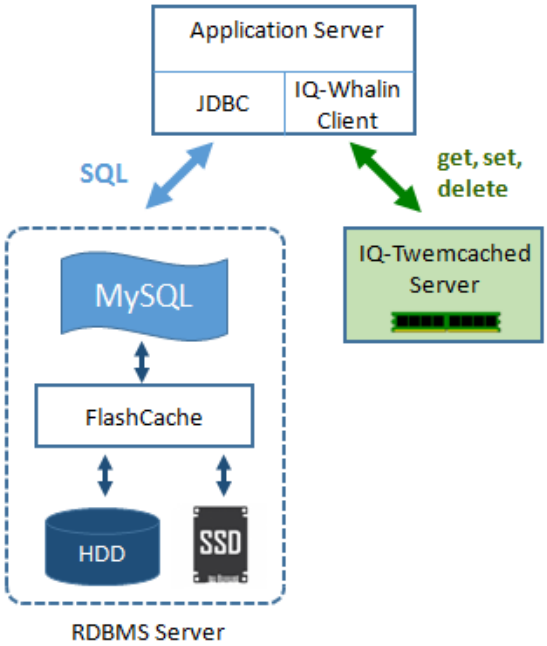

Figure 3: Application-side cache (AsC) and Host-side cache (HsC) deployed together.

with the Write-around policy (and IQ-Twemcached), the cache is cold after a reload of the database. Hence, an experiment includes a warm-up phase to populate the cache. We report on this with both BG and TPC-C, see the next two sections and discussion of Figure 6.

All machines are 4 (8 hyper-threaded) Core Intel i7-3770 3.40 GHz CPU with 16 GB of memory and multiple 1 Gbps networking cards. The server hosting MySQL is configured with two cards. The one hosting IQ-Twemcached is configured with three cards. This prevents an experiment utilizing the network cards fully.

5.1 BG

BG Social Actions	99.9% Read	99% Read	90% Read
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 mixes of interactive social networking actions.

BG [6, 8, 18, 17, 7] is a benchmark for interactive social-networking actions, such as those processed by Facebook and Twitter (www.bgbenchmark.org). Example actions in-

clude View Profile, Invite Friend and Accept Friend. With List Friends and View Friends Requests actions, we retrieve at most 10 profiles. This resembles the behavior of typical social networking sites. An experimentalist specifies the mix of actions to evaluate a data store. Table 1 shows three workloads considered in this paper: 99.9% read, 90% read, and 90% read. These translate into read to write ratio of 1000:1, 100:1, and 10:1, respectively. The 500:1 read to write ratio of Facebook [10] falls in-between 1000:1 and 100:1. As reported below, the observations with these two workloads is almost identical.

BG summarizes the performance of a data store in one metric, Social Action Rating (SoAR). SoAR is defined as the highest throughput provided by a data store while satisfying a pre-specified service level agreement, such as 95% of actions to be performed faster than 100 msec for 10 minutes [6]. This SLA is used in all experiments. The distribution of accesses to the data is skewed using a Zipfian distribution with exponent 0.27. This means approximately 63% of requests are referencing 20% of the database.

We used a 1 Million and a 10 Million member social graph with 10 friends per member for our experiments. The resulting databases are 2.7 GB and 32 GB, respectively. The one Million member database is small enough to fit in the MySQL buffer pool. However, the 10 Million member database is twice the size of the physical memory of the server hosting MySQL.

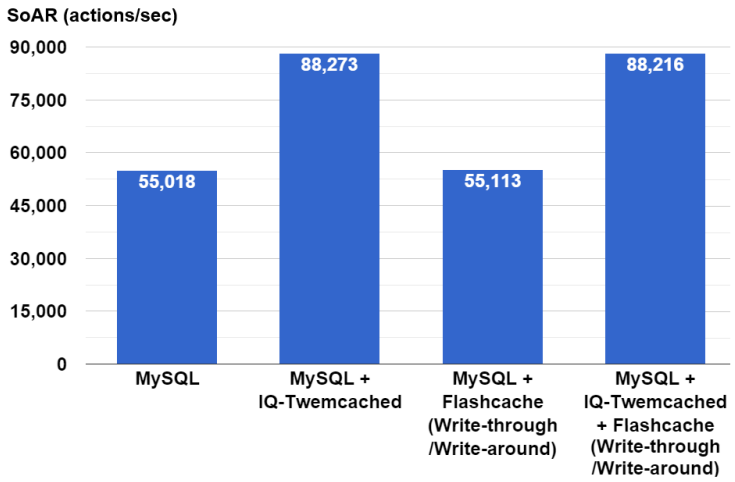

Figure 4: SoAR of the 99.9% workload with alternative system configurations, 1 Million members, and 10 friends per member.

Figure 4 shows the observed SoAR with a 99.9% read workload and the 1 Million member social graph that is small enough to fit in the memory of MySQL (almost identical results are observed with the 99% workload.) These results show Flashcache using both the Write-around and Write-through policies provides marginal performance enhancements when compared with MySQL by itself (compare 1st and 3rd bar). MySQL by itself (1st bar) utilizes the CPU cores of its server fully (100%) when BG establishes its SoAR. Flashcache provides no performance benefit since the database is in memory. This also explains why we see no performance difference between the Write-around and Write-through policies.

IQ-Twemcached enhances the SoAR of MySQL (compare 1st and 2nd bars) by reducing

the load imposed on the MySQL server CPU. It looks up results using the IQ-Twemcached server instead of processing them by issuing queries to MySQL.

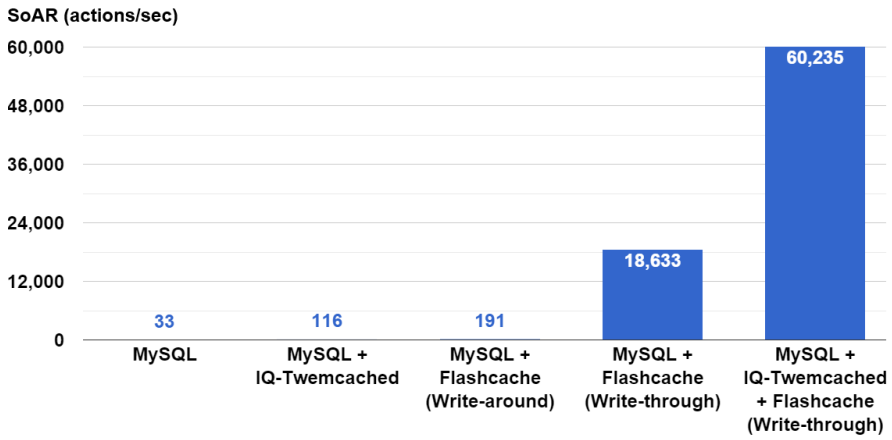

Figure 5: SoAR of the 99.9% workload with alternative system configurations, 10 Million members, and 10 friends per member.

With the 10 Million member social graph, the database no longer fits in the buffer pool of MySQL. This causes processing of queries to incur expensive disk I/Os that reduce the SoAR of MySQL to 33, see the first bar of Figure 5. It is interesting that IQ-Twemcached (2nd bar of Figure 5) enhances SoAR by tens of actions per second while Flashcache using Write-through (4th bar) enhances SoAR by more than ten thousand actions per second. There are several reasons for this. First, Flashcache renders the database resident in its 90 GB partition while IQ-Twemcached cannot fit all the key-value pairs in its 10 GB memory. Second, with IQ-Twemcached, the 0.1% write actions invalidate key-value pairs. These key-value pairs must be re-computed once referenced and re-computing them using MySQL is expensive.

Flashcache configured with the Write-through policy in combination with IQ-Twemcached results in a system that is dramatically superior to either cache by itself. The disk of the MySQL server is the bottleneck when IQ-Twemcached is deployed by itself. Flashcache (with Write-through) causes the CPU of the MySQL server to bottleneck. This enables IQ-Twemcached to boost performance dramatically by reducing the load imposed on the CPU of the MySQL server[3]. Flashcache further complements IQ-Twemcached by processing its issued queries due to cache misses faster.

In Figure 5, Flashcache using the Write-around policy is almost 100x lower than the Write-through policy (compare 3rd and 4th bars) because its warmup period does not render the database Flash resident. The Write-though policy has the entire database Flash resident

[3]With the 10 million member social graph, the CPU utilization of MySQL server drops from almost 100% with Flashcache (Write-through policy) to 45% with both Flashcache and IQ-Twemcached. Flashcache by itself causes the MySQL server to maintain a sustained queues of 80 I/O requests (100% Flash bandwidth utilization) and a low disk bandwidth utilization. Once extended with IQ-Twemcached, the Flash remains fully utilized with a queue length that varies from 8 to 20 with an average of 10 pending requests. The disk is now fully utilized with more than 1 (and typically fewer than 2) queued requests.

because each experiment copies the database at its start, see Section 5. Figure 6 shows

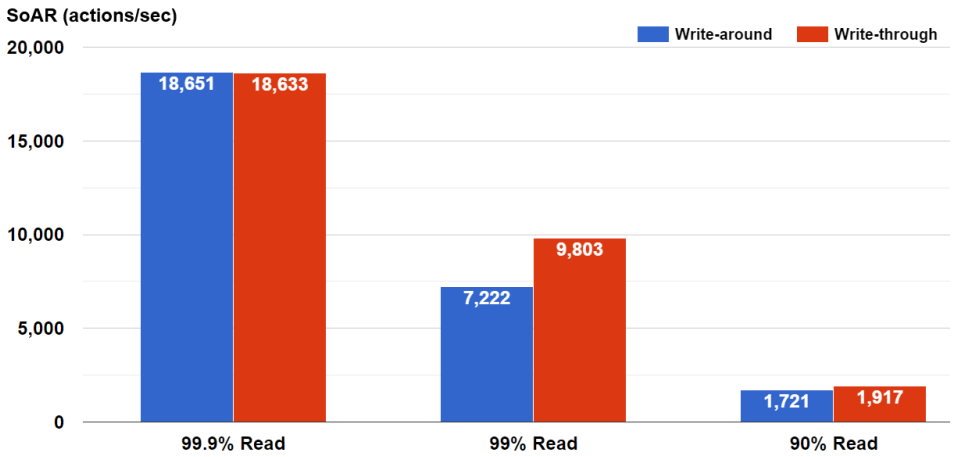

Figure 6: SoAR of three different workloads using MySQL configured with Flashcache, 10 Million members, and 10 friends per member.

system SoAR with a warmup that issues all possible queries to MySQL to enable the Write-around policy to render the database Flash resident. Now, the Write-around policy provides a compareable performance with the Write-through policy with the 99.9% read workload because writes are rare. With workloads that exhibit a lower read to write ratios (99% and 90%), use of invalidation causes the Write-around to provide a SoAR that is lower than the Write-through. However, this difference is no longer as dramatic: less than 50% instead of 100x.

5.2 TPC-C

Transaction	Mix	SLA
New-Order	45%	5 sec
Payment	43%	5 sec
Order-Status	4%	5 sec
Delivery	4%	5 sec
Stock-Level	4%	20 sec

Table 2: TPC-C workload.

TPC-C is a benchmark that simulates the activity of a wholesale supplier [4]. The benchmark consists of five transactions: two read-only transactions (Order-Status and Stock-Level) and three update-intensive transactions (New-Order, Payment, and Delivery). TPC-C specifies a service level agreement that requires 90% of each transaction to observe a response time faster than that shown in Table 2. This table also shows the mix of transactions recommended by TPC-C.

The performance metric for the TPC-C benchmark is transactions-per-minute-C (tpmC), the number of New-Order transactions processed per minute. We quantify the metric using

20 and 50 warehouses. The resulting databases are 2.1 GB and 5.1 GB, respectively. The tpmC trends observed with the two databases are almost identical. Though, the performance benefit observed with Flashcache is higher with 50 warehouses. Hence, below, we focus on results observed using the 20 warehouses database only.

We used the implementation of TPC-C by OLTPBenchmark [15] and extended it with C/O leases of Section 3 to provide strong consistency with IQ-Twemcached.

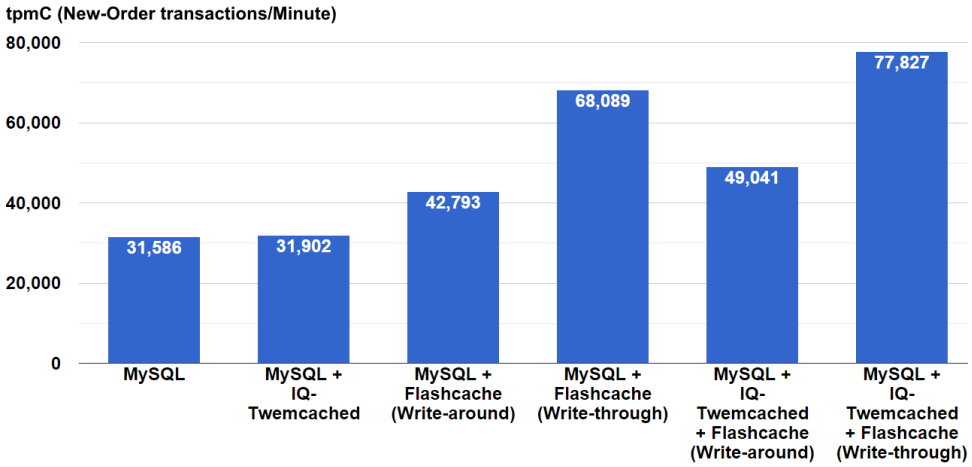

Figure 7: tpmC for different system configurations with 20 warehouses.

Figure 7 shows MySQL with IQ-Twemcached (2nd bar) provides marginal performance enhancements when compared with MySQL (1st bar). Frequent writes invalidate the cached key-value pairs that minimizes its benefits. Once this configuration is extended with Flashcache (6th bar), the resulting configuration provides a 2 fold enhancement relative to MySQL. Flashcache expedites processing of queries issued by the application due to misses reported by IQ-Twemcached. In these experiments, the bottleneck resource changes from the DBMS disk (IQ-Twemcached by itself) to Flash when both caches are deployed together[4].

Both (a) MySQL with IQ-Twemcached (2nd bar) and (b) MySQL with Flashcache using the Write-around policy (3rd bar) use invalidation to maintain their respective caches consistent. However, Flashcache provides more than 30% performance enhancement while IQ-Twemcaches provides no enhancement because the overhead of invalidation and subsequent replenishing of the cache is higher with IQ-Twemcached. This is attributed to the following factors: 1) query processing to compute a key-value pair, 2) serializing the key-value pairs to insert in the IQ-Twemcached, and 3) network round-trip times to invalidate and insert a key-value pair in the IQ-Twemcached. Flashcache incurs none of these.

Flashcache with the Write-around policy is inferior to the Write-through policy, compare the 3rd and 4th bars of Figure 7. References for cached pages invalidated by the Write-around policy must be fetched from the disk while the Write-through policy avoids this overhead. Moreover, unlike the discussions of Figure 6 with BG, increasing the duration of warm-up

[4]With IQ-Twemcached, the CPU of the MySQL server is approximately 45% utilized with a disk queue of one or more requests. Once extended with Flashcache, this disk queue is replaced with a sustained Flash queue of five or more requests. Moreover, the MySQL server CPU utilization is increased to 80% (68%) with Write-through (Write-around).

does not change the observed results because TPC-C is write-intensive and requires 92% of the transactions in the mix to include writes [15, 31].

6 Future Research

This paper quantifies the tradeoffs associated with a HsC named Flashcache and an AsC named IQ-Twemcached when deployed individually or in combination with one another. Flashcache using the Write-around policy is comparable with IQ-Twemcached because both use invalidation to implement cache consistency. It is fascinating that for some workloads, the two caches together are much more beneficial than the sum of benefits provided by each cache alone.

A future research direction is to evaluate the refill [22] consistency technique of IQ-Twemcached with the write-through policy of Flashcache. Refill updates key-value pairs impacted by an update when issuing SQL DML (insert/delete/update) commands to the DBMS. Note that the compatibility of C/O leases with refill is different than that with invalidation.

At the time of this writing, the write-back policy of HsC has no comparable AsC alternative. Hence, it was not included in our evaluation. Currently, we are investigating a comparable AsC technique that requires the cache server to queue SQL DML commands and process them asynchronously. A challenge is how to compute a key-value pair that observes a cache miss with a pending queue of SQL DML commands. One must design algorithms to determine if the pending DML commands impact the value of the key. For those that do, the system may prioritize them to be issued to the DBMS first.

Another interesting dimension is the architecture of AsC. IQ-Twemcached implements a client-server architecture that requires an application to communicate with the cache using message passing. An alternative is the Shared-Address-Space (SAS) where the cache is integrated into the application [12, 19, 39], eliminating the overhead of serialization, network transmission, and deserialization of values [21]. A comparison of SAS (e.g., KOSAR [19]) with HsC provides further insights into the tradeoffs associated with the alternative caching solutions.

Finally, it is worth comparing AsC and HsC to caching techniques inside the DBMS: mid-tier and client-side caches, see Section 2. A key research question is how these techniques compare to AsC as they all strive to minimize the load on the database server.

References

[1] Flashcache. https://github.com/facebook/flashcache.

[2] IQ-Twemcached. http://dblab.usc.edu/users/IQ/.

[3] M. Callaghan. Private Communications, Feb 6, 2016.

[4] TPC-C. http://www.tpc.org/tpcc.

[5] Twemcache. https://twitter.com/twemcache.

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

[7] S. Barahmand and S. Ghandeharizadeh. Expedited Benchmarking of Social Networking Actions with Agile Data Load Techniques. *CIKM*, 2013.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] Dell. Dell Fluid Cache for Storage Area Networks, http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san, 2014.

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

[17] S. Ghandeharizadeh and S. Barahmand. A Mid-Flight Synopsis of the BG Social Networking Benchmark. *Fourth Workshop on Big Data Benchmarking*, October 2013.

[18] S. Ghandeharizadeh, R. Boghrati, and S. Barahmand. An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions. *TPC Technology Conference*, September 2014.

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

[20] S. Ghandeharizadeh, S. Irani, and J. Lam. Memory Hierarchy Design for Caching Middleware in the Age of NVM, USC Database Lab Tech Report 2015-01, http://dblab.usc.edu/Users/papers/CacheDesTR2.pdf.

[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. Strong Consistency in Cache Augmented SQL Systems. *Middleware*, December 2014.

[23] G. Graefe. The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *DaMoN*, page 6, 2007.

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

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

[26] 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.

[27] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, June 1981.

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

[29] Linux. Flexible I/O Tester Synthetic Benchmark, http://linux.die.net/man/1/fio.

[30] D. Liu, J. Tai, J. Lo, N. Mi, and X. Zhu. VFRM: Flash Resource Manager in VMware ESX Server. In *IEEE Network Operations and Management Symposium*, 2014.

[31] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[32] Michael Kirkland. Flashcache at Facebook: From 2010 to 2013 and beyond. https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920/, 2013. Online; accessed 16 January 2016.

[33] 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*, 2013.

[34] Oracle Inc. Oracle Database Smart Flash Cache, 2010.

[35] 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.

[36] S. Daniel and S. Jafri. Using NetApp Flash Cache (PAM II) in Online Transaction Processing. NetApp White Paper, 2009.

[37] W. Stearns and K. Overstreet. Bcache: Caching Beyond Just RAM. https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/, 2010.

[38] STEC. EnhanceIO SSD Caching Software, https://github.com/stec-inc/EnhanceIO, 2012.

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

[40] Eric Van Hensbergen and Ming Zhao. Dynamic policy disk caching for storage networking. *URL: http://visa. cs. fiu. edu/ming/dmcache*, 2006.

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