A Mid-Flight Synopsis of the BG Social Networking Benchmark*

Shahram Ghandeharizadeh and Sumita Barahmand

Database Laboratory Technical Report 2013-03

Computer Science Department, USC

Los Angeles, California 90089-0781

{shahram,barahman}@usc.edu

November 15, 2013

Abstract

BG is a benchmark that rates the performance of a data store for processing interactive social networking actions such as view a member's profile, invite a member to be friends, accept a friend request, and others. It is motivated by a proliferation of data stores from a variety of academic and industrial contributors including social networking companies, e.g., Voldemort by LinkedIn. BG is designed to provide a system architect with insights into alternative design principles such as the use of a weak consistency technique instead of a strong one, different physical data models such as relational and JSON, factors that impact vertical and horizontal scalability of a data store, the consistency versus availability tradeoff in the CAP theorem, among others. While BG is a recently introduced benchmark (less than a year old as of this writing), it combines elements of maturer benchmarks and extends them to simplify its use by the practitioners and experimentalists. This paper provides a synopsis of the BG benchmark by identifying its strengths and limitations in our daily use cases. The identified limitations shape our research activities and the obtained solutions shall be incorporated into future BG releases. Thus, this workshop paper is a mid-flight glimpse into our current research efforts with BG.

A Introduction

In an article that appeared in the July 2012 issue of the Communications of the ACM, David Patterson observes when a discipline has good benchmarks, debates are settled and the discipline makes rapid progress [16]. Today, we have an abundance of architectures for data stores and services with only a handful of benchmarks to substantiate their many claims. Academia, cloud service providers such as Google and Amazon, social networking sites such as LinkedIn and Facebook, and computer industry continue to contribute systems and services with novel architectures

*This paper appeared in the Fourth Workshop on Big Data Benchmarking, October 2013, San Jose, CA.

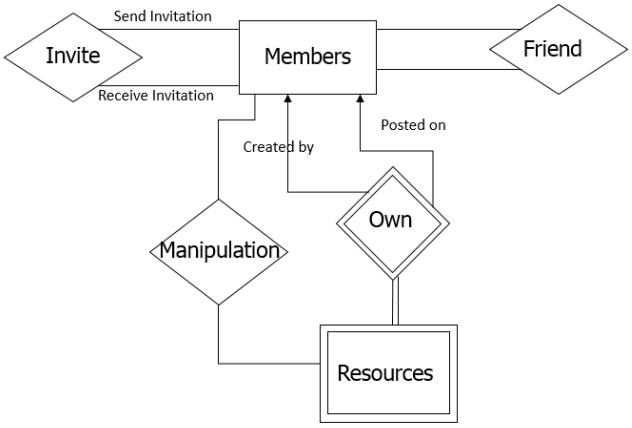


Figure 1: Conceptual design of BG's database.

and assumptions. In 2010, Rick Cattell surveyed 23 systems [7] and we are aware of 10 new[1] ones since that writing. In his survey, Cattell identified a "gaping hole" with a scarcity of benchmarks to substantiate the claims made by the different systems. We have designed and implemented a social networking benchmark named BG [2] (visit http://bgbenchmark.org) to address certain aspects of the hole that is too large to address with just one benchmark.

BG's workload consists of actions that either read or write a small amount of data from big data, typically termed simple operations [21, 10]. Today's BG is designed for high throughput data stores that provide interactive response times. A long term objective is to extend BG with complex analytics that require processing of a large amount of data using machine learning algorithms. This would make BG suitable to evaluate Hadoop and other implementation of the MapReduce [8] framework, see Section G for details.

We developed BG in 2012 and released a stable version of it in January 2013. Its conceptual schema and eleven actions are an abstraction of today's social networking sites such as Google+, Facebook and others. In [2], we provide a comprehensive list of the surveyed sites and a matrix that describes compatibility of BG's actions with those supported by a site. Figure 1 shows BG's conceptual schema. The concept of members with registered profiles befriending one another is at the core of this schema. Its implementation in a physical data store is dictated by an experimentalist. BG is data store agnostic and one may tailor both the physical schema and the implementation of actions to highlight the strengths of a data store. At the time of this writing, an implementation of BG's schema and actions is available for the following data stores:

- MySQL, Oracle, PostgreSQL, and VoltDB as relational data stores.
- MongoDB and CouchBase as document stores.

[1] Apache Jackrabbit and RavenDB, Titan, Oracle NoSQL, FoundationDB, STSdb, EJDB, FatDB, SAP HANA, CouchBase.

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	0.04%	0.4%	4%
Accept Friend Request (AFR)	Write	0.02%	0.2%	2%
Reject Friend Request (RFR)	Write	0.02%	0.2%	2%
Thaw Friendship (TF)	Write	0.02%	0.2%	2%
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.

- Microsoft Azure as a cloud service provider.
- Hibernate as an Object Relational Mapping (ORM) framework.
- Cache augmented data stores with memcached, EhCache, Twemcache, and KOSAR.

The first column of Table 1 shows the eleven actions that constitute BG. The name of the actions is self explanatory. All actions that reference members are binary consuming two member ids as input. For example, the two member ids specified with the View Profile action identify the member who is viewing a profile and the member whose profile is being viewed. Those actions that consume a resource id either read the resource and its comments or modify a comment on that unique resource. The different actions may either read or write data from a data store as highlighted by the second column of Table 1. We refer the interested reader to [2] for a detailed description of each action.

The last three columns of Table 1 show three different workloads corresponding to a very low, low, and a high percentage of write actions. (According to Facebook, more than 99% of its workload consists of read actions [9].) Each column specifies a fixed percentage of occurrence for each action in a workload. We use the presented three workloads in our experiments on a daily basis. All three are symmetric workloads that cause an experiment to complete with approximately the same number of confirmed friendships and pending friendships as those in the beginning of the experiment. The number of these relationships is impacted by the frequency of the following actions: Invite Friend, Accept Friend Request, Reject Friend Request, and Thaw Friendship actions. For this number to remain unchanged, the rate at which BG generates friendships should equal the rate at which it thaws friendships. This is realized by satisfying the following two conditions: 1) percentage of Thaw Friendship and Accept Friend Request must be identical, and 2) percentage of Invite Friend must equal the sum of percentage of Reject Friend Request and Accept Friend Request. In Table 1, the frequency of Post Comment on a Resource (PCR) and Delete Comment from a Resource (DCR) are intentionally kept at zero to demonstrate that one may specify workloads consisting of either a single or a few actions. When specifying frequencies of PCR and DCR, a symmetric workload should define the same frequency of occurrence for each action.

BG is a stateful benchmark that generates valid actions. For example, it extends a friendship from Member A to Member B only when they are not friends. It realizes this by maintaining a representation of the social graph in its memory. BG uses this representation to ensure its emulated simultaneous members and resources are unique at an instance in time.

A novel feature of BG is its ability to quantify the amount of unpredictable (stale, inconsistent, erroneous) data produced by a data store. BG evaluates a data store for a workload that specifies an SLA. An example SLA may require 95% of actions to be performed faster than 100 milliseconds with no more than 0.01% unpredictable data for Δ=10 minutes. BG includes a heuristic search technique to quantify the maximum throughput (actions per second) observed with a data store while satisfying the pre-specified SLA. This is termed the Social Action Rating, SoAR, of the data store.

We have employed BG to investigate design and implementation of novel architectures for data intensive applications, e.g., to compare a relational representation of a social graph with its JSON representation [5], quantify the tradeoffs associated with alternative consistency techniques for a cache augmented relational data store [12], and others. In these use cases, we have identified several limitations with BG's design. These shape our research efforts to extend BG to maintain it as a state of the art benchmark. Most are in the context of the novel features of BG that make it unique. Below, we describe these in turn, detailing BG's scalable request generation in Section B, its closed emulation of socialites and an alternative open emulator in Section C, its rating mechanism in Section D, its validation phase in Section E, and additional actions in Section F. Section G provides our long term future research.

B Scalability

BG employs a shared-nothing architecture and scales to a large number of nodes, preventing either the CPU, network, or memory resources of a single node from limiting its request generation rate. Its software architecture consists of one coordinator and N clients, termed BGCoord and BGClient, respectively. In our experiments with an 8 core CPU, a multi-threaded BGClient is able to utilize all cores fully as long as the client component of a data store does not suffer from the convoy phenomena [6] and the data store is able to process requests at the rate generated by BG. When the client component of a data store limits vertical scalability, as long as there is a sufficient amount of memory, one may execute multiple instances of BGClients on a single node to utilize all cores. BG scales horizontally by executing multiple BGClients across different nodes. BGCoord is responsible for initiating the BGClients, monitoring their progress, gathering their results at the end of an experiment, and aggregating the obtained results to compute the SoAR of a data store.

Once the BGClient instances are started, they generate requests independently with no synchronization. This is made possible using the following two concepts. First, a BGClient implements a decentralized partitioning strategy that declusters a benchmark social graph into N disjoint sub-graphs where N is the number of BGClients. A BGClient is assigned a sub-graph to generate requests referencing members of its assigned sub-graph only. While the data store

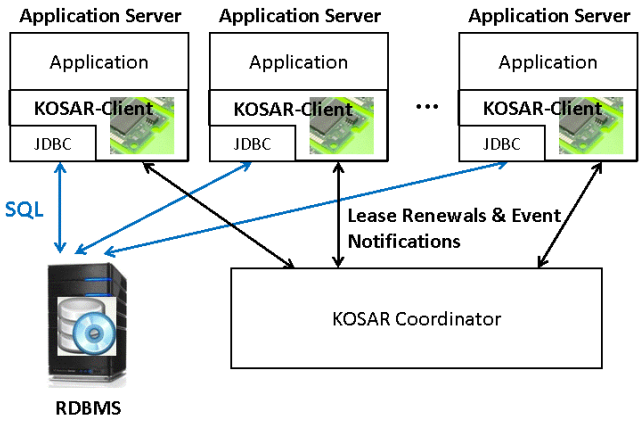


Figure 2: A data intensive architecture using KOSAR.

is not aware of this partitioning, the data generated and stored in the data store does correspond to the N disjoint graphs. One may conceptualize each sub-graph as a province whose citizens may perform BG's actions with one another only. This means citizens of different provinces may not view one another's profile or become friends with one another.

Second, BG employs a novel decentralized implementation of the Zipfian distribution, named D-Zipfian [3, 24], that ensures the distribution of requests to the different members is independent of N. Thus, the distribution of access with one node is the same as that with several nodes. D-Zipfian in combination with partitioning of the social graph enables BG to utilize N nodes to generate requests without requiring coordination until the end of the experiment, see [2, 4, 3] for details.

While BG scales to a large numbers of nodes, its two concepts may fail to evaluate some data stores objectively. As an example, consider the architecture of Figure 2 where an application is extended with a cache such as KOSAR or EhCache [12]. This caching framework consists of a KOSAR coordinator that maintains which application server has cached a copy of a data item in its KOSAR JDBC wrapper, KOSAR-Client for short. When one application server updates a copy of the data item, its KOSAR-Client informs the KOSAR coordinator of the impacted data item. In turn, the KOSAR coordinator invalidates a copy of this data item that resides in the KOSAR-Client of other application servers. With a skewed pattern of access to members and a workload that exhibits a low read to write ratio, a centralized KOSAR coordinator may become the bottleneck and dictate the overall system performance. The aforementioned two concepts employed by BG fail to cause the formation of such a bottleneck. To elaborate, each application server references data items that are unique to itself since its assigned sub-graph is unique and independent of the other sub-graphs. Hence, once an application server updates a cached data item, BG does not exercise the KOSAR coordinator informing KOSAR-Client of another application server.

To address the above limitation, we are extending BG to employ N BGClients with one social graph. The key concept is to hash partition members and resources across the N BGClients. Each BGClient is aware of the hash

function and employs the original Zipfian distribution (instead of D-Zipfian) to generate member ids. When a BGClient BGC_i references a data item that does not belong to its assigned partition, it contacts the BGClient that owns the referenced data (say BGC_j) to lock that data item for exclusive use by BGC_i and to determine if its intended action is possible. BGC_j grants the lock request if there is no existing lock on the referenced data item and the action is possible, enabling BGC_i to proceed to generate a request with the identified data item to the data store. Once the request is serviced, BGC_i contacts BGC_j to release the exclusive lock on the referenced data item to make it available for use by other BGClients. This design raises the following interesting questions:

- When BGC_j fails to grant an exclusive lock[2] to the referenced data item due to an existing lock, how should the framework handle the conflict? Three possibilities are as follows. First, it may block BGC_i until the referenced data item becomes available. Second, it may return an error to BGC_i to generate a different member/resource id and try again. Third, it may simply abort this action and generate a new action all together. We intend to quantify the tradeoff associated with these three possibilities and their impact on both the distribution of requests and the benchmarking framework.

- What is the scalability characteristic of the proposed technique? The proposed request generation technique requires different BGClients to exchange messages to lock and unlock data items and to determine the feasibility of actions. We plan to quantify this overhead and its impact on the scalability of this request generation technique. This intuition should enable us to propose refinements to enhance scalability.

- How different are the obtained results with N disjoint social graphs (current version of BG) and one social graph (the proposed change)? This question applies to those systems that may use the current version of BG. We intend to repeat our published experiments such as those reported in [5] to quantify differences if any.

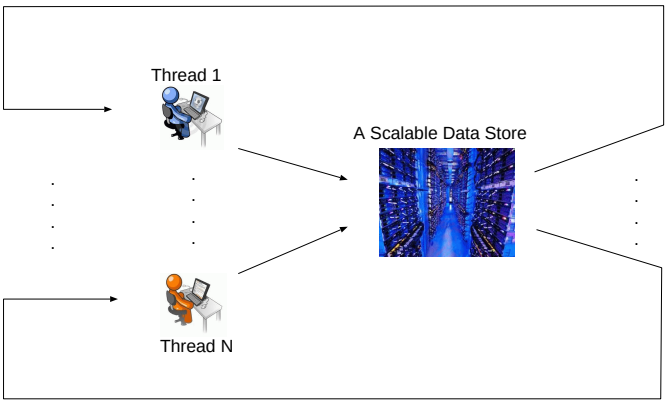
An investigation of these questions shapes our short term research direction.

C Closed versus Open

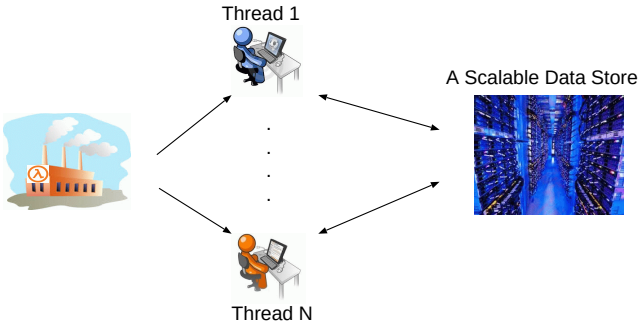
BGClients generate requests using a fixed number of threads T. Each thread emulates a random member of a social networking site performing one of the eleven actions. The randomly selected member is conditioned using the D-Zipfian distribution. This is termed a closed emulation model because a thread does not emulate a new member generating a new action until its emulation of a current member completes. This model may include a think time between emulation of different members issuing actions. Historically, this is a model[3] of a financial institution with a fixed number of tellers (ATM machines) with T concurrent customers (threads) performing financial transactions simultaneously [13].

[2] BGC_j may return an error code when the action is not possible. For example, Thaw Friendship using Member A may not be feasible because A has no friends. In these cases, BGC_i may either abort the action or may reference a new member for the same action.

[3] This model is a representative of a web server configured with a maximum number of threads.



3.a) Closed



3.b) Open

Figure 3: Closed and open emulation of socialites issuing actions to a data store.

An open emulator is a more realistic model of a social networking site [17] (and web sites in general). With this model, the emulator generates requests based on a pre-specified arrival rate, λ. This model is depicted in Figure 3 where a factory generates members who issue a social networking action independently. (A member who is performing an action is termed a *socialite*.) The factory does not wait for the data store to service a request issued by a socialite. Instead, it generates λ socialites issuing requests per unit of time using a distribution such as random, uniform, or Poisson. A Poisson distribution results in a pattern of requests that is bursty. This means λ is an average and the number of simultaneous socialites at an instance in time might be higher than λ.

While the open emulator is more realistic, its design and implementation requires a careful study. This is because today's data stores service requests at such a high rate that the emulator must support λ values in the order of a million without exhausting its CPU resources. In addition to the scalability discussions of Section B, the emulator must generate requests in a burst consistent with the Poisson distribution. At the time of this writing, we are evaluating the feasibility of such an open emulator and its implementation in BG.

D Rating Mechanism

BG *rates* a data store to compute its Social Action Rating (SoAR) and Socialites rating for processing a workload. A workload consists of a mix of the eleven actions (see Table 1 and its discussion in Section A), an exponent for the the D-Zipfian distribution to control its degree of skew when referencing members, and a pre-specified SLA, see Section A for an example SLA. The SoAR of a data store is the highest throughput provided by that data store for the specified workload. The Socialite rating of a data store is the maximum number of simultaneous socialites (threads) that may generate requests corresponding to the specified workload. The pre-specified SLA imposes constraints on the acceptable response times and the amount of unpredictable data to constrain the SoAR and Socialite ratings of a data store. Given several data stores, the data store with the highest SoAR and Socialite rating is the superior one.

BG's rating process and its assumptions are detailed in [2]. Briefly, the rating process consists of a heuristic search that conducts multiple experiments. Each experiment uses the same workload specified by an experimentalist. With those workloads that impact the state of the database, the rating process might be required to re-load the database prior to each experiment. The repeat loading of a benchmark database may constitute a significant portion of the rating process. For example, the time to generate a one million member database with a data store requires 11 days [4]. If the rating process conducts ten experiments each 30 minutes in duration, the time to generate the database each time would require almost 4 months.

Three alternatives that re-generate the database expeditiously are detailed in [4]. One is the Database Image Loading (*DBIL*) technique that maintains the original image of a database and copies it as the current database in advance of each experiment to reduce the load time. The time to copy the one million member database is 30 minutes [4], reducing the time to conduct ten experiments to 11 days and ten hours. The 11 days incurred to create the database for the first time is a one time overhead. By maintaining the image and re-using it, the duration of rating of the data store

with different workloads is reduced dramatically. For example, the time to conduct ten experiments is now ten hours. We refer the interested reader to [4] for a detailed description of DBIL and two other techniques.

A key research question is the duration of each conducted experiment. A possible answer is to employ the duration specified by the SLA, Δ. However, if an experimentalist select a high value for this input parameter, then the rating process may consume more time than necessary. It is desirable to run experiments for shorter durations than Δ to identify a region in the search space that establishes the true SoAR of a data store. Ideally, the duration of an experiment should be the smallest possible value, δ, that reflects the behavior of a data store as if the experiment was running for Δ time units. The ideal δ is both data store and workload dependent and can be analyzed in a pre-processing step, prior to the rating process. This step involves multiple experiments issuing the given workload against the data store to select the smallest duration that results in a *steady* system behavior defined as one whose resource utilization and observed throughput do not change in time. For such a system the recently observed behavior will continue to hold into the future.

E Validation

A novel feature of BG is its ability to quantify the amount of unpredictable data (stale, inconsistent, erroneous) produced by a data store. A data store may produce unpredictable data for a variety of reasons. Examples include use of a weak consistency technique such as eventual [23, 20] and use of a cache [12] in a manner that results in dirty reads [15] and inconsistent cache states [11].

BG is able to measure the amount of unpredictable data because it is a stateful benchmark that is aware of the initial state of a data item in the database and its updates to the database. It maintains the start and end of each action to enumerate the finite number of ways a read may overlap multiple concurrent write actions that reference the same data item. BG enumerates these to compute a range of possible 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*.

BG decouples generation of requests to quantify the performance of a data store from the validation phase, performing validation in an off-line manner. When generating requests, each BG thread generates a log record for each of its actions: read log records for read actions and write log records for write actions. These log records are written to separate files. One file for the read log records and a second file for the write log records.

The validation phase assumes in-memory data structures to compute the amount of unpredictable data as follows. First, it maintains an interval tree for (1) each member and write actions that impact her friendships, (2) each member and write actions that impact her pending friend invitations, and (3) each resource that is annotated with a write action. It constructs interval trees for a member/resource on demand as it reads the write log records in memory. The start and end time stamp of the write log records are indexed by an interval tree. Once the write log records are staged in memory, the validation phase retrieves the read log records. It employs the member id (resource id) and the action to identify the interval tree with the relevant write log records. Next, it uses the start and end time stamp of each read log

record to enumerate the number of ways it overlaps with the different write actions, computing a range of valid values that should be observed by the read action. If the value observed by the read action (recorded in the log record) does not match one of the valid values then this read action has observed an unpredictable value.

The current implementation of the validation phase is fast when there is a sufficient amount of memory to stage the write log records in interval trees. This enables the validation phase to read the log files once to process both read and write log records in one pass. This multi-threaded process may utilizes multiple cores fully because (1) write log records are inserted into different interval trees based on their referenced member/resource id, and (2) different read log records may share and read the same interval tree simultaneously.

By performing validation in an off-line manner, BG reduces the number of nodes required to generate request to evaluate a data store. A key limitation of the current validation technique is that it may exhaust the available physical memory, causing the operating system to exhibit a thrashing behavior that results in an unacceptably long validation process. This is specially true with high throughput multi-node data stores and cache augmented data stores such as KOSAR that process requests in the order of millions of actions per second. We intend to extend the validation phase to analyze the size of files produced during an experiment to estimate the amount of required memory to ensure it does not exhaust the physical memory. In passing, it is interesting to note that the log records pertaining to each unique member can be processed independently to identify unpredictable data. Hence, a MapReduce [8] framework such as Hadoop is effective in implementing the validation phase.

F Additional Actions

The eleven actions of BG are a good start to evaluate a data store. However, there are other actions that are common to many social networking sites that we intend to abstract. An implementation of these would extend BG with new actions. Here, we focus on two new actions to be released soon, namely, Share Resource (SR) and Retrieve Feed (RF). Both are in support of feed following [18, 19, 1]. This action is supported by sites such as Google+, Twitter, Facebook, My Yahoo and others. It enables users to create personalized feed by selecting one or more event streams they wish to follow.

Figure 4 shows the high level (incomplete) ER diagram for feed following. While the conceptual model looks complex, it is based on the concept of aggregation that establishes the many-to-many relationship between producers and consumers of news feeds. The producers are a specialization of the Accounts entity set, identifying Pages and Members as two different categories because pages may have a significantly larger number of followers[4] and do not perform some of the actions performed by members. Examples of pages are celebrity fan pages and company and brand pages among the others. Resources such as images and tweets posted by a page are shared with their followers. This is represented by the red rectangle that aggregates the follow relationship between members and pages as an entity that participates in the "Share" relationship (red line) with the Resources entity set. Similarly, resources posted by the

[4] As of November 12, 2013, Katy Perry had more than forty million Twitter followers.

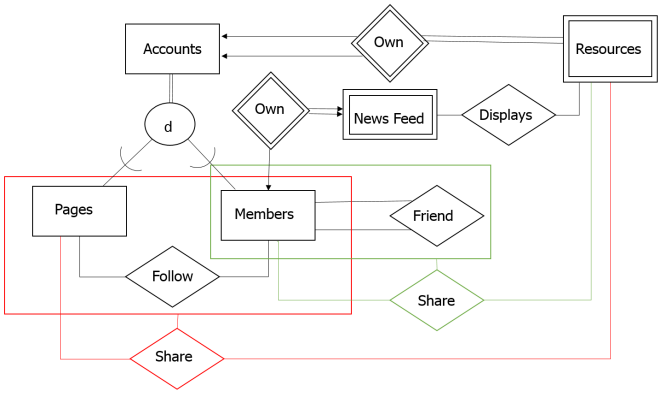


Figure 4: Conceptual design of BG's feed following.

members are shared with their circle of friends which are also members. This is denoted by the green aggregation that participates in the "Share" relationship with the Resources entity set. Each member owns a "News feed" that contains resources shared by those whom they are following.

In the near future, we intend to investigate alternative techniques to implement feed following; see [19] for a pull, push, and a hybrid of these two techniques. We are also exploring techniques that compute an approximate feed. We intend to use BG to quantify both the scalability of these techniques and the performance gain of the approximate technique relative to its amount of unpredictable data.

G Conclusions and Future Research

Social networks are emerging in diverse applications that strive to provide a sense of community for their users. These diverse applications range from financial web sites such as online trading system to academic institutions. BG is a benchmark to evaluate the performance of a data store for processing interactive social networking actions such as viewing a member's profile, extending a friendship request to a member, accepting a friendship request, and others as shown in Table 1. We use BG on a daily basis to evaluate the performance of novel architectures that enable high throughput, low latency data stores. Obtained insights enable us to introduce novel designs and implementations. We identified key features of BG and how we are refining these to ensure BG remains a state of the art benchmark. Each section focuses on one feature of BG. However, a change in one aspect of BG impacts the other features of BG. For example, the two design principles that enable BG to generate requests in a scalable manner (see Section B) also enables multiple BGClients to perform the validation phase (see Section E) independently and in parallel with one another. The proposed modifications of Section B on how BG generates requests requires a revisit of how BG performs its validation phase.

We intend to extend today's BG beyond simple analytics to include additional Web 2.0 workloads including com-

plex analytics that require the use of machine learning algorithms. An example is a recommender system that analyzes the social graph and attribute values to suggest friends for a member. A challenge here is to generate the social graph in a manner where the output of the recommendation system is known, extending BG with metrics such as precision and recall. The graph database may include annotations on the edges of the graph with attribute values. For example, the friendship relationship between two members A and B might be tagged with a family relationships (such as daughter), number of likes given by Member A to postings by B's friends, number of comments posted by B's friends on A's resources, and others. We may change how the value of textual attributes for a comment are generated in a manner that is more realistic using techniques such as those utilized by the TREC benchmark. This may evaluate alternative (1) natural language processing and (2) information retrieval techniques in the context a recommender system for a social graph.

Finally, we are investigating the viability of a *Benchmark Generator*, BG+, that inputs an abstraction of an application, its actions and their dependencies, metrics of interest to be quantified, and a control parameter. Its output is a benchmark specific to that application. This is appropriate for those applications with diverse use cases such as data sciences [22, 14]. In essence BG+ is an extensible toolkit that is configured with its input to enable a data scientist to develop a benchmark for their activities rapidly. Similar to BG and YCSB, it would expose the implementation of the database schema and the abstracted actions to be implemented by the data scientists. This enables BG+ to support diverse types of data models such as structured (relational), un-structured, JSON, extensible, images, audio and video as input. The control parameter may manipulate factors such as noise in the data (similar to today's system load controlled by parameter T) to analyze the behavior of an algorithm. This would enable BG+ to output a rating mechanism that computes a single value (a maxima such as SoAR) by manipulating the value of the control parameter.

References

[1] X. Bai, F. P. Junqueira, and A. Silberstein. Cache Refreshing for Online Social News Feeds. In *CIKM*, pages 787–792, 2013.

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

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

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

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

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

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

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

[9] R. Nishtala et. al. Scaling Memcache at Facebook. *NSDI*, 2013.

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

[11] S. Ghandeharizadeh and J. Yap. Gumball: A Race Condition Prevention Technique for Cache Augmented SQL Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, 2012.

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

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

[14] C. Greenberg. Overview of the NIST Data Science Evaluation and Metrology Plans, Data Science Symposium, NIST, March 4-5, 2014.

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

[16] D. Patterson. For Better or Worse, Benchmarks Shape a Field. *Communications of the ACM*, 55, July 2012.

[17] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. *NSDI*, 2006.

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

[19] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding Frenzy: Selectively Materializing Users' Event Feeds. In *SIGMOD Conference*, pages 831–842, 2010.

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

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

[22] A. Talukder. Overview of the NIST Data Science Program, Data Science Symposium, NIST, March 4-5, 2014.

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

[24] J. Yap, S. Ghandeharizadeh, and S. Barahmand. An Analysis of BGs Implementation of the Zipfian Distribution. *USC DBLAB Technical Report 2013-02, http://dblab.usc.edu/Users/papers/zipf.pdf*, 2013.