BG: A Scalable Benchmark for Interactive Social Networking Actions

Yazeed Alabdulkarim, Sumita Barahmand and Shahram Ghandeharizadeh

Database Laboratory Technical Report 2018-01

Computer Science Department, USC

Los Angeles, California 90089-0781

Abstract

BG is a benchmark that rates a data store for processing interactive social networking actions such as view a member profile and extend a friend invitation to a member. It elevates the amount of stale, inconsistent, and erroneous (termed *unpredictable*) data produced by a data store to a first class metric, quantifying it as a part of the benchmarking phase. It summarizes the performance of a data store in one metric, Social Action Rating (SoAR). SoAR is defined as the highest throughput provided by a data store while satisfying a pre-specified service level agreement, SLA.

To rate the fastest data stores, BG scales both vertically and horizontally, generating a higher number of requests per second as a function of additional CPU cores and nodes. This is realized using a shared-nothing architecture in combination with two multi-node execution paradigms named Integrated DataBase (IDB) and Disjoint DataBase (D^2B). An evaluation of these paradigms shows the following tradeoffs. While the D^2B scales superlinearly as a function of nodes, it may not evaluate data stores that employ client-side caching objectively. IDB provides two alternative execution paradigms, Retain and Delegate, that might be more appropriate. However, they fail to scale as effectively as D^2B . We show elements of these two paradigms can be combined to realize a hybrid framework that scales almost as effectively as D^2B while exercising the capabilities of certain classes of data stores as objectively as IDB .

1 Introduction

There has been a proliferation of data stores with novel software architectures and data models for processing *simple* operations, reads and writes of a small amount of data from big data [19, 46, 17]. These systems range from SQL to NoSQL, cache augmented SQL [43, 39, 4, 33, 30], document stores [19], graph databases [16, 28], and others. Cattell surveyed more than twenty systems in [19], providing a qualitative discussion of the alternative designs and their merits. This survey identifies scarcity of benchmarks as a “gaping hole” to substantiate the scalability claims of data stores.

There are two classes of benchmarks: micro and macro [36, 45]. Microbenchmarks emphasize a specific functionality of a data store. For example, the Wisconsin benchmark [15] quantifies the response time of a SQL system to join two tables or look up a record. Macrobenchmarks provide an abstraction of an application class such as social networking, e.g., [8, 5, 26]. While their specific actions can be used in isolation to model a microbenchmark, they go one step further to generate a mix of actions that resemble everyday use case of their target applications [45, 42, 36]. The focus of this study is on macrobenchmarks.

Design of macrobenchmarks is challenging because they must produce meaningful actions, scale to evaluate the fastest data stores, and evaluate data stores with diverse software architectures and design decisions. A meaningful action is one that is valid in the context of the database and does not result in integrity constraint violations. These exceptions are undesirable because they do not model the behavior of a real application that performs useful work. For illustration, we provide examples from YCSB [22] and BG [8, 6]. An action produced by YCSB is to insert a row with a primary key in a table. When scaling YCSB horizontally, if two nodes generate the same primary key for two different concurrent inserts then one will fail due to integrity constraint violation. YCSB++ [41] addresses this limitation by range partitioning the possible set of primary key values and assigning each range to a different node for request generation. When reads are generated based on these pre-splits, the execution of benchmark instances populate caches of Figure 1 with mutually exclusive data, failing to exercise the CORE component responsible for maintaining replicated cached entries consistent.

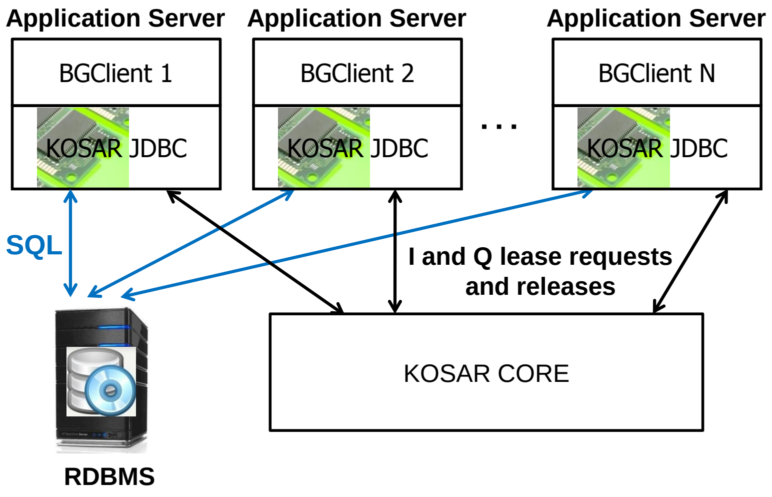

Figure 1: Caches such as Terracotta BigMemory [48], JBoss [18] and KOSAR [30] run in the address space of an application. They employ a core that maintains replicated cached entries consistent with one another. If the benchmarking framework prevents replication of entries across BGClient$_i$ caches then the system is not evaluated objectively.

Similarly, BG strives to provide meaningful interactive social networking actions. An example is the Thaw Friendship (TF) action between two members, say M_i and M_j. BG generates TF(M_i, M_j) only if M_i and M_j are friends. Once again, a challenge is how to identify members who are friends to generate the TF action and scale horizontally. Similar to YCSB++, BG constructs disjoint logical social graphs and assigns each subgraph to a different node for request generation [8, 6]. It is comparable to representing a large metropolitan as a collection of villages where members of a village are allowed to have social interactions with one another only. This approach also fails to evaluate the cache augmented architecture of Figure 1 objectively because it prevents replication of members and their friendships across the cache instances, failing to exercise the CORE responsible for managing replicated data.

The **primary contribution** of this study are three paradigms for generating meaningful requests using a shared-nothing architecture and their tradeoffs. These paradigms are named integrated database (IDB), disjoint database (D^{2}B), and a combination of these two extremes named Hybrid. All three partition data across the N benchmarking nodes that generate actions. However, they differ in how they create the benchmark database and the actions to rate a data store. IDB constructs an integrated database and requires the N nodes to use message passing to generate meaningful actions. D^{2}B constructs N disjoint databases and assigns each to a different benchmarking node. Moreover, it implements global constraints across concurrent threads effectively. For example, it enforces uniqueness of members issuing concurrent requests in a scalable manner. D^{2}B was illustrated by the motivating examples using YCSB and BG that required each node to generate actions referencing its assigned logical database partition. Its design challenge is how to enable the N nodes to generate a Zipfian distribution with minimal communication. D-Zipfian [9] is a technique that addresses this challenge.

Our evaluation of these two paradigms in the context of the BG benchmark shows that D^{2}B scales superlinearly as a function of nodes, producing a larger number of requests per unit of time as we increase the number of nodes. However, it is not able to evaluate cache augmented data stores such as the one in Figure 1 for those applications that replicate data across caches and exercise CORE to ensure consistency. For these applications, IDB is more appropriate. A limitation of IDB is that it does not scale horizontally. It generates the highest number of requests per unit of time with one node. With two or more nodes, its rate of request generation is one half that of a single node. One may enhance scalability of IDB using a global clock across nodes to generate log records to implement BG's validation phase and eliminate global constraints such as requiring uniqueness of members issuing concurrent requests. This eliminates IDB's need for message passing to generate read actions, enabling it to scale as a function of nodes because social networking workloads are dominated by read actions. (According to [17], ratio of reads to writes is 500:1 at Facebook.)

The strengths of IDB and D^{2}B can be combined into a Hybrid. The main insight is that one may construct two groups of actions and use IDB for one group and D^{2}B for the other group. For example, the actions can be categorized into read and write actions. Next, the benchmark can be configured to use IDB when generating a read action and D^{2}B when generating a write action. This enables Hybrid to replicate data across caches of Figure 1. Writes issued by different nodes may not impact the same cached data item. However, they exercise the CORE to maintain all cached replicas (by read actions) consistent with one another. Note that an inverse of this example is also feasible with IDB for writes and D^{2}B for reads with a write-heavy workload. While this prevents replication of data across caches, it causes multiple benchmark nodes to generate write requests that may reference the same data.

While the alternative paradigms are general purpose and can be applied to alternative benchmarks, we focus on their use with BG [8, 6]. This is because its target application is interactive social networking actions that motivate the use of diverse data stores including cache augmented [39], in-memory graph databases [2], and extensible record stores [37, 20, 23]. Moreover, in addition to traditional performance metrics such as response time and throughput, BG quantifies the amount of unpredictable data (stale, inconsistent, or simply erroneous data) produced by a data store [8, 7, 3, 35, 38, 44, 50]. Hence, novel designs such as IDB and Hybrid must preserve this functionality of the benchmarking framework. Finally, there are other concurrent efforts in developing a social networking benchmarks such as [5, 26] that are similar to BG and can use our proposed paradigms and their tradeoffs.

The rest of this paper is organized as follows. In Section 2, we provide an overview of BG, its multi-node infrastructure to generate interactive social networking actions, and its validation phase. Section 3 presents the IDB, D^2B, and Hybrid execution paradigms. Section 4 provides a qualitative and a quantitative comparison of these paradigms. In Section 5, we present related work. Brief words of conclusion and future research directions are contained in Section 6.

2 Overview of BG

The conceptual model of BG's social graph has evolved to be complex and supports both simple and complex social networking actions. To simplify discussion and without loss of generality, this paper focuses on the core of the conceptual model that consists of members with profiles and their friendship with other members. This simple subset consists of one entity set named Members and two relationship sets named Invite and Friend. The Members entity set pertains to those with a profile registered with the social networking site. A member may extend a friendship invitation to another member, resulting in a Pending Friendship relationship from the inviter to the invitee. When the invitee accepts the pending friendship, a friendship relationship is formed between the two members and their pending friendship is removed.

BG benchmarks a data store by generating a synthetic database consisting of M *members* with a registered profile and ϕ friendships per member. Both the value of M and ϕ are configurable by an experimentalist. BG may be configured to generate a social graph either with or without images. With images, a member profile consists of a large (12 KB) profile image and a small (2 KB) thumbnail image. Images are not relevant because our focus is on BG's rate of request generation with alternative paradigms and their scalability characteristics.

BG Social Actions	99.9% Read	99% Read	90% Read
View Profile, VP	33.3%	33%	30%
List Friends, LF	33.3%	33.3%	30%
View Friend Req, VFR	33.3%	33.3%	30%
Invite Friend, IF	0.04%	0.4%	4%
Accept Friend Req, AFR	0.02%	0.2%	2%
Reject Friend Req, RFR	0.02%	0.2%	2%
Thaw Friendship, TF	0.02%	0.2%	2%

Table 1: Three mixes of interactive social networking actions.

Database parameters	
M	Number of members in the database.
ϕ	Number of friends per member.
ρ	Number of resources per member.
Workload parameters	
ϵ	Think time between actions constituting a session.
ψ	Inter-arrival time of 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	Duration 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.

BG consists of 17 actions [6]. In this study, we focus on 7 of these actions as shown in Table 1. Three are read actions: View Profile, List Friends, and View Pending Friendships. The remaining four are write actions: Accept Friend Request, Invite Friend, Reject Friend Request, Thaw Friendship. Except for View Pending Friendship, all actions are binary and have two members as input. The first is the *socialite*[1] that is performing the action (M_p) and the second is the member that is the target of the action (M_t). For example Socialite M_p may view the profile of M_t. These actions are detailed in [8, 6]. A key consideration is how to generate meaningful write requests, see Section 2.1 for details.

BG's workload generator implements a closed simulation model with a fixed number of threads T. (See Table 2 for a list of BG parameters and their definitions.) Each thread emulates a sequence of socialites performing a social action[2] shown in the first column of Table 1. While a database may consist of millions of members, at most T simultaneous socialites issue requests with BG's workload generator. The value of T is a configurable parameter that the experimentalist (or a rating mechanism [10, 12]) increases in order to impose a higher load on a data store. With T set to one, the experimentalist measures the *average service time* of the system to process a workload. With T set to values higher than one, the experimentalist establishes the *throughput* of a data store defined as the number of actions processed per second.

BG summarizes the performance of a data store using its Social Action Rating, SoAR[3], metric [8]. SoAR computes the highest throughput achievable by a data store while satisfying a pre-specified service level agreement, SLA. An SLA requires 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 10 minutes (Δ=600 seconds). An experimentalist may compare the SoAR of different systems

[1]A member actively engaged in a social networking action.

[2]BG also supports the concept of a session that consists of a sequence of actions, see [8, 6] for details.

[3]BG also computes the socialite rating of a data store, see [8, 6] for details.

with one another by specifying a fixed SLA.The system with the highest SoAR is the superior one.

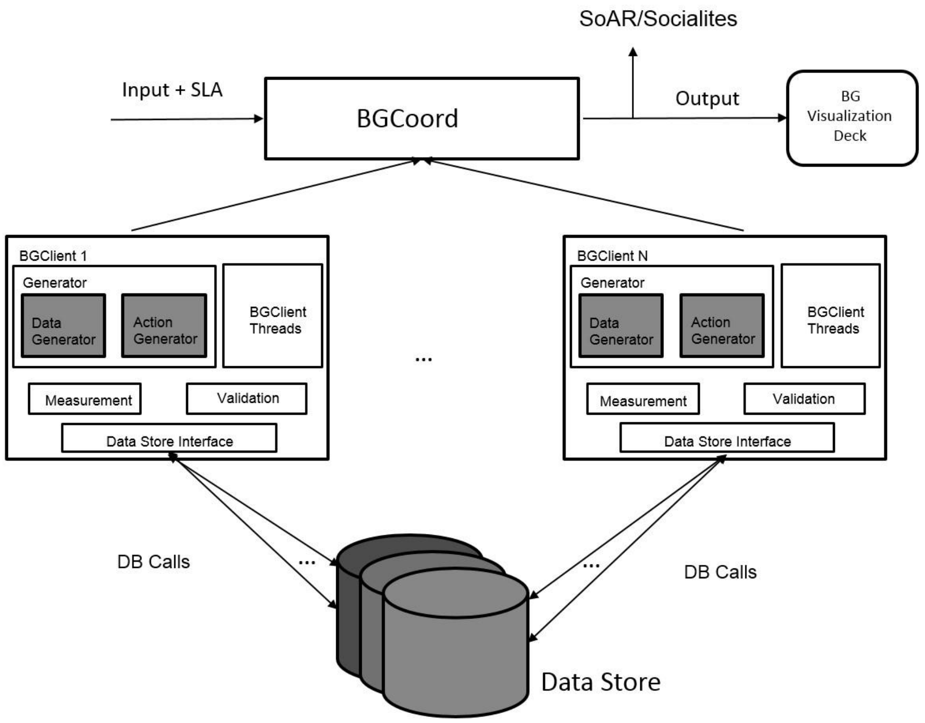

Figure 2: BG's shared-nothing architecture.

Figure 2 shows the software architecture of BG, consisting of a coordinator (BGCoord), a visualization deck, and N BGClients. BGCoord computes SoAR of a data store using a search technique [12]. The visualization deck enables an experimentalist to specify parameter settings for BGCoord, initiate rating of a data store, and monitor the rating process. Below, we focus on BGClient as it constitutes our focus.

BGClient is a configurable, extensible and modular component. We describe each in turn. An input file provided by BGCoord configures the Generator component of BGClient (see Figure 2) to produce both the data loaded into the data store (Data Generator) during the load phase and the workload issued to the data store (Action Generator) during the benchmarking phase. BGClient is extensible because one may introduce new actions besides the seven discussed in this paper. For example, in [28], we extended BG with complex analytics such as Get Shortest Distance or Compute Common Friends between two members. BGClient is modular because it consists of loosely-coupled components allowing each component to be modified with minimal impact on the rest of the system. For example, introducing new actions has no impact on the Data Generator and the Measurement components.

Table 1 shows three different workloads. Each is a different configuration file used to configure the Action Generator of BG. We use these three workloads throughout this study. A key property of these workloads is that they are symmetric in the sense that they issue the write actions in a manner to maintain the same number of friendships and pending friendships in the social graph. This is realized by satisfying the following two constraints: 1) % Invite Friend = % Reject Friend Request + % Accept Friend Request, and 2) % Thaw Friendship = % Accept Friend Request.

2.1 Meaningful Write Actions

A challenge with using multiple threads is how to ensure the write actions are meaningful. We discuss the challenge and our solution in turn. A BGClient maintains an in-memory state of the social graph to generate meaningful write actions. In essence, it is aware of the pending friendship requests and confirmed friendships between any two members and updates this information as it issues write actions. However, multiple threads may race with one another to convert actions intended to be meaningful into meaningless actions. An example may include (a) one thread ζ_1 emulating Member M_j rejecting a friend invitation by Member M_i while (b) another thread ζ_2 emulating Member M_j accepting M_i's friend invitation. In the context of each thread, the action is meaningful. However, depending on how the data store serializes these two concurrent threads (say ζ_1 before ζ_2) then one (ζ_2) becomes meaningless. To prevent such undesirable race conditions, a thread that generate a write action must acquire an exclusive lock on either one or both of its referenced members prior to issuing the write action. This section provides details assuming one multithreaded BGClient. Section 3 extends this discussion to N multithreaded BGClients using the alternative execution paradigms.

A thread of BGClient locks different members of the social graph per write action as follows:

- Invite Friend (IF) acquires exclusive locks on both its input members, the inviter (socialite M_p) and the invitee (M_t).
- Accept Friend Request (AFR) acquires exclusive lock on the invitee (M_p).
- Reject Friend Request (RFR) acquires exclusive lock on the invitee (M_p).
- Thaw Friendship (TF) acquires exclusive locks on both members (socialite M_p and M_t).

Both AFR and RFR lock only their socialite member (M_p), the invitee. Hence, a different socialite (thread ζ_1) may perform a write action on any of the target members of these actions. For example, two Accept Friend Requests such as AFR(M_2,M_1) and AFR(M_1,M_3) may execute simultaneously because their socialites (invitees M_2 and M_1) are different.

A BG thread releases its exclusive lock(s) on its member(s) once the data store completes the processing of its write action. While a thread is holding the write lock on a member of the social graph, another thread may not issue a write action using that same member. However, another thread may issue a read action using that same member. BG is configurable to ensure a socialite is unique across all threads at any instant in time. We do not discuss this feature of BG and refer the interested reader to [6] for details.

When a thread conflicts with another for a member, it does not wait as this would defeat the purpose of having additional threads impose a higher load. Instead, it selects another member for its write action. With a large number of threads, a thread may conflict with other threads repeatedly. It iterates through members until it finds an idle member and locks it to generate its action.

2.2 Validation and Unpredictable Data

A novel feature of BG is its ability to quantify the amount of unpredictable data produced by a data store. It is able to do so using its knowledge of the social graph, the write actions that transition the social graph from one state to another, and how its issued read and write actions overlap one another. Currently, the Validation component of the BGClient, see Figure 2, computes two metrics: percentage of unpredictable data and freshness confidence. The former is the percentage of read actions that observe a value other than what is expected based on a serial schedule of concurrent read and write actions [8]. The latter is the probability of a read action observing an accurate value a fixed amount of time, say t seconds, after an update occurs [49]. BG implements both using an offline technique, meaning, it computes them after the benchmarking phase has ended. This minimizes the amount of memory and computing resources required during the benchmarking phase. Below, we detail the offline technique by focusing on unpredictable data. We refer the reader to [6] for the details of the freshness confidence.

A BGClient computes the amount of unpredictable data as follows. (Section 3 describes extensions with the alternative multi-node execution paradigms.) During the benchmarking phase, each thread of a BGClient that invokes a read or a write action generates a log record and writes it to a read or a write log file. Each log record consists of a unique identifier, the action that produced it, the data item referenced by the action, and its start and end timestamp of the action (multiple log records can be created if multiple data items are modified see BG paper [8]). The read log record contains its observed value from the data store. The write log record contains the change to an existing value of its referenced data item. During the validation phase, for each read log record, a BGClient computes the write log records that overlap it. Given W overlapping write log records, there are (W+1)! possible serial schedules. Using its knowledge of the original state of the social graph and all the write actions issued prior to the read request, it computes the W possible values that the read log record should have observed. If none match the value observed by the read log record then the data store has produced an unpredictable data.

With a transaction processing system that guarantees serial schedules, a correct implementation of BG action produces no unpredictable data. This holds true using snapshot isolation.

3 Multi-Node Execution Paradigms

This section details BG's D^{2}B, IDB, and Hybrid as alternative shared nothing execution paradigms to generate requests. We present two implementations of IDB named Delegate and Retain. We report scalability of the SoAR of different paradigms when generating requests without a data store. The specified SLA requires 99% of actions to observe a response time faster than 100 milliseconds. In all experiments, the termination condition is reached because the observed throughput plateaus. The SLA is not violated because BG generates requests in a non-blocking manner and scales vertically. Lack of a data store means the reported SoAR and scalability characteristics are the empirical upper bound on BG's request generation rate.

3.1 Disjoint Databases, D^{2}B

Given N BGClients, the D^{2}B technique generates the benchmark social graph to consist of N self-contained social graphs and loads these onto the target data store. When rating the data store, it assigns each of the N *logical* social graphs to one of the BGClients. Each BGClient generates requests in parallel using its assigned social graph only. A BGClient does not reference members of a social graph assigned to a different BGClient. A good analogy of D^{2}B is to conceptualize a large metropolitan as N villages where members of each village are allowed to view each other's profile and form friendships with members of the same village only.

D^{2}B raises two design challenges that we describe in turn. First, to generate a realistic workload, BG must emulate scenarios where certain members are more active and participate either as

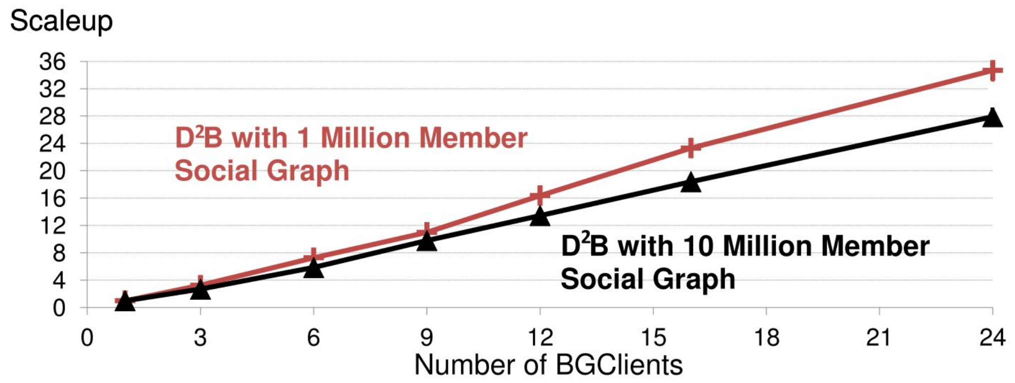

Figure 3: Scalability of the request generation rate of D^{2}B with a 1 Million and 10 Million social graph.

socialites or targets of those socialites more frequently than others. To do so, it employs the Zipfian distribution [51] and enables an experimentalist to generate different degrees of skew by manipulating the exponent (θ) of the distribution. The challenge is how to generate a skewed (i.e., Zipfian) distribution of references to members independent of N. This is desirable because it makes results gathered with a different number of BGClients comparable with one another. Second, BG must generate a sustained load in order to rate its target data store. The challenge is how to ensure different BGClients generate requests proportional to their processing capability such that all N BGClients start and finish at approximately the same time.

Both design decisions are addressed using a decentralized implementation of Zipfian named D-Zipfian [9]. Intuitively, D-Zipfian assigns a total probability of $\frac{1}{N}$ to each of the N BGClients and requires each generator to reference data items with a scaled probability. In the case of heterogeneous generators, the total probability of each generator is proportional to its processing capability. Empirical measurements of the chi-square statistic show the effectiveness of D-Zipfian in addressing both challenges, see [9] for details. Generation of member ids using a Zipfian distribution is CPU intensive and time consuming[4] and impacts BG's scalability characteristics. Its current implementation maintains an array of M elements for a social graph with M members. The elements of the array contain sorted values conditioned using the exponent (θ) of the Zipfian distribution. A binary search of these values using a randomly generated value produces member ids in a skewed manner. This process is highly parallelizable, scaling both vertically as a function of the number of cores and horizontally as a function of the number of nodes.

With one Google Compute node consisting of four cores, a single node BGClient generates a maximum of 276,000 and 266,000 requests per second with 1 million and 10 million member social graphs, respectively. These numbers are comparable even though one social graph is ten times larger than the other. This gap widens with an increased number of BGClients such that the maximum request generation rate with 24 nodes is 9.5 and 7.5 million requests per second with 1 million and 10 million members, respectively. (Recall there is no data store in these experiments with BG generating requests as fast as possible utilizing the four CPU cores of each node fully.) This is reflected in the horizontal scalability of D^{2}B as shown in Figure 3. This graph shows D^{2}B

[4]Profiling of one BGClient shows Zipfian constitutes more than 60% of request generation time.

scales superlinearly with both the 1 and 10 million social graphs because the social graph assigned to each BGClient becomes proportionally smaller with additional nodes, expediting request generation using the Zipfian distribution. The scalability with the 1 million member social graph is superior because there are fewer members, i.e., size of the array for the binary search is smaller.

The reported numbers are an upper bound on the request generation capability of BG because there is no data store and no network traffic (except for the initial coordination by BGCoord that is not reflected in the reported numbers). Hence, the processor speed dictates the rate of request generation. In order to increase BG's rate of request generation, one must either use processors with a higher processing capability or change the binary search with a more efficient search technique. A tree search technique with branching factor t changes the complexity of search from O($log_2 M$) to O($log_t M$).

The key advantage of D^{2}B is that N BGClients are coordinated (a) once at the beginning of the benchmarking phase with one round of message passing and (b) a second time at the end of the benchmark phase with another round of message passing. During the benchmarking phase, BGClients generate requests in parallel and independent of one another. Once they are done, each BGClient performs its validation phase independent of the other BGClients as its read actions do not conflict with the write actions of the other BGClients (because their assigned social graphs are mutually exclusive). This enables BG to scale horizontally to provide a proportionally higher throughput as a function of N.

To generate meaningful actions, each BGClient uses shared memory among threads to implement the non-blocking lock-based protocol of Section 2.1. It is trivial to use the same locking mechanism to ensure uniqueness of concurrent socialites (member ids) issuing requests. This models those applications where a user logins at most once to issue a social networking action.

The main limitation of D^{2}B is that it may not be suitable to evaluate those data store architectures that deploy a cache in their client component. See discussions of KOSAR, Figure 1, in Sections 1 and 4.1.

3.2 Delegate and Retain: IDB Techniques

We present two variants of Integrated DataBases (IDB) named Delegate and Retain. Given N BGClients, both variants assign each member to a node, requiring that node to serve as the *owner* of that member. This is implemented using a hash function applied to member id. When issuing an action, a BGClient (BGC_r) may reference a member owned by another BGClient (BGC_o). To implement the non-blocking lock-based protocol of Section 2.1 to ensure the action is meaningful, BGC_r exchanges messages with BGC_o. Retain and Delegate differ in that Retain requires BGC_r to issue the action while Delegate requires BGC_o to issue the action.

In their simplest form, Retain and Delegate generate a round-trip message for every action that involves two nodes. With the read actions, the messages enable the owner (BGC_o) to generate log records that facilitate the validation phase of BG to compute the amount of unpredictable data produced by a data store[5]. With the write actions, the messages enable the owner (BGC_o) to implement the protocol of Section 2.1 to generate meaningful actions.

Delegate generates fewer round-trip messages between BGClients when compared with Retain. With Retain, a BGClient thread obtains the exclusive lock on remote members referenced by its

[5]One may eliminate these messages assuming a global clock, see Section 4.2.

write action via message passing, issues the action to the data store, waits for the data store to process its action, and releases the exclusive lock on the remote members via message passing. With Delegate, the same thread requires the BGClient that owns the referenced member to issue the write action.

In our experiments, Delegate generates approximately 40% fewer messages per second when compared with Retain. However, Delegate may open significantly higher number of connections to the underlying data store than Retain. When the network bandwidth is not the bottleneck resource, Delegate produces a slightly ($< 10\%$) higher rate of requests than Retain.

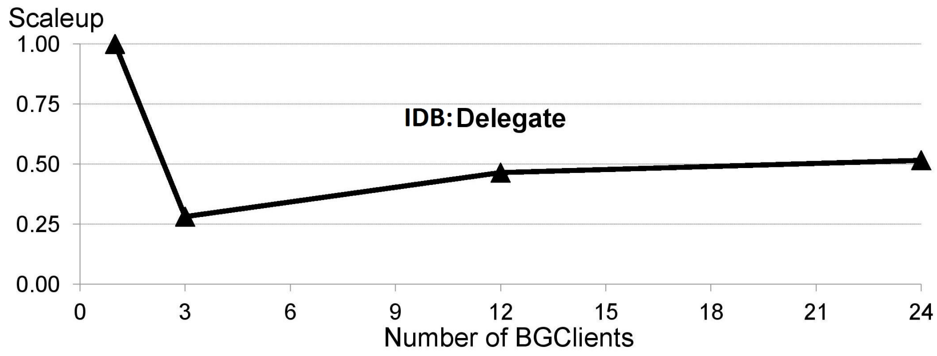

Figure 4: Scalability of IDB (Delegate) with a 1 Million member social graph.

The overhead of passing messages between BGClients is significant. It prevents both Retain and Delegate from providing a higher SoAR beyond one node. Figure 4 shows horizontal scalability of BG's SoAR as a function of the number of nodes with a 1 million member social graph. The observed SoAR is highest with one node at 163,000 actions per second. It drops dramatically to 48,000 actions per second with 3 nodes. Beyond 3 nodes, the scalability of SoAR improves relative to 3 nodes. However, even with 24 nodes, SoAR is one half that with one node.

Read actions are a significant portion ($\geq$90%) of the workload, see Table 1. One way to enhance the scalability of Retain and Delegate is to eliminate messages required to generate these actions. This requires global clocks across nodes [23] for generation of log records and dropping the requirement that concurrent socialites are unique. We consider this variant when comparing IDB with D^2B in Section 4.2.

3.2.1 Local versus Global Member ID Generation

When a BGC_1 references a remote member (say M_j) belonging to BGC_2, another concurrent thread may have obtained an exclusive lock on M_j. BG's software architecture is nonblocking[6] and BGC_2 must select a different member, say M_i, to generate an action. BG is configurable to require BGC_2 to generate M_i using its assigned members (termed *Local*) or using the Zipfian distribution that considers the entire social graph (termed *Global*). Global may select a member M_i

[6]In order to impose a sustained load on the data store.

that belongs to a different BGClient, resulting in additional messages. However, it approximates the experimentalist specified Zipfian distribution more accurately.

Figure 5 shows a chi-squared analysis of the distribution of member ids generated by Local and Global as a function of the number of BGClients. This was computed using a small social graph (M=10K) by issuing 1 million requests. The resulting distribution was compared with the analytical Zipfian (the mathematical distribution generated by $p_i(M,\theta) = \frac{\frac{1}{i^{(1-\theta)}}}{\sum_{m=1}^{M}(\frac{1}{m^{(1-\theta)}})}$) to compute its chi-squared. A lower chi-squared value means the resulting distribution is closer to the true distribution. As a comparison, we include the chi-squared analysis with an implementation that does not resolve conflicts. This is the lower bound on the reported deviation from the analytical Zipfian. Figure 5 shows Global is not sensitive to the number of nodes and approximates the lower bound. The chi-squared of local deviates from the lower bound as a function of the number of nodes. This deviation becomes smaller with larger social graphs, e.g., Global and Local are almost identical with M=1 million.

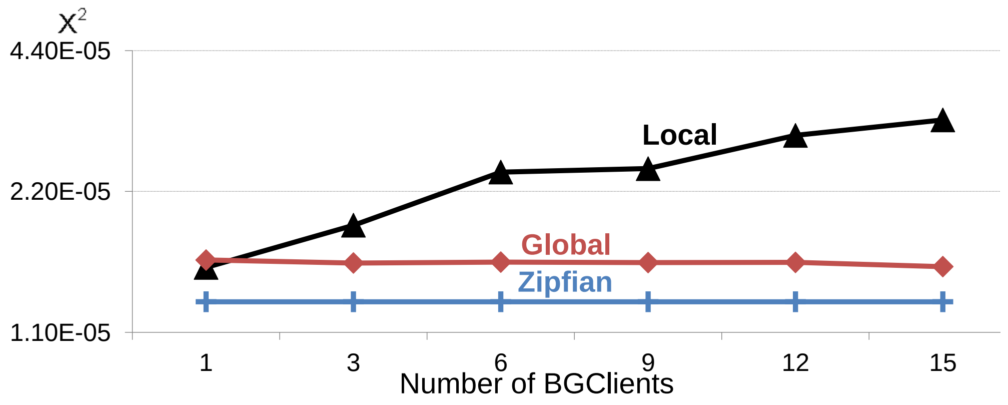

Figure 5: A chi-squared analysis of the distribution of member ids generated using Local and Global policies.

3.3 Hybrid

Hybrid combines IDB with D^{2}B by constructing two groups of actions and using each technique with a different group. It is based on the insight that the D-Zipfian (used with D^{2}B) can be used in combination with the regular Zipfian (used with IDB) to generate a request pattern that is the required Zipfian. It constructs the database using[7] D^{2}B. When benchmarking a data store, it first generates an action. If the action belongs to D^{2}B then it employs D-Zipfian to generate member ids. Otherwise, it uses Zipfian.

To illustrate, in order to benchmark the architecture of Figure 1, one may construct two different groups consisting of read and write actions. The read actions may be assigned to IDB, resulting in replication of data across the caches. The write requests may be assigned to D^{2}B producing write actions in a scalable manner while exercising the CORE component responsible for maintaining the different replicas consistent with one another.

[7]It cannot hash partition members across nodes because the read and write logs for a member may be generated by two different BGClients, requiring the shipment of the log records amongst BGClients to perform validation.

We performed a chi-squared analysis of the distribution of member ids generated by Hybrid (Zipfian for read actions and D-Zipfian for write actions) and one Zipfian distribution for all actions. Both distributions were computed using a small social graph (M=10K) by issuing 100 million requests. With the D-Zipfian distribution, we emulated three BGClients issuing requests proportionally. The chi-squared analysis compares each distribution with the the analytical Zipfian. These results show combining Zipfian with D-Zipfian produces a distribution comparable to using one Zipfian distribution.

4 A Comparison of D^{2}B with IDB

IDB is similar to D^{2}B in that they require each BGClient to be responsible for a fragment of the social graph. It is different than D^{2}B in several ways. First, it does not require the social graph to be loaded based on the value of N, i.e., the number of BGClients. Second, a BGClient may generate actions that reference members belonging to other BGClients. However, the overhead of message passing prevents both implementations of IDB, Delegate and Retain, from scaling as effectively as D^{2}B. Hybrid on the other hand combines techniques of both and is able to scale almost as effectively as D^{2}B, see Section 4.2 for details.

This section compares the D^{2}B, IDB and Hybrid paradigms both qualitatively and quantitatively. For the IDB , we only consider the Delegate approach. We do not discuss Retain because its throughput is comparable to Delegate with Delegate outperforming it slightly in all experiments.

4.1 A Qualitative Comparison

An experimentalist may use BG to compute the service time of a data store for processing different mixes of actions by configuring the benchmark to issue requests using one thread, or characterize the throughput of the system with a low, moderate, and a high system load by varying the number of threads used to generate requests. To establish the SoAR of a data store, an experimentalist must verify that the rate of request generation by BG is greater than the rate at which a data store can process requests while satisfying the specified SLA. In other words, BG cannot be the limiting factor when establishing the SoAR of a data store. Once this is verified, an experimentalist must select between the D^{2}B and IDB BG to evaluate a data store. The IDB BG is attractive because it is highly scalable. However, its assumption to construct N social graphs disjoint from one another is not suitable for an objective evaluation of all software architectures. One such architecture is shown in Figure 1. In this architecture the client JDBC driver embodies a cache that maintains the results of the different SQL queries issued by BG [32, 18, 30, 1]. When a query is issued, the JDBC driver checks the cache to see if its result is found in the cache. If not, it issues the query to the relational database management system (RDBMS) and caches the result set for future execution of the same query. In the presence of updates to the RDBMS, the framework uses its CORE to notify all caches to invalidate their impacted query result sets. To prevent undesirable race conditions that insert stale query result sets in the individual caches, the framework may use the IQ leases [33]. With this framework, when a JDBC client observes a cache miss, it must obtain an I lease prior to querying the database. Moreover, an update must obtain a Q lease in order to invalidate a query result set. These two leases enable the framework to guarantee strong consistency while providing enhanced performance for those workloads that exhibit a high read to write ratio.

To evaluate this cache augmented architecture, it is natural for an experimentalist to implement BGClients as shown in Figure 1, integrating the cache in each BGClient. With D^{2}B BG, this prevents multiple BGClients from issuing the same query, enforcing a single replica of a cached query and its result set. Moreover, this copy resides in the cache (BGClient) that created it. This renders the CORE that coordinates multiple caches (BGClients) with a replica of the same query redundant.

Retain forces all BGClients to use the same social graph, causing a frequently executed query to be cached across multiple BGClients. This means the CORE must maintain the identity of BGClients with a replica of a query and its result set. Moreover, every time an update invalidates the result of a query, the CORE must propagate this invalidation to all the caches (BGClients) with a replica of the query and its result set.

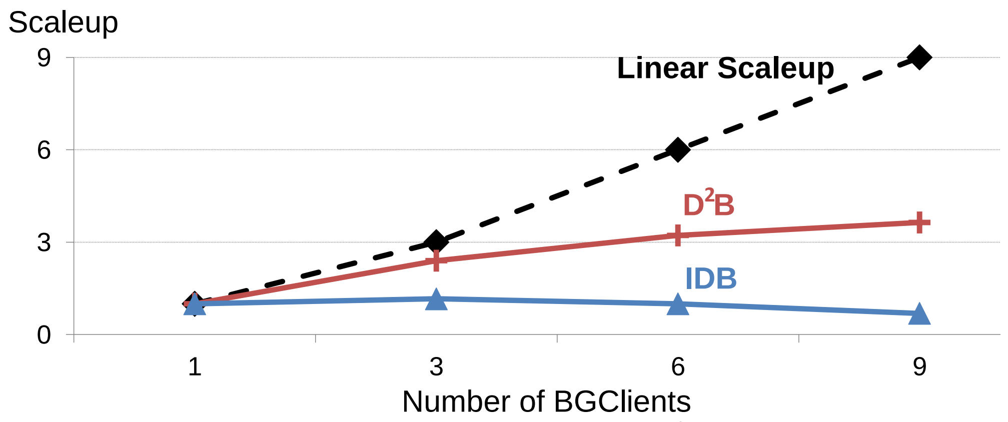

Figure 6: Scalability of KOSAR with D^{2}B and IDB.

Figure 6 shows the scalability of KOSAR when the number of BGClients is varied from 1 to 9. In these experiments, we eliminated the RDBMS from the picture, generating the results of queries deterministically and as fast as possible. This enables us to focus on the performance of the KOSAR architecture and its scalability. In this experiment, the CORE consists of four separate nodes and the social graph consists of 1 million members. (The number of nodes in this experimental platform varied from 5 to 13 nodes: 4 for KOSAR CORE and 1 to 9 for BGClients generating requests.) The system scales better with D^{2}B than with Retain. This is because, with Retain, the BGClients share one social graph and generate requests that the CORE detects as conflicting. With D^{2}B, different BGClients do not issue requests that conflict with one another. Moreover, with D^{2}B, each invalidation impacts only one BGClient while with Retain the same invalidation may impact all BGClients. This increased overhead as a function of additional BGClients causes Retain to report no change in throughput as a function of the number of BGClients. Since D^{2}B does not incur the same overhead, it shows the same configuration to scale sublinearly.

Whether D^{2}B or Retain is evaluating KOSAR [30] (and others similar to it such as EhCache [1] and JBoss [18]) objectively depends on the target application that is being modeled. Figure 1 shows each BGClient models an Application Server. If they reference data that is mutually exclusive by issuing a unique query and caching its result set then D^{2}B is appropriate. Otherwise, the IDB Retain is more appropriate.

The experimental results of Figure 1 focused on the KOSAR architecture removing the RDBMS from the evaluation. With an RDBMS, when the workload renders the RDBMS to be the bottleneck and dictates the scalability of the architecture, then D^{2}B, Delegate, and Retain would produce similar scalability results.

4.2 A Quantitative Comparison

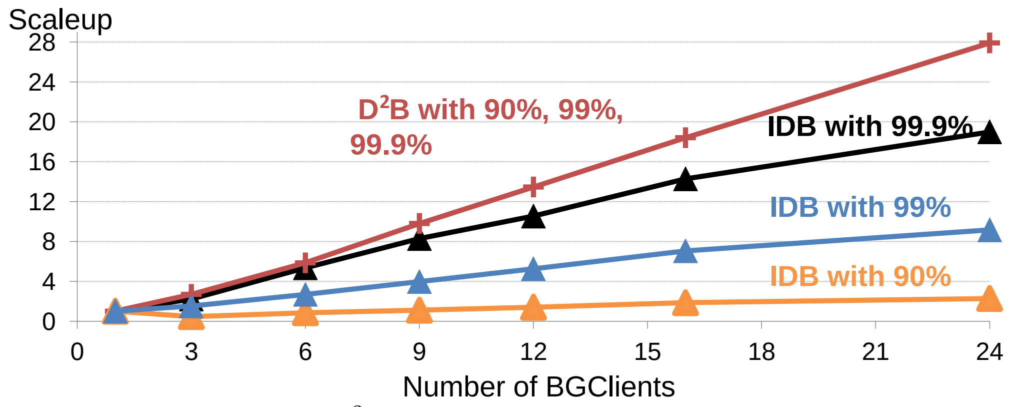

Figure 7: Horizontal scalability of D^{2}B and Delegate as a function of the number of nodes with a 1 million member social graph.

This section characterizes both the vertical and horizontal scalability of BG's alternative execution paradigms. Vertical scaling is the ability to produce a larger number of requests per unit of time by increasing the resources of a single node. In the context of BG, impactful resources include a node's number of CPU cores, number of network interface cards, and the amount of memory. Horizontal scaling is the ability of BG to use multiple nodes to generate a larger number of requests per unit of time. In analyzing these two forms of scalability, we discard a data store from our consideration, establishing the empirical upper bounds on BG's capabilities. A data store would distort the analysis, shifting the focus from the D^{2}B, Delegate and Retain. (See [8, 6, 14, 31] for use of BG with a data store.)

With vertical scaling, the alternative execution paradigms of Section 3 do not apply because the focus is on a single BGClient (the alternative paradigms apply in the context of multiple nodes and horizontal scalability). A BGClient is constrained by memory because it maintains data structures (arrays) at run time to generate stateful requests. The size of these data structures is a linear function of the number of members, M. The bandwidth of the network is important for communicating with a target data store. The messages exchanged to implement Delegate and Retain are relatively small in size (tens of bytes). The CPU is the main limiting resource. BG's highest request generation rate is established with the same number of threads as the number of cores. This means that one thread utilizes a single core fully, i.e., 100% utilization. With additional cores, additional threads utilize these cores fully due to several design decisions. First, our implementation of the BGClient does not employ Java classes such as Vector and Arraylist that constrain vertical scalability. Second, a BGClient employs an array of semaphores to enable multiple threads to manipulate elements of the shared array data structures (i.e., a Java container object). This is realized by hashing the index of the referenced array element to the array of semaphores to identify the semaphore to acquire. Similar to bloom filters, there might be collisions. However, these are minimized by adjusting the size of the array of semaphores depending on the number of threads and cores. Third, a BGClient does not block a thread when it cannot obtain an exclusive lock on a member or issue a valid write action. The thread is provided with a different member or is required to produce a different action. At the end of the experiment, a BGClient reports the mix of actions generated, enabling the experimentalist to decide whether the benchmarking results are consistent with the specified input. These design decisions enable a BGClient to scale vertically as a function of CPU cores.

With multiple nodes, with a fixed number of members, the rate of request generation with D^2B scales superlinearly as a function of the number of nodes, see Figure 7. This is due to the time complexity of our implementation of Zipfian becoming smaller with fewer members per BGClient, see discussions of Section 3.1 and Figure 3. Delegate and Retain do not scale horizontally. Increasing the number of nodes from 1 to 3 drops the rate of requests from 180,000 actions per second to 50,000 actions per second. Increasing the number of nodes to 24 increases this rate to 80,000 actions per second with minimal increase in the request generation rate beyond 24 nodes. This is due to Retain/Delegate's overhead of message passing to: 1) reserve members with write actions, and 2) generate log records with read actions. The overhead of these messages prevents Delegate and Retain from generating a higher number of requests as a function of additional nodes.

One may minimize the number of messages required by Delegate and Retain by using global clocks (GPS and atomic clocks [23]). This eliminates the need for message passing to generate log records for read actions. Each BGClient generates log records for its read actions using the global clock. Once the benchmarking phase completes, each BGClient that owns a member fetches the read log records pertaining to its members from other BGClients to perform its validation phase. Figure 7 shows this approach enables Delegate and Retain to scale sublinearly as a function of the number of nodes. They are closer to D^2B with a higher percentage of read actions (99.9%) because the write actions must continue to use message passing to obtain exclusive locks on their referenced members.

Hybrid paradigm is similar to IDB, except that it uses D-Zipfian distribution to eliminate messages associated with write operations. Generating members for write actions with a D-Zipfian distribution will eliminate the messages exchanged between BGClients to maintain the state of the social graph. Having two distributions: Zipfian to generate members for read operations and D-Zipfian to generate members for write operations enables Hybrid paradigm to scale linearly (using global clocks) without impacting its ability to generate read actions using one social graph.

IDB paradigm is scaling sublinearly as a result of the update actions, as seen in Figure 7. The scalability diminishes as we increase the ratio of write operations and the number of clients. Hybrid eliminates this messaging overhead by generating write actions similar to D^2B which enables it to scale linearly independent of the ratio of updates. Hybrid is not able to scale super-linearly as D^2B because increasing the number of clients will not minimize the overhead of the Zipfian generator as significantly as D^2B. The overhead of the Zipfian generator will be minimized as we increase number clients for members generated for write actions only.

Key feature	TPC-C [24]	RUBBoS [40]	RUBiS [21]	YCSB [22]	YCSB++ [41]	BG	LinkBench [5]	LDBC [26]
Unpredictable reads	✗	✗	✗	✗	✗	✓	✗	✗
Horizontally scalable	✓	✓	✓	✗	✓	✓	✗	✗
SLA	✓	✗	✗	✗	✗	✓	✗	✗
Inconsistency Window	✗	✗	✗	✗	✓	✓	✗	✗
Rating mechanism	✗	✗	✗	✗	✗	✓	✗	✗
Visualization Tool	✗	✓	✓	✗	✓	✓	✗	✗

Table 3: BG's Key features and a comparison with benchmarks for simple operation.

5 Related Work

Table 3 shows the key features of BG and its comparison with benchmarks for *simple* operations. We do not include benchmarks for complex operations/analytics that read and write either a large fraction of data or the entire data, e.g., BigBench [34], TPC-H [24], GenBase [47] to name a few.

We developed BG in 2012 and released a version of it in January 2013 as a part of our publication [8]. This was followed with three other social networking benchmarks: LinkBench [5], OLTPBench [25], and LDBC [26]. These benchmarks do not quantify the amount of unpredictable data produced by a data store. Moreover, they do not consider horizontal scalability of their architecture. The rest of this section provides an overview of these benchmarks and compares them with BG.

LinkBench [5] is a synthetic social networking benchmark based on traces from a production multi-node relational data store, MySQL. It is a (mostly) stateless parameterized workload that evaluates the persistent storage layer only, bypassing the memcached [39] and TAO [16] caching layers. In contrast, BG is a stateful benchmark that emulates the entire storage stack, including caches as illustrated in Section 4.1. BG and LinkBench reflect different design goals. BG's stateful requests generates temporal and spatial locality which is a dominant factor when considering the entire storage stack. LinkBench, on the other hand, is focused on post-cache with no interest in locality of references.

OLTPBench [25] is an extensible benchmark to generate workloads in support of simple operations. It includes an implementation of several benchmarks. Three of these are social networking benchmarks: LinkBench [5], Twitter, and Wikipedia. We discuss the last two as we have already discussed LinkBench. The Twitter workload is based on an anonymized snapshot of the Twitter social graph from August 2009 that contains 51 million users and 2 billion follows relationships. It consists of a synthetic workload generator that is based on an approximation of Twitter's daily use cases. The Wikipeida workload is based on the open source MediaWiki. It uses the real schema, transactions, and queries of the website. This benchmark models the most common operations in Wikipedia for article and "watchlist management. BG is different than both in that it is data store agnostic, enabling an experimentalist to design and implement a different data model for its conceptual schema and load it with an arbitrary number of members and friendships.

LDBC [26] is a synthetic social network benchmark. Similar to BG, an experimentalist may use this benchmark to generate social graphs of arbitrary size consisting of member profiles and their friendships. LDBC generates its write actions in an offline manner, producing streams of requests with dependencies on one another. These dependencies must be satisfied when issuing requests to generate meaningful actions. An implementation of this technique using shared memory and multiple threads is presented in [26], showing it scales vertically. However, the horizontal scalability of the proposed technique along with its message passing overhead is not clear. BG is different in that it employs synchronization primitives, e.g., exclusive locks on members, to generate requests in an online manner. Moreover, we consider both the vertical and horizontal scalability of request generation.

When compared with our own prior work, the Delegate and Retain are the novel contributions of this study. While D^{2}B was described in [8], we did not report on its horizontal scalability. In [10, 12], we detail BG's rating mechanism that computes the SoAR of a data store. This is an automated process that generates rounds of experiments that increase the number of threads T and compares the observed data store performance with the experimentalist specified SLA to rate the data store. With a workload that consists of write actions, the framework must reload the data store in each round. We specify three agile data loading techniques to expedite this process. This discussion is complementary to the presented execution paradigms.

We have used BG to evaluate and compare an industrial strength relational data store with a document store named MongoDB [8, 14], a graph data store named Neo4j [29], and an Object Relational Mapping framework named Hibernate [31]. We have also used BG to characterize the scalability characteristics of MongoDB and HBase [13]. We have also used BG to evaluate alternative techniques to implement feed following [11]. All these studies employ the D^{2}B paradigm introduced in [8]. In [27], we provided a mid-flight status update of the BG benchmark and articulated the need for alternative execution paradigms to the D^{2}B. Delegate and Retain of this study address this need. Moreover, we present a quantitative and a qualitative comparison with the D^{2}B design to enable an experimentalist to select between these alternatives.

6 Conclusion and Future Research

BG is a macrobenchmark to evaluate alternative novel database management systems for processing interactive social networking actions. Novel features of BG include its ability to quantify the amount of unpredictable data produced by a data store, rate a data store using the SoAR metric, and scale to rate the fastest data stores. This study focused on the shared-nothing architecture of BG, three execution paradigms to generate requests, and their tradeoffs. One of the techniques, D^{2}B, provides superlinear scalability as a function of the number of nodes. However, it is not appropriate for evaluating architectures that employ a client side cache, e.g., KOSAR [30], EhCache [1], and JBoss [18]. With these, the Retain technique of the IDB paradigm is more appropriate. (We also presented an intuitive variant of Retain named Delegate that is not appropriate for evaluating Client side caches.) A limitation of Retain is that it does not scale due to the overhead of message passing to synchronize generation of concurrent write actions that are meaningful, e.g., a socialite invokes Accept Friend Request on a member with a pending friendship invitation to the socialite. Retain's scalability is enhanced by assuming a global clock for generation of log records to measure the amount of stale data produced by a data store and eliminating global constraints such as the uniqueness of members emulated as socialites.

A future research direction is an in-depth analysis of more complex social networking actions such as news feed following [11].

References

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

[2] AMSDEN, Z., BRONSON, N., III, G. C., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., HOON, J., KULKARNI, S., LAWRENCE, N., MARCHUKOV, M., PETROV, D., PUZAR, L., AND VENKATARAMANI, V. TAO: How Facebook Serves the Social Graph. In *SIGMOD Conference* (2012).

[3] ANDERSON, E., LI, X., SHAH, M. A., TUCEK, J., AND WYLIE, J. J. What Consistency Does Your Key-value Store Actually Provide? In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability* (2010), HotDep'10, pp. 1–16.

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

[5] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *SIGMOD* (2013), pp. 1185–1196.

[6] BARAHMAND, S. Benchmarking Interactive Social Networking Actions, Ph.D. thesis, Computer Science Department, USC, 2014.

[7] BARAHMAND, S., AND GHANDEHARIZADEH, S. Benchmarking Correctness of Operations in Big Data Applications. In *MASCOTS 2014, Paris, France*.

[8] BARAHMAND, S., AND GHANDEHARIZADEH, S. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *Proceedings of 2013 CIDR* (January 2013).

[9] BARAHMAND, S., AND GHANDEHARIZADEH, S. D-Zipfian: A Decentralized Implementation of Zipfian. In *ACM SIGMOD DBTest Workshop* (2013).

[10] BARAHMAND, S., AND GHANDEHARIZADEH, S. Expedited Benchmarking of Social Networking Actions with Agile Data Load Techniques. *CIKM 2013* (2013).

[11] BARAHMAND, S., AND GHANDEHARIZADEH, S. Extensions of BG for Testing and Benchmarking Alternative Implementations of Feed Following. *ACM SIGMOD Workshop on Reliable Data Services and Systems (RDSS)* (June 2014).

[12] BARAHMAND, S., AND GHANDEHARIZADEH, S. *On Expedited Rating of Data Stores*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 1–32.

[13] BARAHMAND, S., GHANDEHARIZADEH, S., AND LI, J. On Scalability of Two NoSQL Data Stores for Processing Interactive Social Networking Actions, USC DBLAB Technical Report 2014-11, http://dblab.usc.edu/Users/papers/TwoNoSQL.pdf, 2014.

[14] BARAHMAND, S., GHANDEHARIZADEH, S., AND YAP, J. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. In *CIKM* (2013).

[15] BITTON, D., TURBYFILL, C., AND DEWITT, D. J. Benchmarking Database Systems: A Systematic Approach. In *VLDB* (1983), pp. 8–19.

[16] BRONSON, N., AND ET. AL. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC* (2013).

[17] BRONSON, N., LENTO, T., AND WIENER, J. L. Open Data Challenges at Facebook. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015* (2015), pp. 1516–1519.

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

[19] CATTELL, R. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec. 39* (May 2011), 12–27.

[20] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (2008).

[21] CONSORTIUM, O. S. M. RUBiS Benchmark, http://rubis.ow2.org.

[22] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing* (2010).

[23] CORBETT, J. C., AND ET. AL. Spanner: Google's globally-distributed database. In *OSDI* (2012).

[24] COUNCIL, T. P. P. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp.

[25] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRÉ-MAUROUX, P. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB 7*, 4 (2013), 277–288.

[26] ERLING, O., AVERBUCH, A., LARRIBA-PEY, J., CHAFI, H., GUBICHEV, A., PRAT, A., PHAM, M., AND BONCZ, P. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD* (2015).

[27] GHANDEHARIZADEH, S., AND BARAHMAND, S. A Mid-Flight Synopsis of the BG Social Networking Benchmark. In *Advancing Big Data Benchmarks* (2013).

[28] GHANDEHARIZADEH, S., BOGHRATI, R., AND BARAHMAND, S. An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions. *TPC Technology Conference* (September 2014).

[29] GHANDEHARIZADEH, S., BOGHRATI, R., AND BARAHMAND, S. An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions. In *TPCTC* (2014).

[30] GHANDEHARIZADEH, S., AND ET. AL. A Demonstration of KOSAR: An Elastic, Scalable, Highly Available SQL Middleware. In *ACM Middleware* (2014).

[31] GHANDEHARIZADEH, S., AND MUTHA, A. An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions. In *International Conference on Information Integration and Web-based Applications & Services* (2014).

[32] GHANDEHARIZADEH, S., AND YAP, J. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop* (June 2013).

[33] GHANDEHARIZADEH, S., YAP, J., AND NGUYEN, H. Strong Consistency in Cache Augmented SQL Systems. *Middleware* (2014).

[34] GHAZAL, A., RABL, T., HU, M., RAAB, F., POESS, M., CROLOTTE, A., AND JACOBSEN, H. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD* (2013), pp. 1197–1208.

[35] GOLAB W., RAHMAN, M. R., YOUNG, A. A., KEETON, K., WYLIE, J. J., AND GUPTA, I. Client-centric Benchmarking of Eventual Consistency for Cloud Storage Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), SOCC '13, pp. 28:1–28:2.

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

[37] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010).

[38] LIU, S., NGUYEN, S., GANHOTRA, J., RAHMAN, M. R., GUPTA, I., AND MESEGUER, J. Quantitative Analysis of Consistency in NoSQL Key-Value Stores. In *Quantitative Evaluation of Systems, 12th International Conference, QEST 2015, Madrid, Spain, September 1-3,*

2015, Proceedings (2015), pp. 228–243.

[39] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *NSDI* (2013).

[40] OBJECTWEB. RUBBoS: Bulletin Board Benchmark, http://jmob.ow2.org/rubbos.html.

[41] PATIL, S., POLTE, M., REN, K., TANTISIRIROJ, W., XIAO, L., LÓPEZ, J., GIBSON, G., FUCHS, A., AND RINALDI, B. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Cloud Computing* (New York, NY, USA, 2011), ACM.

[42] PATTERSON, D. For Better or Worse, Benchmarks Shape a Field. *Communications of the ACM 55* (July 2012).

[43] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI* (2010), USENIX.

[44] RAHMAN, M. R., GOLAB, W. M., AUYOUNG, A., KEETON, K., AND WYLIE, J. J. Toward a principled framework for benchmarking consistency. In *Proceedings of the Eighth Workshop on Hot Topics in System Dependability, HotDep 2012, Hollywood, CA, USA, October 7, 2012* (2012).

[45] SELTZER, M., KRINSKY, D., SMITH, K., AND ZHANG, X. The Case for Application Specific Benchmarking. In *HotOS* (1999).

[46] STONEBRAKER, M., AND CATTELL, R. 10 Rules for Scalable Performance in Simple Operation Datastores. *Communications of the ACM 54* (June 2011).

[47] TAFT, R., VARTAK, M., SATISH, N. R., SUNDARAM, N., MADDEN, S., AND STONEBRAKER, M. GenBase: A Complex Analytics Genomics Benchmark. In *SIGMOD* (2014), pp. 177–188.

[48] TERRACOTTA. BigMemory, http://terracotta.org/products/
bigmemory.

[49] WADA, H., FEKETE, A., ZHAO, L., LEE, K., AND LIU, A. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR 2011* (2011).

[50] WADA, H., FEKETE, A., ZHAO, L., LEE, K., AND LIU, A. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: The Consumers' Perspective. In *CIDR* (2011).

[51] ZIPF, G. K. Relative Frequency as a Determinant of Phonetic Change. Harvard Studies in Classified Philiology, Volume XL, 1929.