BG: A Benchmark to Evaluate Interactive Social Networking Actions*

Sumita Barahmand, Shahram Ghandeharizadeh

Database Laboratory Technical Report 2012-06

Computer Science Department, USC

Los Angeles, California 90089-0781

{barahman,shahram}@usc.edu

February 14, 2013

Abstract

BG is a benchmark that rates a data store for processing interactive social networking actions using a pre-specified service level agreement, SLA. An example SLA may require 95% of issued requests to observe a response time faster than 100 milliseconds. BG computes two different ratings named *SoAR* and *Socialites*. In addition, it elevates the amount of unpredictable data produced by a data store to a first class metric, including it as a key component of the SLA and quantifying it as a part of the benchmarking process.

One may use BG for a variety of purposes ranging from comparing different data stores with one another, evaluating alternative physical data organization techniques given a data store, quantifying the performance characteristics of a data store in the presence of failures (either CP or AP in CAP theorem), among others. This study illustrates BG's first use case, comparing a document store with an industrial strength relational database management system (RDBMS) deployed either in stand alone mode or augmented with memcached. No one system is superior for all BG actions. However, when considering a mix of actions, the memcached augmented RDBMS produces higher ratings.

A Introduction

Social networking sites such as LinkedIn, Facebook, Twitter (see [34] for a list) are cloud service providers for person-to-person communication. There are different approaches to building these sites ranging from SQL to NoSQL, Cache Augmented SQL [24, 18, 15] (CASQL), graph databases [1] and others. (See [9] for

*A shorter version of this paper appeared in the *biennial Conference on Innovative Data Systems Research*, CIDR'13, Asilomar, California, January 2013.

a survey.) Some provide a tabular representation of data while others offer alternative data models that scale out [10]. Some may sacrifice strict ACID [17] properties and opt for BASE [9] to enhance performance. Independent of a qualitative discussion of these approaches and their merits, a key question is how do these systems compare with one another quantitatively. BG is a benchmark designed to answer this question for interactive social networking actions that either read or update a very small amount of the entire dataset [30, 14]. In addition to traditional metrics such as response time and throughput, BG quantifies the amount of *unpredictable* data produced by a solution. This metric refers to either stale, inconsistent, or invalid data produced by a data store, see Section F.

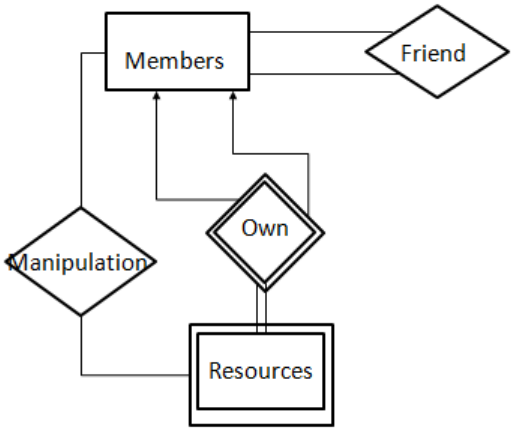
BG emphasizes interactive actions of a social networking application such as browse a profile, generate a friend request and accept one, and (not so sociable) practices such as thaw a friendship and reject a friend request. Table 1 shows the different actions and their overlap with several popular social networking sites.

BG's database consists of a fixed number of *members* with a registered profile. Its workload generator implements a closed simulation model with a fixed number of threads T. Each thread emulates a sequence of members performing a social action shown in Table 1. At any instance in time, an emulated member who is actively engaged in a social action is called a *socialite.* While a database may consist of millions of members, at most T simultaneous socialites issue requests with BG's workload generator.

One may use BG to compute either a Social Action Rating (SoAR) or a Socialites rating of a data store given a prespecified service level agreement (SLA). An SLA requires at least α percentage of requests to observe a response time equal to or faster than β with the amount of unpredictable data less than τ for some fixed duration of time Δ. For example, an SLA might require 95% (α=95%) of actions to be performed faster than 100 msec (β=0.1 second) with no more than 0.1% (τ=0.1%) unpredictable data for 1 hour (Δ=3600 seconds). SoAR pertains to the highest throughput (actions per second) of a data store that satisfies this SLA. Socialites is the highest number of threads (largest T value) that satisfies this SLA, see Figure 12.a.

BG quantifies the amount of unpredictable data in a system at the granularity of a social action. It does so by considering concurrent socialites and all possible race conditions to compute a range of values for a retrieved data item, e.g., number of friends for a member's profile. If a data store fetches a value that falls outside this range then it has produced unpredictable data.

BG is inspired by prior benchmarks that evaluate cloud services such as YCSB [11] and YCSB++ [23], e-commerce sites [2], and object-oriented [8] and transaction processing systems [16]. Its contributions are two folds. First, it emphasizes interactive social actions that retrieve a small amount of data. Second, it promotes the amount of unpredictable data produced by a solution as a first class metric for comparing different data stores with one another. The value of this metric is impacted by BG's knobs such as the exponent of the Zipfian distribution used to generate referenced members and the inter-arrival time between two socialites emulated by a thread. These knobs enable one to approximate a realistic use case of an application to quantify unpredictable data practically.



1.a Conceptual data model of BG's database.

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

```
Resources:{
        "rid" : ""
        "creatorid" : ""
        "walluserid": ""
        "type" : ""
        "body": ""
        "doc": ""
        "manipulation":{
                "mid": ""
                "modifierid": ""
                "type": ""
                "content": ""
                "timestamp": ""
        }
    }
```

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

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

1.b JSON-Like data model of BG's database.

```
Members(userid,username,pw,lastname,gender,dob,jdate,ldate,address,email,  profileImage, thumbnail)
Friends(userid1,userid2,status)
Resources(rid, creatorid, wallUserid, body, doc)
Manipulation(mid, modifierid, rid, resourcecreatorid, timestamp,type,content)
```

1.c Relational data model of BG's database.

Figure 1: Conceptual and logical data models of BG's database.

BG might be used for a variety of purposes ranging from comparing different data stores with one another to characterizing the performance of a data store under different settings: Normal mode of operation with alternative physical data organizations [6], in the presence of a failure (either CP or AP in CAP [21]), and when exercising the elasticity of a data store by adding or removing nodes incrementally.

This paper illustrates BG's use case by comparing the following 3 different data stores with one another:

- SQL-X: An industrial strength relational database management system with ACID properties and a SQL query interface. Due to licensing restrictions, we cannot reveal its identity and name it SQL-X.
- MongoDB version 2.0.6, a document store for storage and retrieval of JavaScript Object Notations, JSON. MongoDB is a representative NoSQL system. See [9] for a survey.
- CASQL: SQL-X extended with memcached server version 1.4.2 (64 bit). BG employs Whalin memcached client versions 2.5.1 to communicate with the memcached server. We configured the Whalin client to compress (uncompress) key-value pairs when storing (retrieving) them in (from) memcached.

The rest of this paper is organized as follows. Section B presents the conceptual data model of BG and its logical design for relational and JSON-like data models. Social networking actions that constitute BG are detailed in Section C. Section D enumerates BG's sessions that consist of a sequence of actions. Section E describes limitations of a centralized single node benchmarking framework and presents a parallel

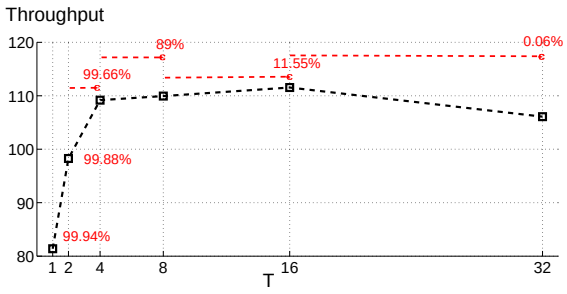


Figure 2: Throughput of SQL-X as a function of T with View Profile action, 12 KB profile image size, β=100 msec, τ=0%, θ=0.27. Confidence (α) is shown in red.

implementation of BG using a shared-nothing architecture. In Section F, we describe how BG quantifies the amount of unpredictable data produced by a data store. A heuristic search technique to rate data stores is presented in Section G. Section H presents related work. Brief conclusions along with our future research directions are detailed in Section I.

B Conceptual Data Model and Performance Metrics

Figure 1.a shows the ER diagram of BG's database. The Member entity set contains those users with a registered profile. It consists of a unique identifier and a fixed number of string attributes whose length can be adjusted to generate different member record sizes. In addition, each member may have either zero or 2 images. With the latter, one is a thumbnail and the second is a higher resolution image. Typically, thumbnails are displayed when listing friends of a member and the higher resolution image is displayed when a member visits a profile. Thumbnail images are typically small (in the order of KBs) and their use (instead of larger images, in the order of tens and hundreds of KBs and MBs) has a dramatic impact on system performance, see discussions of Section C.1.

A member may either extend an invitation to or be friends with another member. Both are captured using the "Friend" relationship set. An attribute of this relationship set (not shown) differentiates between invitations and friendships.

A resource may pertain to an image, a posted question, a technical manuscript, etc. These entities are captured in one set named "Resources". In order for a resource to exist, it must be "Owned" by a member, a binary relationship between a member and a resource. A member may post a resource, say an image, on the profile of another member, represented as a ternary relationship between two members and a resource. (In this relationship, the two members might be the same member where the member is posting the resource on her own profile.) A member (either the owner or another) may comment on a resource (not shown). A member may restrict the ability to comment on a resource only to her friends. This is implemented using

the "Manipulation" relationship set.

Figures 1.b and 1.c show the logical design of the ER diagram with both MongoDB's JSON-like and relational data models. An experimentalist builds a database by specifying the number of members (M) in the social network, number of friends per member (ϕ), and resources per member (ρ). Some of the relationships might be generated either uniformly or using a Zipfian distribution. For example, one may use a Zipfian distribution with exponent (θ) 0.27 to assign 80% of friendships ($M \times \phi$) to 20% of members.

One may specify BG workloads at the granularity of an action, a session, or a mix of these two possibilities. A session is a sequence of actions with ϵ think time between actions and ψ inter-arrival time between sessions. Table 1 shows BG's list of actions and its compatibility with several social networking sites. Section D enumerates the different sessions supported by BG. One may extend BG with new sessions consisting of an arbitrary mix of actions.

Similar to YCSB [11], BG exposes both its schema and its actions to be implemented by a developer. Thus, a developer may target an arbitrary data store, specify its physical data model for the conceptual data model of Figure 1.a, provide an implementation of the actions of Table 1, and run BG to evaluate the target data store. As detailed in Section E, these functionalities are divided between a Coordinator, named BGCoord, and N slave processes, named BGClients.

When generating a workload, BG is by default set to prevent two simultaneous threads from emulating the same member concurrently. This is to model real life user interactions as closely as possible. An experimentalist may eliminate this assumption by modifying a setting of BG.

BG rates a system with *at least α percentage of actions observing a response time equal to or less than β with at most τ percentage of requests observing unpredictable data in Δ time units.* For example, an experimentalist may specify a workload with the requirement that at least 95% (α=0.95) of actions to observe a response time equal to or less than 100 msec (β=0.1 second) with at most 0.1% (τ=0.001) of requests observing unpredictable data for 1 hour (Δ=3600 seconds). With such a criterion, BG computes two possible ratings for a system:

1. SoAR: Highest number of completed actions per second that satisfy the specified criterion. Given several systems, the one with the highest SoAR is desirable.

2. Socialites: Highest number of simultaneous threads that satisfy the specified SLA. It quantifies the multi-threading capability of the data store and whether it suffers from limitations such as the convoy phenomena [7] that diminishes its throughput rating with a large number of simultaneous requests. Given several systems, the one with the highest Socialites rating is more desirable.

These ratings are not a simple function of the average service time ($\bar{S}$) of a workload. The specified confidence (α), the tolerable response time (β), and the amount of unpredictable data (τ) observed from a system impacts its SoAR and Socialites rating. To illustrate, Figure 2 shows the throughput of SQL-X as a function

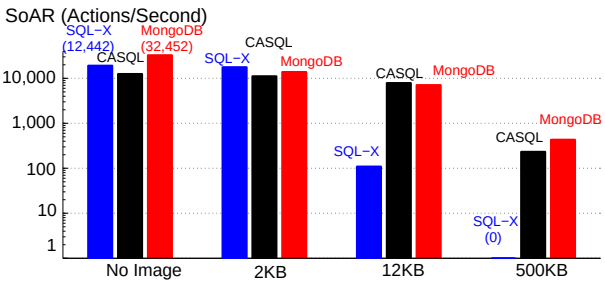


Figure 3: SoAR of 3 different systems with view profile and different profile image sizes, M=10,000, β=100 msec, α=95%, ϵ=ψ=0, θ=0.27.

of the number of threads T for a read only action, τ=0. We show the different confidence values for β=0.1 second. As we increase the number of threads, the throughput of the system increases. Beyond 4 threads, a queue of requests forms causing an increase in system response time. This is reflected in a lower α value. With 32 threads, almost all (99.94%) requests observe a response time higher than 100 msec.

C Actions

This section provides a specification of BG's social actions, see Table 1. We present their implementation using SQL-X, MongoDB, and CASQL; see Section A for a description. Subsequently, Section C.4 presents 3 workloads consisting of a mix of actions.

For the first two actions, we present SoAR numbers using the following SLA: 95% of requests observe a response time equal to or faster than 100 msec with the amount of stale data less than 0.1%. Member ids are generated using a Zipfian distribution with exponent 0.27. Reported numbers were obtained from 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 gigabit networking card. Even though these PCs have the same exact model and were purchased at the same time, there is some variation in their performance. To prevent this from polluting our results, the same one node hosts the different data stores for all ratings. This node hosts both memcached and SQL-X to realize CASQL. Either all or a subset of the remaining 5 nodes are used as BGClients to generate requests 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 becomes fully utilized. When SoAR is zero, this means the data store failed to satisfy the SLA with one single threaded BGClient issuing requests, N=T=1.

Action	Facebook	Google+	Twitter	LinkedIn	YouTube	FourSquare	Delicious	Academia.edu	Reddit.com
View Profile (VP)	✓	✓	✓	✓	✓	✓	✓	✓	✓
List Friends (LF)	✓	✓	✓	✓	✓	✓	✓	✓	✗
View Friend Requests (VFR)	✓	✗	✗	✓	✗	✓	✗	✗	✗
Invite Friend (IF)	✓	Add to Circle	Follow	✓	Subscribe	✓	Follow	Follow	Follow
Accept Friend Request (AFR)	✓	✗	✗	✓	✗	✓	✗	✗	✗
Reject Friend Request (RFR)	✓	✗	✗	✓	✗	✓	✗	✗	✗
Thaw Friendship (TF)	✓	Remove from Circle	Unfollow	✓	Unsubscribe	✓	Unfollow	Unfollow	Unfollow
View Top-K Resources (VTR)	✓	✓	✓	✓	✓	✓	✓	✓	✓
View Comments on a Resource (VCR)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Post Comment on a Resource (PCR)	✓	✓	Reply to a tweet	Recommend a colleague's work	Post Comment on a video	Add Comment on a check-in	Add tag to a link	Post answer to a question	✓
Delete Comment from a Resource (DCR)	✓	✓	Delete the reply for a tweet	Withdraw recomm-endation	Remove comment on a video	Delete comment on a check-in	Remove tag from a link	Delete answer to a question	✓

Table 1: Socialite actions and their compatibility with several social networking sites.

C.1 View Profile, VP

View Profile (VP) emulates a socialite visiting the profile of either herself or another member. Its input include the socialite's id and the id of the referenced member, U_r. BG generates these two ids using a random number conditioned using the Zipfian distribution of access with a pre-specified[1] exponent (specified by the experimentalist who is benchmarking a system). Socialite's id may equal the id of U_r, emulating a socialite referencing her own profile. Its output is the profile information of U_r. This includes U_r's attributes and the following two aggregate information: U_r's number of friends, U_r's number of resources (e.g., images). If the socialite is referencing her own profile (socialite id equals U_r's id) then VP retrieves a third aggregate information: U_r's number of pending friend invitations.

VP retrieves all attributes of U_r except U_r's thumbnail image. This includes U_r's profile image assuming the database is created with images, see Section B. An implementation of VP with the different data stores is as follows. With MongoDB (SQL-X), it looks up the document (row) corresponding to the specified U_r userid. With MongoDB, VP computes the number of friends and pending invitations by counting the number of elements in pendingFriends and confirmedFriends arrays, respectively. It counts the number of resources posted on U_r's wall by querying the Resources collection using the predicate "walluserid = U_r's userid". With SQL-X, VP issues different aggregate queries. With CASQL, VP constructs two different keys using U_r's userid: self profile when socialite's id equals U_r's userid and browse profile when socialite's id does not equal U_r's userid. Depending on whether socialite's id equals U_r's userid, it looks up the appropriate key in memcached. If a value is returned, it proceeds to uncompress and deserialize it, producing it as its output.

[1]The exponent θ used in this section is 0.27.

Otherwise, it performs the same set of steps as those with SQL-X, computes the final output, serializes it, and stores it in memcached as the value associated with the appropriate key. This key-value pair is used by future references.

Presence of a profile image and its size impact SoAR of different data stores for VP dramatically [26, 6]. Figure 3 shows the performance of three different systems for a BG database consisting of no-images, and a 2 KB thumbnail image with different sizes for the profile image: 2 KB, 12 KB, and 500 KB. These settings constitute the x-axis of Figure 3. The y-axis reports SoAR of different systems.

With no images, MongoDB provides the best performance, outperforming both SQL-X and CASQL by almost a factor of two. With 12 KB images, SoAR of SQL-X drops dramatically from thousands to hundreds[2]. With 500 KB image sizes, SQL-X cannot perform even one VP action per second that satisfies the 100 msec response time (with 1 thread), producing a SoAR of zero. SoAR of MongoDB and CASQL also decrease as a function of larger image size because they must transmit a larger amount of data to the BGClient using the network. However, their decrease is not as dramatic as SQL-X.

CASQL outperforms SQL-X because these experiments are run with a warm up phase that issues 500,000 requests to populate memcached with key-value pairs pertaining to different member profiles. Most requests are serviced using memcached (instead of SQL-X). While this does not payoff[3] with small images, with 12 KB and 500 KB image sizes, it does enhance performance of SQL-X considerably.

C.2 List Friends, LF

List Friends (LF) emulates a socialite viewing either her list of friends or another member's list of friends. This action retrieves the profile information of each friend. In the presence of images, it retrieves only the thumbnail image of each friend. At database creation time, BG empowers an experimentalist to configure a database with a fixed number of friends per member (ϕ). Figure 4 shows SoAR of the alternative data stores for LF as a function of a different number of friends (ϕ) per member. (The median Facebook friend count is 100 [31, 4].) A larger ϕ value lowers the rating of all data stores. Overall, CASQL provides the best overall performance with 50 and 100 friends per member. Even though MongoDB performs no joins, its SoAR is zero for all the examined ϕ values. Below, we describe implementation details of each system.

SQL-X must join the Friends table with the Members table (see Figure 1.c) to compute the socialite's list of friends. We assume the friendship relationship between two members is represented as 1 record[4] in Friends table, see Figure 1.c. CASQL caches the final results of the LF action and enhances SoAR of SQL-X by less than 10% with ϕ values of 50 and 100. With ϕ=1000, SQL-X slows down considerably

[2]We use SQL-X with the physical data design shown in Figure 1.c. This design can be enhanced to improve performance of SQL-X by ten folds or more. See [6] for details.

[3]There are several suggested optimizations to the source code of memcached to improve its performance [25, 3]. Their evaluation is a digression from our main focus. Instead, we focus on the standard open source version 2.5.1 [22].

[4]See [6] for a discussion of representing friendship as 2 records and its impact on SoAR.

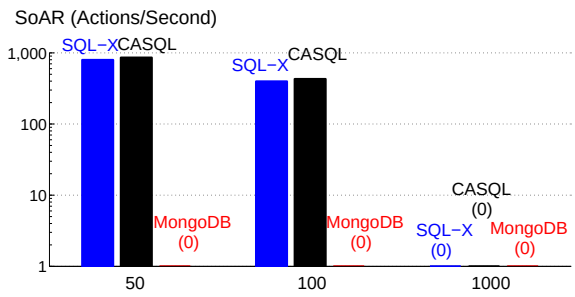


Figure 4: SoAR of List Friends with 3 different data stores as a function of number of friends per member (ϕ), M=10,000, β=100 msec, α=95%, ϵ=ψ=0, θ=0.27.

and can no longer satisfy the 100 msec response time requirement. The CASQL alternative is also unable to meet this SLA because each key-value is larger than 1 MB, the maximum key-value size supported by memcached. This renders memcached idle, redirecting all requests issued by CASQL to SQL-X, producing zero for system SoAR. One may modify memcached to support key-value pairs larger than 2 MB (ϕ=1000 and each thumbnail is 2 KB) to realize an enhanced SoAR with CASQL.

With MongoDB, an implementation of LF may retrieve the confirmed friends either one document at a time or as a set of documents. With both approaches, the BG client starts by retrieving the confirmedFriends array of the referenced member, see Figure 1.b. With one document at time, the client processes the array and for each userid, retrieves the profile document of that member. With a set at a time, the client provides MongoDB with the array of userids to retrieve a set containing their profile documents. These two alternatives cannot satisfy the 100 msec SLA requirement, producing a SoAR of zero for different values of ϕ. With fewer friends per member, say 10, SoAR of MongoDB is 6 actions per second.

C.3 Other actions

View Friend Requests, VFR: This action retrieves a socialite's pending friend request. It retrieves the profile information of each member extending a friend request invitation along with her thumbnail (assuming the database is configured with images). Both the implementation and the behavior of SQL-X, MongoDB, CASQL with VFR are similar to the discussion of LF.

Invite Friend, IV: This action enables a socialite to invite another member, say A, of the social network to become her friend. With MongoDB, this action inserts the socialite's userid into A's array of pendingFriends, see Figure 1.b. With both SQL-X and CASQL, this operation inserts a row in the Friends table with status set to "pending", see Figure 1.c. CASQL invalidates the memcached key-value pairs corresponding to A's self profile (with a count of pending invitations) and A's list of pending invitation. A subsequent VP invocation that references these key-value pairs observes a cache miss, computes the latest key-value pairs, and inserts them in the cache.

Database parameters	
M	Number of members in the database.
ϕ	Number of friends per member.
ρ	Number of resources per member.
Workload parameters	
O	Total number of sessions emulated by the benchmark.
ϵ	Think time between social actions constituting a session.
ψ	Inter-arrival time between users emulated by a thread.
θ	Exponent of the Zipfian distribution.
Service Level Agreement (SLA) parameters	
α	Percentage of requests with response time $\leq \beta$.
β	Max response time observed by α requests.
τ	Max % of requests that observe unpredictable data.
Δ	Min length of time the system must satisfy the SLA.
Environmental parameters	
N	Number of BGClients.
T	Number of threads.

Table 2: BG's parameters and their definitions.

Accept Friend Request, AFR: A socialite A uses this action to accept a pending friend request from member B of the social network. With MongoDB, this action inserts (a) A's userid in B's array of confirmedFriends, and (b) B's userid in A's arrays of confirmedFriends, see Figure 1.b. Moreover, it removes B's userid from A's array of pendingFriends. With both SQL-X and CASQL, this operation updates the "status" attribute value of the row corresponding to B's friend request to A to "confirmed", see Figure 1.c. CASQL invalidates the memcached key-value pairs corresponding to self profiles of members A and B, profiles of members A and B as visited by others, list of friends for members A and B, list of pending invitations for member A.

Reject Friend Request, RFR: A socialite uses RFR to reject a pending friend request from a member B. BG assumes the system does not notify Member B of this event. With MongoDB, we implement RFR by simply removing B's userid from the socialite's array of pendingFriends, see Figure 1.b. With both SQL-X and CASQL, RFR deletes the friend request row corresponding to B's friend request to the socialite, see Figure 1.c. CASQL invalidates the key-value pairs corresponding to socialite's self profile and pending friend invitations from memcached.

Thaw Friendship, TF: This action enables a socialite A to remove a member B as a friend. With MongoDB, TF removes A's userid from B's array of confirmedFriends and vice versa, see Figure 1.b. With both SQL-X and CASQL, TF deletes the row corresponding to the friendship of user A and B (with status equal to "confirmed") from Friends table, see Figure 1.c. CASQL invalidates the key-value pairs corresponding to the list of friends for users A and B, self profile of users A and B, and profiles of users A and B as visited by other users (because their number of friends has changed).

View Top-K Resources, VTR: When BG populates a database, it requires each member to create a fixed

number of resources. Each resource is posted on the wall of a randomly chosen member, including one self's wall. View Top-K Resources (VTR) enables a socialite to retrieve and display her top k resources posted on her wall. Both the value of k and the definition of "top" are configurable. Top may correspond to those resources with the highest number of "likes", date of last view/comment (recency), or simply its ID. At the time of this writing, BG supports the last one. With MongoDB, VTR queries the Resources collection in a sorted order to retrieve top k resources owned by the socialite. With SQL-X and CASQL, VTR queries the Resources table and uses top k ordered using their rid. CASQL constructs a unique key using the action and socialite userid, serializes the results as a value, and inserts the key-value pair in memcached for future reference.

View Comments on Resource, VCR: A socialite displays the comments posted on a resource with a unique rid using VCR action. BG generates rids for this action by randomly selecting a resource owned by a member (selected using a zipfian distribution). With MongoDB, we looked into two different implementations. The first implementation supported the schema shown in Figure 1.b where the comments for every resource are stored within the manipulation array attribute for that resource. With this implementation, VCR retrieves the elements of manipulation array of the referenced resource, see Figure 1.b. The second implementation creates a separate collection for the comments named Manipulations, see Figure 5. With this implementation VCR queries the Manipulations collection for all those documents whose rid equals the referenced resourceid. (A comparison of these alternative physical data designs is a future research direction.) With SQL-X, VCR employs the specified identifier of a resource to query the Manipulation table and retrieve all attributes of the qualifying rows, see Figure 1.c. CASQL constructs a unique key using rid to look up the cache for a value. If it observes a miss, it invokes the procedure for SQL-X to construct a value. The resulting key-value pair is stored in memcached for future reference.

Post Comment on a Resource, PCR: A socialite uses PCR to comment on a resource with a unique id. BG generates rids by randomly selecting a resource owned by a member selected using a Zipfian distribution. It generates a random array of characters as the comment for a user. The number of characters is a configurable parameter. With MongoDB, PCR is implemented by either generating an element for the manipulation array attribute of the selected resource, see Figure 1.b or generating a document, setting its rid to the unique identifier of the referenced resource and inserting it into the Manipulations collection, see Figure 5. With SQL-X and CASQL, PCR inserts a row in the Manipulation table. CASQL invalidates the key-value pair corresponding to comments on the specified resource id.

Delete Comment from a Resource, DCR: This action enables a socialite to delete a unique comment posted on one of her owned resources chosen randomly. With MongoDB, an implementation of DCR either removes the element corresponding to the comment from the manipulation array attribute of the identified resource, see Figure 1.b or removes the document corresponding to the comment posted on the referenced resource from the Manipulations collection, see Figure 5. With SQL-X and CASQL, DCR deletes a row of

Figure 5: An alternative JSON-Like data model of BG's database.

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 Friend 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	49.9%	49%	45%
View Comments on a Resource, VCR	Read	0%	0%	0%
Post Comment on a Resource, PCR	Write	0%	0%	0%
Delete Comment from a Resource, DCR	Write	0%	0%	0%

Table 3: Three mixes of social networking actions.

the Manipulation table. CASQL invalidates the key-value pair corresponding to comments on the specified resource id.

C.4 Mix of Actions

One may evaluate a data store by specifying a mix of actions. Three different mixes are shown in Table 3. To simplify discussion, actions are categorized into read and write. These mixes exercise write actions that impact the friendship relationship of two members of the social network, invalidating the cached CASQL result of read actions such as View Profile, List Friends, and View Friend Requests. Each mix consists of a different percentage of write actions, ranging from very low (0.1%) to high (10%).

We use MongoDB with its strict write concern which requires each write to wait for a response from the server [19]. Without this option, MongoDB produces stale data (less than 0.01%).

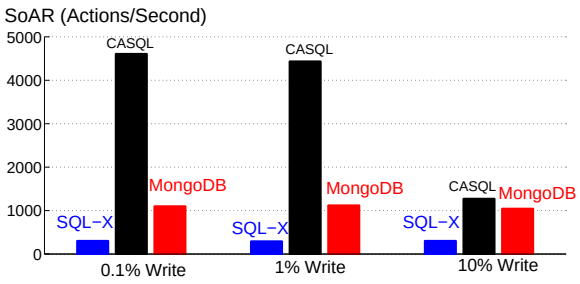


Figure 6: SoAR for 3 mixes of read and write actions, M=10,000, 12 KB image size, ϕ=100, ρ=β=100 msec, α=95%, τ=0.01%, ϵ=ψ=0, θ=0.27.

Figure 6 shows SoAR of the different systems with the 3 mixes for a database with 10,000 members and 100 friends per member. MongoDB outperforms SQL-X for the different mixes by almost a factor of 3. The CASQL is sensitive to the percentage of write actions as they invalidate cached key-value pairs, causing read actions to be processed by the RDBMS. With a very low (0.1%) write mix, CASQL outperform MongoDB by more than a factor of 3. With a high percentage of write actions, SoAR of CASQL is slightly higher than MongoDB.

The observed trends with SQL-X and MongoDB change depending on the mix of VP and LF actions. In Figure 6, MongoDB outperforms SQL-X because the frequency of VP is significantly higher than LF. If this was switched such that LF is more frequent than VP, then SQL-X would outperform MongoDB. A system evaluator should decide the mix of actions based on the characteristics of a target application.

D Sessions

A *session* is a sequence of actions performed by a socialite. BG employs the Zipfian distribution to select one of the M members to be the socialite. The selected session is based on a probability computed using the frequencies specified for the different sessions in a configuration file. A key conceptual difference between actions and sessions is the concept of think time, ϵ. This is the delay between the different actions of a session emulated on behalf of a socialite. BG supports the concept of inter-arrival time (ψ) between socialites emulated by a thread with both actions and sessions.

Currently, BG supports 8 sessions. The first session is the starting point for the remaining 7 sessions. These sessions are as follows:

1. ViewSelfProfileSession, $\{$VP(m_i),VTR(m_i)$\}$: A Member m_i visits her profile page to view her profile image (if available), number of pending friend requests, number of confirmed friends, and number of resources posted on her wall. Next, the member lists her top k resources.

2. ViewFrdProfileSession, $\{$VP(m_i),VTR(m_i),LF(m_i),VP(m_j),VTR(m_j) $\mid$ $m_j \in$ LF(m_i))$\}$: After view-

ing self profile and top k resources, Member m_i lists her friends and picks one friend randomly, m_j. Next, m_i views m_j's profile and m_j's top k resources. If m_i has no friends, the session terminates without performing the two actions on m_j.

3. PostCmtOnResSession, {VP(m_i),VTR(m_i),VP(m_{rand}),VTR(m_{rand}),VCR(r_{rand}) | $r_{rand} \in$ VTR(m_{rand}), PCR(r_{rand}),VCR(r_{rand}) }: After viewing self profile and top k resources, Member m_i views the profile of a randomly chosen member m_{rand}, lists m_{rand}'s top k resources, and picks one resource randomly, r_{rand}. If there are no resources, the rest of actions are not performed. Otherwise, m_i views comments posted on r_{rand}, posts a comment on r_{rand} and views all comments on r_{rand} a second time.

4. DeleteCmtOnResSession, {VP(m_i),VTR(m_i),VCR(r_{rand}),DCR(r_{rand}) | $r_{rand} \in$VTR(m_i),VCR(r_{rand}) }: After viewing self profile and top k resources, Member m_i views comments on one of her own randomly selected resource, r_{rand}, deletes a comment from this resource (assuming it exists), and views comments on r_{rand} again. If r_{rand} has no comments, she skips the remaining actions and the session terminates.

5. InviteFrdSession, {VP(m_i),VTR(m_i),LF(m_i),IF(m_j),VFR(m_j) | $m_j \cap$ LF(m_i) = $\emptyset$}: After viewing self profile and top k resources, Member m_i lists her friends, and selects a random member m_j who has no pending or confirmed relationship[5] with m_i. (If all members of the database are m_i's friend then the remaining two actions are not performed.) She invites m_j to be friends and concludes by listing her own pending friend requests.

6. AcceptFrdReqSession, {VP(m_i),VTR(m_i),LF(m_i),VFR(m_i),AFR(m_j) | $m_j \in$ VFR(m_i),VFR(m_i),LF(m_i) }: After viewing self profile and top k resources, Member m_i lists her friends and pending friend requests. Next, she picks a pending friend request by member m_j and accepts this friend request (if any). She reviews her friend request a second time and concludes by listing her friends. If m_i has no pending friend requests, she skips the remaining actions and the session terminates.

7. RejectFrdReqSession, {VP(m_i),VTR(m_i),LF(m_i),VFR(m_i),RFR(m_j) | $m_j \in$ VFR(m_i),VFR(m_i),LF(m_i) }: After viewing self profile and top k resources, Member m_i lists her friends and pending friend invitation to select an invitation from member m_j. She rejects friend request from m_j, views her own friend requests and lists her friends a second time. If m_i has no pending friend requests, she skips the remaining actions and the session terminates.

8. ThawFrdshipSession, {VP(m_i),VTR(m_i),LF(m_i),TF(m_j) | $m_j \in$ LF(m_i),LF(m_i) }: After viewing self profile and top k resources, Member m_i lists her friends and select a friend m_j randomly. Next,

[5]Includes friendship, pending invitation from m_i to m_j, and pending invitation from m_j to m_i.

m_i thaws friendship with m_j. This session concludes with m_i listing her friends. If m_i has no friends, she skips the remaining actions and the session terminates.

Note the dependency between the value of m_i and m_j with ViewFrdProfileSession, InviteFrdSession, RejectFrdReqSession, and ThawFrdshipSession. For example, with ViewFrdProfileSession, m_j must be a friend of m_i. If m_i has no friends, the session terminates without performing the remaining actions.

Moreover, some of the sessions cannot be implemented by simply using the state of the database in the data store because multiple concurrent threads may race with one another to change the database state simultaneously. BG is not previewed to how the data store serializes the concurrent actions and whether the data store implements the concept of transactions (ACID). Hence, to detect unpredictable data accurately, BG maintains in-memory data structures that track the state of the database and employs them to decide the identity of entities manipulated by the actions and sessions, see Section F for details. As an example, consider the DeleteCmtOnResSession. Conceptually, it enables a socialite to delete a comment created on one of the resources posted on her wall. Delete Comment from a Resource, DCR, action of this session consumes a randomly generated resource, r_{rand}. BG does not generate r_{rand} using the state of the database. Instead, it maintains in-memory data structures that track the state of the database to generate r_{rand}. It uses semaphores to manage the integrity of these data structures and prevents multiple socialites from deleting the same comment on a resource simultaneously. In essence, r_{rand} is a member of VTR(m_j) and guaranteed to exist prior to invocation of DCR.

BG is an extendible framework and one may specify sessions consisting of a different mix of actions.

E Parallelism

Today's data stores use techniques that may fully utilize resources (CPU and network bandwidth) of a single node benchmarking framework. For example, Whalin client for memcached (CASQL) is configured to compress key-value pairs prior to inserting them in the cache. It decompresses key-value pairs upon their retrieval to provide the uncompressed version to its caller, i.e., BG. Use of compression minimizes CASQL's network transmissions and enhances its cache hit rate by reducing the size of key-value pairs with a limited cache space. It also causes the CPU of the node executing BG to become 100% utilized for certain workloads. This is undesirable because the resulting SoAR reflects the capabilities of the benchmarking framework instead of the data store.

To address this issue, BG implements a scalable benchmarking framework using a shared-nothing architecture. Its software components are as follows:

1. A coordinator, BGCoord, computes SoAR and Socialites rating of a data store by implementing both an exhaustive and a heuristic search technique. Its inputs are the SLA specifications and parameters

Figure 7: BG's Visualization Deck.

of an experiment, see Table 2. It computes the fraction of workload that should be issued by each worker process, named BGClient, and communicates it with that BGClient. BGCoord monitors the progress of each BGClient periodically, aggregates their current response time and throughput, and reports these metrics to BG's visualization deck for display, see Item 3. Once all BGClients terminate, BGCoord aggregates the final results for display by BG's visualization deck.

2. A BGClient is slave to BGCoord and may perform three possible tasks. First, create a database. Second, generate a workload for the data store that is consistent with the BGCoord specifications. Third, compute the amount of unpredictable data produced by the data store. It transmits key metrics except for the amount of unpredictable data to BGCoord periodically. At the end of the experiment, it computes all metrics and transmits them to BGCoord.

3. BG visualization deck enables a user to specify parameter settings for BGCoord, initiate rating of a data store, and monitor the rating process, see Figure 7.

Once BGCoord activates N BGClients, each BGClient generates its workload independently to enable the benchmarking framework to scale to a large number of nodes. We realize this by constructing the physical database of Section B to consist of N logical self-contained fragments. Each fragment consists of a unique collection of members, resources, and their relationships. BG can realize this because it generates the benchmark database. BGCoord assigns a logical fragment to one BGClient to generate its workload. This partitioning enables BG to implement uniqueness of concurrent socialites, i.e., the same member does not manipulate the database simultaneously. Note that construction of logical fragments has no impact on the size of the physical database and its parameter settings such as number of friendships.

With BG, an experiment may specify a Zipfian distribution with a fixed exponent and vary the number of BGClients, value of N. BGClients implement a decentralized Zipfian, D-Zipfian [5], that produces the

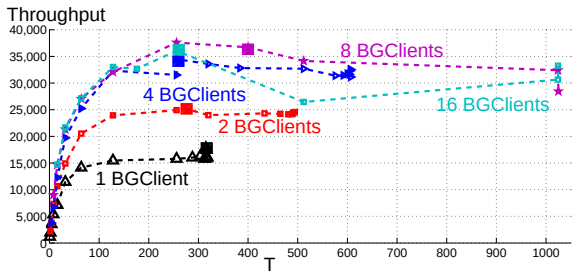


Figure 8: MongoDB's Throughput as a function of T with view profile (VP) action and different number of BGClients, N. M=10,000, No image, β=100 msec, α=95%, ϵ=ψ=0, θ=0.27.

same distribution of references with different values of N. This enables us to compare results obtained with different number of BGClients with one another. We implement D-Zipfian to incorporate heterogeneity of nodes (hosting BGClients) where one node produces requests at a rate faster than the other nodes. D-Zipfian assigns more load to the fastest node by assigning a larger logical fragment to it and requiring it to produce more requests. Hence, the N BGClients complete issuing requests at approximately the same time. For details of D-Zipfian, see [5].

Figure 8 shows the throughput of MongoDB as a function of socialites. Presented results pertain to different number of BGClients performing view profile (VP) action with D-Zipfian and exponent 0.27. The Socialites rating is the length of each curve along the x-axis. While it is 317 with 1 BGClient, it increases 3.2 folds to 1024 with 8 (16) BGClients. A solid rectangular box denotes the SoAR rating with a given number of BGClients. It also increases as a function of N; from 15,800 with 1 BGClient to 33,200 with 16 BGClients. With 1 BGClient, client component of MongoDB is limiting the observed ratings. We know it is not the hardware platform because we can run multiple BGClients on one node to observe higher ratings. Four physical nodes are used in the experiments of Figure 8. Both SoAR and Socialites rating remain unchanged from 8 to 16 BGClients. D-Zipfian ensures the same distribution of requests is generated with 1 to 16 BGClients.

F Unpredictable Data

Unpredictable data is either stale, inconsistent, or simply invalid data produced by a data store. For example, the design of a CASQL may incur dirty reads [18] or suffer from race conditions that leave the cache and the database in an inconsistent state [15], a data store may employ an eventual consistency [32, 29] technique that produces either stale or inconsistent data for some time [23], and others. The requirements of an application dictate whether these techniques are appropriate or not. A key question is how much unpredictable data is produced by a data store for interactive social networking actions. This section describes how BG quantifies an answer to this question.

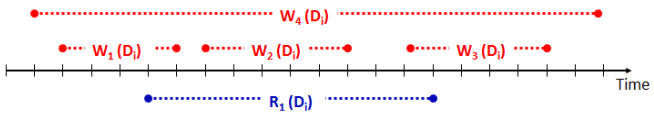


Figure 9: A write of D_i may overlap the read of D_i in four possible ways.

Conceptually, BG is aware of the initial state of a data item in the database (by creating them using deterministic functions) and the change of value applied by each update operation. There is a finite number of ways for a read of a data item to overlap with concurrent actions that write it. BG enumerates these to compute a range of acceptable values that should be observed by the read operation. If a data store produces a different value then it has produced unpredictable data. This process is named *validation* and its details are as follows.

BG implements validation in an off-line manner after it rates a data store, preventing it from exhausting the resources of a BGClient. During the benchmarking phase, each thread of a BGClient invokes an action that generates one log record. There are two types of log records, a read and a write log record corresponding to either a read or a write of a data item. A log record consists of a unique identifier, the action that produced it, the data item referenced by the action, its socialite session id, and start and end time stamp of the action. The read log record contains its observed value from the data store. The write log contains either the new value (named *Absolute Write Log*, AWL, records) or change (named *Delta Write Log*, DWL, records) to existing value of its referenced data item.

The start and end time stamps of each log record identifies the duration of an action that either read or wrote a data item. They enable BG to detect the 4 possible ways that a write operation may overlap a read operation, see Figure 9. During validation phase, for each read log record that references data item D_i, BG enumerates all completed write log records that reference D_i and overlap the read log record, computing a range of possible values for this data item. If the read log record contains a value outside of this range then its corresponding action has observed unpredictable data. To elaborate, BG uses the set of q DWL records to compute all serializable schedules that a data store may generate. The theoretical upper bound on the number of schedules is $q!$. However, BG computes fewer schedules when two or more DWL records do not overlap: The end time stamp of one is prior to the start of the second. This produces an accurate range of possible values for the read operation. This is best illustrated with an example. Consider the four log records of Table 10 where 3 DWL records overlap 1 read log record. Theoretically, there is a maximum of six (3!) possible ways for the updates to overlap one another. However, the actual number of possibilities is two, {{DWL1,DWL2,DWL3}, {DWL2,DWL1,DWL3}}, because DWL3 has a start time stamp after both DWL1 and DWL2. Thus, assuming the value of D_1 is zero at time zero, acceptable values for the read are

Operation id	Type	Data item	Start	End	Value
Read1	Read	D_1	0	10	3
DWL1	Write	D_1	1	3	-1
DWL2	Write	D_1	2	4	1
DWL3	Write	D_1	5	6	2

Figure 10: Example log records.

{-1, 0, 1, 2}, flagging the observed value 3 as unpredictable. If one had assumed 3! possible schedules incorrectly then value 3 would have appeared in the acceptable set, confirming Read1 as valid incorrectly.

Log records produced by one BGClient are independent of those produced by the remaining $N - 1$ BGClients because BGCoord partitions members and resources among the BGClients logically. Thus, there are no conflicts across BGClients and each BGClient may perform validation independently to compute number of actions (sessions) that observe unpredictable data. BGCoord collects these numbers from all BGClients to compute the overall percentage of actions (sessions) that observed unpredictable data.

Depending on the value of Δ, a BGClient may produce a large number of log records. These records are scattered across multiple files. Currently, there are two centralized implementations of the validation phase using interval-trees [12] as in-memory data structures and a persistent store using a relational database. The latter is more appropriate when the total size of the write log records exceeds the available memory of a BGClient. We also have a preliminary implementation of the validation phase using MapReduce [13] that requires the log files to be scattered across a distributed file system. Below, we describe the centralized implementation of the validation phase.

Both in-memory and persistent implementations of validation are optimized for workloads dominated with actions that read data items [1]. These optimizations are as follows. First, if there are no update log records then there is no need for a validation phase; the validation phase terminates by deleting the read log file(s) and reporting 0% unpredictable reads. Second, write log records are processed first to construct a main memory data structure (independent of interval-trees or the RDBMS) that maintains each updated data item and its value prior to the first write log record and after the last write log record, start time stamp of the first write log record, and the end time stamp of the last write log record. This enables BG to quickly process read log records that either reference data items that were never updated (do not exist in the main memory data structure), or were issued before the first or after the last writer (there is only one possible value for these and available in the main memory data structure). Third, multiple threads may process the read log records by accessing the aforementioned data structure with no semaphores as they are simply looking up data. This makes the validation phase suitable for multi-core CPUs as it employs multiple threads to process the read log files simultaneously.

Figure 11 shows the percentage of unpredictable data produced by a CASQL system that employs a time

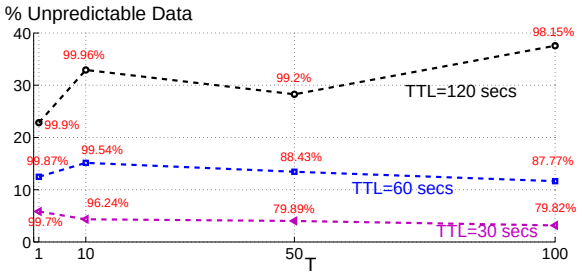


Figure 11: Percentage of unpredictable data (τ) as a function of the number of threads with memcached (CASQL). Mixed workload with 10% write actions, see Table 3. M=10,000, 12 KB image size, ϕ=ρ=100, β=100 msec, θ=0.27, α is in red.

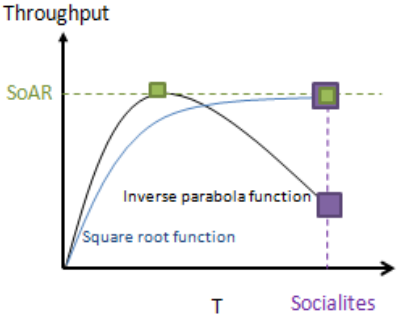
to live (TTL) to maintain its key-value pairs up to date (instead of the invalidation discussions of Section C that require design, development, debugging, and testing of software). Figure 11 shows the behavior of three different TTL values, 30, 60, and 120 seconds, as a function of the number of BG threads, T. We assume 10% of actions are writes (see Table 3). Obtained results show a higher TTL value increases the likelihood of both a write causing a key-value pair to become stale and the stale key-value being referenced. This explains the larger amount of unpredictable data with higher TTL values. Note that with the invalidation implementation of Section C, the amount of observed unpredictable data is less than 0.001% in all experiments. This can be reduced to zero by extending memcached with a race condition prevention technique such as Gumball [15].

A higher TTL value also enhances performance of CASQL by increasing the number of references that observe a cache hit. This is shown with a higher percentage of request that observe a response time faster than 100 msec (α) with T=100: α increases from 79.8% with a 30 second TTL to 98.15% with a 2 minute TTL. In essence, a higher TTL value enhances performance of CASQL by producing a higher amount of stale data.

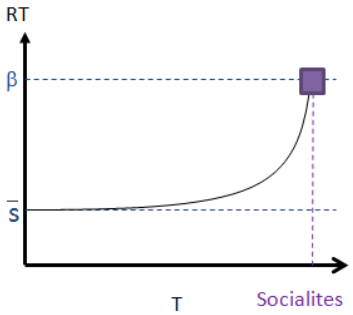
G Rating a data store

Once the BGClient for a data store has been developed (debugged and tested), N instances of it are deployed across one or more servers. Next, BGCoord is provided with the identity (IP and port) of these BGClient instances and an SLA consisting of values for α, β, τ, and Δ (see Table 2). BGCoord employs the N BGClients to compute SoAR and Socialites rating of the data store. It rates a data store by conducting several experiments, each with a fixed number of threads T. These enable BGCoord to compute SoAR and Socialites rating of the data store. Details are as follows.

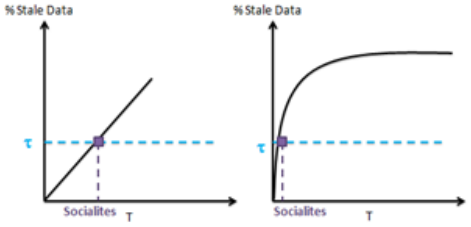
Each experiment uses T threads (spread across the N BGClients) to issues actions during Δ time units. At the end of the experiment, each BGClient reports its observed number of unpredictable reads, and the



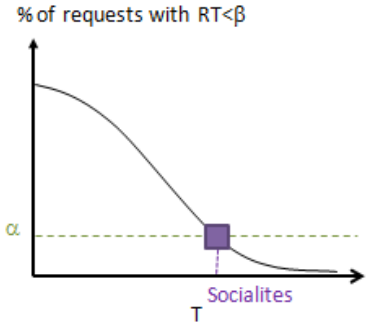
12.a Throughput as a function of T.



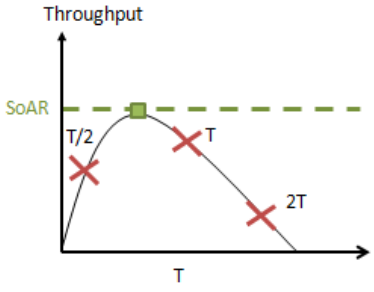
12.b $\bar{R}T$ as a function of T.



12.c τ as a function of T.



12.d α as a function of T.



12.e SoAR search space.

Figure 12: Assumptions of heuristic search.

percentage of requests that observed a response time equal to or faster than β. This experiments is *successful* as long as the following[6] hold true: 1) with each BGClient, the percentage of unpredictable reads should be less than or equal to the SLA specified tolerable amount of unpredictable reads, and 2) with each BGClient, the percentage of requests that observe a response time less than or equal to β is greater than or equal to α. Otherwise, this experiment has *failed* to meet the specified SLA.

One approach to compute SoAR and Socialites rating of a data store is to conduct experiments starting with T=1 and increment T by one every time an experiment succeeds. It would maintain the highest observed throughput and the highest T value. And, it terminates once an experiment fails (see Assumption 1 below) to satisfy the SLA, reporting the highest observed throughput as SoAR and the largest T as Socialites rating of the data store. A limitation of this strategy is that it requires a substantial amount of time. For example, in Figure 8, MongoDB supports a Socialites rating of 1000, T=1000. An exhaustive search starting with 1 thread and assuming Δ=10 minutes would require almost 7 days.

BGCoord employs heuristic search to expedite rating of a data store by conducting fewer experiments than an exhaustive search. This technique makes the following 3 assumptions about the behavior of a data store as a function of T:

1. Throughput of a data store is either a square root function or a concave inverse parabola of the number of threads, see Figure 12.a.

2. Average response time of a workload either remains constant or increases as a function of the number of threads, see Figure 12.b.

3. Percentage of stale data produced by a data store either remains constant or increases as a function of the number of threads, see Figure 12.c.

These are reasonable assumptions that hold true in most cases. Below, we formalize the second assumptions in greater detail. Subsequently, we detail the heuristic for SoAR and Socialites rating. Finally, we describe sampling using δ values (smaller than Δ) to further expedite the rating process.

Figure 12.b shows the average response time ($\bar{RT}$) of a workload as a function of T. With one thread, $\bar{RT}$ is the average service time ($\bar{S}$) of the system for processing the workload. With a handful of threads, $\bar{RT}$ may remain a constant due to use of multiple cores and sufficient network and disk bandwidth to service requests with no queuing delays. As we increase the number of threads, $\bar{RT}$ may increase due to either (a) an increase in $\bar{S}$ attributed to use of synchronization primitives by the data store that slow it down [7, 20], (b) queuing delays attributed to fully utilized server resources where $\bar{RT}$=$\bar{S}$+$\bar{Q}$ and $\bar{Q}$ is the average queuing delay, or both. In the absence of (a), the throughput of the data store is a square root function of T, see Figure 12.a. In scenario (b), $\bar{Q}$ is bounded with a fixed number of threads since BG

[6] We treat each BGClient individually because its fragment of the social network is independent of others, see Section E.

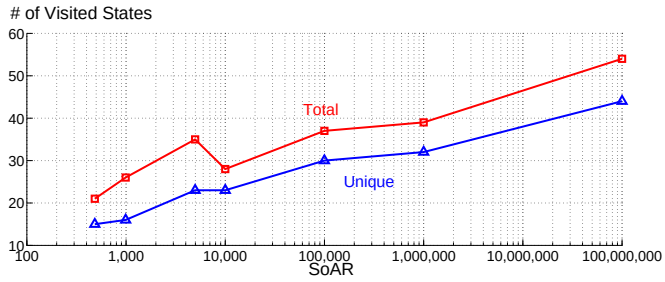


Figure 13: Number of experiments conducted to compute SoAR.

emulates a closed simulation model where a thread may not issue another request until its pending request is serviced. Moreover, as $\bar{RT}$ increases, the percentage of requests observing an RT lower than or equal to β decrease, see Figure 12.d.

The heuristic search technique to compute Socialites rating of a data store starts with an experiment using one thread, T=1. If the experiment succeeds, it doubles the value of T. It repeats this process until an experiment fails, establishing an interval for the value of T. The minimum value of this interval is the previous value of T that succeeded and its maximum is the value of T that failed. The heuristic performs a binary search of T in this interval to compute the highest T value that enables an experiment to succeed. This is the Socialites rating of the data store. It is accurate as long as Assumption 1 is satisfied, see Figure 12.a.

The heuristic to compute SoAR is similar to Socialites with several key differences. First, BGCoord maintains the highest observed throughput with each T value, λ_T. It stops doubling T once an experiment produces a throughput lower than λ_T or fails to satisfy the pre-specified SLA as is the case with the square root curve of Figure 12.a. This is the point denoted as $2T$ in Figure 12.e. It may not simply focus on the interval (T,2T) because the peak throughput might be in the interval ($\frac{T}{2}$,T), see Figure 12.e. Instead, it identifies the peak throughput as follows. It conducts experiments with $T \pm \eta$ threads to determine the slope of the curve in each direction. If both slopes are negative then T is the peak and reported as the SoAR of the data store. Otherwise, it focuses on the interval that contains the peak and performs a hill climbing process to identify the peak.

The heuristic to compute SoAR visits tens of states even though SoAR might be in the order of hundreds of millions of actions per second. To illustrate, we used the following quadratic function to model throughput of a data store as a function of the number of threads: throughput=$-T^2 + bT$. An experiment employs a fixed number of threads T that serves as the input to the function to report the observed throughput (all computed negative values are reset to zero). The vertex of this function is the maximum throughput, SoAR, and is computed by solving the first derivative of the quadratic function: T=$\frac{b}{2}$. The heuristic must compute this value of T as the SoAR of a system modeled using b.

We select different values of b to model diverse systems whose SoAR varies from 500 to 100 million

actions per second. Figure 13 shows the number of visited states when η=1. When SoAR is 100 million, the heuristic conducts 54 experiments to compute the value of T that maximizes the output of the function. Ten states are repeated from previous iterations with the same value of T. To eliminate these, the heuristic maintains the observed results for the different values of T and performs a look up of the results prior to conducting the experiment. This reduces the number of unique experiments to 40. This is 2.6 times the number of experiments conducted with a system modeled to have a SoAR of 500 (which is several orders of magnitude lower than 100 million).

During its search process, BGCoord may run the different experiments with a shorter duration (δ) than Δ to expedite the rating process, $\delta < \Delta$. Once it identifies the ideal value of T with δ for SoAR (Socialites), it runs a final experiment with Δ to compute the final SoAR (Socialites rating) of a data store. A key question is what is the ideal value of δ? Ideally, it should be small enough to expedite the time required to rate a data store and, large enough to enable BG to rate a data store accurately. There are several ways to address this. For example, one may compare the throughput computed with δ and Δ for the final experiment and, if they differ by more than a certain percentage, repeat the rating process with a larger δ value. Another possibility is to employ a set of values for δ: $\{\delta_1, \delta_2, ..., \delta_i\}$. If the highest two δ_i values produce identical ratings, then they establish the value of δ for that experiment. The number of δ values in the set should be small enough to render the rating process faster than performing the search with Δ.

The value of δ is an input to BGCoord. If it is left unspecified, BG uses Δ for the rating process. As an example, numbers of Figure 8 are generated using δ=3 minute. The solid rectangle boxes (SoAR ratings with different N) are generated using Δ=10 minutes.

H Related Work

BG falls in the *vector based* approach of [27] that models application behavior as a list of actions and sessions (the 'vector') and randomly applies each action to its target data store with the frequency a real application would apply the action. The input workload file of BG specifies the frequency of different actions and session, configuring BG to emulate a wide range of social networking applications. (See Table 3 for three example mixes.) This flexibility is prevalent with both YCSB [11] and YCSB++ [23]. In fact, our implementation of BG employs the core components of YCSB and extends them with new ones such as the actions of Section C, D-Zipfian, BGCoord, and BG's visualization deck. Those with hands on experience with YCSB find BG familiar with the following key modifications and extensions:

1. A more complex conceptual schema specific to social networks.

2. Simple table operations of YCSB have been replaced with social actions and sessions.

3. BG consumes an SLA to compute two ratings for a data store: SoAR and Socialites. If no SLA is specified, BG execute the same as YCSB.

4. BG quantifies the amount of unpredictable data produced by a data store.

5. BG employs a shared-nothing architecture and constructs self-contained fragments of its database to ensure concurrent socialites emulated by independent BGClients are unique, see Section E. This eliminates the need for coordination between BGClients during benchmarking phase, enabling BG to scale to a large number of nodes.

Some of BG's extensions to YCSB are similar to those that differentiate YCSB++ from YCSB. For example, the concept of multiple BGClients managed by BGCoord is similar to how YCSB++ supports multiple YCSB clients. However, there are also differences. First, YCSB++ includes mechanisms specific to evaluate table stores such as HBase. These include function shipping and fine grained access control. Instead of these, BG focuses on interactive social networking actions of Section C and their implementation with alternative data stores. While extension 5 of BG (see the previous paragraph) is similar to ingest-intensive extension of YCSB++, it goes beyond simple ranges that partition data across multiple nodes: Friendships and resources of members are logically partitioned to construct N self-contained independent social networks where N is the number of BGClients.

Second, YCSB++ consists of an elegant mechanism to quantify the inconsistency window: The lag in acknowledged data store changes that are not seen by other clients for some time due to use of a weak consistency semantic such as eventual consistency [32]. BG captures the impact of such design decisions by quantifying the amount of unpredictable data. Both metrics are in synergy and may co-exist in a benchmark.

Finally, while both YCSB and YCSB++ lack the concept of an SLA to rate a data store, SLAs are the essence of TPC-A/C benchmarks [16]. For example, TPC-A measures transactions per second (tps) subject to a response time constraint. BG is similar as it employs SLAs to obtain its rating. It is different than TPC because it focuses on social networking actions and incorporates unpredictable data as a component of SLAs.

I Future Research

While there are many data stores, there is a "gaping hole" with scarcity of benchmarks to substantiate the claims of these data stores [9]. Social networking companies continue to contribute data stores to address their requirements for interactive member actions, e.g., Cassandra and TAO [1] by Facebook and Voldemort by LinkedIn. BG is a benchmark to evaluate these alternative implementations and their claims objectively. The most important feature of BG is its ability to scale to characterize the performance of a data store accurately.

Our immediate short term activities are as follows. First, we are studying alternative physical design of data with both the relational [6] and JSON-Like models. Second, we are extending the validation phase of BG to utilize its log records to compute the lag for an acknowledged update to be visible to all clients [33]. Third, we are extending BG with additional interactive actions such as posting and viewing a tweet with Twitter, newsfeed with Facebook, job change with LinkedIn [28]. Fourth, we are using BG in a number of studies to evaluate elasticity of data stores and their behavior in the presence of failures.

J Acknowledgments

We thank Jason Yap for his implementation of CASQL BGClient. We are grateful to anonymous CIDR 2013 reviewers for their insights and valuable comments.

References

[1] Z. Amsden, N. Bronson, G. Cabrera 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.

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

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

[4] L. Backstrom. Anatomy of Facebook, http://www.facebook.com/note.php?note_id=10150388519243859, 2011.

[5] S. Barahmand and S. Ghandeharizadeh. D-Zipfian: A Decentralized Implementation of Zipfian, USC DBLAB Technical Report 2012-04, http://dblab.usc.edu/users/papers/dzipfian.pdf, 2012.

[6] S. Barahmand, S. Ghandeharizadeh, and J. Yap. Physical Relational Database Design for Interactive Social Networking Actions, USC DBLAB Technical Report 2012-08, http://dblab.usc.edu/users/papers/RelationalBG.pdf.

[7] M. W. Blasgen, J. Gray, M. F. Mitoma, and T. G. Price. The Convoy Phenomenon. *Operating Systems Review*, 13(2):20–25, 1979.

[8] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *SIGMOD Conference*, pages 12–21, 1993.

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

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

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

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 15, pages 290–295. MIT Press, 2001.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004.

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

[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*, 2012.

[16] J. Gray. The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann 1993, ISBN 1055860-292-5.

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, pages 677–680. Morgan Kaufmann, 1993.

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

[19] C. Harris. Overview of MongoDB Java Write Concern Options, November 2011, http://www.littlelostmanuals.com/2011/11/overview-of-basic-mongodb-java-write.html.

[20] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, pages 24–35, 2009.

[21] N. Lynch and S. Gilbert. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT New*, 33:51–59, 2002.

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

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

[24] 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.

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

[26] 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.

[27] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The Case for Application Specific Benchmarking. In *HotOS*, 1999.

[28] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan. Feed Following: The Big Data Challenge in Social Applications. In *DBSocial*, pages 1–6, 2011.

[29] M. Stonebraker. Errors in Database Systems, Eventual Consistency, and the CAP Theorem. *Communications of the ACM, BLOG@ACM*, April 2010.

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

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

[32] W. Vogels. Eventually Consistent. *Communications of the ACM, Vol. 52, No. 1*, pages 40–45, January 2009.

[33] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: The Consumers' Perspective. In *CIDR*, 2011.

[34] The Free Encyclopedia Wikipedia. List of Social Networking Websites, http://en.wikipedia.org/wiki/list_of_social_networking_websites.