A Comparison of Two Physical Data Designs for Interactive Social Networking Actions*

Sumita Barahmand, Shahram Ghandeharizadeh, Jason Yap

Database Laboratory Technical Report 2012-08

Computer Science Department, USC

Los Angeles, California 90089-0781

{barahman,shahram,jyap}@usc.edu

October 7, 2013

Abstract

This paper compares the performance of an SQL solution that implements a relational data model with a document store named MongoDB. We report on the performance of a single node configuration of each data store and assume the database is small enough to fit in main memory. We analyze utilization of the CPU cores and the network bandwidth to compare the two data stores. Our key findings are as follows. First, for those social networking actions that read and write a small amount of data, the join operator of the SQL solution is not slower than the JSON representation of MongoDB. Second, with a mix of actions, the SQL solution provides either the same performance as MongoDB or outperforms it by 20%. Third, a middle-tier cache enhances the performance of both data stores as query result look up is significantly faster than query processing with either system.

A Introduction

There is an abundance of data stores with both the computer industry and the research arena contributing novel architectures and data models. In [10], Cattell surveys and classifies 22 data stores to motivate a quantitative analysis of the alternative designs and implementations. We study a specific aspect of this vast multi-faceted topic, namely, a comparison of an industrial strength relational database management system (RDBMS) named[1] SQL-X and a NoSQL document store named MongoDB. While SQL-X implements a relational data model [12], MongoDB implements a

[*] A shorter version of this paper appeared in the ACM International Conference on Information and Knowledge Management (CIKM), San Francisco, CA, Oct 2013.

[1] Due to licensing agreement, we cannot disclose the identity of this system.

JSON representation of data [14]. Each offers a rich set of design choices. We use the BG [5] benchmark to exercise the different capabilities of each data store. This social networking benchmark consists of a database and eleven actions (see Table 1) that either read or write a small amount of data from the database.

While SQL-X does not scale horizontally, MongoDB scales to a large number of nodes. In addition to impacting the performance of a single node instance of each data store, physical organization of data impacts the horizontal scalability of MongoDB. While both are important, we focus on the performance of a single node instance of each data store for the following reasons. First, it provides insights into the tradeoffs associated with two alternative logical data designs, namely, relational and JSON. An interesting finding is that the use of the join operator is not slower than the JSON representation, see Section D.

Second, while BG's interactive social networking actions are simple, they interact in complex ways to offer a wide range of design choices. We show it is beneficial to move the work of read actions to write actions when the workload is dominated by read actions. (According to Facebook, more than 99% of their workload is dominated by queries [3, 28].) Materialized views are not appropriate because they provide either a very low performance or a high amount of stale data, see Section E.

Members($\underline{userid}, username, pw, firstname, lastname, job, gender, jdate, ldate, address, email, tel, profileImage, thumbnailImage$)

Friends($\underline{\widehat{inviterID}, \widehat{inviteeID}}, \widehat{status}$)

Resource($\underline{rid}, \widehat{creatorid}, \widehat{walluserid}, type, body, doc$)

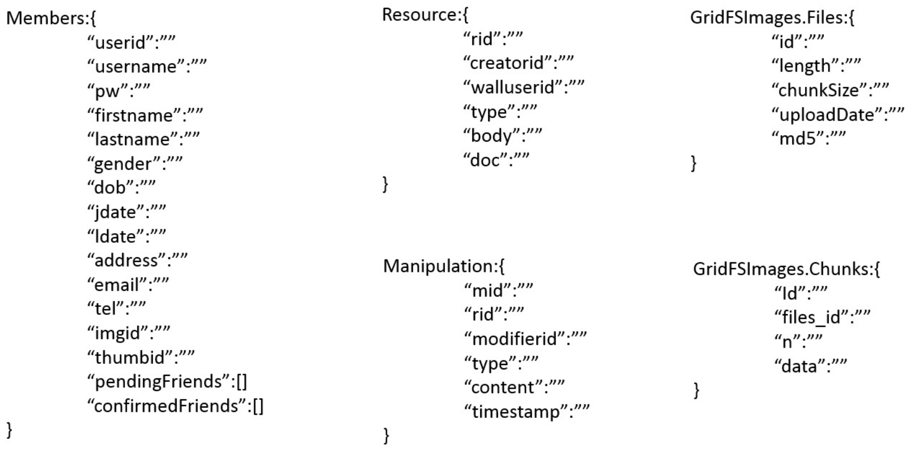
Manipulation($\underline{mid}, modifierid, \widehat{rid}, \widehat{creatorid}, timestamp, type, content$)

Figure 1: Basic SQL-X database design. The underlined attribute(s) denote the primary key of a table. Attributes with a hat denote the indexed attributes.

Third, BG's social networking actions impose a small amount of work on a node and should be processed by a single node of a multi-node data store. Otherwise, the overhead of parallelism limits the scalability of a data store [19, 15, 33]. By understanding factors that enhance the performance of a single node, we provide a solid foundation to investigate alternative designs that impact the scalability of a data store. These alternatives include partitioning strategies, replication, and secondary indexes [15, 29].

Our primary contribution is a quantitative comparison of two different data store architectures to provide insights into their working operations. Data store architects may use these results to enhance the performance of their existing data store for social networking actions. Social networking sites may use these results to fine tune the performance of their existing deployments. For example, when computing the friends of a member [3], obtained results suggest join of two tables might be fast enough as long as images are represented effectively, see Section C.

Related work: An experimental comparison of RDBMS solutions with the NoSQL systems for interactive data-serving environments and decision support workloads is presented in [18]. Its key finding is that the SQL systems provide significant performance advantages and that NoSQL systems are fairly competitive in many cases. It employs the



```
Members:{
        "userid":""
        "username":""
        "pw":""
        "firstname":""
        "lastname":""
        "gender":""
        "dob":""
        "jdate":""
        "ldate":""
        "address":""
        "email":""
        "tel":""
        "imgid":""
        "thumbid":""
        "pendingFriends":[]
        "confirmedFriends":[]
}

Resource:{
        "rid":""
        "creatorid":""
        "walluserid":""
        "type":""
        "body":""
        "doc":""
}

Manipulation:{
        "mid":""
        "rid":""
        "modifierid":""
        "type":""
        "content":""
        "timestamp":""
}

GridFSImages.Files:{
        "id":""
        "length":""
        "chunkSize":""
        "uploadDate":""
        "md5":""
}

GridFSImages.Chunks:{
        "Id":""
        "files_id":""
        "n":""
        "data":""
}
```

Figure 2: Basic MongoDB design of BG's database.

YCSB benchmark [13] for its evaluation of interactive environments. Our evaluation focuses on interactive social networking actions and considers a richer conceptual data model and workload. We explore a subset of a large space of possibilities including the use of caches, quantifying their tradeoffs.

The rest of this paper is organized as follows. Section B provides an overview of the BG social networking benchmark. It includes an organization of data, termed Basic, with both SQL-X and MongoDB. Sections C to F present physical design enhancements to the Basic design of each system. This discussion includes a quantitative analysis to identify the best design decisions (termed Boosted) for each system. We use these designs to compare SQL-X with MongoDB in Section G. Brief conclusions and future research directions are presented in Section I.

B Overview of BG Benchmark

BG [5] is a benchmark to rate data stores for interactive social networking actions and sessions. These actions and sessions either read or write a very small amount of the entire data set. BG uses a thread to emulate a member of a social networking site viewing either her own profile or that of another member, listing either her friends or those of another member, inviting another member to be friends, viewing her top-k resources (image, posting), and others. The first column of Table 1 lists all actions supported by BG. These actions are common to sites such as Facebook, LinkedIn, Twitter, FourSquare, and others [5].

BG models a database consisting of a fixed number of members (M) with a registered profile. Each member profile may consist of either zero or 2 images. With the latter, one image is a thumbnail and the second is a higher resolution image. While thumbnails are displayed when listing friends of a member, the higher resolution image is displayed when a member visits a user's profile. An experiment starts with a fixed number of friends (ϕ) and resources per member (ρ). This study assumes a database of 10,000 profiles with 2 KByte thumbnail images and 12 KByte

BG Social Actions	Type	Very Low (0.1%) Write	Low (1%) Write	High (10%) Write
View Profile (VP)	Read	40%	40%	35%
List Friends (LF)	Read	5%	5%	5%
View Friends Requests (VFR)	Read	5%	5%	5%
Invite Friend (IF)	Write	0.02%	0.2%	2%
Accept Friend Request (AFR)	Write	0.02%	0.2%	2%
Reject Friend Request (RFR)	Write	0.03%	0.3%	3%
Thaw Friendship (TF)	Write	0.03%	0.3%	3%
View Top-K Resources (VTR)	Read	40%	40%	35%
View Comments on Resource (VCR)	Read	9.9%	9%	10%
Post Comment on a Resource (PCR)	Write	0%	0%	0%
Delete Comment from a Resource (DCR)	Write	0%	0%	0%

Table 1: Three mixes of social networking actions.

profile images. We also consider databases with no images. All experiments start with 100 friends[2] and resources per user, ϕ=ρ=100. (The time to load the database with MongoDB is slightly faster than SQL-X [7].) We have conducted experiments with a 100K member database. The reported observations and trends do not change as long as the benchmark database is smaller than the server memory.

BG computes a Social Action Rating (SoAR) of a data store based on a pre-specified service level agreement (SLA) by manipulating the number of threads (i.e., emulated members) that perform actions simultaneously. SoAR is the maximum system throughput (actions per second) that satisfies the SLA. All SoAR ratings in this paper are established with the following SLA: 95% of requests observe a response time of 100 milliseconds or faster with unpredictable (stale) data lower than 0.1%. An ideal physical data design is one that maximizes SoAR of a system. Data designs using materialized views and cache augmented data stores may produce stale data. The former is because the RDBMS may propagate updates to the materialized view asynchronously. The latter is due to write-write race conditions between the data store and the cache [20].

Figure 1 shows the relational design of BG's database. The underlined attributes are the primary keys of the identified tables. Index structures are constructed on these attributes to facilitate efficient processing of read actions. For example, with view profile action referencing a member with a specific userid, say 5, a hash index facilitate efficient retrieval of the Member corresponding to this userid. Members table may store images as BLOBs. Alternatives are discussed in Section C. Computing either list of friends or pending friends requires a join between Members and Friends table. Section E explores the use of materialized views and their alternatives to migrate the work of read actions to write actions for computing simple analytics. We report SoAR of these designs with SQL-X.

Figure 2 shows the JSON design of BG's database tailored for use with MongoDB. For each member M_i, this design maintains three different arrays: 1) pendingFriends maintains the id of members who have extended a friend invitation to M_i, 2) confirmedFriends maintains the id of members who are friends with M_i, and 3) wallResourceIds maintains the id of resources (e.g., images) posted on M_i's profile. One may store profile and thumbnail image of

[2] Median Facebook friend count is 100 [35, 4].

each member either in the file system, MongoDB's GridFS, or as an array of bytes. Figure 2 shows the last two choices. When images are stored in the GridFS, the profileimageid and thumbnailimageid are stored as attributes of the Members collection (instead of the array of bytes shown in Figure 2). Section C shows one design provides a SoAR significantly higher than the other two.

In the next 3 sections, we provide additional details about BG's actions and their implementation using both the relational and JSON representations. We discuss changes to the physical organization of data and their impact on the SoAR of SQL-X and MongoDB. We analyze SoAR of SQL-X with different mixes of actions, see Table 1. Post Comment and Delete Comment actions are eliminated because we have no improved designs to offer for these actions.

To simplify discussion, this paper classifies BG's actions into those that either read or write data. A read action is one that queries data and retrieves data items without updating them. A write action is one that either inserts, deletes, or updates data items. Column 2 of Table 1 identifies different read and write actions.

All reported SoAR numbers are based on a dedicated hardware platform consisting of six PCs connected using a Gigabit Ethernet switch. Each PC consists of a 64 bit 3.4 GHz Intel Core i7-2600 processor (4 cores with 8 threads) configured with 16 GB of memory, 1.5 TB of storage, and one[3] Gigabit/second networking card. One node acts as our data store server (either MongoDB or SQL-X) at all[4] times. All other nodes are used as BGClients to generate workload for this node. With all reported SoAR values greater than zero, either the disk, all cores, or the networking card of the server hosting a data store become fully utilized. We report on the use of two networking cards to eliminate the network as a limiting resource. When SoAR is reported as zero, this means a design failed to satisfy the SLA.

C Manage Images Effectively

There is folklore that an RDMBS efficiently handles a large number of small images, while file systems are more efficient for storage and retrieval of large images [31]. With BG, we show physical organization of profile and thumbnail images in a data store impacts its SoAR rating dramatically. For example, if thumbnail images are not stored as a part of the profile structure representing a member then the performance of the system for processing the List Friend (LF) action is degraded significantly. This holds true with both MongoDB and SQL-X. Performance of SQL-X is further enhanced when profile images are stored in the file system. The same does not hold true with MongoDB. Below, we provide experimental results to demonstrate these observations.

The LF action of BG retrieves the thumbnail image and the profile information of each friend of a member (see attributes shown in Figure 1). Figure 3 shows the SoAR rating of LF with SQL-X and MongoDB with 100 friends per member. While SQL-X performs a join between two tables (Members and Friends of Figure 1) to perform this action, MongoDB looks up an array of member identifiers (confirmedFriends of Figure 2 for the referenced Member JSON instance) and retrieves the JSON object for each member. With SQL-X, we consider thumbnails stored in either the

[3] In some experiments, the server hosting either SQL-X or MongoDB is configured with two networking cards, each is a one Gigabit/second card.

[4] The same node is used as either SQL-X or MongoDB server in all experiments.

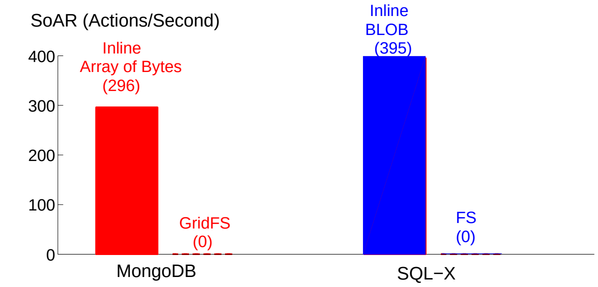


Figure 3: SoAR of List Friends (LF) with different organization of 2 KB thumbnail image, M=10K, ϕ=100.

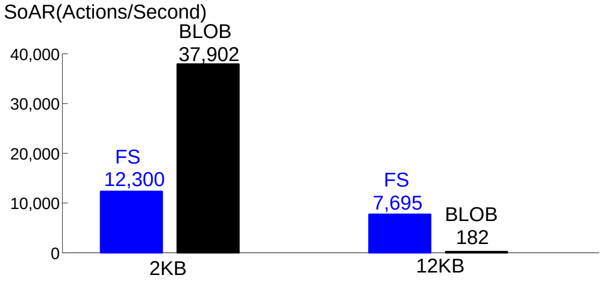


Figure 4: SoAR of SQL-X for processing a workload consisting of 100% View Profile (VP) action with images stored as either BLOBs or in the FS, M=10K, ϕ=100.

file system or inline with the record representing the member. With MongoDB, we consider thumbnails stored in its Grid File System (GridFS) or as an array of bytes in the JSON-like representation of a member. With both systems, storing the thumbnail image as a part of the Member profile enhances SoAR rating of the system from zero to a few hundred. In these experiments, the CPU of the the data store becomes 100% utilized. Note that, with a single node, the join operation of SQL-X is not necessarily slower than MongoDB's processing of confirmedFriends array to retrieve documents corresponding to the friends of the member.

The performance of SQL-X for processing View Profile (VP) action of BG is enhanced when large profile images are *not* stored in the RDBMS, see Figure 4. An alternative is to store them in the file system with a member record maintaining the name of the file containing the corresponding profile image [31, 8]. Figure 4 shows the SoAR of SQL-X with these two alternatives for two different image sizes: 2 KB and 12 KB. (As a comparison, with no images, SoAR of SQL-X is 119,746 for this workload.) A small image size, 2 KB, enables SQL-X to store the image inline with the member record, outperforming the file system by a factor of 3. SQL-X stores images inline as long as they are smaller than 4 KB. Beyond this, for example with our assumed 12 KB image sizes, the performance of SQL-X diminishes dramatically, enabling the file system to outperform it by more than 40 folds.

MongoDB's GridFS provides effective support for images. Its SoAR is comparable to storing these images in the file system. It outperforms the file system by more than a factor of two with very large profile images, e.g., 500 KB. It is worth noting that SQL-X outperforms MongoDB with image sizes smaller than 4 KB by inlining them in profile

Social Action	One Record per Friendship	Two Records per Friendship

Member 1’s number of friends	SELECT count(*) FROM Friends
	WHERE (inviterID=1 or inviteeID=1)
	and status=‘C’
		SELECT count(*) FROM Friends
		WHERE inviterID=1 and status=‘C’

Member 1’s list of friends	SELECT m.* FROM Members m, Friends f
	WHERE ((f.inviterID=1 and m.userid=f.inviteeID) or
	(f.inviteeID=1 and m.userid=f.inviterID))
	and f.status=‘C’
		SELECT m.* FROM Members m, Friends f
		WHERE f.inviterID=1 and
		f.status=‘C’ and
		m.userid=f.inviteeID

Member 1 invites Member 2	INSERT INTO Friends values (1, 2, ’P’)	

Member 2 accepts Member 1’s invitation	UPDATE friendship
	SET status=‘C’
	WHERE inviterID=1 and inviteeID=2
		1. UPDATE friendship SET status=‘C’
		WHERE inviterID=1 and inviteeID=2
		2. INSERT into friendship (inviteeID,
		inviterID, status) values (1, 2, ‘C’)

Member 2 rejects Member 1’s Invitation	DELETE FROM Friends
	WHERE inviterID=1 and inviteeID=2 and status=‘P’	

Member 1 thaws friendship with Member 2	DELETE
	FROM Friends
	WHERE ((inviterID=1 and inviteeID=2) or (inviterID=2 and inviteeID=1)) and status=‘C’	

Table 2: One record and two record representation of a friendship with one table, Friends table of Figure 1.

records. Beyond this limit, MongoDB outperforms SQL-X. Similar to the thumbnail discussions, if profile image sizes are known to be small in advance then one may inline them with MongoDB by representing them as an array of bytes in the Members collection, see Figure 9. Key considerations include MongoDB’s limit of 16 Megabytes for the size of a document and the impact of large documents on actions that do not require the retrieval of the profile image. For example, the List Friend (LF) action does not require the profile image. MongoDB provides an interface to remove some attribute values of a document while constructing a query. For example, one may query the Members collection for a document with userid 1 and not retrieve the profile image of the qualifying document by issuing the following expression: db.member.find({”userid”:1,”profileimage”:false}).

D Friendship

The concept of friendship between two members is central to a social networking site. Most of BG’s actions model this concept, see Table 2. An important consideration is how to represent the thumbnail image of each member listed as a friend of a referenced member. This was discussed in Section C. Hence, this section focuses on a BG database configured with no images.

Social Action	One Record per Friendship	Two Records per Friendships

Member 1's number of friends	SELECT count(*)
	FROM Frds
	WHERE frdID1=1 or frdID2=1
		SELECT count(*)
		FROM Frds
		WHERE frdID1=1

Member 1's list of friends	SELECT m.* FROM Members m, Frds f
	WHERE ((f.frdID1=1 and m.userid=f.frdID2)
	or
	(f.frdID2=1 and m.userid=f.inviterID))
		SELECT m.*
		FROM Members m, Frds f
		WHERE f.frdID1=1 and m.userid=f.frdID2

Member 1 invites Member 2	INSERT INTO PdgFrds values (1, 2)	
Member 2 accepts Member 1's invitation	1. DELETE FROM PdgFrds WHERE inviterID=1 and inviteeID=2 2. INSERT into Frds (frdID1, frdID2) values (1, 2)	1. DELETE FROM PdgFrds WHERE inviterID=1 and inviteeID=2 2. INSERT into Frds (frdID1, frdID2) values (1, 2), (2, 1)

Member 2 Rejects Member 1's Invitation	DELETE FROM PdgFrds
	WHERE inviterID=1 and inviteeID=2	

Member 1 thaws friendship with Member 2	DELETE FROM Frds WHERE (frdID1=1 and frdID2=2) or (frdID1=2 and frdID2=1)	

Table 3: One record and two record representation of a friendship with two tables, Frds and PdgFrds tables of Figure 8.

D.1 Relational Design: A Tale of One or Two

With a relational design, one may represent pending and confirmed friendships as either one or two tables. With each alternative, a friendship might be represented as either one or two rows. We elaborate on these designs below. Subsequently, we establish their SoAR rating. Obtained results show that a two table design is superior to a one table design.

Figure 1 shows a design that employs one table. It employs an attribute named “status” to differentiate between pending and confirmed friendships: A ‘C’ value denotes a confirmed friendship while a ‘P’ value denotes a pending friendship. The second column of Table 2 shows the SQL commands issued to implement the alternative BG actions with this design. Note the use of disjuncts (“or”) in the qualification list of the SQL queries. A designer may simplify these queries and eliminate disjuncts by representing a friendship with two records. The resulting queries are shown in the third column of Table 2. The design changes the implementation of the Accept Friendship Request action (fourth row of Table 2) into a transaction consisting of two SQL statements. In our implementation, all transactions are implemented as stored procedures in SQL-X.

An alternative to the one table design is to employ two different tables and separate pending friend invitations from confirmed invitations, see physical design of Figure 8 and queries of Table 3. This eliminates the “status” attribute of the one table design. However, the data designer is still faced with the dilemma to represent a friendship either as one row or two rows in the table corresponding to the confirmed friends. The second and third column of Table 3 shows the SQL commands with these two possibilities. A key difference is that SQL queries are simpler with the two record design.

When comparing the alternative designs, the two record design requires more storage space than the one record design. However, its resulting SQL queries are simpler to author and reason about. With one user issuing requests (single threaded BG), the larger number of records does not impact the service time of issued queries and update commands because index structures facilitate retrieval and manipulation of the relevant records. In a multi-user setting with a mix of read and write actions, see Table 1, the two table design outperforms the one table design when the frequency of write action is high enough to result in conflicts. Figure 5 shows SoAR of these two alternatives with each friendship represented as two records. Observed SoAR with a mix of very low (0.1%) write actions is almost identical for the two designs due to the use of index structures and a low conflict rate. With a mix of high (10%) write actions, the two table design outperforms the one table design by more than 30%. We speculate this is due to ACID property of transactions slowing down the one table design as it is used concurrently to process both pending and confirmed friendship transactions. The two table design reduces this contention among concurrently executing actions. For example, the query to compute the number of pending friend invitations for a member is no longer blocked by the transaction that thaws friendship between two members.

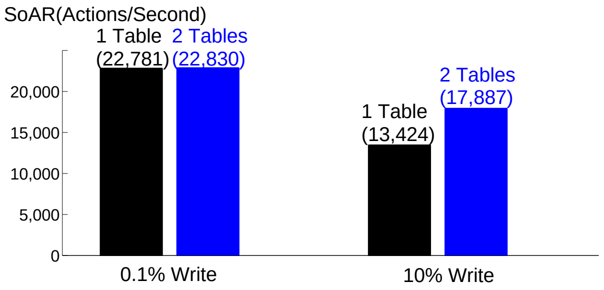


Figure 5: SoAR of SQL-X with either one or two tables for pending and confirmed friendships with two workloads, M=10K and ϕ=100. Each friendship is represented as two records.

D.2 MongoDB: List Friends

With MongoDB, BG's List Friend (LF) action is most interesting because it must retrieve the documents pertaining to the friends of a referenced member. These can be retrieved either one document at a time or all documents at once. With the former, LF is implemented by issuing a query to retrieve the basic profile information for each confirmed friend. With the latter, the entire list of friends is used with the $in operator to construct the query issued to MongoDB. This operator selects all the documents whose identifiers match the values provided in the list. With an under utilized system (a few BG threads), the second approach provides a response that is approximately 1.5 times faster than the first. This is because the first approach incurs the overhead of issuing multiple queries across the network for each document. The SoAR of these two alternatives is almost identical because the CPU of the server hosting MongoDB becomes 100% utilized.

MongoDB supports a host of write concerns, see [27] for details. We investigate two, termed *normal* and *safe* in MongoDB's documentation. Both are implemented by MongoDB's java client. The normal write concern returns control once the write is issued to the driver of the client. The safe write concern returns control once it receives an acknowledgment from the server. With a low system load (BG with one thread), the normal write concern improves the average response time of MongoDB by 13%. It does not, however, improve the processing capability of the MongoDB server and has no impact on its SoAR when compared with the safe write concern. Moreover, it produced a very low ($<$ 0.1%) amount of unpredictable reads.

E Migrate Work of Reads to Writes

Due to a high read to write ratio of the workload of social networking sites [28], one may enhance the average service time of the system by migrating the workload of reads to writes. With RDBMSs, one way to realize this is by using materialized views, MVs. Section E.1 discusses this approach and shows that it slows down write actions so dramatically that it is difficult to argue they are interactive. It presents an alternative named *Manual* that does not suffer this limitation. However, Manual requires additional software and incurs the overhead of a development life

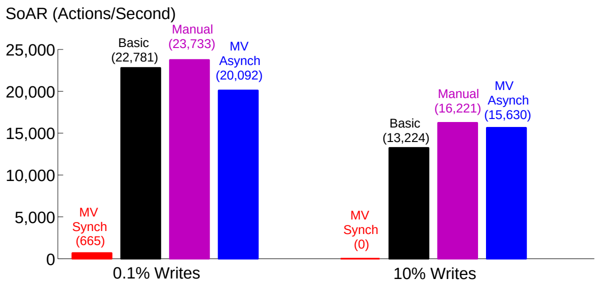
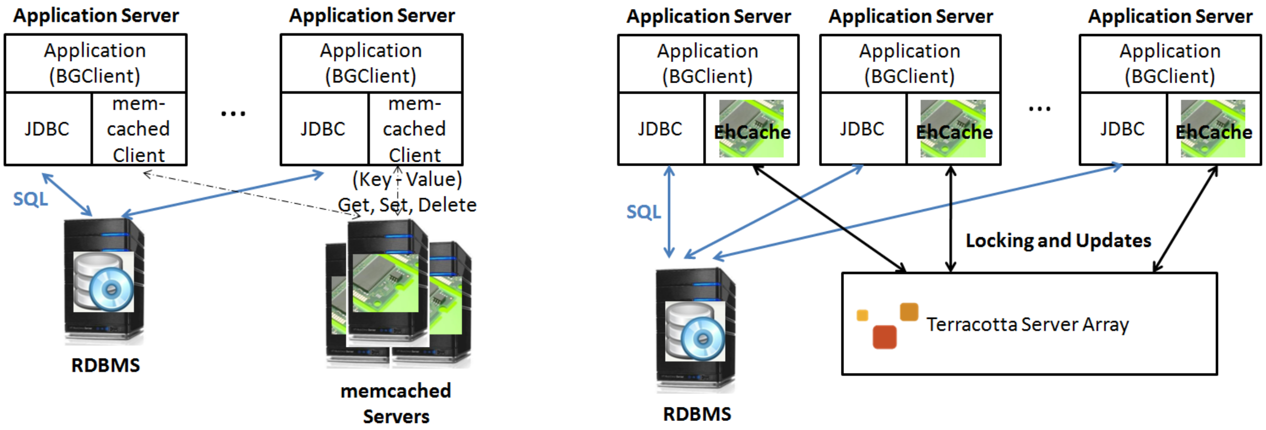


Figure 6: SoAR with the Basic SQL-X design of Figure 1, materialized views (*MV*) for aggregates as attributes with both synchronous and asynchronous mode of refresh, and developer maintained (*Manual*) aggregates as attributes, M=10K, ϕ=100, BG database has no images.



7.a) Client-Server (CS) architecture

7.b) Shared Address Space (SAS) architecture

Figure 7: Alternative cache augmented SQL architectures.

cycle.

E.1 Read Mostly Aggregates as Attributes

Social networking sites present their members with individualized “small analytics” [32]. These are aggregate information such as a member’s number of friends. BG models these using its View Profile (VP) action that provides each member with her count of resources, friends, and pending friend invitations. One may implement these in two ways: 1) Compute the aggregates each time the VP action is invoked, 2) Store the value of aggregates, look them up to process VP, and maintain them up to date in the presence of write actions that impact their value. An example SQL query that implements the former is illustrated in the first row of Table 2. The latter migrates the workload of read actions to write actions. It is appropriate when write actions are infrequent. Below, we present two alternatives to implement the second approach.

One may use Materialized Views (MVs) of SQL-X to store the value of BG’s simple analytics and require the RDBMS to maintain their value up to date. This was implemented as follows. First, we define one MV for each aggregate of the VP action. The resulting 3 views have two columns: user-id and the corresponding aggregate attribute value. Next, we author a MV that joins these three views with the original Member table (using the user-id attribute value), implementing a table that consists of each member's attributes along with 3 additional attribute values representing each aggregate for that member. This table is queried by the VP action to look up the value of its simple analytic instead of computing it.

One may configure SQL-X to refresh MVs either synchronously or asynchronously in the presence of updates. The asynchronous refresh is in the order of hours, causing the MV to contain stale data. BG quantifies these as *unpredictable* reads. Below, we discuss this in combination with the observed SoAR.

With no profile image and a read workload that invokes the VP action only, the authored MV improves SoAR of SQL-X more than six folds from 19,020 to 119,746 actions per second. With infrequent (0.1%) writes, asynchronous mode of processing updates enables MVs to enhance SoAR of SQL-X by almost a factor of two, see Figure 6. However, this causes 31% of read actions to observe unpredictable (stale) data. The amount of unpredictable data increases to 72% with a high frequency (10%) of write actions, enhancing SoAR of SQL-X by a modest 11%.

The synchronous refresh mode of MVs eliminates unpredictable data. However, as shown in Figure 6, it diminishes SoAR of SQL-X dramatically. This is because it slows down write actions. As an example, the service time of the Accept Friend Request write action is slowed down from 1.7 millisecond to[5] 1.94 seconds with an under-utilized system, i.e., one BG thread. These service times are not interactive, rendering MVs inappropriate for BG's workload.

An alternative to MVs, named *Manual*, is for a software developer to implement aggregates as attributes by extending the Member table with 3 additional columns, one for each aggregate. When a member registers a profile, these attribute values are initialized to zero. The developer authors additional software (either in the application software or in the RDBMS in the form of stored procedures and triggers) for the write actions that impact these attribute values to update them by either incrementing or decrementing their values by one. For example, the developer extends a write action that invites Member 1 to be friends with Member 2 to increment the number of pending friends for Member 1 by one as a part of the transaction that updates the Friends table, see Section D.

Manual speeds up the VP action by transforming 4 SQL queries into one. The four queries include retrieval of the referenced member's profile attribute values, count of friends, count of pending friend invitations, and count of resources. In our experiments, Manual enhanced SoAR of SQL-X for processing the VP action by the same amount as MVs with asynchronous update. However, it produces no stale reads. When compared with Basic, Manual provides at most a 22% improvement as the VP action constitutes 35% to 40% of the workload, see Table 1.

A drawback of Manual is the additional software and its associated software development life cycle (design, implementation, testing and debugging, and maintenance). Its key advantages include interactive response times for both the read and write actions with no unpredictable reads.

[5] A 1,141 fold slow down.

F Cache Augmented Database Management Systems, CADBMS

With both MongoDB and SQL-X, a developer may avoid issuing a query to the data store by caching its output, *value*, given its unique input, *key*. This is the main motivation for middle tier caches [23, 11, 36, 17, 16, 25, 1, 2, 30, 22, 28, 21]. This section focuses on a specific subclass that employs in-memory Key-Value Stores (KVS) with a simple put, get, delete interface [21, 28]. Its use case is as follows. The developer modifies each read action to convert its input to a key and use this key to look up the KVS for a value. If the KVS returns a value then the value is produced as the output of the action without executing the main body of the read action that issues data store queries. Otherwise, the body of the read action executes, issues data store queries to compute a value (i.e., output of the read action), stores the resulting key-value pair in the KVS for future use, and returns the output to BG.

The developer must modify each write action to invalidate key-value pairs that are impacted by its insert, delete, update command to the data store. For example, the write action that enables Member 1 to accept Member 2's friendship request must invalidate 5 key-value pairs. These correspond to Member 1's profile, list of friends and list of pending friends, and Member 2's profile and list of friends.

The maximum number of unique key-value pairs is a function of the number of members/resources and read actions. With a database of 10,000 members, the view profile action of BG may populate the KVS with 10,000 unique key-value pairs. With View Comment on Resource (VCR) action and 100 resources per member, the KVS may consist of a million unique key-value pairs. The actual number of cached key-value pairs might be lower due to a skewed pattern of data access, e.g., a workload that employs a Zipfian distribution [6] to reference data items.

There are two categories of in-memory KVSs: Client-Server (CS) and Shared Address Space (SAS) [21], see Figure 7. With CS, the application server communicates with the cache via message passing. A popular CS KVS is memcached [26, 28]. With SAS, the KVS runs in the address space of the application. Examples include Terracotta's Ehcache [34] and JBoss Cache [9]. SAS KVSs implement the concept of a transaction to atomically update all replicas of a key-value in different application instances. Both CS and SAS architectures may support replication of key-value pairs and implement consistent hashing to enhance availability of data and implement elasticity. A discussion of these topics is a digression from our focus. Instead, we focus on the performance of a single cache instance. With memcached, the cache server is a process hosted on a different server than the one hosting the data store. With Ehcache, the cache instance executes in the address space of the BGClient.

In the following, we focus on the impact of the KVS with a very low (0.1%) and a high (10%) frequency of writes. With these workloads, both MongoDB and SQL-X provide comparable SoARs as either the CPU or the network bandwidth of the server hosting the KVS becomes 100% utilized. Hence, without loss of generality, we present SoARs observed with SQL-X using either memcached or Ehcache.

Table 5 presents SoAR of the alternative designs when the database is configured with either no images or 12 KB profile image sizes with two different mixes of workloads. These results show Ehcache provides the highest SoAR, outperforming memcached by more than a factor of 13 (5) with images (no images). This is because it runs

Members($\underline{userid}, username, pw, firstname, lastname, job, gender, jdate, ldate, address, email, tel, thumbnailImage$)

Frds($\underline{\widehat{frdID}1, \widehat{frdID}2}$)

PdgFrds($\underline{\widehat{inviterID}, \widehat{inviteeID}}$)

Resource($\underline{rid}, \widehat{creatorid}, \widehat{walluserid}, type, body, doc$)

Manipulation($\underline{mid}, modifierid, \widehat{rid}, \widehat{creatorid}, timestamp, type, content$)

Figure 8: Boosted SQL-X database design with profile images stored in the file system and thumbnail images as inline blobs. One record in the Frds table represents the friendship between two members. The underlined attribute(s) denote the primary key. Attributes with a hat denote the indexed attributes.

in the same address space as the BGClient, avoiding the overhead of transmitting key-value pairs across the network and deserializing them. In these experiments, the four core CPU of the server hosting BGClient (and the Ehcache) becomes 100% utilized, dictating the overall system performance. (This bottleneck explains why there is no difference between SQL-X and MongoDB once extended with Ehcache.) It is interesting to note that the SoAR of Ehcache with 12 KB images is almost twice lower than that with no images. This is due to network transmission of images for invalidated key-value pairs, increasing network utilization from 30% to 88%.

With memcached, the four core CPU of its server becomes 100% utilized when there are no images, dictating its SoAR rating. With 12 KB profile images, the network bandwidth becomes 100% utilized dictating SoAR of memcached. In these experiments, memcached could produce key-value pairs at a rate of up to 2 Gbps as its server was configured with two Gbps networking cards.

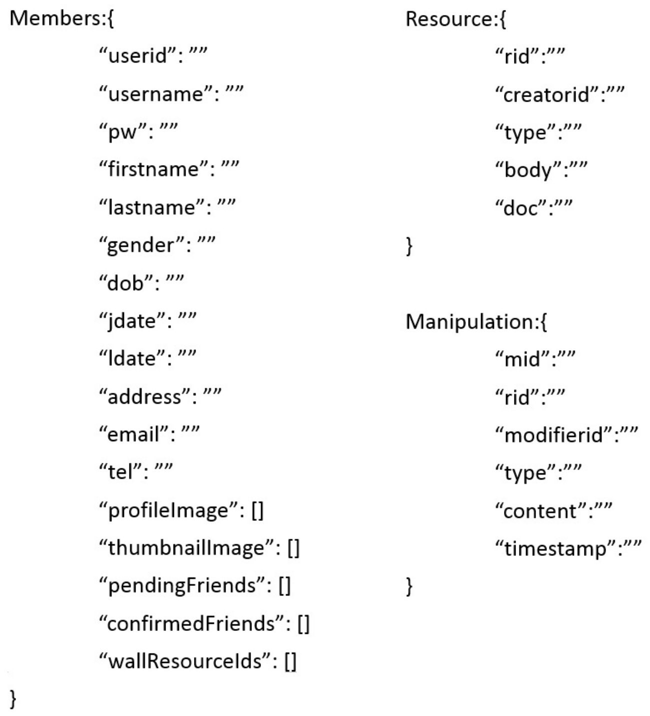
G A Comparison

Table 5 shows SoAR of the Basic SQL-X and MongoDB data designs when compared with their Boosted alternatives. (See Figures 1 and 8 (2 and 9) for the Basic and Boosted SQL-X (MongoDB) data designs.) Boosted incorporates all of the best practices presented in the previous sections except for the use of caches[6]. With both SQL-X and MongoDB, the Basic data design is inferior to the Boosted alternative because it is inefficient and utilizes its 4 core CPU fully.

With Boosted and no images, the CPU of the server hosting the data store becomes 100% utilized, dictating its SoAR. This is true with both SQL-X and MongoDB and the two workloads, 0.1% and 10% frequency of writes. These results suggest SQL-X processes BG's workload more efficiently than MongoDB because its SoAR rating is two folds higher.

With 12 KB profile images, both SQL-X and MongoDB continue to utilize their CPU fully with the Basic data design. With Boosted, the network becomes 100% utilized and dictates their SoAR rating. These results suggest SQL-X transmits less data than MongoDB to process BG's workload because its SoAR rating is 50% higher.

[6] The presented SoAR for memcached and Ehcache use the Boosted data design.



```
Members:{                                   Resource:{
            “userid”: ””                                “rid”:””
            “username”: ””                              “creatorid”:””
            “pw”: ””                                    “type”:””
            “firstname”: ””                             “body”:””
            “lastname”: ””                              “doc”:””
            “gender”: ””                    }
            “dob”: ””
            “jdate”: ””                     Manipulation:{
            “ldate”: ””                                 “mid”:””
            “address”: ””                               “rid”:””
            “email”: ””                                 “modifierid”:””
            “tel”: ””                                   “type”:””
            “profileImage”: []                          “content”:””
            “thumbnailImage”: []                        “timestamp”:””
            “pendingFriends”: []            }
            “confirmedFriends”: []
            “wallResourceIds”: []
}
```

Figure 9: Boosted MongoDB design of BG’s database.

We have conducted experiments with a 100K member database. The reported trends and observations hold true for this and other databases as long as the available server memory is larger than the size of the benchmark database.

H Break-Even point

The average service time of a data store is a function of the mix of read and write actions that constitute its workload. Both the use of caches (KVS of Section F) and an implementation of aggregates as attributes (Manual of Section E.1) slow down write actions in order to speedup read actions. This is beneficial as long as both the speedup and the frequency of read actions is high enough to compensate for the slow down observed with write actions. Otherwise, the migration of work may degrade (instead of enhance) overall system performance. An obvious question is how does the factor of slow down interact with the factor of speedup? We answer this question by characterizing what factor of slow down eclipses the observed speedup. This is the break even point. A technique that slows down write actions by a higher factor than this break even point is undesirable because it does not enhance system performance. Derivation of the break even point is as follows.

Consider the average service time of a lightly loaded system that does not incur queuing delays. Let u and $\overline{U}$ denote the frequency of write actions and their average service time, respectively. The probability of a read action is

u	Frequency of write actions.
$\overline{Q}$	Avg service time of read actions with the Basic design.
$\overline{U}$	Avg service time of write actions with the Basic design.
$\overline{S}$	Avg service time of a workload.
X	Factor of speedup in $\overline{Q}$ with a change of data design.
Y	Factor of slowdown in $\overline{U}$ with a change of data design.

Table 4: Parameters and their definitions

$(1-u)$. Assuming $\overline{Q}$ denotes the average service time of read actions, the average service time of the system, $\overline{S}$, is:

$$\overline{S} = u * \overline{U} + (1-u)\overline{Q} \tag{1}$$

The values for $\overline{U}$ and $\overline{Q}$ may vary with different physical data design changes such as those outlined in Sections E.1 and F.

Assume Equation 1 denotes average service time without a physical data design change. This is the *Basic* database design of Figure 1. A physical data design change (say aggregates as attributes of Section E.1) enhances the value of $\overline{Q}$ and increases $\overline{U}$. Let X (Y) denote the factor of speedup (slowdown) in the average read (write) service time observed with this physical data design change. The new average service time is:

$$\overline{S} = u * Y * \overline{U} + (1-u)\frac{\overline{Q}}{X} \tag{2}$$

One may solve for values of Y that cause Equations 1 and 2 to break-even:

$$Y = \frac{X-1}{X} \times \frac{(1-u)}{u} \times \frac{\overline{Q}}{\overline{U}} + 1 \tag{3}$$

For example, if the average service of read actions with the Basic design is 150 msec ($\overline{Q}$=150 msec) and a physical data design technique, say use of memcached, enhances this time 8.6 folds (X=8.6) then the same technique may slowdown write actions by a factor of 23,381 and provide the same service time as the Basic database design when the frequency of write actions is 0.4% (u=0.004) and the average service time of an update is 1.4 msec ($\overline{U}$=1.4 msec).

A value of Y higher (lower) than that computed using Equation 3 means the new database design is slower (faster) than the Basic design. In our example, values of Y greater than 23,381 imply that the Basic design is faster than using memcached.

Equation 3 quantifies several trivial and intuitive observations:

1. A proposed physical data design technique outperforms the Basic when write actions are rare, i.e., small values of u.

2. A proposed physical data design technique may incur a higher slowdown in service time of write actions (higher values of Y) and continue to outperform the Basic design as long as it enhances the service time of read actions

by a wider margin, (higher values of X). However, this observation has limits: A linear increase in value of X does not compensate for a linear increase in value of Y. In our example, speeding read actions up (value of X) from 8.6 to 100 folds enables a modest increase in slowdown of write actions (value of Y) from 23,144 to 26,185 folds. Note that more than a 10 fold increase in X did not provide a 10 fold increase in Y. And, this observations becomes exaggerated with larger values of X. In our example, increasing X from a 100 to a 1000 folds facilitates a negligible increase of Y from 26,185 to 26,423 folds. Equation 3 shows this phenomena with the component $\frac{X-1}{X}$ that approaches the value 1 (i.e., becomes irrelevant) with large values of X.

3. The tolerable slowdown in write actions (value of Y) is almost an inverse linear function of the frequency of write actions exhibited by a workload (value of u). Thus, given a workload, if its frequency of write actions is halved then the proposed physical data design may tolerate almost twice as much slowdown in average service time of its write actions and outperform the Basic design. With Equation 3, this is captured with $\frac{1-u}{u}$ when $0 < u \leq 0.2$.

4. The average service time of read actions ($\overline{Q}$) and write actions ($\overline{U}$) that constitute the workload impacts the tolerable slowdown in write actions (Y) linearly. If read actions are more complex than write actions with a significantly higher average service time then a proposed physical data design may slow down write actions by higher factors and continue to outperform the Basic design. This is captured by $\frac{\overline{Q}}{\overline{U}}$ in Equation 3.

The first observation is a property of social networking applications. While observation 2 reasons about a proposed physical data design change, Observation 3 provides intuition about different workloads with varying frequency of write actions (u). Observation 4 focuses on the average service time of write and reads actions with the RDBMS as they impact the overall average service time of both the Basic design and the proposed physical data design change.

The simple analytical model of this section enables a database designer to reason about a proposed change in the physical organization of data to estimate whether it provides an enhancement for the overall system.

I Conclusion and Future Research

This experimental paper compares organization of a social networking database with two alternative data store architectures: an industrial strength SQL solution and a NoSQL document store named MongoDB. We used the BG benchmark for this comparison. The observed SoAR ratings are impacted by two key parameters of BG. First, the mix of actions that constitute the workload. Second, configuration of member profiles with either two images or no images. We analyzed alternative enhancements to both the relational representation of SQL-X and JSON representation of MongoDB. A summary of these enhancements are as follows, starting with SQL-X:

- Separate pending and confirmed friendships into two tables: This design modification provides a 33% improvement in performance with a mixed workload that consists of a high (10%) fraction of write actions.

		Basic SQL-X	Basic MongoDB	Boosted SQL-X	Boosted MongoDB	memcached	Ehcache
No Image	0.1% Write	12,322	6,512	33,694	14,665	55,634	271,760
	10% Write	13,976	5,895	28,503	13,117	49,006	286,260
12 KB Profile Image	0.1% Write	305	0	11,820	7,700	11,888	147,845
	10% Write	300	0	10,977	7,438	10,271	144,672

Table 5: SoAR of alternative designs, M=10K, ϕ=ρ=100.

- Store profile images that cannot be inlined in the file system: This improves SoAR of SQL-X 40 folds. Note that thumbnail images must be inlined and stored with the record representing a member. Otherwise, SoAR of SQL-X drops to zero.

- Represent simple analytics using aggregate functions as attribute values and maintain them up to date using additional software (either at the application level or as triggers).

A negative finding is that materialized views slow down the response time of write actions so dramatically that they can no longer be considered interactive.

With MongoDB, a key finding is to store the thumbnail image of a member as an array of bytes in the member's JSON object. This results in a SoAR of seven thousand actions per second. If thumbnail images are stored in either MongoDB's GridFS or the file system of the operating system, SoAR of MongoDB diminishes to zero.

With both MongoDB and SQL-X, one may extend the data store with a cache to look up query results instead of processing queries to compute results. We investigated both memcached and Ehcache. While both enhance system performance, the improvement is dramatic with Ehcache because it eliminates the overhead of transmitting key-values across the network and deserializing them.

When comparing the best SQL-X and MongoDB physical data designs, SoAR of SQL-X is 2.5 times higher than MongoDB when BG's database is configured with no images. With both systems, the CPU of the server hosting the data store becomes 100% utilized. This suggests SQL-X processes BG's workload more efficiently than MongoDB. When BG's database is configured with 12 KB images, the network (2 Gbps) becomes 100% utilized with both data stores. In this scenario, SoAR of SQL-X is 30% higher than MongoDB. This suggests SQL-X transmits less data than MongoDB when processing BG's workload.

A key feature of MongoDB, memcached, and Ehcache is their ability to horizontally scale to a large number of nodes. VoltDB is an SQL solution that also scales to a large number of nodes [24, 29]. An on-going research effort is to quantify the scalability of these systems using BG. This will explore a host of new physical data designs such as different partitioning strategies, replication, and secondary indexes [29]. It will include an interaction of these design choices with the boosted designs presented in this study.

J Acknowledgments

We thank Mark Callaghan and the anonymous reviewers of CIKM 2013 for their insights and valuable comments.

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *ICDE*, 2005.

[2] C. Amza, G. Soundararajan, and E. Cecchet. Transparent Caching with Strong Consistency in Dynamic Content Web Sites. In *Supercomputing*, ICS '05, pages 264–273, New York, NY, USA, 2005. ACM.

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

[4] L. Backstrom. Anatomy of Facebook, http://www.facebook.com/note.php?note_id=1015038 8519243859, 2011.

[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] S. Barahmand and S. Ghandeharizadeh. Expedited Rating of Data Stores Using Agile Data Loading Techniques. *CIKM*, 2013.

[8] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*. USENIX, October 2010.

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

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

[11] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[12] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), June 1970.

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

[14] D. Crockford. *The Application/JSON Media Type for JavaScript Object Notation (JSON)*. Internet Engineering Task Force (IETF), RFC4627, July 2006.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *VLDB*, 3(1-2), September 2010.

[16] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, D. VanderMeer, K. Ramamritham, and D. Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *VLDB*, pages 667–670, 2001.

[17] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[18] A. Floratou, N. Teletria, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the Elephants Handle the NoSQL Onslaught? In *VLDB*, 2012.

[19] S. Ghandeharizadeh and D. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. *VLDB*, 1990.

[20] 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.

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

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

[23] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 49–60, 1997.

[24] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *VLDB*, 1(2), 2008.

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

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

[27] MongoDB. Class WriteConcern, http://api.mongodb.org/java/2.10.1/com/mongodb/ WriteConcern.html.

[28] 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 *Tenth USENIX Symposium on Networked Systems Design and Implementation*, April 2013.

[29] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, 2012.

[30] 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.

[31] R. Sears, C. V. Ingen, and J. Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem. Technical Report MSR-TR-2006-45, Microsoft Research, 2006.

[32] M. Stonebraker. What Does 'Big Data' Mean? *Communications of the ACM, BLOG@ACM*, September 2012.

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

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

[35] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The Anatomy of the Facebook Social Graph. *CoRR*, abs/1111.4503, 2011.

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