Cache Augmented Database Management Systems

Shahram Ghandeharizadeh
University of Southern California
Los Angeles, California
shahram@usc.edu

Jason Yap
University of Southern California
Los Angeles, California
jyap@usc.edu

ABSTRACT

Cache Augmented Database Management Systems, CADBMSs, enhance the velocity of simple operations that read and write a small amount of data from big data. They are most suitable for those applications with workloads that exhibit a high read to write ratio, e.g., interactive social networking actions. This study surveys state of the art with CADBMSs and presents *physical data independence* as the next step in their evolution. We detail the requirements of this evolution, technological trends and software practices, and our research efforts in this area.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

General Terms

Algorithms, Design, Performance

Keywords

Middle-tier caches, physical data independence

1. INTRODUCTION

In the era of no "one-size-fits-all", organizations extend a relational database management system (RDBMS) with a key-value store (KVS) to enhance the velocity of "simple operations" [11, 27] that either read or write a very small amount of big data. The resulting cache augmented database management system, CADBMS, targets applications with a high read to write ratio. An example application is social networking with interactive actions such as browse a profile, view a friend's resource (say a picture) and post a comment on it, generate a friend request and accept one, and others. (According to [3], more than 99% of Facebook actions are read operations.) A popular distributed in-memory KVS is memcached and in use by well known Internet destinations such as YouTube, Wikipedia, and others [25, 4]. Its simple interface provides put, get, and delete of key-value pairs computed using data in the RDBMS.

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

A code segment may execute several queries and perform arbitrarily complex application logic. As long as it is deterministic, a key identifies a unique input to the code segment and its value is the corresponding output.

To illustrate the performance enhancement obtained using CADBMS, Figure 3 shows the throughput of several systems with the same workload using a social networking benchmark named BG [5, 6], see Section 1.1 for details. A document store named MongoDB performs slower than an industrial strength RDBMS named SQL-X. However, once either system is augmented with Ehcache (SAS CADBMS of Section 3), overall performance is enhanced dramatically. The 22 fold performance improvement for a single node of MongoDB complements its ability to scale horizontally.

This paper identifies CADBMSs as an important class of systems that are fast evolving. They are still in their infancy, providing the database community (both industry and research) with significant opportunity to extend and enhance this technology. We believe physical data independence is the next major evolution of CADBMSs, see Section 4 for details.

1.1 Overview

This paper presents performance numbers to make a case for CADBMSs. These are obtained using BG [5], a benchmark to evaluate the performance of data stores for processing interactive social networking actions. BG is partic-

[1]Performs less wasteful work [19], see Section 5.

DBSocial '13 New York, NY USA

ularly suitable to evaluate CADBMSs because it quantifies the amount of stale data produced by these systems. In all experiments, the BG database is configured with 10,000 members, 100 friends per member, and 100 resources per member. The imposed workload consists of 99% read and 1% write actions, see Table 1. It requires the following SLA: 95% of actions to complete in 100 msec or faster. BG computes the maximum observed throughput supported by a data store. This is termed the Social Action Rating (SoAR) of the data store. A higher SoAR value means a higher performant system.

We focus on the following data stores:

1. SQL-X: An industrial strength relational database management system (RDBMS) with ACID properties and a SQL query interface. Due to licensing restrictions, we cannot disclose its identity and name it SQL-X.

2. MongoDB version 2.0.6, a document store, representing a NoSQL solution. See [11] for a taxonomy.

3. A Client-Server (CS) CADBMS consisting of SQL-X extended with memcached [22] server version 1.4.2 (64 bit). In the presence of updates to the RDBMS, we consider two approaches to maintain the cached key-value pairs consistent: Either using application software or RDBMS triggers, see Section 3 for details.

4. A Shared Address Space (SAS) CADBMS with application consistency: SQL-X extended with Ehcache version 2.6.2 [28] and application software to maintain key-value pairs consistent. See Section 3.

All numbers are gathered from a dedicated hardware platform consisting of nine PCs connected using a gigabit Ethernet switch. Two PCs acts as servers: one hosts a data store (SQL-X) and the other hosts an instance of memcached. Each server PC consists of a 64 bit 3.4 GHz Intel Core i7-3770 processor (4 cores with hyperthreading) configured with 16 GB of memory, 1.5 TB of storage, and three teamed gigabit networking cards (for a total bandwidth of 3 Gbps). The remaining 7 PCs are used as clients by BG to generate requests for the server nodes. In the reported experiments, the available memory of the server exceeds the database size. Hence, disk I/O is not the limiting factor, enabling us to compare a CADBMS system with either SQL-X or MongoDB.

The rest of this paper is organized as follows. Section 2 describes related work and how key-value pairs are different than materialized views. Subsequently, Section 3 details today's state of the art with CADBMSs. Section 4 outlines several research directions that would benefit from greater scrutiny by the database community. We make a case for why CADBMSs will be successful in Section 5. Section 6 contains brief words of conclusion.

2. RELATED WORK

A key-value pair of CADBMS shares similarities with a materialized view, MV, of an RDBMS. Both enhance the velocity of data intensive applications. However, their target applications are different. While MVs enhance the performance of applications that manipulate a large amount of data such as decision support applications and their On-Line Analytical Processing (OLAP) tools, key-values of a CADBMS enhance the performance of interactive applications that retrieve a small amount of data from big data.

BG Social Actions	Type	Low (1%) Write
View Profile, VP	Read	40%
List Friends, LF	Read	5%
View Friend Requests, VFR	Read	5%
Invite Friend, IF	Write	0.4%
Accept Friend Request, AFR	Write	0.2%
Reject Friend Request, RFR	Write	0.2%
Thaw Friendship, TF	Write	0.2%
View Top-K Resources, VTR	Read	49%

Table 1: A mix of social networking actions.

In the reported results of Figure 3, if we authored MVs using SQL-X, we must choose between either a very high amount of stale data (more than 70%) or a very low SoAR rating (28). This is because we must specify whether the MV is refreshed asynchronously or synchronously. The rate of asynchronous refresh is in the order of hours, producing a substantial amount of stale data. With synchronous refresh, MVs slow down updates to the database so dramatically (in the order of seconds) that it is difficult to argue that the observed response times are interactive. The diminish the SoAR of SQL-X almost 1000 folds.

The proposed CADBMS is different from the middle tier caches of the previous decade with a SQL interface [2, 1, 8, 9, 14, 21]. These caches process SQL queries while the KVS component of a CADBMS looks up key-value pairs. It strives to avoid the traditional wasteful work performed by a SQL solution such as locking and managing the buffer pool [19], see Section 5.

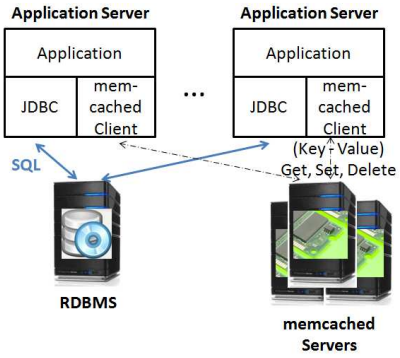
The RDBMS component of a CADBMS implements MVs and its KVS implements key-value pairs. Hence, they may co-exist. We explore this synergy in Section 4.

3. TODAY'S CADBMS

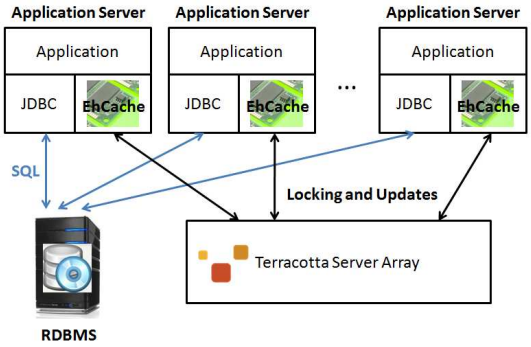
At the time of this writing, memcached [22] is a popular open source distributed in-memory KVS server. There exist a number of open source clients for memcached, e.g., Spymemcached, Xmemcached, Whalin[2] memcached and others. An application employs the client component to partition (shard) key-value pairs across several instances of memcached servers. Figure 1.a shows this *Client-Server* (CS) architecture. An alternative architecture, named *Shared Address Space* (SAS), requires the KVS to run in the address space of the application, see Figure 1.b. Example KVSs include Terracotta Ehcache [28] and JBoss Cache [10]. They operate in either stand-alone or in a distributed mode. With the latter, a key-value pair might be replicated either across a subset or all application+KVS instances. The KVS may implement the concept of a transaction to atomically update all replicas of a key-value in different instances.

With both SAS and CS, the KVSs provide a simple interface to insert, get, and delete key-value pairs. When compared with one another, SAS may slow down writes in order to improve the performance of reads. This is because CS requires a get to retrieve its referenced value across the network, uncompress and deserialize it to a format suitable for use by the application. The SAS architecture eliminates

[2]All experimental results with memcached reported in this paper are conducted using Whalin client.



1.a) Client-Server (CS) architecture



1.b) Shared Address Space (SAS) architecture

Figure 1: Alternative CADBMS architectures.

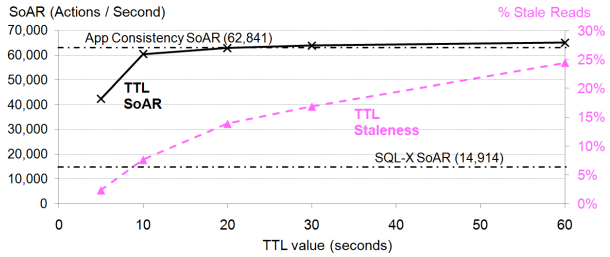


Figure 2: Throughput and percentage of reads that observe stale data with different TTL values using BG and the workload of Table 1.

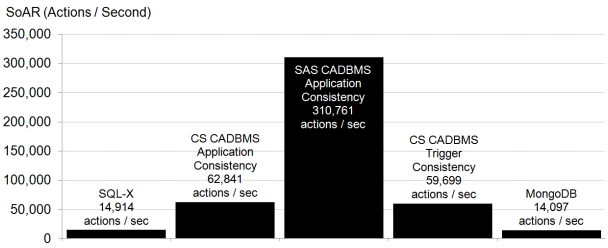


Figure 3: Throughput of SQL-X, MongoDB, and alternative CADBMS architectures with the workload of Table 1. With CS CADBMSs (memcached), App and Trig provide comparable results.

these overheads as the data is stored locally and in the native application format. Writes with SAS might be slowed down because they must propagate to all replicas of a key-value pair in different KVS instances, see Figure 1.b. When writes are rare, SAS may outperform CS dramatically (several folds), see discussions of Figure 3 below.

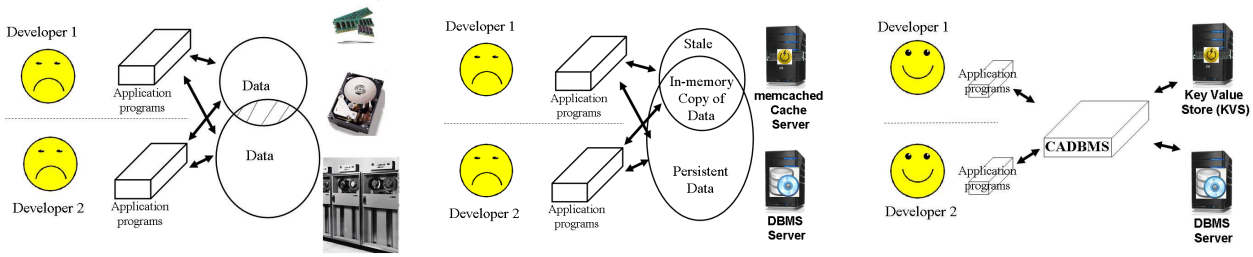
A challenge of using a CADBMS is how to maintain the key-value pairs consistent with both incremental and bulk updates to the database. One may taxonomize today's approaches into time to live and invalidation techniques. With the former, the developer extends the application to provide a Time To Live, TTL, for each key-value pair inserted in KVS. The KVS invalidates a key-value pair once its TTL expires, causing a subsequent reference for it to observe a miss, re-compute the value and insert it in the KVS. Figure 2 shows the throughput and amount of stale data observed with a client-server CADBMS (SQL-X using memcached with Whalin client) with different TTL values. Its experimental setting is identical to the one shown in Figure 3 except for the use of TTL. The x-axis of Figure 2 shows different TTL values, ranging from 5 seconds to 1 minute. As TTL increases, the throughput of the CADBMS is enhanced due to a higher KVS hit rate. It also causes a larger percentage of reads to observe stale data because the key-value pairs are stale and inconsistent with their tabular representation in SQL-X.

One may implement an invalidation based technique in either the application or the DBMS. With application consistency, App, the developer identifies code segments of the application that update the database and extends them to either invalidate, refresh, or propagate the change to the key-value pairs in the KVS. With the second technique, named Trig, the database administrator authors triggers (notification mechanisms) to either invalidate [16] or update [18] the KVS. We implemented these two techniques using SQL-X. Figure 2 shows that App provides a SoAR similar to TTL of 20 seconds with a significantly lower amount of stale data (0.01% vs 14%). All values of TTL outperform the base SQL-X. Both App and Trig provide comparable performance, see the two CS CADBMS bars in Figure 3. And, both produce some stale data because they suffer from race conditions between writes to SQL-X and memcached. These are identified in [15] with a description of the Gumball technique to detect and avoid them.

The amount of stale data observed with Trig(0.8%) is higher than with App (0.01%) because: 1) SQL-X employs Multiversion Concurrency Control (MVCC) technique [7], and 2) update transactions have a longer lifetime with Trig as they invalidate key-value pairs synchronously, i.e. incur network round-trip times to communicate with the KVS. This longer lifetime increases the likelihood of another read transaction using MVCC to compute and storing a stale value in the cache after the trigger transaction commits.

4. EXTENDING CADBMS TECHNOLOGY

Today's CADBMSs are in their infancy and resemble the data intensive applications of 1970s that existed at the dawn



4.a) Prior to RDBMSs. 4.b) CADBMSs today. 4.c) Future CADBMSs.

Figure 4: Decades of database technology.

of RDBMSs, see Figures 4.a and 4.b. They lack *physical data independence* that hides the details of the storage devices and their physical idiosyncrasies[3] from user applications. In essence, when a CADBMS employs a transactional RDBMS, the application developer authors software to maintain the normalized tables of the RDBMS (magnetic disk of 1970s) consistent with the key-value pairs stored in a KVS (main memory of 1970s). Physical data independence enables a CADBMS to hide details of RDBMS and KVS from the application developer to provide functionalities such as transparent cache consistency (maintaining the content of KVS and RDBMS consistent with one another seamlessly), dynamically adjust the content of KVS to enhance overall system performance, and support different form of consistency ranging from weak to strong. This is beneficial and superior to today's state of the art for two reasons. First, it reduces the complexity of the application software to expedite software development life cycle, empowering application developers to introduce features more rapidly at reduced costs.

Second, it enhances robustness of the deployed system by preventing software assumptions that compromise availability of the data. An example comes from Facebook where, due to dependence of data on different physical forms of storage, a software component was authored with the assumption that configuration data from the cache is obsolete and erroneous while its counterpart in the database is always correct. Every time this component observed erroneous data from the cache, it would query the database to refresh the cache with correct data. On September 23, 2010, erroneous configuration data was inserted into the database, causing this component to overwhelm the DBMS with repeated queries for the correct data [20]. Physical data independence would have eliminated the flawed assumption which resulted in 2.5 hours of down time.

A key component of physical data independence is the concept of transparent caching. This concept refers to a spectrum of possible solutions. At one end, it implies a *seamless caching* solution where the developer has no awareness of the KVS and the DBMS, utilizing a high level language to communicate with the CADBMS. Examples include COSAR-CQN [17], SQLTrig:QR [16] and CacheGenie [18]. At the other end of the spectrum, this concept may imply *transparent cache consistency* only. This means the developer is aware of the components of the CADBMS and authors software to utilize these components effectively for their target application. However, CADBMS guarantees the consistency of the key-value pairs in the KVS with the data stored in the DBMS. Examples include SQLTrig:SemiData [16] and TxCache [23]. While transparent cache consistency facilitates the examples presented in Section 1, seamless caching cannot because the developer is not aware of the KVS to identify code segments with explicit get and put operations. The example systems share several similarities. First, they target SQL RDBMS. Second, they direct all updates to the RDBMS that either invalidates [17], incrementally updates [18], or refreshes [16] the key-value pairs.

Amongst the identified research prototypes, COSAR-CQN is particularly interesting because it tries to provide seamless caching using the query change notification (QCN) mechanism of its RDBMS. This is a relatively new feature supported by Oracle 11g and Microsoft SQL Server 2005 and 2008 editions. A key finding of [17] is that the QCN mechanism of today's RDBMSs is in its infancy and slows down updates so dramatically (tens of seconds) that it is difficult to argue they are interactive. While performing updates asynchronously address this issue, it introduces two limitations. First, the resulting solution produces a large amount of stale data. Second, developing an application using such a solution might be challenging because a transaction may not observe its own update. We are almost certain that the QCN mechanism of RDBMSs will mature in the years to come, enabling commercial vendors to offer CADBMSs.

Both ends of the spectrum are challenged to provide strict with DBMSs that employ MVCC [7]. This is because a lookup that observes a cache miss for a key-value pair, $k_1 - v_1$, may race with an update that changes the state of the database and invalidates $k_1 - v_1$. MVCC enables the lookup to race with the update in the RDBMS to compute a stale value and insert this value in the cache after the update has invalidated $k_1 - v_1$. One approach to address this limitation is to extend the RDBMS to provide transaction ids in order to enable the KVS to detect and avoid inconsistent states [23]. An obvious question is whether inconsistent states can be prevented with no software changes to the RDBMS. This is desirable because it enables the CADBMS to become RDBMS agnostic, functioning with a diverse range of RDBMSs that implement MVCC.

Today's CADBMSs research prototypes using SQL motivate the following research questions: How do the alternative systems compare with one another (both qualitatively and quantitatively)? What workload characteristics enable one to outperform the other? What other solutions are possible between seamless caching and transparent cache consistency? Is it possible for a pre-processor to perform the

[3]For example, volatile versus non-volatile property.

developers task of identifying code segments that must correspond with a key-value pair? How should NoSQL solutions realize seamless caching and transparent cache consistency once augmented with a look up cache?

Write through CADBMS: An open research topic is how a CADBMS may utilize its KVS as a write through cache. The idea is for the CADBMS to update key-value pairs in the KVS. These updates are then propagated to the DBMS. This reduces the work imposed on the DBMS. Moreover, the KVS may batch several updates to a key-value pair into one, performing one update to the DBMS for the batch. This update might be performed asynchronously when the DBMS is lightly loaded.

A write through CADBMS poses two challenges. First, how does the system perform the update in a manner that is either seamless or provides transparent consistency for the DBMS? Second, how would the KVS provide for durability of transactions in the presence of its node failures? While loss of durability might be acceptable for certain applications, e.g., social networking sites, it would be ideal to introduce techniques that prevent them.

Data availability: A key question is how the CADBMS operates when a KVS instance becomes disconnected from the DBMS for a short period of time (due to transient loss of network connectivity)? And, how does it continue operation once the connectivity is restored? The DBMS and KVS may detect loss of connectivity using heart beat messages. Furthermore, to provide consistent reads to those applications that demand it, once the KVS detects loss of connectivity, it may stop servicing requests by reporting cache misses for all key look ups.

Once connectivity is re-established, one may implement alternative approaches to resume normal mode of operation while guaranteeing consistent reads. One approach is to require the KVS instance to discard all its key-value pairs because it has no knowledge of RDBMS notifications issued during loss of network connectivity. This diminishes system performance during the time the KVS instance is repopulated with key-value pairs. This recovery period might be unreasonably long given today's large memory capacities, i.e., tens and hundreds of Gigabytes, motivating the following two alternatives.

An alternative approach is for the DBMS to detect network dis-connectivity with a KVS instance and store the notifications for this KVS in a table or a collection. (To simplify discussion and without loss of generality, we use the term table for the rest of this discussion.) Once connectivity is re-established, the KVS contacts the DBMS and reads the table to update its key-value pairs (while processing in-progress notifications due to on-going DBMS updates). It starts to process request once it completes applying all the notifications buffered in the DBMS table. A key question with this technique is how long should the DBMS accumulate notifications? Given the inexpensive price of disks, an answer might be as long as the time required to replay these notifications during recovery time does not exceed the time for the KVS to reconstruct its entire cache, i.e., the first approach. This supports scenarios where the KVS is removed from the CADBMS permanently.

Yet another possibility might be to use other KVS instances (and CADBMS client components of the participating applications that are a component of the physical data independence solution) to route notifications from the RDBMS to the KVS instance that cannot communicate with the RDBMS. This requires the participants to employ a collaborative routing protocol similar to those used by peer-to-peer networks, e.g., CAN [24], Chord [26], etc. It is interesting to note that such a protocol facilitates elasticity of a CADBMS to accommodate addition (removal) of KVS instances incrementally with no down time.

5. A CASE FOR CADBMS

CADBMSs are an important class of systems because they enable interactive big data applications that would otherwise be too expensive to implement. Their key-value pairs are as essential as materialized views in an RDBMS. They are database agnostic and operate with SQL, NoSQL, and NewSQL solutions. And, they synergize with today's technological trend that offers servers with an ever increasing memory capacities at inexpensive prices. Below, we elaborate on this and others to argue for viability of CADBMSs.

1. CADBMSs leverage main memory effectively: Several studies have observed that today's RDBMSs spend less than 15% of their resources performing useful work [19] with a bulk of their resources utilized for locking, logging, managing buffer pool, and incurring the overhead of multi-threading [27]. Stonebraker and Cattell observe that a main memory database with conventional multi-threading, locking and recovery is only marginally faster than its disk-based counterpart [27]. To make use of servers with an ever increasing memory capacities (e.g., HP Integrity Server with 384 GB memory), one requires the revolutionary approach of CADBMSs that looks up query results and sidesteps query execution all together.

2. CADBMSs provide shared-nothing scalability and high data availability: Shared-nothing is a multi-node architecture with each node having its own memory, multi-core CPU, and mass storage device. It is used by SQL, NoSQL, and NewSQL engines and scales by sharding data across nodes. A fragment of data is assigned to one node and replicated across one or more node(s) to enable the system to continue operation in the presence of node failures [13]. A CS CADBMS architecture employs the same proven concepts to scale and continue operation in the presence of failures. An example CADBMS is Couchbase [12].

The SAS CADBMS architecture may not scale as well as a shared-nothing architecture for writes due to the communication overhead to maintain multiple replicas of a key-value pair consistent. This may not be an issue for applications with workloads that exhibit a high read to write ratio. Its improved performance for a single node is sufficiently high that it would offer a SoAR similar to the CS architecture with fewer nodes. Its more than four fold performance enhancement in Figure 3 results in lower hardware cost, rack space, cooling, and power consumption. If each node fails with the same probability, the SAS CADBMS observes fewer failures than the CS architecture for a given time period, influencing the amount of redundancy installed and the amount of administrative time required to deal with failures [27].

3. CADBMSs are open source and may offer ACID consistency using an RDBMS: ACID properties are non-trivial to implement and require substantial effort and investment. CADBMSs do not implement ACID from scratch. Instead, they leverage existing RDBMSs that support ACID properties. As an example, [23] implements a transactional

cache that offers ACID properties with snap shot isolation. It requires a slight extension to those RDBMSs that implement snap shot isolation, e.g., PostgreSQL.

6. CONCLUSION

CADBMSs enable simple operations that read and write a small amount of data from big data. They are most suitable for those applications whose workload exhibits a high read to write ratio, e.g., interactive social networking actions [5]. The next step in their evolution is to provide physical data independence, requiring extensions of today's technology to integrate database management systems and key-value stores seamlessly. The envisioned future CADBMSs are consistent with today's technological trends and software practices.

7. ACKNOWLEDGMENT

We thank Sumita Barahmand and our anonymous reviewers for their valuable comments.

8. 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, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *ICDE*, pages 821–831, 2003.

[3] Z. Amsden, N. Bronson, G. C. III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, L. Puzar, and V. Venkataramani. TAO: How Facebook Serves the Social Graph. In *SIGMOD Conference*, 2012.

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

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

[6] S. Barahmand and S. Ghandeharizadeh. D-Zipfian: A Decentralized Implementation of Zipfian. *ACM SIGMOD DBTest Workshop*, June 2013.

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

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

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

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

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

[13] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), March 1990.

[14] C. Garrod, A. Manjhi, A. Ailamaki, B. M. Maggs, T. C. Mowry, C. Olston, and A. Tomasic. Scalable Query Result Caching for Web Applications. *PVLDB*, 1(1):550–561, 2008.

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

[16] S. Ghandeharizadeh and J. Yap. SQL Query To Trigger Translation: A Novel Consistency Technique for Cache Augmented DBMSs. In *USC DBLAB Technical Report 2012-09, http://dblab.usc.edu/users/papers/sqltrig.pdf, Submitted for Publication*, 2012.

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

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

[19] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008.

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

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

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

[23] 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.

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

[25] P. Saab. Scaling memcached at Facebook, https://www.facebook.com/note.php?note_id=39391378919.

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

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

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