On Expedited Rating of Data Stores*

Sumita Barahmand†, *Shahram Ghandeharizadeh*

Database Laboratory Technical Report 2014-09

Computer Science Department, USC

Los Angeles, California 90089-0781

December 10, 2015

Abstract

To rate a data store is to compute a value that describes the performance of the data store with a database and a workload. A common performance metric of interest is the highest throughput provided by the data store given a pre-specified service level agreement such as 95% of requests observing a response time faster than 100 milliseconds. This is termed the action rating of the data store. This paper presents a framework consisting of two search techniques with slightly different characteristics to compute the action rating. With both, to expedite the rating process, the framework employs agile data loading techniques and strategies that reduce the duration of conducted experiments. We show these techniques enhance the rating of a data store by one to two orders of magnitude. The rating framework and its optimization techniques are implemented using a social networking benchmark named BG.

*This paper is accepted to appear in the Springer Transactions on Large-Scale Data and Knowledge-Centered Systems 2016.

†Supported in part by the Google 2013 Ph.D. fellowship in Cloud Computing.

1 Introduction

1.1 Motivation

The landscape of data stores has expanded to include SQL, NoSQL, NewSQL, cache augmented, graph databases, and others. A survey of 23 systems is presented in [11] and we are aware of a handful more[1] since that survey. Some data stores provide a tabular representation of data while others offer alternative data models that scale out [12]. Some may sacrifice strict ACID [16] properties and opt for BASE [11] to enhance performance. Independent of a qualitative discussion of these approaches and their merits, a key question is how do these systems compare with one another quantitatively? A single metric that captures both response time and processing capability of a data store is *action rating* [5, 4]. It is defined as the highest throughput provided by a data store given a pre-specified service level agreement, SLA. An SLA is a performance agreement between the application developer and the customer. An example SLA may require 95% of issued requests to observe a response time faster than 100 milliseconds for a pre-specified window of time Δ, say 10 minutes. The main contribution of this paper is presenting a framework consisting of two search techniques to compute the action rating for a data store. This framework employs strategies to reduce the number of conducted experiments and utilizes agile data loading techniques to reduce the duration of the experiments.

The action rating reduces the performance of different data stores to one number, simplifying the comparison of different data stores, their data models, and design principles. Workload characteristics that are application specific provide a context for the rating. For example, the BG benchmark [5, 4] uses a social graph to generate a workload consisting of interactive social networking actions to evaluate a data store. This is termed the Social Action Rating, SoAR, and can be used to compare different data stores with one another. To illustrate, Table 1 shows the computed SoAR of a document store named MongoDB, an extensible record store named HBase, a graph data store named Neo4j, and an industrial

[1] TAO [2], F1 [25], Apache's RavenDB and Jackrabbit, Titan, Oracle NoSQL, FoundationDB, STSdb, EJDB, FatDB, SAP HANA, and CouchBase.

Data Store	100% View Profile	100% List Friends
SQL-X	5,714	401
MongoDB	7,699	295
HBase	5,653	214
Neo4j	2,521	112

Table 1: BG's SoAR (actions/sec) with two workloads using a database consisting of 100K members each with 100 friends and 100 resources. The machine hosting the data store is a 64 bit 3.4 GHz Intel Core i7-2600 processor (4 cores hyperthreaded as 8) configured with 16 GB of memory, 1.5 TB of storage, and one Gigabit networking card.

strength Relational Database Management System (RDBMS) named[2] SQL-X. For the imposed workload consisting of a single action that looks up the profile of a member, View Profile, MongoDB is the high performant data store. SQL-X outperforms MongoDB for a workload consisting of 100% List Friends action even though it joins two tables, see [6] for details.

One may establish the action rating of a data store using either an open or a closed emulation model. With an open emulation model, a benchmarking framework imposes load on a data store by generating a pre-specified number of requests per unit of time, termed arrival rate. This arrival rate, λ, is an average over some period of time and might be modeled as a bursty pattern using a Poisson distribution. With a closed emulation model, the benchmarking framework consists of a fixed number of threads (or processes), T, that issue requests to a data store. Once a thread issues a request, it does not issue another until its pending request is serviced. Moreover, a thread may emulate think time by sleeping for some time between issuing requests. Both the number of threads and the think time control the amount of load imposed on a data store. With both emulation models, one increases system load (either λ or T) until the data store violates the specified SLA. The highest observed number of requests[3] processed per unit of time is the action rating of a data store.

The action rating is not a simple function of the average service time of a data store for processing a workload. To illustrate, Figure 1 shows the throughput of SQL-X with a

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

[3]This is λ with the open emulation model. With the closed emulation model, it is the highest observed throughput, see discussions of Figure 1.

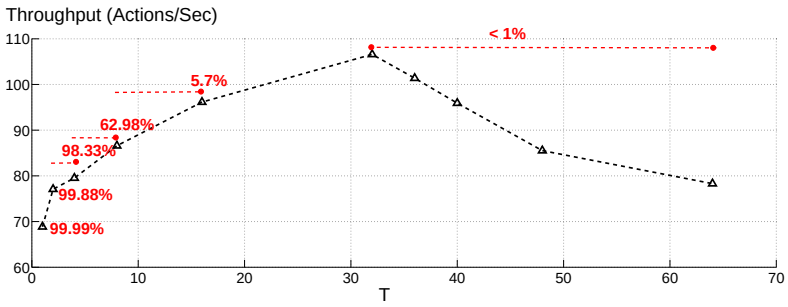


Figure 1: Throughput as a function of the imposed load (T). Percentage of requests that satisfy the SLA requirement of 100 msecs or faster are shown in red.

closed emulation model processing a social networking workload generated by BG. On the x-axis, we increase system load by increasing the number of threads T. The y-axis shows the number of actions processed per second by SQL-X. In addition, we show the different percentage of requests that observe a response time faster than 100 milliseconds. With 1 to 4 threads, this percentage is above 95%. With 8 threads, it drops to 62%. Beyond 32 threads, the throughput of SQL-X drops as the average service time of the workload starts to increase [30] with less than 1% of the requests observing a response time faster than 100 milliseconds.

To quantify the action rating of a data store, one must conduct experiments that impose an increasing amount of system load until the highest throughput that satisfies the pre-specified SLA is identified. A naïve technique may perform an exhaustive search of possible system loads by enumerating them starting with the lowest: λ=T=1. Alternatively, one may use a search technique such as the Golden Section [32, 31] which conducts fewer experiments than naïve and is as accurate. Reducing the number of experiments expedites the rating process because each experiment has a duration in the order of minutes and may have to load a benchmark database on the data store that is being evaluated. The load time is a function of the database size and the target data store. For example, the time to load a modest sized BG database consisting of 100,000 members with 100 friends and 100 resources per member is approximately 3 hours with MongoDB. With MySQL, this time is 8 hours. If these rating techniques conduct hundred experiments to rate MongoDB and fifty experiments to rate MySQL then the time to re-create the database at the beginning of each experiment alone

is more than a week.

Ultimately, the time to rate a data store is dependent on the employed search technique that dictates the number of conducted experiments, duration of each experiment, frequency of re-loading the benchmark database in between experiments, the time to load the benchmark database, and the true action rating of the data store. A high performant data store with a million as its rating requires more time to rate than a data store with 100 as its rating.

1.2 Contribution

The primary contributions of this study are three folds. First, it introduces search techniques that rate a data store both accurately and quickly. When compared with naïve, they reduces the rating of MongoDB and MySQL from days and weeks into hours. Second, it discusses the components of the rating framework. This includes providing an answer to the following two questions:

- How the framework uses the heuristic search techniques to expedite the rating process by reducing the number of conducted experiments?
- How it reduces the duration of each experiment? See discussions of the Delta Analyzer in Section 3.

Moreover, this framework employs agile data loading techniques to create the benchmark database at the beginning of each experiment. In Section 5.2, we show that hardware solutions are not a substitute for a smart strategy to create the benchmark database.

Third, this study details an implementation of the proposed techniques using the BG benchmark. We present experimental results to compare the heuristic search technique when compared with its extensions that employ the agile data loading techniques and techniques that reduce the duration of each experiment. Using Amdahl's law [1], we show these extensions to speedup the heuristic search technique by one to two orders of magnitude.

The rest of his paper is organized as follows. Section 2 presents two variants of our heuristic search technique. In Sections 3 and 4, we present a technique to reduce the duration of

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.
r	Climbing factor
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.
Υ	Warmup duration.
Λ	Total rating duration.

Table 2: BG's rating parameters and their definitions. Some of these parameters are used in Section 6.

each experiment and three agile data loading techniques, respectively. Section 5 describes an implementation of these techniques using the BG benchmark. An analysis of these techniques including their observed speedup is presented in Section 6. We present our related work in Section 7. Brief words of conclusion and future research directions are presented in Section 8 and Section 9, respectively.

2 Rating process

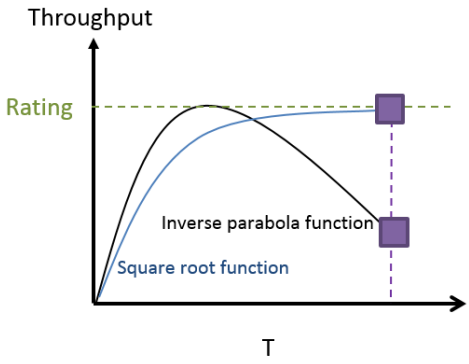
To rate a data store is similar to computing a local maxima. One may compare it to a search space consisting of nodes where each node corresponds to the throughput observed by an experiment with an imposed system load, either T with the closed or λ with the open simulation model. The node(s) with the highest throughput that satisfies the pre-specified SLA identifies the rating of the system. In the following, we focus on a closed emulation model and assume a higher value of T imposes a higher system load on a data store. Section 9 describes extensions in support of an open emulation model.

This section presents two techniques to navigate the search space. The assumptions of both techniques are described in Section 2.1. Subsequently, Section 2.2 details the two techniques and describes an alternative when the stated assumptions are violated. Section 2.3 compares these two alternatives with one another by quantifying the number of nodes that they visit and their accuracy when rating a data store.

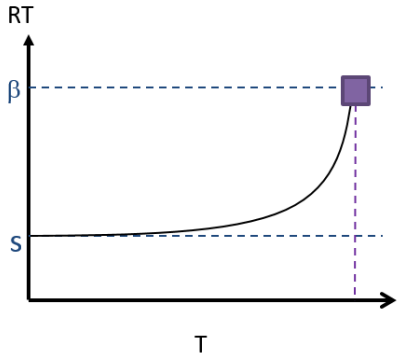
2.1 Assumptions

Our proposed search techniques make the following two assumptions about the behavior of a data store as a function of system load:

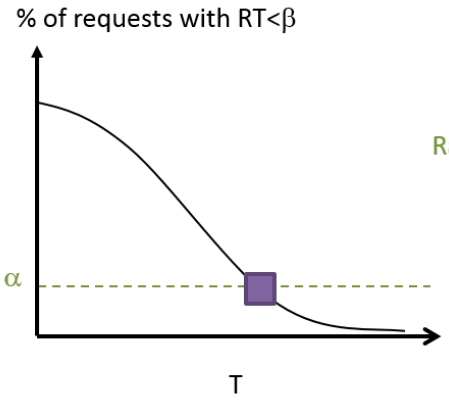
1. Throughput of a data store is either a square root or a concave inverse parabola function of the system load, see Figure 2.a.
2. Average response time of a workload either remains constant or increases as a function of the system load, see Figure 2.b.



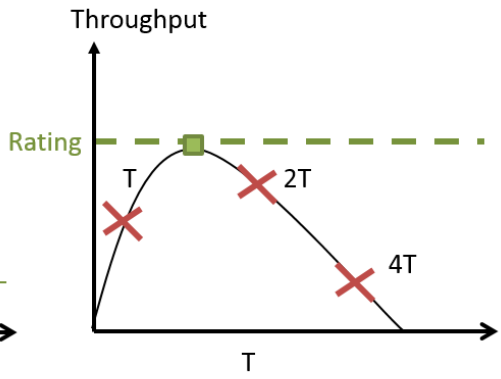
2.a : Throughput as a function of T



2.b : Average response time as a function of T



2.c : α as a function of T



2.d : Action rating's search space

Figure 2: Assumptions of BG's rating technique.

These are reasonable assumptions that hold true almost always. Below, we formalize both assumptions in greater detail.

Figure 2.b shows the average response time ($\bar{RT}$) of a workload as a function of T. With one thread, $\bar{RT}$ is the average service time ($\bar{S}$) of the system for processing the workload. With a handful of threads, $\bar{RT}$ may remain a constant due to use of multiple cores and sufficient network and disk bandwidth to service requests with no queuing delays. As we increase the number of threads, $\bar{RT}$ may increase due to either (a) an increase in $\bar{S}$ attributed to use of synchronization primitives by the data store that slow it down [9, 19], (b) queuing delays attributed to fully utilized server resources where $\bar{RT}=\bar{S}+\bar{Q}$ and $\bar{Q}$ is the average queuing delay, or (c) both. In the absence of (a), the throughput of the data store is a square root function of T else it is an inverse parabola function of T, see Figure 2.a. In scenario (b), with a closed emulation model where a thread may not issue another request until its pending request is serviced, $\bar{Q}$ is bounded with a fixed number of threads. Moreover, as $\bar{RT}$ increases, the percentage of requests observing an $\bar{RT}$ lower than or equal to β decrease, see Figure 2.c.

2.2 Search Techniques

The two techniques that employ the assumptions of Section 2.1 to rate a data store are as follows. The first technique, named *Golden*, is a search technique assured to compute the rating for the system[4]. The second technique, named *Approximate*, is a heuristic variant that is faster and less accurate. Both techniques realize the search space by conducting experiments where each experiment imposes a fixed load (T) on the system to observe a throughput that may or may not satisfy the pre-specified SLA. Each experiment is a node of the search space. While the search space is potentially infinite, for a well behaved system, it consists of a finite number of experiments defined by a system load (value of T) high enough to cause a resource such as CPU, network, or disk to become 100% utilized. A fully utilized resource dictates the maximum throughput of the system and imposing a higher load by

[4]We also developed a greedy algorithm of our own named Gauranteed [4]. Gauranteed provides similar performance to Golden except that it visits a higher number of states.

increasing the value of T (with a closed emulation model) does not increase this observed maximum. A finite value of T limits the number of nodes in the search space.

Both Golden and Approximate navigate the search space by changing the value of T, imposed load. When navigating the search space by conducting experiments, an experiment is *successful* if it satisfies the pre-specified SLA provided by an experimentalist. Otherwise, the experiment has *failed*. Both techniques traverse the search space in two distinct phases: a hill climbing phase and a local search phase.

Algorithm 1 *Compute SoAR*

1: ***procedure*** COMPUTE–SOAR
2: $T_{max} \leftarrow T \leftarrow 1$
3: $\lambda_{T_{max}} \leftarrow 0$
4: ***while*** *true* ***do***
5: $(\lambda_T, SLA_T) \leftarrow$*Conduct an experiment using* T *to compute* λ_T *and* SLA_T
6: ***if*** $\lambda_T > \lambda_{T_{max}}$ *and SLA experiments are satisfied* ***then***
7: $T_{max} \leftarrow T$
8: $\lambda_{T_{max}} \leftarrow \lambda_T$
9: $T \leftarrow T \times r$
10: ***else***
11: $End \leftarrow T$
12: ***if*** *technique is Golden* ***then***
13: $Start \leftarrow \frac{T}{r \times r}$
14: *return Golden–LocalSearch(Start,End)*
15: ***else***
16: $Start \leftarrow \frac{T}{r}$
17: *return Approximate–LocalSearch(Start,End)*

end

The local search phase differentiates Golden from Approximate. Approximate conducts fewer experiments during this phase and is faster. However, its rating incurs a margin of

error and is not as accurate as Golden. Below, we describe the hill climbing phase that is common to both techniques. Subsequently, we describe the local search of Golden and Approximate in turn.

Algorithm 2 *Perform Local Search For Golden*

1: ***procedure*** GOLDEN–LOCALSEARCH*(Start, End)*
2: $GoldenRatio \leftarrow \frac{-1+\sqrt{5}}{2}$
3: $Point1 \leftarrow End + GoldenRatio \times (Start - End)$
4: $Point2 \leftarrow Start + GoldenRatio \times (End - Start)$
5: ***while*** $(End - Start > 1)$ ***do***
6: $(\lambda_{Point1}, SLA_{Point1}) \leftarrow$*Conduct an experiment using* $Point1$
7: *to compute* λ_{Point1} *and* SLA_{Point1}
8: $(\lambda_{Point2}, SLA_{Point2}) \leftarrow$*Conduct an experiment using* $Point2$
9: *to compute* λ_{Point2} *and* SLA_{Point2}
10: ***if*** $(\lambda_{Point1} > \lambda_{Point2})$ ***then***
11: $End \leftarrow Point2$
12: $Point2 \leftarrow Point1$
13: $Point1 \leftarrow End + GoldenRatio \times (Start - End)$
14: ***else***
15: $Start \leftarrow Point1$
16: $Point1 \leftarrow Point2$
17: $Point2 \leftarrow Start + GoldenRatio \times (End - Start)$
18: $(\lambda_{Start}, SLA_{Start}) \leftarrow$*Conduct an experiment using Start*
19: *to compute* λ_{Start} *and* SLA_{Start}
20: $\lambda_{T_{max}} \leftarrow \lambda_{Start}$
21: ***return*** $\lambda_{T_{max}}$

end

One may implement the hill climbing phase by maintaining the thread count (T_{max}) that results in the maximum observed throughput ($\lambda_{T_{max}}$) among all conducted experiments, i.e.,

visited nodes of the search space. It starts an experiment using the lowest possible system load, one thread ($T = 1$) to issue the pre-specified mix of actions. If this experiment fails then the rating process terminates with a rating of zero. Otherwise, it enters the hill climbing phase where it increases the thread count to $T = r \times T$ where r is the hill climbing factor and an input to the technique. (See below for an analysis with different values of r.) It repeats this process until an experiment either fails or observes a throughput lower than $\lambda_{T_{max}}$, establishing an interval for the value of T that yields the rating of the system. Once this interval is identified, the hill climbing phase terminates, providing the local search space with the identified interval, see lines 15 and 18 of Algorithm 1.

The start of the local search interval is computed differently with Approximate and Golden, see lines 12-18 of Algorithm 1. With Approximate, the starting thread count is $\frac{T}{r}$ and the ending thread count is T and the peak throughput is assumed to reside in the interval $(\frac{T}{r}, T)$. With Golden, the starting thread count is $\frac{T}{r^2}$, the ending thread count is the current T and the peak throughput is assumed to reside in the interval $(\frac{T}{r^2}, T)$. Next, we describe how these local search intervals are navigated by Golden and Approximate.

Golden identifies the peak throughput by maintaining the throughput values for triples of thread counts (nodes) whose distances form a golden ration [32, 31]. Next, it successively narrows the range of values inside which the maximum throughput satisfying the SLA requirements is known to exist, see Algorithm 2 for details.

Approximate navigates the interval identified by the hill climbing phase differently, see Algorithm 3. It treats the start of the interval as the point with the highest observed throughput among all points that have been executed, $\frac{T}{r}$ and its end as the point with the lowest thread count that failed or resulted in a lower throughput, T. It then executes an experiment with the mid-point in this interval. If this experiment succeeds and observes a throughput higher than $\lambda_{T_{max}}$, then the heuristic changes the start of the interval to focus on to this mid-point, does not change the end of the interval (T) and repeats the process. Otherwise, it changes the end point of the interval to be this mid-point, does not change the starting point of the interval and repeats the process until the interval shrinks to consist of

one point. It repeats the experiment with this last point as the value of T and compares the observed throughput with $\lambda_{T_{max}}$ to identify the threadcount that maximized the throughput. The heuristic approach is not guaranteed to find the peak throughput (rating) for a system. Its margin of error depends on the behavior of the data store and the climbing factor r. Below, we describe an example to illustrate why Approximate incurs a margin of error.

Algorithm 3 *Perform Local Search For Approximate*

```
1:  procedure APPROXIMATE–LOCALSEARCH(Start,End)
2:      while true do
3:          intervalLength ← End − Start
4:          if intervalLength < 2 then
5:              break
6:          else
7:              T ← Start + intervalLength/2
8:          (λ_T, SLA_T) ←Conduct an experiment using T to compute λ_T and SLA_T
9:          if λ_T > λ_{T_max} & SLA requirements satisfied then
10:             Start ← T
11:             T_max ← T
12:             λ_{T_max} ← λ_T
13:         else
14:             End ← T
        return λ_{T_max}
```

end

Consider a scenario where the experiment succeeds with T threads and increases the thread count to $2T$. With $2T$ the experiments succeeds again and observes a throughput higher than the max throughput observed with T, see Figure 2.d. Thus, the hill climbing phase increases the thread count to $4T$ (assuming a climbing factor of 2, r=2). With $4T$, the experiment produces a throughput lower than the maximum throughput observed with 2T. This causes the hill climbing phase to terminate and establishes the interval ($2T$, $4T$) for the local search of Approximate. If the peak throughput is in the interval (T, 2T) then

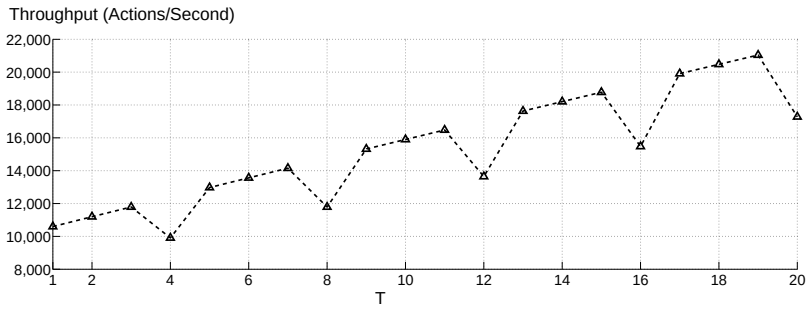


Figure 3: Behavior of a system violating the rating assumptions.

Approximate returns 2T as the peak, failing to compute SoAR accurately. The difference between the true peak and 2T is the margin of error observed with Approximate. Golden avoids this error by navigating the interval (T, 4T)

Both Golden and Approximate maintain the observed throughput with a given value of T in a hash table. (This is not shown in the Algorithms 1-3.) When exploring points during either the hill climbing phase or local search, an algorithm uses this hash table to detect repeated experiments. It does not repeat them and simply looks up their observed throughput, expediting the rating process significantly.

When comparing throughputs, both Golden and Approximate take some amount of variation into consideration. For example, λ_a is considered higher than λ_b only if it is $\epsilon\%$ higher than it. It is the responsibility of the experimentalist to identify the tolerable variation, ϵ.

With a system that violates the assumptions of Section 2.1, both Golden and Approximate may fail to identify system rating. For example, Figure 3 shows a system where the observed throughput is not an increasing function of system load. In such a case, both techniques may become trapped in a local maxima and fail to identify the peak throughput of the system. A possible approach may use simulated annealing to perform (random) jumps to escape the local maxima. We do not discuss this possibility further as we have not observed a system that violates the stated assumptions.

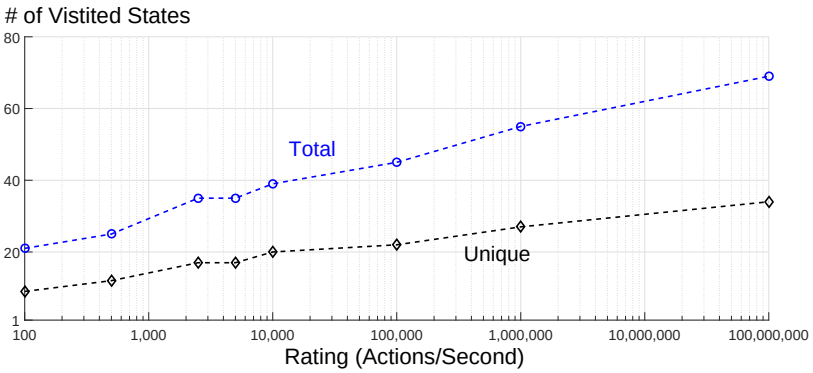


Figure 4: Number of conducted experiments using Golden.

2.3 Comparison

To compare Golden and Approximate, we use a simple quadratic function, $-aT^2+bT+c=y$ ($a = 1$ and $b > 1$), to model the throughput of a data store as a function of number of threads issuing requests against it. The vertex of this function is the maximum throughput and is computed by solving the first derivative of the quadratic function: $\mathrm{T} = \frac{b}{2}$. Golden and Approximate must compute this value as the rating of the system. We select different values of b and c to model diverse systems whose ratings vary from 100 to 100 million actions per second. We start with a comparison of Golden and Approximate, showing Golden conducts 11% to 33% more experiments than Approximate but computes the correct rating at all times. While Approximate is faster it computes an action rating with some margin of error, see discussions of Table 3 below.

Figure 4 shows the number of visited nodes. When the true rating is 100 million actions per second, Golden conducts 69 experiments to compute the value of T that realizes this rating. 35 experiments are repeated from previous iterations with the same value of T. To eliminate these, the algorithm maintains the observed results for the different values of T and performs a look up of the results prior to conducting the experiment. This reduces the number of unique experiments to 34. This is 2.8 times the number of experiments conducted with a system modeled to have a rating of 500 actions per second (which is several orders of magnitude lower than 100 million).

Figure 5 shows the number of unique experiments executed with each of the techniques

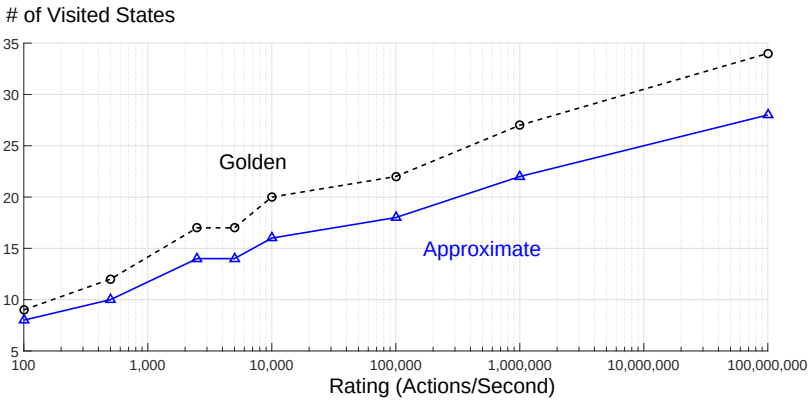


Figure 5: Number of unique experiments conducted by using Golden and Approximate, r=2.

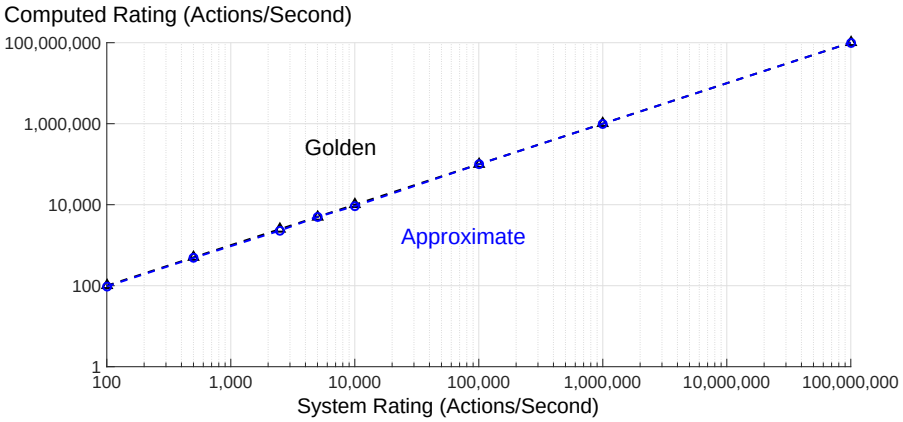


Figure 6: Rating with Golden and Approximate, $r = 2$.

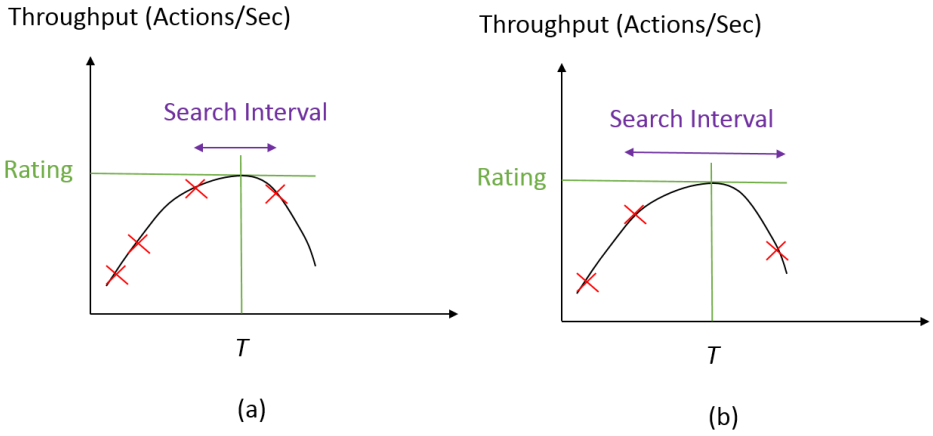


Figure 7: The impact of the value of climbing factor on the rating, (a) small versus (b) large climbing factor.

System Rating	% Reduction in Experiments			% Error in SoAR		
	r=2	r=4	r=10	r=2	r=4	r=10
100	11.11%	33.33%	25.00%	0.00%	0.00%	0.00%
500	16.67%	25.00%	25.00%	0.00%	0.80%	0.20%
2500	17.65%	25.00%	25.00%	7.84%	7.84%	1.00%
5000	17.65%	20.00%	27.78%	0.08%	0.00%	18.00%
10000	20.00%	25.00%	23.53%	7.84%	1.44%	0.00%
100000	18.18%	25.00%	23.53%	0.40%	0.02%	0.63%
1000000	18.52%	21.74%	21.74%	0.06%	0.06%	0.00%
100000000	17.65%	22.22%	12.00%	0.05%	0.06%	0.00%

Table 3: A comparison of Approximate with Golden using different climbing factor r.

with Approximate conducting fewer experiments. Figure 6 shows the ideal (expected) rating as well as the computed ratings by the two techniques for the different curves. As shown in Figure 6, Golden always computes the expected rating for the system. With Approximate, the highest observed percentage error in the computed rating was 8%. With most experiments, the percentage error was less than 1%. However, Golden conducts more experiments and visits more nodes in order to find the solution, see Figure 5. This is because Golden executes a larger number of experiments before it discards intervals that do not contain the peak.

The total number of visited nodes is computed by summing the number of experiments executed in the hill climbing phase and the number of experiments executed in the local search phase. For both techniques, this value is dependent on the value of the climbing factor r. A small climbing factor may increase the number of experiments executed in the hill climbing phase before the search interval is identified. Whereas a large climbing factor may increase the number of experiments executed in the local search phase as it may result in a larger search interval, see Figure 7.

Table 3 shows a comparison of Approximate with Golden using different climbing factors r. The first column is a known system rating. The next three columns show the percentage reduction in the number of experiments conducted by Approximate when compared with Golden with 2, 4, and 10 as values of r. The last three columns report the percentage error introduced by Approximate when computing SoAR for the same values of r. These results

show Approximate reduces the number of experiments by as much as one third. While the percentage error in the computed SoAR by Approximate is close to zero in most experiments, it was as high as 18% in one instance with r=10.

3 Experiment duration, δ

With the search techniques of Section 2, the throughput of each experiment with an imposed system load must be quantified accurately. A key parameter is the duration of each experiment denoted as δ. The ideal duration, value of δ, is both data store and workload dependent. It must be long enough to satisfy two constrains. First, it must generate a mix of requests that corresponds to the specified workload. Here, a high throughput data store shortens the duration of δ because it generates many requests per unit of time to approximate the specified workload quicker than a low throughput data store. Second, the ideal δ value must be long enough to enable a data store to produce a throughput that is permanent[5] with longer δ values. The specified workload and its mix of actions play an important role here. If there is a significant variation in the service time of the data store for the specified actions then the value of δ must be longer to capture this variation. This section describes a technique named Delta Analyzer, DA, to compute the ideal δ.

DA consists of a core timing component, DA-Core, that is invoked repeatedly. (DA-Core might be realized using the component that is used by the search techniques of Section 2 to compute the throughput of a node, see Section 5 for details.) The input of DA-Core is the user defined workload and the amount of load that it must impose, T. Its output is the value of δ, the observed resource utilization, and the mix of requests issued. DA-Core generates the input workload and system load for t time units repeatedly, doubling the value of t each time. The starting value of t is configurable and might be 1 second by default. DA-Core terminates when two conditions hold true. First, the mix of requests issued for the last q experiments is approximately the same as the input workload. Second, the observed

[5]One may define permanent using a tolerable threshold, say ±10%, on the variation in the observed throughput.

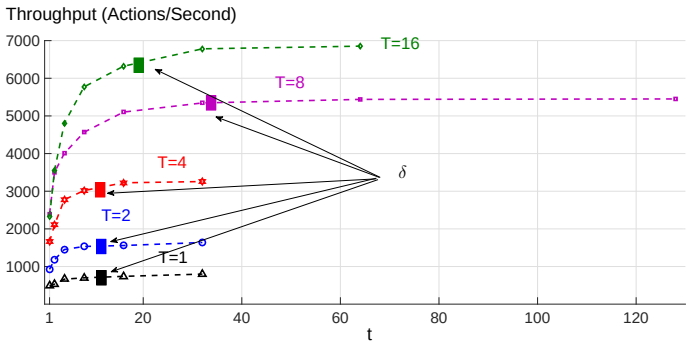


Figure 8: DA with BG and a social graph of 10,000 members, 100 friends per member, and 100 resources per member. Workload consists of 0.1% write actions.

throughput and resource utilization does not change beyond a certain threshold for these q iterations. These last q iterations include t, $2t$, ..., $2^{q-1}t$. The value of q is a configurable parameter of DA-Core. The value of t is the ideal δ value for the specified workload and system load T.

DA invokes DA-Core with a low system load, T=1, to quantify the value of t and utilization of resources. It then identifies the resource with the highest utilization and uses its reciprocal to estimate the multiplicative increase ρ in the value of T to cause this resource to become fully utilized. It invokes DA-Core with the new value of ρT to establish a new value for δ. This process repeats until a resource becomes fully utilized. The definition of full utilization of a resource is configurable and might be set at 80% usage. Once this termination condition is reached, DA terminates and reports the value of $\delta = \frac{2^{q-1}t}{2^{q-1}} = t$ as the ideal experiment duration.

To illustrate, Figure 8 shows the behavior of DA with q=3 and 5% tolerable change in the observed throughput and system utilization with BG using a workload that has a low frequency (0.1%) of write actions [6]. The x-axis of this figure shows the different values of t. The y-axis is the observed throughput with different values of t employed by DA-Core, starting with t=1. The curves show the different system loads considered by DA, i.e., invocation of DA-Core with T=1, 2, 4, 8, and 16 threads. When DA invokes DA-Core with a low system load, T=1, the termination condition is satisfied with t=32 seconds. In this experiment, the ideal δ is 8 seconds, i.e., $\frac{t=32}{2^{(3-1)}}$. The network utilization is at 11% and

Data Store	10K Members	100K Members
MongoDB	2	157
SQL-X	7	153.5
MySQL	15	2509

Table 4: Time to load (minutes) a BG social graph with 10,000 and 100,000 members into three data stores.

higher than both CPU and disk. As DA doubles the system load by changing the value of T to 2, 4, 8, and 16, the network utilization increases sub-linearly to 22%, 44%, 75%, and 91%. It is interesting to note that the throughput observed with 8 and 16 threads is almost identical with small values of t, i.e., 1 and 2 seconds. The observed throughput is higher with T=16 threads and values of t higher than 4 seconds. The ideal delta is 16 seconds and twice that observed with a low system load. One may configure DA to report the maximum of the candidate delta values or the delta value observed with the highest system load as the duration of experiments, δ, conducted by the heuristic search technique.

4 Agile data loading

With those workloads that change the state of the database (its size and characteristics[6]) one may be required to destroy and reconstruct the database at the beginning of each experiment to obtain meaningful ratings. To illustrate, Workload D of YCSB [13], inserts new records into a data store, increasing the size of the benchmark database. 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 imposed 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 destroy and recreate the benchmark database at the beginning of each experiment.

This repeated destruction and creation of the same database may constitute a significant

[6]Cardinality of a many-to-many relationship such as the number of friends per member of a social networking site.

portion of the rating process, see Table 4 . 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 [6]. 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 the introduction of agile data load techniques to expedite the rating mechanism.

This section presents three agile data loading techniques. The first technique, named *RepairDB*, restores the database to its original state prior to the start of an experiment. One may implement RepairDB using either point-in-time recovery mechanism of a data store or techniques that are agnostic to a data store. Section 5.1 presents an implementation of the latter.

The second 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[7]. Depending on the percentage of writes and the data store characteristics, RepairDB may be slower than DBIL. When the percentage of write actions is very low, RepairDB might be faster 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. First, the workload must be symmetric: It must issue write actions that negate one another in the long run. For example, with BG, a symmetric workload 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 repeated experiments. This prevents both an increased database size and the possible depletion of the benchmark database from friendship relationships to thaw. See Section 5.3 for other

[7]One may implement a similar loading technique named Virtual Machine Image Loading (VMIL) by creating virtual machines with a clean copy of the database. The resulting VM image is used repeatedly across different experiments. When compared with DBIL, the VM image is larger than just the database image and may require a longer time to copy.

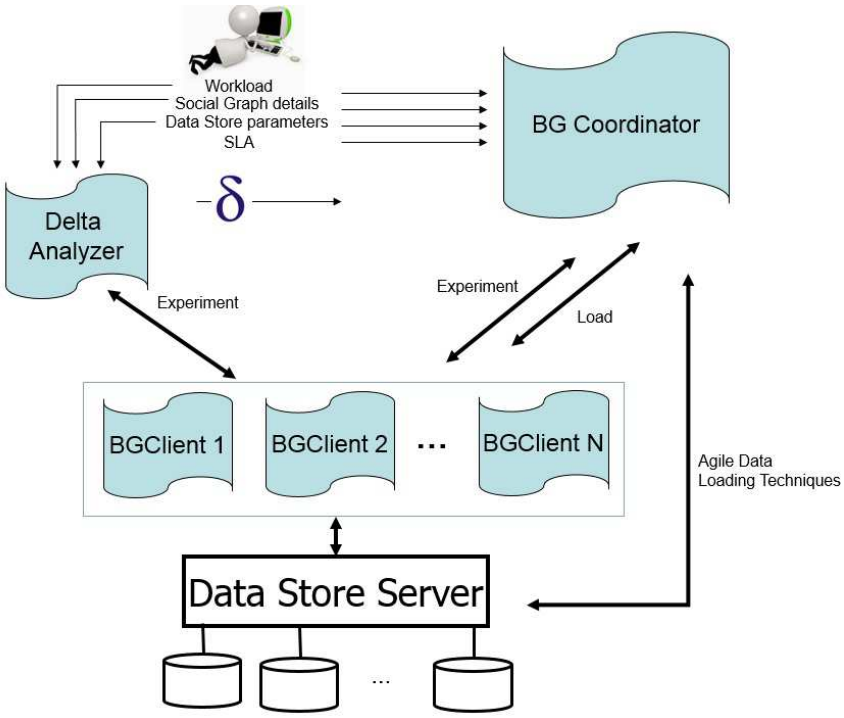


Figure 9: Architecture of the BG benchmark.

conditions that constrain 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 6 shows the hybrid approaches provide a factor of five to twelve speedup in rating a data store.

An implementation of these techniques in BG is detailed in Section 5. Section 6 presents an evaluation of these techniques.

5 An Implementation

Figure 9 shows the software architecture of an implementation of the concepts presented in the previous three sections using the BG benchmark [5]. BG is a scalable benchmark that employs a shared-nothing hardware platform to generate sufficient requests to rate high throughput data stores. Its components include N BG Clients (BClient), one BG Coordi-

nator (BGCoord), and one Delta Analyzer (DA). We describe these in turn. Subsequently, Sections 5.1-5.3 describe an implementation of the three agile data loading techniques.

BGCoord implements the search techniques of Section 2 that conduct repeated experiments by issuing commands to BGClients to conduct experiments to compute the SoAR of a data store. One of BGCoord's inputs is the duration of each experiment computed using the delta analyzer (DA) of Section 3. Each BGClient instance consists of a workload generator and an implementation of BG's eleven social networking actions specific to a data store.

To be portable to different operating systems such as Linux and Windows, BG is implemented using the Java programming language. The different components communicate using message passing (instead of operating system specific remote invocations) to conduct different rounds of experiments. This is realized using a BGListener (not shown in Figure 9) that is co-located with each BGClient instance. BGCoord and DA communicate with the BGListener using message passing. BGListener communicates with a spawned BGClient directly. There is one BGListener per BGClient.

Currently, the DA is separated from the BGCoord. Its input include the workload, details of the social graph, and the data store parameters[8]. A workload consists of a mix of eleven social networking actions and the degree of skew for referencing different data items. DA conducts experiments by imposing a different amount of load on a data store using the BGClients. Each BGClient is multi-threaded and the number of threads dictates how much load it imposes on a data store. Each BGClient gathers its observed throughput of a data store along with the utilization of the resources of a server hosting the data store. DA uses this information to computer δ, the duration of experiments to be conducted by BGCoord, per discussions of Section 3.

BGCoord inputs the workload, details of the social graph and the data store, the user specified SLA and δ to rate the data store. It implements the agile data loading techniques of Section 4. The software architecture of Figure 9 is general purpose. One may adapt it for use with YCSB [13] and YCSB++ [24] by specifying a high tolerable reponse time, i.e., max

[8]The data store parameters are those required to connect to the database such as the connection URL, database name, data store username and password and all other data store specific parameters such as MongoDB's write concern and read preference.

integer, for the input SLA. In addition, the core classes of both YCSB and YCSB++ should be extended to (a) gather resource utilization of servers hosting the data store, and (b) communicate the gathered resource utilization and the observed throughput to BGCoord.

5.1 Repair Database

Repair Database, *RepairDB*, marks the start of an experiment (T_{Start}). At the end of the experiment, it may employ the point-in-time recovery [21, 20] 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.

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 [5] data during its validation phase at the end of an experiment. One may implement point-in-time recovery by 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 5 shows RepairDB improves the load time of MongoDB[9] by at least a factor of 2 with 1000 friends per member. This

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

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

Table 5: Factor of improvement in MongoDB's load times with RepairDB compared with re-creating the database, M=100K, ρ=100.

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 BG-Clients, 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 [5], 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.

5.2 Database Image Loading

Various data stores provide specialized interfaces to create a "disk image" of the database [23]. Ideally, the data store should provide a high-throughput external tool [24] that the benchmarking framework employs to generate the disk image. Our target data stores (MongoDB, MySQL, an industrial strength RDBMS named SQL-X) do not provide such a tool. Hence, our proposed technique first populates the data store with benchmark database and

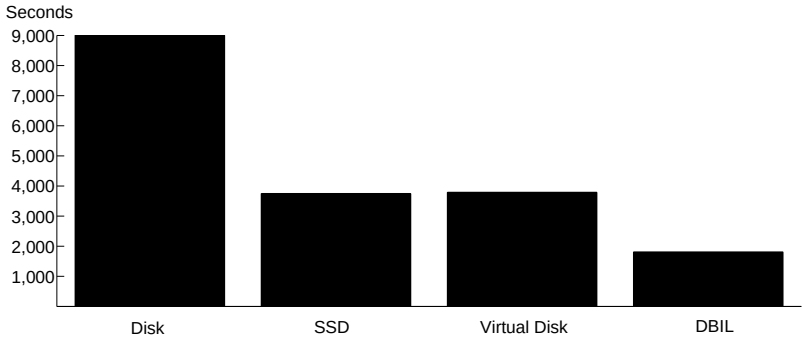


Figure 10: Time to load a 100K member social graph with 3 different hardware configurations and an agile data loading technique named DBIL.

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. Figure 10 compares the amount of time it takes to load a social graph consisting of 100,000 members using DBIL with using BG to construct the database for a data store that stores its data on disk, an MLC SSD and a virtual disk[10]. All three are slower than the DBIL technqiue.

The reason copying an image of a database using DBIL is faster than constructing the social graph using BG is because it does a sequential read and write of a file. BG's construction of the social graph is slower because it generates users and friendships dynamically[11]. This may cause a data store to read and write the same page (corresponding to a user) many times in order to update a piece of information (a user's JSON object) repeatedly (modify friends). In addition, it also must construct index structure that is time consuming[12].

With DBIL, the load time depends on how quickly the system copies the files pertaining to

[10]With disk and the MLC SSD the disk on the node hosting the data store becomes the bottleneck. With the virtual disk, the CPU of the node hosting the data store becomes the bottleneck as now all the data is written to the memory. DBIL is faster than this approach as it eliminates the overhead of locking and synchronization on the data store.

[11]Constructing the social graph using BG without actually issuing calls to the data store takes less than a second showing that BG does not impose any additional overhead while loading the social graph into the data store.

[12]In addition, with MongoDB, there is also the overhead of locking and synchronization on the data store.

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[13]. 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 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

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

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 [22]. 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. It is also important to start the shards (Mongod instances) before attaching them to the data store cluster. The programmer specifies this sequence of actions by implementing "MultiShardStart" and "MultiShardStop" methods of BGCoord.

5.3 Load Free

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 BG framework to implement LoadFree.

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 was 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 (TF) 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 TF 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 4, 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 [17] or suffer from race conditions that leave the cache and the database in an inconsistent state [15], a data store may employ an eventual consistency [27, 26] technique that produces either stale or inconsistent data for some time [24], 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 6) as it eliminates the load time between experiments. One may violate the above three aforementioned crite-

rion 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 6. 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 starts the BGClient and the BGClient terminates once it has either executed for a pre-specified[14] amount of time or has issued a pre-specified number of requests [13, 24]. 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[15] 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 Execute On Experiment (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

[14]Described by the workload parameters.

[15]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.

invitation to Member B if they are already friends) and quantify the amount of unpredictable data at the end of each experiment, see [5] for details.

6 Evaluation

This section quantifies the speedup observed with the 3 proposed loading techniques and the DA using a fixed workload. With the data loading techniques, 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. With the DA, we focus both on Simple[16] and agile data loading techniques to quantify the observed speedup.

In the following, we start with an analytical model that describes the total time required by the heuristic search techniques to rate a data store. Next, we describe how this model is instantiated by the data loading techniques. Subsequently, we describe how the ideal δ impacts the overall rating duration. We conclude by presenting the observed enhancements and quantifying the observed speedup relative to not using the proposed techniques for three different single node data stores[17]. Each node running a data store consists of an Intel i7-4770 CPU (quad core, 3.8 GHz), 16 Gigabytes of memory, a 1 Terabyte disk drive, and 1 network interface card.

6.1 Analytical Model

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 (such as the Bulkload of YCSB++ [24]) 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

[16]We refer to constructing the database by issuing queries against it as the Simple loading technique.
[17]SQL-X, MongoDB 2.0.6 and MySQL 5.5.

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	308	498	212	242	284
500K	ζ	361	351	351	361	351
	ν	10	165	0	2.5	41.2
	Λ	526	2221	406	444	860
1000K	ζ	14804	14773	14773	14804	14773
	ν	31	588	0	7.75	147
	Λ	15200	21296	14828	14887	16445

Table 6: BG's rating of MongoDB (minutes) with 1 BGClient for a fixed workload, ϕ=100, ρ=100, Υ =1 minute, δ=3 minutes, Δ=10 minutes, and η=11. The hybrid techniques used either DBIL or RepairDB for approximately 25% of experiments.

by the workload parameters M, P, ϕ, ι, ϱ 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 6.

- 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.
- The warmup duration required for each round of experiment, Υ. This duration may be determined either by experience or using DA. One may use DA to compute the amount of time it takes for a system to warm up.
- Total number of rating experiments conducted by the heuristic search techniques, 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.
- The duration of the final rating round per the pre-specified SLA, Δ.

The total rating duration is:

$$\Lambda = \zeta + (\omega \times \nu) + (\eta \times (\Upsilon + \delta)) + (\Upsilon + \Delta) \quad (1)$$

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 6. Its value is zero with LoadFree[18].

By setting ν equal to ζ, Equation 1 models a simple use of BG's heuristic search technique that does not employ the agile data loading techniques described in this chapter. This technique would require 1939 minutes (1 day and eight hours) to rate MongoDB with 100K members. The third row of Table 6 shows this time is reduced to a few hours with the proposed loading techniques. This is primarily due to considerable improvement in load times, see the first two rows of Table 6. Note that the initial load time (ζ) with DBIL is longer because it must both load the database and construct the disk image of the database. The last six rows of Table 6 show the observed trends continue to hold true with databases consisting of 500K and 1 million members. In addition, if $\delta < \Delta$ is used as the duration of each rating experiment, then the overall duration for of the rating process will improve.

An obvious question is the impact of the discussed techniques while leaving other pieces alone relative to the simple use of the heuristic techniques (ν=ζ) and when $\delta = \Delta$? 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, and k is speedup while in faster mode. The next two paragraphs will describe how f and k are computed for various agile data loading techniques and the ideal δ computed by DA.

Focusing on the data loading techniques alone, the fraction of work done in the faster mode is computed as $f = \frac{\omega \times \zeta}{\Lambda}$, and the speedup while in faster mode is computed using $k = \frac{\zeta}{\nu}$. 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. In [7], we show the value of S to level off with the value of k in the order of hundred thousand.

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

M	DBIL	RepairDB	LoadFree	LoadFree + DBIL	LoadFree + RepairDB
100K	6.5	3.9	9.1	8.3	6.8
500K	8.3	1.9	10.5	9.8	5
1000K	11.7	8.3	12	11.9	10.8

Table 7: Observed speedup (S) when rating MongoDB using agile loading techniques.

When only changing the duration of rating experiments from Δ to ideal δ, the fraction of work done in the faster mode is computed as $f = \frac{\eta \times (\Upsilon + \Delta)}{\Lambda}$. Speedup while in faster mode is computed as $k = \frac{(\Upsilon + \Delta)}{(\Upsilon + \delta)}$.

When we use both an agile data loading technique and the ideal δ computed by the DA in our rating experiments, the following are used to compute the overall speedup compared to the simple use of heuristic search without the use of agile data loading techniques: $f = \frac{(\omega \times \zeta) + (\eta \times (\Upsilon + \Delta))}{\Lambda}$ and $k = \frac{(\omega \times \zeta) + (\eta \times (\Upsilon + \Delta))}{(\omega \times \nu) + (\eta \times (\Upsilon + \delta))}$.

6.2 Speedup with Loading Techniques

Table 7 shows the observed speedup (S) for the experiments reported in Table 6. LoadFree provides the highest speedup followed by DBIL and RepairDB. The hybrid techniques follow the same trend with DBIL outperforming RepairDB. Speedups reported in Table 7 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 6. 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 the heuristic search techniques.

Reported trends with MongoDB hold true with both MySQL and an industrial strength RDBMS named [19] SQL-X. The time to load these data stores and rate them with 100K member database is shown in Table 8. For all three data stores Load Free provides the highest speedup followed by DBIL and RepairDB. And the hybrid techniques follow the

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

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	308	498	212	242	284
MySQL	ζ	2514	2509	2509	2514	2509
	ν	4.7	1206	0	1.2	302
	Λ	2620	15830	2564	2582	5881
SQL-X	ζ	158.5	153.5	153.5	158.5	153.5
	ν	5	30	0	1.3	7.5
	Λ	274	544	214	232	296

Table 8: BG's rating (minutes) of MongoDB, MySQL and SQL-X with 1 BGClient for a fixed workload, M=100K, ϕ=100, ρ=100, ω=11, Υ=1 minute, δ = 3 minutes, Δ = 10 minutes, and η=11. The hybrid techniques used either DBIL or RepairDB for approximately 25% of the experiments.

Data Store	DBIL	RepairDB	LoadFree	LoadFree+ DBIL	LoadFree+ RepairDB
MongoDB	6.5	3.9	9.1	8.3	6.8
MySQL	11.5	1.9	11.8	11.7	5.1
SQL-X	7.3	3.6	9.2	8.6	6.6

Table 9: Speedup (S) when rating MongoDB, MySQL and SQL-X with M=100K.

same trend. 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 9.

6.3 Speedup with Loading Techniques and DA

This section analyzes the observed speedup with the δ value computed using the DA, highlighting its usefulness to expedite the rating process. We assume the heuristic search technique conducts 11 experiments to rate the data store, comparing the alternative data loading techniques using two different values of δ. First, when δ is set to the SLA duration specified by the experimentalist, i.e., δ=Δ. Second, when δ is computed using the DA as 16 seconds. Table 10 shows the latter with different data loading techniques compared with Simple loading technique for four different values of Δ: 3, 10, 100 and 1,000 minutes. Note that there is a column for Simple, comparing the technique that uses BG to load the database at the beginning of each experiment with the two δ values.

$\delta = \Delta$ (mins)	Simple	DBIL	RepairDB	LoadFree	LoadFree+DBIL	LoadFree+RepairdDB
3	1.02	7.3	4.2	11	9.8	7.8
10	1.1	7.5	4.3	11.1	9.9	8
100	1.5	8.6	5.6	11.4	10.5	9.02
1,000	4.8	11.03	9.53	11.9	11.6	11.2

Table 10: Speedup when rating MongoDB using agile data loading techniques and DA compared to Simple with δ=Δ.

The observed speedup increases as we increase the value of Δ because it causes both f and k to increase, approaching a speedup of 12. With Δ=3 minutes, Simple observes the lowest speedup as its load time is significant and does not benefit from the use of the DA computing the duration of each experiment to be 11 times faster (δ=16 seconds instead of 3 minutes). At the other extreme, LoadFree observes the highest speedup because its database load time is zero and benefits greatly from a δ of 16 seconds. For the same reason, LoadFree observes approximately the same speedup when Δ approaches 1,000 minutes.

7 Related Work

Both the action rating metric and the use of a heuristic search technique to compute this metric for a data store were originally described in [5]. Both were tightly coupled with BG. Subsequently, we realized that both concepts are orthogonal to an application specific benchmark and its abstractions. To illustrate, all concepts described in this paper are applicable to YCSB [13], YCSB++ [24], LinkBench [33] and benchmarks for graph databases [3, 18] among the others.

The agile data loading techniques were originally described in [7]. They address the challenge of loading a benchmark database that is a recognized topic by practitioners dating back to Wisconsin Benchmark [8, 14] and 007 [10, 28, 29], and by YCSB [13] and YCSB++ [24] more recently. YCSB++ [24] 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 DBIL. 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.

To the best of our knowledge, the delta analyzer to compute the duration of each experiment is novel and not described elsewhere in the literature. This component is different than a warmup phase and its duration. Assuming the presence of a warmup phase, it considers how long each experiment must run in order to obtain meaningful numbers.

8 Conclusion

Experimentalists require fast and accurate frameworks to quantify the performance tradeoffs associated with alternative design decisions that implement novel data stores. Response time and throughput are two key metrics. Action rating is a single number that monetizes these two metrics into one. It quantifies the highest observed throughput while a fixed percentage of issued requests observe a response time faster than a pre-specified threshold. When comparing two data stores with one another, the one with the highest action rating is superior. The same holds true when comparing two different algorithms. The primary contribution of this study is a framework to compute action rating. To speedup the rating process, we described the following optimizations: 1) reduce the number of conducted experiments using Approximate instead of Golden with the understanding that its computed SoAR may not be as accurate as Golden, 2) reduce the duration of each conducted experiment using the Delta Analyzer, 3) minimize the time required to create the database at the beginning of each experiment for those workloads that require it. With these improvements and a fixed amount of time, an experimentalist may rate many more design alternatives to gain insights.

9 Future Work

With the agile data loading techniques, one may expedite the processes using muliple disks, a RAID disk subsystem, a RAM disk or even a flash memory. Our immediate future work is to analyze these techniques and to quantify their tradeoffs.

We are extending this study by doing a switch from a closed emulation model to an open one which requires the following changes. First, each BGClient is extended with a main thread that generates requests based a pre-specified arrival rate using a distribution such as Poisson. This thread generates requests for a collection of threads that issue requests and measure the response time. With Poisson, it is acceptable for a coordinator to require N BGClients to generate requests using an implementation of Poisson with input $\frac{\lambda}{N}$. Now, the heuristic changes the value of λ instead of the number of threads T. The Delta Analyzer uses this same infrastructure to compute the duration of each experiment. The three agile data loading techniques remain unchanged.

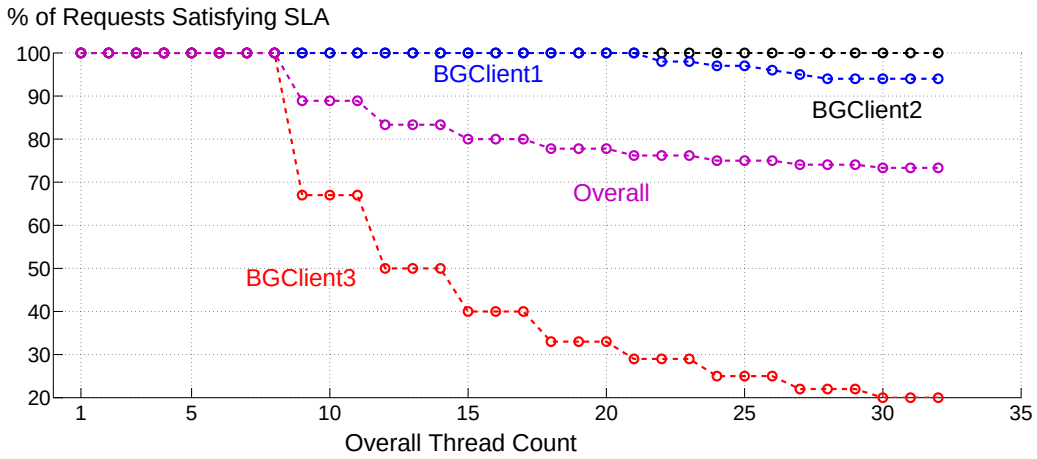


Figure 11: SoAR rating with three BGClients where one node is slower than the other two.

Moreover we intend to extend the rating framework by detecting when a component of the framework is a bottleneck. In this case, the obtained action ratings are invalid and the rating process must be repeated with additional computing and networking resources. For example, with the implementation of Section 8, there might be an insufficient number of BGClients [5] or one of the BGClients might become the bottleneck because its hardware

is slower than that used by other BGClients. For example, Figure 11 shows the SoAR rating with 3 BGClients where one is unable to produce requests at the rate specified by the BGCoord. This misleads BGCoord and its heuristic to compute 8 instead of 16 as the SoAR rating of the data store. A preventive technique may compute the required number of BGClients given a workload and an approximate processing capability of a data store. Other possibilities include detecting the slower BGClient and either not using it or requiring it to produce a load that does not exhaust its processing capability. These extensions enhance the accuracy of the framework when rating a data store.

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] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J. Larriba-Pey. Benchmarking Database Systems for Social Network Applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013.

[4] S. Barahmand. Benchmarking Interactive Social Networking Actions, Doctorate thesis, Computer Science Department, USC, 2014.

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

[6] S. Barahmand, S. Ghandeharizadeh, and J. Yap. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. In *Proceedings of the 22Nd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '13, 2013.

[7] S. Barahmand and S. Ghandeharizadeh. Expedited Rating of Data Stores Using Agile

Data Loading Techniques. In *Proceedings of the 22Nd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '13, pages 1637–1642, 2013.

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

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

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

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

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

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

[14] D. 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.

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

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

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

[18] F. Holzschuher and R. Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, 2013.

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

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

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

[22] MongoDB. Sharded Cluster Administration, http://docs.mongodb.org/manual/administration/sharde clusters/, 2011.

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

[24] 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.

[25] J. Shute, R. Vingralek, B. Samwel, B. Handy, Ch. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.*, 6(11), August 2013.

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

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

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

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

[30] H. Jung, H. Han, A.D. Fekete, G. Heiser, and H.Y. Yeom. A scalable lock manager for multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[31] J. Kiefer. Sequential Minimax Search for a Maximum. In *Proceedings of the American Mathematical Society 4*, 1953.

[32] Golden Section Search. http://en.wikipedia.org/wiki/golden_section_search, 2015.

[33] T. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *SIGMOD*, pages 1185–1196, 2013.