Expedited Rating of Data Stores Using Agile Data Loading Techniques*

Sumita Barahmand, Shahram Ghandeharizadeh

Database Laboratory Technical Report 2013-01

Computer Science Department, USC

Los Angeles, California 90089-0781

{barahman,shahram}@usc.edu

September 8, 2013

Abstract

To benchmark and rate a data store, one must repeat experiments that impose a different amount of load on the data store. Workloads that modify the benchmark database may require the same database to be loaded repeatedly. This may constitute a significant portion of the time to rate a data store. This paper presents several agile data loading techniques to expedite the rating process. These techniques include generating the disk image of the database once and re-using it, restoring the updated data items to their original value, maintaining in-memory state of the database across different experiments to avoid repeated loading of the database all together, and a hybrid of the third technique in combination with the other two. These techniques are general purpose and apply to a variety of cloud benchmarks. We investigate their implementation and evaluation in the context of one, the BG benchmark. Obtained results show a factor of two to twelve speedup in the rating process. As an example, when evaluating MongoDB with a million member BG database, we show these techniques expedite BG's rating from 4 months (123 days) of continuous running to less than 11 days for the first rating experiment. Subsequent ratings of MongoDB with different workloads using the same database is much faster, in the order of hours (less than 1 day).

A Introduction

To *rate* a data store is to compute a value that describes the performance of the data store on a specific hardware platform for a workload. An example value is the maximum processing capability (throughput)

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

of the data store. To quantify this value, one conducts experiments with the amount of imposed load as the only variable that changes across experiments. For example, with YCSB [7], YCSB++ [15], and BG [3], one may vary the number of threads (T) across the experiments to control the amount of load. With a higher value of T, the benchmark generates a higher load for the data store.

With those workloads that change the state of the database (its size, characteristics[1] or storage space[2]), one might be required to destroy and reconstruct the database at the beginning of each experiment to obtain meaningful ratings. For example, Workload D of YCSB inserts new records into a data store, increasing the database size. Use of this workload across different experiments with a different number of threads causes each experiment to impose its workload on a larger database size. If the data store becomes slower as a function of the database size then the observed trends cannot be attributed to the different amount of offered load (T) solely. Instead, they must be attributed to both a changing database size (difficult to quantify) and the offered load. To avoid this ambiguity, one may recreate the same database at the beginning of each experiment. This repeated destruction and creation of the same database may constitute a significant portion of the rating process. As an example, the time to load a modest sized BG database consisting of 10,000 members with 100 friends and 100 resources per member is 2 minutes with MongoDB. With an industrial strength relational database management system (RDBMS) using the same hardware platform, this time increases to 7 minutes. With MySQL, this time is 15 minutes, see Appendix A for details. If the rating of a data store conducts 10 experiments, the time attributed to loading the data store is ten times the reported numbers, motivating this study of agile data load techniques to expedite the rating mechanism.

We study the following data loading techniques. The first technique, named Database Image Loading, *DBIL*, relies on the capability of a data store to create a disk image of the database. DBIL uses this image repeatedly across different experiments. The second technique, named *RepairDB*, restores the database to its original state prior to the start of an experiment. Our proposed implementation of RepairDB is agnostic to a data store and does not require a database recovery technique. Depending on the percentage of writes and the data store characteristics, RepairDB may be slower than DBIL.

The third technique, named *LoadFree*, does not load the database in between experiments. Instead, it requires the benchmarking framework to maintain the state of the database in its memory across different experiments. In order to use LoadFree, the workload and its target data store must satisfy several conditions. One requirement is for the workload to be symmetric: It must issue write actions that negate one another in the long run. An example symmetric workload with BG issues Thaw Friendship (TF) action as frequently as Invite Friend (IF) and Accept Friend Request (AFR). The TF action negates the other two actions across

[1] Running BG with an asymmetric workload may result in some users having a larger number of friends compared to the others. Use of this kind of a workload across different back to back experiments results in each experiment to impose its workload on a different data set.

[2] Using workloads that insert and delete data from the data store, the underlying storage space of the data store may change from the initial state in such a way that performance of an experiment may differ greatly from a previously executed experiment.

repeated experiments. This prevents both an increased database size and the possible depletion of the benchmark database from friendship relationships to thaw. See Section E for other conditions that govern the use of LoadFree.

In scenarios where LoadFree cannot be used for the entire rating of a data store, it might be possible to use LoadFree in several experiments and restore the database using either DBIL or RepairDB. The benchmarking framework may use this *hybrid* approach until it rates its target data store. Section F shows the hybrid approaches provide a factor of five to twelve speedup in rating a data store.

The primary contribution of this paper is several agile data loading techniques for use with the cloud benchmarks. We describe an implementation of each with BG.

Related work: The overhead of loading a benchmark database is a recognized topic by practitioners dating back to Wisconsin Benchmark [5, 8] and 007 [6, 18, 19], and by YCSB [7] and YCSB++ [15] more recently. YCSB++ [15] describes a bulk loading technique that utilizes the high throughput tool of a data store to directly process its generated data and store it in an on-disk format native to the data store. This is similar to our DBIL technique. DBIL is different in two ways. First, DBIL does not require a data store to provide such a tool. Instead, it assumes the data store provides a tool that creates the disk image of the benchmark database once its loaded onto the data store for the very first time. This image is used in subsequent experiments. Second, DBIL accommodates complex schemas similar to BG's schema. Both RepairDB and LoadFree are novel and apply to data stores that do not support either the high throughput tool of YCSB++ or the disk image generation tool of DBIL. They may be adapted and applied to other benchmarking frameworks that rate a data store similar to BG.

The rest of the paper is organized as follows. Section B provides an overview of the rating framework. Subsequently, Section C through E use this framework to detail the 3 techniques in turn. We evaluate these techniques in Section F. Brief conclusion and future research direction are presented in Section G.

B BG Rating Framework

This section provides an overview of BG's framework to rate a data store. This framework is general purpose and can be used in combination with both YCSB and YCSB++. This description concludes this section.

BG [3] rates a data store for processing interactive social networking actions. It computes either a Social Action Rating (SoAR) or a Socialites rating of a data store given a pre-specified service level agreement (SLA). An example SLA may require 95% of actions that constitute a workload to be performed faster than 100 msec with no more than 0.1% unpredictable data for 20 minutes. SoAR is the highest throughput of a data store that satisfies this SLA. Socialites is the highest number of threads that satisfies this SLA.

Table 1 shows the eleven social actions and their frequency of occurrence for a workload used in this paper. These simple actions either read or write a small amount of data from a BG database. This database

BG Social Actions	Type	High (10%) Write
View Profile	Read	35%
List Friends	Read	5%
View Friends Requests	Read	5%
Invite Friend	Write	4%
Accept Friend Request	Write	2%
Reject Friend Request	Write	2%
Thaw Friendship	Write	2%
View Top-K Resources	Read	35%
View Comments on Resource	Read	10%
Post Comment on a Resource	Write	0%
Delete Comment from a Resource	Write	0%

Table 1: A BG workload consisting of a mix of social networking actions.

starts with M members, each with ϕ friends and ρ resources. The write actions of a workload may modify the friendship relationship between members, post comments on resources, or both.

BG scales to a large number of nodes using a shared-nothing architecture to rate the fastest data stores. It does so by logically dividing a social network into N unique communities and assigning each to a different BGClient process. After the first loading of the database, our decentralized implementation [3, 4] enables each BGClient to either use LoadFree by maintaining an in-memory state of its unique assigned community or use DBIL/RepairDB to restore the database to its original state prior to each rating experiment. In addition, it enables each BGClient to either create or generate a workload for its unique community independently. A coordinator, BGCoord, issues commands to BGClients to either create BG's schema, construct a database and load it, or generate a workload for the data store. A BGListener on each node facilitates communication between BGCoord and its spawned BGClient. One may host multiple BGListeners on different ports on a node. A configuration file informs the BGCoord of the different BGListeners and their ports. (It is possible to extend BGClient with the functionality provided by BGListener to eliminate the BGListener all together, see Section G.)

BGCoord conducts several experiments, each with a fixed number of threads T, to compute the SoAR and Socialites rating of a data store. With the SoAR rating, BGCoord uses heuristic search to control the value of T. In [3], we provide details and show the number of conducted experiments is a function of the true SoAR rating of the data store. When this value is in the order of a few thousands, BGCoord conducts between 10 to 20 unique[3] experiments. When the value is in the order of one hundred million, BGCoord may conduct between 50 to 60 unique experiments. The next 3 sections describe three different approaches to expedite the rating process.

To adapt this framework for use with both YCSB and YCSB++, one may relax BG's SLA by specifying

[3]BGCoord caches the results of different experiments and looks up their observed throughput when the heuristic search attempts to repeat an experiment with the same T value.

a high tolerable response time, e.g. max integer. This is because both YCSB and YCSB++ lack the concept of SLA. In addition, extra software is required to communicate the observed throughput between the YCSB/YCSB++ client(s)[4] and their respective BGListener. This enables BGCoord to gather the throughput observed by each client to compute the overall throughput observed from a data store.

C Database Image Loading

Various data stores provide specialized interfaces to create a "disk image" of the database [14]. Ideally, the data store should provide a high-throughput external tool [15] that the benchmarking framework employs to generate the disk image. Our target data stores (MongoDB, MySQL, an industrial strength RDBMS named[5] SQL-X) do not provide such a tool. Hence, our proposed technique first populates the data store with benchmark database and then generates its disk image. This produces one or more files (in one or more folders) stored in a file system. A new experiment starts by shutting down the data store, copying the files as the database for the data store, and restarting the data store. This technique is termed Database Image Loading, *DBIL*. In our experiments, it improved the load time of MongoDB with 1 million members with 100 friends and 100 resources per member by more than a factor of 400.

With DBIL, the load time depends on how quickly the system copies the files pertaining to the database. One may expedite this process using multiple disks, a RAID disk subsystem, a RAM disk or even flash memory. We defer this analysis to future work. Instead, in the following, we assume a single disk and focus on software changes to implement DBIL using BG.

Our implementation of DBIL utilizes a disk image when it is available. Otherwise, it first creates the database using the required (evaluator provided) methods[6]. Subsequently, it creates the disk image of the database for future use. Its implementation details are specific to a data store. Below, we present the general framework. For illustration purposes, we describe how this framework is instantiated in the context of MongoDB. At the time of this writing, an implementation of the general framework is available with MongoDB, MySQL and SQL-X.

We implemented DBIL by extending BGCoord and introducing a new slave component that runs on each server node (shard) hosting an instance of the data store. (The BGClient and BGListener are left unchanged.) The new component is named *DBImageLoader* and communicates with BGCoord using sockets. It performs operating system specific actions such as copy a file, and shutdown and start the data store instance running on the local node.

When BGCoord loads a data store, it is aware of the nodes employed by the data store. It contacts the DBImageLoader of each node with the parameters specified by the load configuration file such as the

[4] YCSB supports one client.

[5] Due to licensing restrictions, we cannot disclose the name of this system.

[6] With BG, the methods are insertEntity and createFriendship. With YCSB, this method is insert.

number of members (M), number of friends per member(ϕ), number of BGClients (N), number of threads used to create the image (T_{Load}), etc. The DBImageLoader uses the values specified by the parameters to construct a folder name containing the different folders and files that correspond to a shard. It looks up this folder in a pre-specified path. If the folder exists, DBImageLoader recognizes its content as the disk image of the target store and proceeds to shutdown the local instance of the data store, copy the contents of the specified folder into the appropriate directory of the data store, and restarts the data store instance. With a sharded data store, the order in which the data store instances are populated and started may be important. It is the responsibility of the programmer to specify the correct order by implementing the "MultiShardLoad" method of BGCoord. This method issues a sequence of actions to the DBImageLoader of each server to copy the appropriate disk images for each server and start the data store server.

As an example, a sharded MongoDB instance consists of one or more Configuration Servers, and several Mongos and Mongod instances [13]. The Configuration Servers maintain the metadata (sharding and replication information) used by the Mongos instances to route queries and perform write operations. It is important to start the Configuration Servers prior to Mongos instances. Next, the shards (Mongod instances) are attached to the data store cluster. The programmer specifies this sequence of actions by implementing "MultiShardStart" and "MultiShardStop" methods of BGCoord.

D Repair Database

Repair Database, *RepairDB*, marks the start of an experiment (T_{Start}) and, at the end of the experiment, it employs the point-in-time recovery [12, 11] mechanism of the data store to restore the state of the database to its state at T_{Start}. This enables the rating mechanism to conduct the next experiment as though the previous benchmark database was destroyed and a new one was created. It is appropriate for use with workloads consisting of infrequent write actions. It expedites the rating process as long as the time to restore the database is faster than destroying and re-creating the same database.

In our experiments (see Section F), RepairDB was consistently slower than DBIL. Hence, RepairDB is appropriate for use with those data stores that do not provide a DBIL feature (or with those experiments where RepairDB is faster than DBIL).

With those data stores that do not provide a point-in-time recovery mechanism, the benchmarking framework may implement RepairDB. Below, we focus on BG and describe two alternative implementations of RepairDB. Subsequently, we extend the discussion to YCSB and YCSB++.

The write actions of BG impact the friendship relationships between the members and post comments on resources. BG generates log records for these actions in order to detect the amount of unpredictable[7] data during its validation phase at the end of an experiment. One may implement point-in-time recovery by

[7] See Section E for the definition of unpredictable data.

No. of friends per member (ϕ)	Speedup Factor
10	12
100	7
1000	2

Table 2: Factor of improvement in load times with RepairDB when compared with re-creating the entire database, target data store is MongoDB, M=100K, ρ=100.

using these log records (during validation phase) to restore the state of the database to the beginning of the experiment.

Alternatively, BG may simply drop existing friendships and posted comments and recreate friendships. When compared with creating the entire database, this eliminates reconstructing members and their resources at the beginning of each experiment. The amount of improvement is a function of the number of friendships per member as the time to recreate friendship starts to dominate the database load time. Table 2 shows RepairDB improves the load time of MongoDB[8] by at least a factor of 2 with 1000 friends per member. This speedup is higher with fewer friends per member as RepairDB is rendered faster.

BG's implementation of RepairDB must consider two important details. First, it must prevent race conditions between multiple BGClients. For example, with an SQL solution, one may implement RepairDB by requiring BGClients to drop tables. With multiple BGClients, one succeeds while others encounter exceptions. Moreover, if one BGClient creates friendships prior to another BGClient dropping tables then the resulting database will be wrong. We prevent undesirable race conditions by requiring BGCoord to invoke only one BGClient to destroy the existing friendships and comments.

Second, RepairDB's creation of friendships must consider the number (N) of BGClients used to create the self contained communities. Within each BGClient, the number of threads (T_{load}) used to generate friendships simultaneously is also important. To address this, we implement BGCoord to maintain the original values of N and T_{load} and to re-use them across the different experiments.

Extensions of YCSB and YCSB++ to implement RepairDB is trivial as they consist of one table. This implementation may use either the point-in-time recovery mechanism of a data store or generate log records similar to BG.

E Load Free Rating

With Load Free, the rating framework uses the same database across different experiments as long as the *correctness* of each experiment is preserved. Below, we define correctness. Subsequently, we describe extensions of the framework of Section B to implement LoadFree.

[8]The factor of improvement with MySQL is 3.

Correctness of an experiment is defined by the following three criteria. First, the mix of actions performed by an experiment must match the mix specified by its workload. In particular, it is unacceptable for an issued action to become a no operation due to repeated use of the benchmark database. For example, with both YCSB and YCSB++, a delete operation must reference a record that exists in the database. It is unacceptable for an experiment to delete a record that we deleted in a previous experiment. A similar example with BG is when a database is created with 100 friends per member and the target workload issues Thaw Friendship (TW) more frequently than creating friendships (combination of Invite Friend and Accept Friend Request). This may cause BG to run out of the available friendships across several experiments using LoadFree. Once each member has zero friends, BG stops issuing TW actions as there exist no friendships to be thawed. This may introduce noise by causing the performance results obtained in one experiment to deviate from their true value. To prevent this, the workload should be symmetric such that the write actions negate one another. Moreover, the benchmarking framework must maintain sufficient state across different experiments to issue operations for the correct records.

Second, repeated use of the benchmark database should not cause the actions issued by an experiment to fail. As an example, workloads D and E of YCSB insert a record with a primary key in the database. It is acceptable for an insert to fail due to internal logical errors in the data store such as deadlocks. However, failure of the insert because a row with the same key exists is not acceptable. It is caused by repeated use of the benchmark database. Such failures pollute the response times observed from a data store as they do not perform the useful work (insert a record) intended by YCSB. To use LoadFree, the uniqueness of the primary key must be preserved across different experiments using the same database. One way to realize this is to require the core classes of YCSB to maintain sufficient state information across different experiments to insert unique records in each experiment.

Third, the database of one experiment should not impact the performance metrics computed by a subsequent experiment. In Section A, we gave an example with YCSB and the database size impacting the observed performance. As another example, consider BG and its metric to quantify the amount of unpredictable reads. This metric pertains to read actions that observe either stale, inconsistent, or wrong data. For example, the design of a cache augmented data store may incur dirty reads [10] or suffer from race conditions that leave the cache and the database in an inconsistent state [9], a data store may employ an eventual consistency [17, 16] technique that produces either stale or inconsistent data for some time [15], and others. Once unpredictable data is observed, the in-memory state of database maintained by BG is no longer consistent with the state of the database maintained by the data store. This prevents BG from accurately quantifying the amount of stale data in a subsequent experiment. Hence, once unpredictable data is observed in one experiment, BG may not use LoadFree in a subsequent experiment. It must employ either DBIL or RepairDB to recreate the database prior to conducting additional experiments.

LoadFree is very effective in expediting the rating process (see Section F) as it eliminates the load time

between experiments. One may violate the above three aforementioned criterion and still be able to use LoadFree for a BG workload. For example, a workload might be asymmetric by issuing Post Comment on a Resource (PCR) but not issuing Delete Comment from a Resource (DCR). Even though the workload is asymmetric and causes the size of the database to grow, if the data store does not slow down with a growing number of comments (due to use of index structures), one might be able to use LoadFree, see Section G. In the following, we detail BG's implementation of LoadFree.

To implement LoadFree, we extend each BGClient to execute either in *one time* or *repeated* mode. With the former, BGListener (see Section B) starts the BGClient and the BGClient terminates once it has either executed for a pre-specified[9] amount of time or has issued a pre-specified number of requests [7, 15]. With the latter, once BGListener starts the BGClient, the BGClient does not terminate and maintains the state of its in-memory data structures that describe the state of the database. The BGListener relays commands issued by the BGCoord to the BGClient using sockets.

We extend BGCoord to issue the following additional[10] commands to a BGClient (via BGListener): reset and shutdown. BGCoord issues the reset command when it detects a violation of the three aforementioned criteria for using LoadFree. The shutdown command is issued once BGCoord has completed the rating of a data store and has no additional experiments to run using the current database.

In between experiments identified by EOE commands issued by BGCoord, BGClient maintains the state of its in-memory data structures. These structures maintain the pending and confirmed friendship relationships between members along with the comments posted on resources owned by members. When an experiment completes, the state of these data structures is used to populate the data structures corresponding to the initial database state for the next experiment. BGClient maintains both initial and final database state to issue valid actions (e.g., Member A should not extend a friendship invitation to Member B if they are already friends) and quantify the amount of unpredictable data at the end of each experiment, see [3] for details.

F An Evaluation

This section quantifies the speedup observed with the proposed 3 techniques using the workload in Table 1. In addition, we consider two hybrids: 1) LoadFree with DBIL and 2) LoadFree with RepairDB. These capture scenarios where one applies LoadFree for some of the experiments and reloads the database in between. In the following, we start with an analytical model that describes the total time required by BG to rate a data store. Next, we describe how this model is instantiated by the data loading techniques. We

[9] Described by the workload parameters.

[10] Prior commands issued using BGListener include: create schema, load database and create index, Execute One Experiment (EOE), construct friendship and drop updates. The EOE command is accompanied by the number of threads and causes BG to conduct an experiment to measure the throughput of the data store for its specified workload (by BGCoord). The last two commands implement RepairDB.

Database parameters	
M	Number of members in the database.
ϕ	Number of friends per member.
ρ	Number of resources per member.
Workload parameters	
O	Total number of sessions emulated by the benchmark.
ϵ	Think time between social actions constituting a session.
ψ	Inter-arrival time between users emulated by a thread.
θ	Exponent of the Zipfian distribution.
Service Level Agreement (SLA) parameters	
α	Percentage of requests with response time $\leq \beta$.
β	Max response time observed by α requests.
τ	Max % of requests that observe unpredictable data.
Δ	Min length of time the system must satisfy the SLA.
Environmental parameters	
N	Number of BGClients.
T	Number of threads.
δ	Duration of the rating experiment.
Incurred Times	
ζ	Amount of time to create the database for the first time.
ν	Amount of time to recreate the database in between experiments.
η	Number of rating experiments conducted by BGCoord.
ω	Number of times BGCoord loads the database.
Λ	Total rating duration.

Table 3: BG's parameters and their definitions.

M	Action	DBIL	RepairDB	LoadFree	LoadFree + DBIL	LoadFree + RepairDB
100K	ζ	165	157	157	165	157
	ν	8	26	0	1.9	6.4
	Λ	290	481	200	228	270
500K	ζ	361	351	351	361	351
	ν	10	165	0	2.5	41.2
	Λ	514	2205	394	431	847
1000K	ζ	14804	14773	14773	14804	14773
	ν	31	588	0	7.75	147
	Λ	15188	21284	14816	14932	16433

Table 4: BG's rating of MongoDB with 1 BGClient using workload of Table 1, ϕ=100, ρ=100, δ=3 minutes, Δ=10 minutes, and η=11. All reported durations are in minutes. The hybrid techniques used either DBIL or RepairDB for approximately 25% of the loading experiments.

M	DBIL	RepairDB	LoadFree	LoadFree + DBIL	LoadFree + RepairDB
100K	6.8	4	9.6	8.7	7
500K	8.4	1.9	10.7	10.1	5
1000K	11.7	8	12	11.9	10.8

Table 5: Observed speedup (S) when rating MongoDB.

present the observed enhancements and conclude by quantifying the observed speedup relative to not using the proposed techniques.

With BG, the time required to rate a data store depends on:

- The very first time to create the database schema and populate it with data. This can be done either by using BGClients to load BG's database or by using high throughput tools that convert BG's database to an on-disk native format of a data store. We let ζ denote the duration of this operation. With DBIL, ζ is incurred when there exists no disk image for the target database specified by the workload parameters M, ϕ, and ρ, and environmental parameter N and others. In this case, the value of ζ with DBIL is higher than RepairDB because, in addition to creating the database, it must also create its disk image for future use, see Table 4.

- The time to recreate the database in between rating experiments, ν. With DBIL and RepairDB, ν should be a value less than ζ. Without these techniques, ν equals ζ, see below.

- The duration of each rating experiment, δ.

- Total number of rating experiments conducted by BGCoord, η.

- Total number of times BGCoord loads the database, ω. This might be different than η with LoadFree and hybrid techniques that use a combination of LoadFree with the other two techniques.

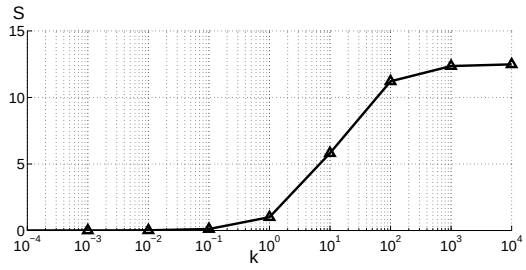

Figure 1: S as a function of k.

- The duration of the final rating round per the pre-specified SLA, Δ.

The total rating duration is:

$$\Lambda = \zeta + (\omega \times \nu) + (\eta \times \delta) + \Delta \tag{1}$$

With LoadFree, ω equals zero. The value of ω is greater than zero with a hybrid technique that combines LoadFree with either DBIL or RepairDB. The value of ν differentiates between DBIL and RepairDB, see Table 4. Its value is zero with LoadFree[11].

By setting ν equal to ζ, Equation 1 models a naïve use of BG that does not employ the techniques of this paper. Such a naïve technique would require 1927 minutes (1 day and eight hours) to rate MongoDB with 100K members. The third row of Table 4 shows this time is reduced to a few hours with the proposed techniques. This is primarily due to considerable improvement in load times, see the first two rows of Table 4. Note that the initial load time (ζ) with DBIL is longer because in addition to loading the database it must construct the disk image of the database.

The last six rows of Table 4 show the observed trends continue to hold true with databases consisting of 500K and 1 million members.

Techniques proposed in this paper enhance BG's load time only. An obvious question is what is the impact of this while leaving other pieces alone relative to the naïve use of BG (ν=ζ)? Amdahl's Law [1] provides the following answer:

$$S = \frac{1}{(1-f) + f/k} \tag{2}$$

where S is the observed speedup, f is the fraction of work in the faster mode, $f = \frac{\omega \times \zeta}{\Lambda}$, and k is speedup while in faster mode, $k = \frac{\zeta}{\nu}$. With LoadFree, ν is zero, causing k to become infinite. In this case, we compute speedup using a large integer value (maximum integer value) for k because S levels off with very large k values. Figure 1 illustrates this by showing the value of S as the value of f with 0.92 (1 million member database) using different k values.

Table 5 shows the observed speedup (S) for the experiments reported in Table 4. LoadFree provides

[11] With LoadFree, a value of ν higher than zero is irrelevant as ω equals zero.

Data Store	Action	DBIL	RepairDB	LoadFree	LoadFree+DBIL	LoadFree+RepairDB
MongoDB	ζ	165	157	157	165	157
	ν	8	26	0	1.9	6.4
	Λ	290	481	200	228	270
MySQL	ζ	2514	2509	2509	2514	2509
	ν	4.7	1206	0	1.2	302
	Λ	2613	15816	2552	2571	5868
SQL-X	ζ	158.5	153.5	153.5	158.5	153.5
	ν	5	30	0	1.3	7.5
	Λ	253	525	197	214	279

Table 6: BG's rating of MongoDB, MySQL and SQL-X with 1 BGClient using workload of Table 1, M=100K, ϕ=100, ρ=100, ω=11, δ=3 minutes, Δ=10 minutes, and η=11. All reported durations are in minutes. The hybrid techniques used either DBIL or RepairDB for approximately 25% of the loading experiments.

Data Store	DBIL	RepairDB	LoadFree	LoadFree+ DBIL	LoadFree+ RepairDB
MongoDB	6.8	4	9.6	8.7	7
MySQL	11.6	1.9	11.8	11.8	5
SQL-X	7.6	3.6	9.6	9	6.8

Table 7: Observed speedup (S) when rating MongoDB, MySQL and SQL-X with M=100K.

the highest speedup followed by DBIL and RepairDB. The hybrid techniques follow the same trend with DBIL outperforming RepairDB. Speedups reported in Table 5 are modest when compared with the factor of improvement observed in database load time between the very first and subsequent load times, compare the first two rows (ζ and ν) of Table 4. These results suggest the following: Using the proposed techniques, we must enhance the performance of other components of BG to expedite its overall rating duration. (It is impossible to do better than a zero load time of LoadFree.) A strong candidate is the duration of each experiment (δ) conducted by BG. Another is to reduce the number of conducted experiments by enhancing BG's heuristic search technique.

Reported trends with MongoDB hold true with both MySQL and an industrial strength RDBMS named [12] SQL-X. The time to load these data stores and rate them with 100K member database is shown in Table 6. While SQL-X provides comparable response time to MongoDB, MySQL is significantly slower than the other two. This enables BG's rating of MySQL to observe the highest speedups when compared with the naïve technique, see Table 7.

[12] Due to licensing restrictions, we cannot disclose the name of this system.

G Conclusion

With social networking companies continuing to introduce novel data stores to meet their requirements for interactive social networking actions (e.g., Cassandra and TAO [2] by Facebook and Voldemort by LinkedIn), it is important to have effective benchmarks to empower the designers of these systems to evaluate the performance tradeoffs associated with their solutions. Cloud benchmarking frameworks are ideal for this purpose. In order to be effective, they must rate a data store quickly. This paper presented several data load techniques to speedup the rating duration for workloads that require repeated loading of the benchmark database. One technique, DBIL, requires a data store to provide specialized interfaces to export the disk image of the database and to import it subsequently. In the absence of such interfaces, one may employ RepairDB that is agnostic to the underlying data store. LoadFree eliminates loading of data in between experiments all together. However, certain conditions must hold true for its use, see Section E for details. One may use LoadFree in combination with the other two techniques.

Our future research directions are as follows. First, we are developing a methodology to enable use of a hybrid technique (e.g., LoadFree+DBIL) with asymmetric workloads. The ideal methodology detects when LoadFree should not be employed by an experiment and switches to DBIL to ensure the integrity of the rating produced by the benchmarking framework. Second, we are analyzing an alternative implementation of RepairDB to improve its performance. It employs log records with additional metadata to *undo* write actions performed during an experiment and revert the database to its initial state. Third, based on our analysis of the observed speedup using Amdahl's law (see discussions of Table 5), we are improving the other components of the rating framework to expedite its performance. Fourth, we are analyzing other techniques such as the bulk data loading technique of YCSB++ [15] that utilizes the high throughput tool of a data store to generate an on-disk format of the benchmark database native to the data store. This analysis includes intermediate formats such as a comma separated CSV file or batching of insertion statements when creating the benchmark database. Finally, we are investigating alternative software architectures in order to reduce the number of components that constitute the rating framework without sacrificing its ease of use. An immediate candidate is to incorporate the functionality of the BGListener in the BGClient, eliminating the BGListener all together.

H Acknowledgments

We thank the anonymous reviewers of CIKM 2013 for their valuable comments.

References

[1] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Spring Joint Computer Confernece*, 30, 1967.

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

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

[4] S. Barahmand and S. Ghandeharizadeh. D-Zipfian: A Decentralized Implementation of Zipfian. In *Sixth International Workshop on Testing Database Systems*, 2013.

[5] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems A Systematic Approach. In *VLDB*, pages 8–19, 1983.

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

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

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

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

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

[11] D. Lomet and F. Li. Improving Transaction-Time DBMS Performance and Functionality. *International Conference on Data Engineering*, 2009.

[12] D. Lomet, Z. Vagena, and R. Barga. Recovery from "Bad" User Transactions. In *SIGMOD*, 2006.

[13] MongoDB. Sharded Cluster Administration, http://docs.mongodb.org/manual/administration/sharded-clusters/, 2011.

[14] MongoDB. Using Filesystem Snapshots to Backup and Restore MongoDB Databases, http://docs.mongodb.org/manual/tutorial/backup-databases-with-filesystem-snapshots/, 2011.

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

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

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

[18] J. Wiener and J. Naughton. Bulk Loading into an OODB: A Performance Study. In *VLDB*, 1994.

[19] J. Wiener and J. Naughton. OODB Bulk Loading Revisited: The Parititioned-List Approach. In *VLDB*, 1995.

Appendix

A Loading of BG using MySQL's InnoDB

With MySQL, we use its InnoDB as it provides ACID properties using row-level locking and imposes foreign key constrains, see http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html for details. We use MySQL Server 5.0 for the experiments in this paper.

BG specifies the primary key and foreign key constraints while creating the database schema, prior to loading the data. To load the data effectively, we changed several settings. First, we increased its communication packet (max_allowed_packet, see https://dev.mysql.com/doc/refman /4.1/en/packet-too-large.html), and its connection and result buffer (net_buffer_length, see http://dev.mysql.com/doc/refman/5.0/en/server-system-variables.html#sysvar _net_buffer_length). Second, we disabled InnoDB's ACID and constraint check features. We re-enable these features once the benchmark database is created and prior to rating MySQL. In addition, BG creates the index structures after the loading of the database is completed using statements similar to `create index friendship_inviteeID on Friendship(inviteeID)`.

With the above modifications, loading of data is improved dramatically. For example, the time required to load a 10,000 member BG database with 100 friends and 100 resources per member is improved from 913 minutes to 15 minutes.

To disable InnoDB's ACID transactional properties and constraint checking capabilities, we issued the following commands:

```
SET FOREIGN_KEY_CHECKS = 0;
SET UNIQUE_CHECKS = 0;
SET SESSION tx_isolation='READ-UNCOMMITTED';
```

To re-enable them, we issued the following commands:

```
SET UNIQUE_CHECKS = 1;
SET FOREIGN_KEY_CHECKS = 1;
SET SESSION tx_isolation='REPEATABLE-READ';
```

Moreover, we modified BGClient's init() function for MySQL to set autocommit to false. Its cleanup method commits pending transactions by invoking conn.commit() and sets autocommit to true.