Benchmarking Interactive Social Networking Actions

Sumita Barahmand

Department of
Computer Science
Adviser: Professor Shahram Ghandeharizadeh

Spring 2014

Abstract

Social networking sites such as Google+, Facebook, Twitter and LinkedIn, are cloud service providers for person to person communications. There are different approaches to building these sites ranging from SQL to NoSQL and NewSQL, Cache Augmented SQL, graph databases and others. Some provide a tabular representation of data while others offer alternative models that scale out. Some may sacrifice strict ACID (Atomicity, Consistency, Isolation, Durability) properties and opt for BASE (Basically Available, Soft-state, Eventual consistency) 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? This dissertation investigates the viability of a benchmark to address this question.

Our primary contribution is the design and implementation of a novel benchmark for interactive social networking actions named BG (http://bgbenchmark.org). BG's design decisions are as follows: First, it rates the performance of a system for processing interactive social networking actions by computing two values: Socialites and Social Action Rating (SoAR) using a pre-specified Service Level Agreement, SLA. An example SLA may require 95% of issued requests to observe a response time faster than 100 milliseconds. Second, BG elevates the amount of unpredictable data produced by a solution to a first class metric, including it as a key component of the SLA (similar to the average response time) and quantifying it as a part of the benchmarking process. It also computes the freshness confidence to characterize the behavior of a weak consistency technique. Third, BG's generated workload is characterized by reads and writes of a very small amount of data from big data. Fourth, BG is a modular, extensible framework that is agnostic to its underlying data store. Fifth, BG employs a logical partitioning of data to scale both vertically and horizontally to thousands of nodes. This is essential for evaluating scalable installations consisting of thousands of nodes. Finally, BG includes a visualization tool to empower an evaluator to monitor an in-progress benchmark and identify bottlenecks.

BG's possible use cases are diverse. One may use BG to compare and contrast various data stores with one another, characterize tradeoffs associated with alternative physical representations of data, or quantify the behavior of a data store in the presence of various failures (either CP or AP of the CAP theorem) among the others. This dissertation demonstrates use of BG in two contexts. First, to rate an industrial strength relational database management system and a document store, quantifying their performance tradeoffs. This analysis includes the use of a middle tier cache (memcached) and its impact on the performance of each system. Second, to gain insight into alternative design decisions for implementing a social action by characterizing their behavior with different social graphs and system loads. BG's proposed framework is quite novel and opens several new research directions that benefit the systems research community.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been an explosion of novel data stores with varying architectures and design decisions for managing the ever increased volume and variety of data produced by applications with unique and strict requirements. Academia, cloud service providers such as Google and Amazon, social networking sites such as LinkedIn and Facebook, and computer industry continue to contribute systems and services with novel assumptions. In 2010, Rick Cattell surveyed 23 systems [25] and we are aware of 10 more[1] since that writing. In his survey, Cattell identified a "gaping hole" with a scarcity of benchmarks to substantiate the claims made by the different systems. Good benchmarks are essential because they settle debates and enable the discipline to make rapid progress [89]. They are a component of a scientific endeavor to understand alternative design decisions, quantify their tradeoffs, and obtain insights to develop improved designs.

A good benchmark provides metrics that are relevant and not misleading. Prior to advent of data stores that sacrificed strong consistency, performance metrics such as response time and throughput sufficed. With advent of data stores that provide weak consistency techniques to enhance performance and data availability in the presence of network partitions [112, 75], a benchmark must quantify the amount of unpredictable data (stale, inconsistent, or simply erroneous data) produced by a data store [12]. This empowers an experimentalist to quantify both the performance and the amount of unpredictable data produced by alternative weak consistency techniques and solutions.

BG [12], a social networking benchmark (visit http://bgbenchmark.org), was designed to quantify these new metrics and address certain aspects of the Cattell's hole that is too large to address with just one benchmark. It was motivated by the need to evaluate the performance of a transparent caching framework [48] developed in the context of a social networking site named RAYS [44]. Several social networking sites have either developed their own data store, e.g., Facebook's Cassandra [69] and LinkedIn's Voldemort [113, 72], or use a NoSQL solution, e.g., FourSquare's MongoDB [82]. BG is intended to provide insights into the performance of these systems.

We developed BG in 2012 and released a stable version of it in January 2013. Its conceptual schema and thirteen actions are an abstraction of today's social networking sites such as Google+, Facebook and others. Table 1.1 provides a comprehensive list of

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

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

Table 1.1: Socialite actions and their compatibility with several social networking sites.

the surveyed sites and a matrix that describes the compatibility of their actions with those abstracted by BG. The first column of Table 1.1 shows the thirteen actions that constitute BG. The name of each action is self explanatory. These are simple actions that read and write a small amount of data. Except for the View News Feed action, all other actions that reference members are binary consuming two member ids as input. For example, the two member ids specified with the View Profile action identify the member who is viewing a profile and the member whose profile is being viewed. Those actions that consume a resource id either read the resource and its comments, modify a comment on that unique resource, or share the resource with other members.

BG's database consists of a fixed number of *members* and *pages* with a registered profile. Its workload generator implements a closed simulation model with a fixed number of threads T. Each thread emulates a sequence of members/pages performing a social action shown in Table 1.1. At any instance in time, an emulated member/page who is actively engaged in a social action is called a *socialite*. While a database may consist of millions of members/pages, at most T simultaneous socialites issue requests with BG's workload generator. Given a social graph, BG generates actions that are valid. For example, it extends a friendship from Member A to Member B only when they are not friends. It realizes this by maintaining a representation of the social graph in its memory. BG uses this representation to ensure its emulated simultaneous members and resources are unique at an instance in time.

While One may use BG for a variety of purposes, this dissertation emphasizes two use cases. First, to rate one or more data stores to either identify the performance limits of a data store, compare the performance of different data stores with one another, or both using

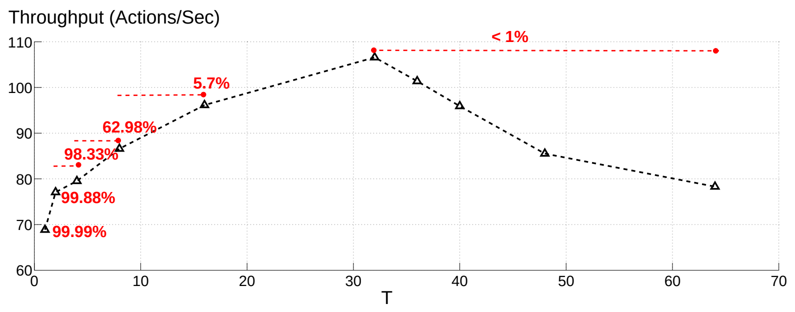


Figure 1.1: Throughput of an RDBMS as a function of T with the View Profile action, 10,000 members, 12 KB profile image size, β=100 msec, τ=0%, θ=0.27. Confidence (α) is shown in red. With 12 KB images, the RDBMS fragments the images into smaller chunks which introduce an additional overhead while retrieving the images. This results in the processor on the node hosting the RDBMS, getting fully utilized and limits its performance.

a single value. Second, to provide insights into the performance of alternative designs and algorithms with a different amount of system load. We describe each in turn.

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

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

2. Socialites: Highest number of simultaneous threads that satisfy the specified SLA. It quantifies the multi-threading capability of the data store and whether it suffers from limitations such as the convoy phenomena [20] that diminishes its throughput rating with a large number of simultaneous requests. Given several systems, depending on the application, the one with the highest Socialites rating may be more desirable.

These ratings are not a simple function of the average service time ($\bar{S}$) of a workload. The specified confidence (α), the tolerable response time (β), and the amount of unpredictable data (τ) observed from a system impact its SoAR and Socialites rating. The key advantage of these ratings is that they reduce the performance of a system to two numbers, simplifying communication of results, allowing definition of clear performance objectives and enabling comparative studies. BG rates a data store by imposing an increasing amount of load starting from a low load to a high load (T), emulating a mix of actions against the data store. It computes the percentage of actions (α confidence values) that observe a response time faster than β and provides insights into the system behavior. To illustrate, Figure 1.1 shows the throughput of an industrial strength relational database management system (RDBMS) as a function of the number of threads T for a read only action, τ=0. We show the different confidence values for β=0.1 second. As we increase the number of threads, the throughput of the system increases. Beyond 4 threads, a queue of requests forms causing an increase in system response time. This is reflected in a lower α value. With 32 threads, almost all (99.83%) requests observe a response time higher than 100 msec.

Second, one may use BG to study the behavior of alternative design choices with a varying amount of system load and different social graphs. A practitioner may specify a tolerable response time (β) and use BG to either reason about the behavior of a design choice or understand and discover trends about the behavior of an algorithm and its implementation. For example, Figures 1.2 and 1.3, demonstrate the behavior of two different design choices, termed Push and Pull, used for implementing feed following actions (View News Feed and Share Resource actions). Push pre-computes the news feed for each member and updates it every time there is a new feed for the member. Pull computes the news feed for a member every time the member requests to view her feed. A SoAR rating is not appropriate for an investigation of these alternatives because the data set size increases as a function of the number of Share Resource actions issued by the socialites. Instead, it is more appropriate to analyze the behavior of these alternatives with different system loads. We elaborate on this in the next two paragraphs.

As shown in Figure 1.2, with a lower imposed load (T), the Pull architecture results in a higher throughput when compared to Push. With intermediate system load ($T = 50$ to $T = 110$), Push becomes superior to Pull. This is because with Push the news feed is already constructed and is retrieved without issuing additional queries. With a high system load ($T > 110$), Push and Pull switch places with Pull providing a higher performance. This is because the View News Feed action displays the top 10 shared resources. While Pull retrieves only these 10 feeds, Push must retrieve the entire news feed and sort it in the application memory each time. This network transmission and processing time causes Push to become inferior. An alternative implementation for Push may sort the shared resources while updating a member's news feed every time a new feed is published. This will reduce the response time for the View News Feed action but will increase the response time for the Share Resource action for Push. One may use BG to evaluate the behavior of this and other designs.

Figure 1.3 shows how the change in the number of friends (followers) per member impacts the observed throughput for these two design considerations. For these social graphs with different fan-outs, the throughput observed using Pull is higher than that for Push. However, as we increase the number of friends per member the observed throughputs for both systems decrease and the gap in performance between Push and Pull becomes negligible. Section 8.4 provides additional details about these experiments.

Today's BG is designed for high throughput data stores that process simple operations that read and write a small amount of data. One may use BG for a variety of purposes ranging from comparing different data stores with one another to characterizing the performance of a data store under different settings such as (1) normal mode of operation with alternative physical data organizations, see Section 8.1, (2) in the presence of a network partition (either CP or AP in CAP [75]), and (3) when exercising the elasticity of a data store by adding or removing nodes incrementally. We have employed BG to compare a

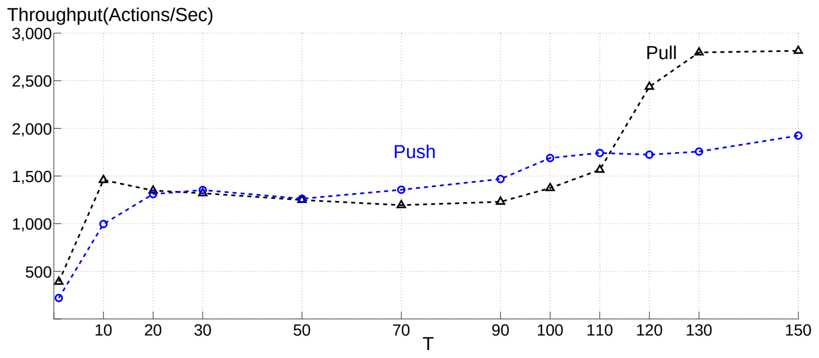


Figure 1.2: Performance of Pull vs. Push with MongoDB for a High (11% Write) Mixed workload of Table 8.9 for $\theta = 0.27$. $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads, 1% of the SR actions are issued by pages.

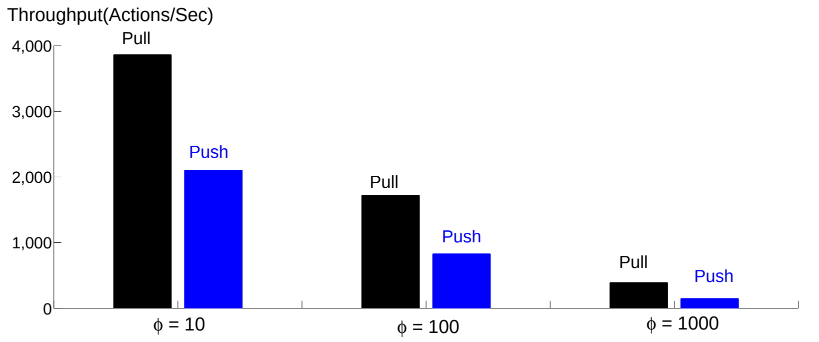


Figure 1.3: Impact of modifying the number of friends per member (ϕ) on the performance of Pull and Push with MongoDB for a workload consisting of 1% Share Resource action and 99% View News Feed Action. $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\rho = 10$ and $\theta = 0.27$. For all workloads, 1% of the SR actions are issued by pages.

relational representation of a social graph with its JSON representation [14], quantify the tradeoffs associated with alternative consistency techniques for a cache augmented relational data store [47], and others. Chapter 8 illustrates these use cases by comparing the following 3 different data stores with one another:

- SQL-X: An industrial strength relational database management system with ACID properties and an SQL query interface. Due to licensing restrictions, we cannot reveal its identity and name it SQL-X.

- MongoDB version 2.4.8, a document store for storage and retrieval of JavaScript Object Notations, JSON. MongoDB is a representative NoSQL system. See [25] for a survey.

- CASQL: SQL-X and MongoDB extended with memcached server version 1.4.2 (64 bit). This implementation employs Whalin memcached client version 2.5.1 to communicate with the memcached server.

1.1 Thesis Contributions

BG is inspired by prior benchmarks that evaluate cloud services such as YCSB [29] and YCSB++ [88], e-commerce sites [4], and object-oriented [23] and transaction processing systems [53]. It is a benchmarking framework developed for social networks which tries to answer questions such as:

1. What is the tradeoff associated with the alternative architectures? Which component of a data store becomes a bottleneck and dictates its SoAR and Socialites ratings?

2. Which systems perform better for what kind of workloads?

3. How do we compare one data store with another for a social networking application?

4. What is the trade-off between ACID and BASE? Does the performance improve by 10% or a factor of 100? What percentage of reads produce unpredictable data? Is it 0.001% or 10% of all issued reads?

5. Many NoSQL solutions claim scalability and elasticity as their main benefit. How well do these solutions scale when compared with one another for social networking workloads?

6. Which systems are truly mature? For example, how long does it take the system to load 1 million entities? A few minutes or several weeks.

BG's contributions are along the following six dimensions: First, it emphasizes interactive social actions that read and write a small amount of data. Second, it promotes the amount of unpredictable data produced by a solution as a first class metric for comparing different data stores with one another. The value of this metric is impacted by BG's knobs such as the exponent of the Zipfian distribution used to generate referenced members and the inter-arrival time between two socialites emulated by a thread. These knobs enable one to approximate a realistic use case of an application to quantify unpredictable data practically. Third, BG computes the freshness confidence to characterize the behavior of a weak consistency technique for a data store. Fourth, BG simplifies evaluation of data stores by reducing their performance to two values: Socialites and Social Action Rating (SoAR) using a pre-specified SLA. Fifth, BG is data store agnostic and its shared nothing architecture enables evaluating data stores with high processing capabilities. Sixth, BG's visualization tool empowers an evaluator to quickly author databases with different characteristics and invoke and monitor the benchmarking process for evaluating a data store.

1.2 Thesis Outline

The rest of this dissertation is organized as follows. Chapter 2 starts with a brief survey of six different well-known benchmarks shown in Table 2.1: RUBiS [4], RUBBoS [87] TPC-C [53], YCSB [29], YCSB++ [88] and LinkBench [8]. For each benchmark we describe its data model and compare its characteristics with BG. Chapter 3 introduces the conceptual schema for BG's social graph, its logical data model and its thirteen interactive social actions. Chapter 4 describes the novel features of BG including its extensible software architecture that scales to evaluate the fastest data stores. Chapter 5 describes the physical data design used by BG to create social graphs. Chapter 6 describes BG's validation mechanism used to compute the amount of unpredictable data and freshness confidence for a solution. Chapter 7 emphasizes on BG's rating mechanism to compute the SoAR and Socialites rating for a solution. Chapter 8 illustrates use of BG to evaluate the performance of a single node data store, demonstrates how BG can be used to understand performance trends for different solutions and explains how BG can be used to study the scalability of multi-node data stores. Finally Chapter 9 concludes by describing the long term research directions that shape the future of BG.

Chapter 2

Related Work

BG falls in the *vector based* approach of [98] that models application behavior as a list of actions and sessions (the ‘vector’) and randomly applies each action to its target data store with the frequency a real application would apply the action. The input workload file of BG specifies the frequency of different actions and sessions, configuring BG to emulate a wide range of social networking applications. (See Table 4.1 for three example mixes.) This flexibility is prevalent with both YCSB [29] and YCSB++ [88]. In fact, our implementation of BG employs the core components of YCSB and extends them with new ones such as the actions of Section 3.2, validation mechanism of Chapter 6, D-Zipfian, BGCoord, and BG’s visualization deck. Those with hands on experience with YCSB find BG familiar with the following key modifications and extensions:

1. A more complex conceptual schema specific to social networks.
2. Simple table operations of YCSB have been replaced with social actions and sessions.
3. BG consumes an SLA to compute two ratings for a data store: SoAR and Socialites. If no SLA is specified, BG executes the same as YCSB by imposing a fixed amount of workload using a fixed number of threads T.
4. BG quantifies the amount of unpredictable data produced by a data store. techniques and solutions.
5. BG also computes the probability of producing valid data as a function of time to characterize the behavior of a weak consistency technique. We term this freshness confidence.
6. BG employs a shared-nothing architecture and constructs self-contained fragments of its database to ensure concurrent socialites emulated by independent BGClients are unique, see Section 4.3. This eliminates the need for coordination between BG-Clients during benchmarking phase, enabling BG to scale to a large number of nodes.
7. BG includes a visualization tool to empower an evaluator to monitor an in-progress benchmark and identify bottlenecks.

	TPC-C	RUBBoS	RUBiS	YCSB	YCSB++	BG	LinkBench
Target App	On-Line Transaction Processing	Online News Forums	E-Commerce Auction Sites	Cloud Services	Cloud Services	Interactive social Networking Actions	Facebook's Production MySQL Deployment
URL	www.tpc.org	jmob.ow2.org/ rubbos.html	rubis.ow2.org	github.com/ brianfrank cooper/YCSB	github.com/ MiloPolte /YCSB	bgbenchmark.org	github.com/ facebook /linkbench
Year introduced	1992	2002	2004	2010	2011	2013	2013

Table 2.1: Overview of 7 benchmarks for simple operations.

Some of BG's extensions to YCSB are similar to those that differentiate YCSB++ from YCSB. For example, the concept of multiple BGClients managed by BGCoord is similar to how YCSB++ supports multiple YCSB clients. However, there are also differences. First, YCSB++ includes mechanisms specific to evaluate table stores such as HBase. These include function shipping and fine grained access control. Instead of these, BG emphasizes interactive social networking actions and their implementation with alternative data stores. While extension 6 of BG (see the previous paragraph) is similar to ingest-intensive extension of YCSB++, it goes beyond simple ranges that partition data across multiple nodes. BG logically partitions friendships and resources of members to construct N self-contained independent social networks where N is the number of BGClients.

Second, YCSB++ consists of an elegant mechanism to quantify the inconsistency window: The lag in acknowledged data store changes that are not seen by other clients for some time due to the use of a weak consistency semantics such as eventual consistency [112]. BG captures the impact of such design decisions by quantifying the amount of unpredictable data and freshness confidence. All three metrics are in synergy and may co-exist in a benchmark.

Finally, both YCSB and YCSB++ lack the concept of an SLA to rate a data store. SLAs are the essence of both A and C benchmarks of TPC [53]. For example, TPC-A measures transactions per second (tps) subject to a response time constraint. BG is similar as it employs SLAs to rate a data store, see Chapter 7. It is different than TPC because it focuses on social networking actions and incorporates unpredictable data as a component of SLA.

Table 2.1 shows seven popular benchmarks developed and used to evaluate data stores. Others [53] include the Wisconsin benchmark [19], TPC-E/H/DS/VMS/Energy [32], BigBench [49], and the Linked Data Benchmark Council (LDBC) [31]. The Wisconsin benchmark is a single-user microbenchmark and amongst the very first DBMS benchmarks. This benchmark was developed to evaluate the various components within a relational database. The TPC-E simulates the OLTP workload of a brokerage firm. TPC-H and TPC-DS are decision support benchmarks. TPC-VMS extends TPC-C, TPC-E, TPC-H, and TPC-DC benchmarks by adding the methodology to obtain performance metrics for virtualized databases. Finally, TPC-Energy augments the existing TPC benchmarks with energy metrics. BigBench extends TPC-DS benchmark with semi-structured and unstructured data and is detailed in Section 2.4. The LDBC focuses on the graph-shaped data from different applications, developing methodologies to evaluate the performance of graph databases.

Below, we present the alternative benchmarks shown in Table 2.1, starting with the most relevant. Each section compares BG with each alternative. We do not repeat a discussion

Key feature	TPC-C	RUBBoS	RUBiS	YCSB	YCSB++	BG	LinkBench
Unpredictable reads	✗	✗	✗	✗	✗	✓	✗
Parallelism	✓	✓	✓	✗	✓	✓	✗
SLA	✓	✗	✗	✗	✗	✓	✗
Inconsistency Window	✗	✗	✗	✗	✓	✓	✗
Rating mechanism	✗	✗	✗	✗	✗	✓	✗
Visualization Tool	✗	✓	✓	✗	✓	✓	✗

Table 2.2: Key features of 7 popular benchmarks for simple operations.

of YCSB and YCSB++ in the aforementioned paragraphs. This chapter concludes with a discussion of the BigBench benchmark.

2.1 LinkBench

Similar to BG, LinkBench [8] is a benchmark developed for social networking systems. Both have a complicated conceptual schema related to that of a social networking system and assume similar workload characteristics. The workloads consist of actions that are similar to social interactions users perform in a social networking system which are fairly simple and short-lived. However, LinkBench's approach differs from that of BG. BG simulates socialites performing social networking actions while LinkBench uses workloads derived from traces of Facebook's production database system. Unlike LinkBench, BG's workload is stateful, see Section 5.2. BG is data store agnostic and emulates the entire storage stack including in-memory caches while LinkBench focuses on their persistent sharded MySQL storage layer only. To elaborate, at Facebook, persistent storage for the social graph is provided by sharded MySQL databases. Facebooks memcached and TAO cache clusters provide a caching layer that can serve most reads, so the MySQL layers production workload is comprised of cache-miss reads and all writes [8].

2.2 TPC-C

The TPC-C [71] benchmark from the Transaction Processing Council is an on-line transaction processing (OLTP) benchmark for comparing alternative OLTP solutions using various hardware and software configurations. In sharp contrast, BG is data store agnostic and one may use it to evaluate the performance of any data store.

TPC-C involves a mix of five concurrent transactions of different types which include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. These transactions do look-up, update, insert and delete, and must exhibit atomicity, consistency, isolation and durability (ACID) properties.

With the assumed ACID properties, TPC-C does not have a metric to compute the amount of stale data such as those available in BG and YCSB++. TPC-C quantifies the processing capability of a system using its throughput, transactions per minute (tpm-C). TPC-C reports this metric along with the total system cost ($/tpm-C) where the system cost is an approximation of the true cost of the vendor-supplied portion of the system to the end-user including maintenance costs [53].

TPC-C introduced the concept of SLA in order to compute system performance. But unlike BG for which the SLA requirement is an input parameter that can be adjusted for different workloads, TPC-C measures throughput of a system while satisfying a **fixed** SLA that requires 90% of each type of transaction to have a response time of at most 5 seconds, except stock-level which can be at most 20 seconds. Finally, similar to the previously discussed benchmarks, TPC-C supports parallelism and may use multiple nodes to impose a higher load on its target transaction processing system.

2.3 RUBiS and RUBBoS

RUBiS is an auction site benchmark modeled after eBay.com. It was developed due to absence of benchmarks for web sites with dynamic content and is used to evaluate application design patterns and application server's performance scalability. Unlike BG which evaluates data stores, the target application for RUBiS is the application server. So it does not have a metric to compute the amount of stale data produced in an application and computes the application performance in terms of the number of requests processed per minute.

RUBiS implements the core functionality of an auction site such as selling, browsing and bidding by implementing 26 user interactions with the application data stored in a relational database management system.

The benchmark generator tool for RUBiS emulates users generating workloads for the dynamic content sites. Similar to BG this tool can run on multiple machines and can be used to emulate multiple concurrent clients and an increasing rate of interactions with the system.

RUBiS is extended with a tool which collects utilization statistics (CPU, memory, network bandwidth, etc.) on each of the client machines while running the benchmark. At the end of the benchmark execution, RUBiS displays detailed statistics about the overall throughput (requests/minute) and response time statistics. Similar to BG's visualization deck, this tool provides immediate insight into the system behavior by providing both big picture and in-depth details.

RUBBoS [87] is very similar to RUBiS and is developed to evaluate the performance of application servers and their scalability. RUBBoS was modeled after slashdot.com and implements the core functionality of an online news forum such as browsing and submitting comments and stories, reviewing stories and rating comments.

RUBBiS and RUBBoS assume their infrastructure produces correct results and have no means of quantifying either stale data or the duration of time that the system produces stale data. Moreover, both benchmarks lack a framework to rate a data store. These concepts are supported by BG, differentiating it from RUBiS and RUBBoS.

2.4 BigBench

BigBench [49] is an end-to-end big data benchmark which was developed based on a product retailer. The business cases for the retailer were the main driver for identifying the five main category of queries that constitute BigBench's workload. These categories are : Marketing, Merchandising, Operations, Supply Chain and New Business Models. They motivate complex operations covering different dimensions of big data analytics. Hence, BigBench covers a variety of data (structured, semi-structured and un-structured) and their associated analytic such as those used in support of decision support applications.

BG is different because it emphasizes simple operations resembling OLTP style workloads. Moreover, BG abstracts the features of a social networking site instead of a retailer. Finally, BG assumes a data store may produce stale data while BigBench assumes its target data store provides the correct results always.

Chapter 3

BG's Conceptual and Logical Data Models

This chapter describes BG's data model used to evaluate the performance of a data store for interactive social networking actions. We start with a conceptual design of data and its reduction to two logical data models: relational and JSON. Subsequently, Section 3.2 and 3.3 describe the different actions and sessions supported by BG, respectively.

3.1 Conceptual Data Model

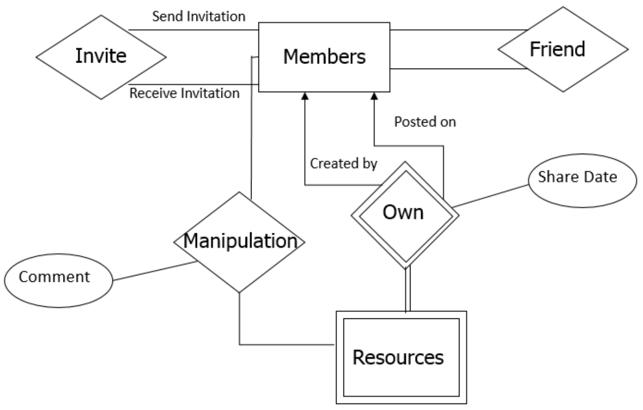
Figure 3.1.a shows the conceptual data model of BG's database using the Entity-Relationship (ER) data model [28]. The Member entity set consists of accounts registered with a social network that belong to individual people. Its attributes include a unique identifier and a number of string attributes such as firstname, lastname and others. The number of these attributes and their lengths are configuration parameters and can be adjusted to generate different database sizes.

In addition, each member may have either zero or 2 images. With the latter, one is a thumbnail and the second is a higher resolution profile image. While thumbnail images are small and in order of KBs, the profile images are in order of tens and hundreds of KBs if not MBs. Typically, thumbnails are displayed when listing friends of a member and the profile image is displayed when visiting a member's profile.

A member may either extend an invitation to or be friends with another member. Figure 3.1.a captures this using the "Invite" and "Friend" relationship sets[1], respectively.

A member may "own" resources such as images, a posted question, a technical manuscript, etc. These entities are grouped in one set named "Resources". The existence of a resource depends on it being "owned" by a member. Hence, Resources is a weak entity set and the participation of a resource in the "owned" relationship is mandatory.s A member may post a resource, say an image, on the profile of another member, represented as a ternary relationship between two members and a resource. In this relationship, the two members might be the same member where the member is posting the resource on her

[1] An alternative captures both relationships with one Friend relationship set and uses an attribute to differentiate between invitations and friendships.



3.1.a Conceptual data model of BG’s database.

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

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

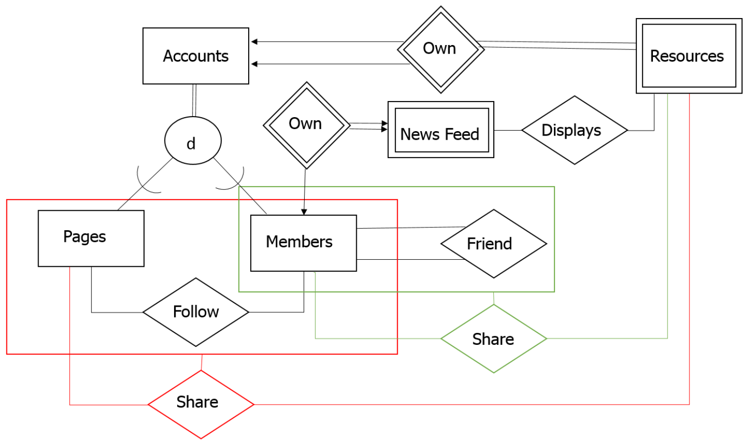
```
GridFSImages.Files.: {
    "id" : “”
    "length“:””
    "chunkSize" :””
    "uploadDate" : “”
    "md5" : “”
}

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

3.1.b JSON-Like data model of BG’s database.

Members(userid,username,pw,firstname,lastname,gender,dob,jdate,ldate,address,tel,email,profileimage,thumbnail)
Friend(userid1,userid2,status)
Resources(rid,creatorid,wallUserid,type,body,doc)
Manipulation(mid,modifierid,rid,resourceCreatorid,timestamp,type,content)

3.1.c Relational data model of BG’s database.

Figure 3.1: Conceptual and logical data models of BG’s database.



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

```
Members:{
          "userid":""
          "username":""
          "pw":""
          "firstname":""
          "lastname":""
          "gender":""
          "dob":""
          "jdate":""
          "address":""
          "email":""
          "tel":""
          "imgid":""
          "thumbid":""
          "pendingFriends":[]
          "confirmedFriends":[]
          "NewsFeed":[]
}

Resource:{
          "rid":""
          "creatorid":""
          "walluserid":""
          "type":""
          "body":""
          "doc":""
}

Manipulations:{
          "mid":""
          "rid":""
          "modifierid":""
          "type":""
          "content":""
          "timestamp":""
}

SharedResources:{
          "srid":""
          "rid":""
          "recipients":[]
}
GridFSImages.Files:{
          "id":""
          "length":""
          "chunkSize":""
          "uploadDate":""
          "md5":""
}
GridFSImages.Chunks:{
          "id":""
          "files_id":""
          "n":""
          "data":""
}
```

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

Members(userid,username,pw,firstname,lastname,gender,dob,jdate,ldate,address,tel,email,profileimage,thumbnail)

Friend(userid1,userid2,status)

Resources(rid,creatorid,wallUserid,type,body,doc)

Manipulation(mid,modifierid,rid,resourceCreatorid,timestamp,type,content)

SharedResources(srid,rid,creatorid)

SharedResourceRecipients(srid,userid)

NewsFeed(userid,srid,rid,creatorid)

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

Figure 3.2: Conceptual and logical data models of BG's database including feed following actions.

Database parameters	
M	Number of members in the database.
P	Number of pages in the database.
ϕ	Number of friends per member.
ρ	Number of resources per member.
ι	Number of followers per page.
ϱ	Number of pages followed by each member.

Table 3.1: BG's database parameters and their definitions.

Term	Definition
Account	Any registered profile with a social network which can be either a member or a page.
Action	A logical social operation implemented by a web page and invoked by a mouse click.
Inter-arrival time	Idle time between two socialite sessions emulated by one thread.
Member	It is an account registered with a social network system that belongs to an individual person. Members can perform all BG's 13 social actions. They can be friends with other members and follow pages.
Page	It is an account registered with a social network that is used to connect people (members) to a topic, which can be a company, celebrity, brand, etc. Pages can be followed only by members, do not follow anyone and do not have any friends.
Resource	An entity that a socialite may browse and post a comment on, e.g., an image.
Session	A sequence of actions by a socialite.
Socialite	A member/page engaged in a social session.
Think time	Idle time between actions in a session.

Table 3.2: Social networking terms and their definitions.

own profile. A member (either the owner or another) may comment on a resource. This is implemented using the "Manipulation" relationship set. A member may restrict the ability to comment on a resource only to her friends. Figures 3.1.b and 3.1.c show the logical design of the ER diagram with both MongoDB's JSON-like and relational data models. An experimentalist builds a database by specifying the number of members (M) in the social network, number of friends per member (ϕ), and resources per member (ρ), see Table 3.1. Some of the relationships might be generated using either a uniform or a skewed distribution. For example, one may use a Zipfian distribution with either (a) exponent 0.99 to model a uniform distribution that assigns 20% of friendships to 20% of members, or (b) exponent of 0.27 to model a skewed distribution to assign 62% of friendships ($M \times \phi$) to 20% of members [63].

Figure 3.2.a shows the remainder of BG's schema that supports feed following actions as detailed in see Section 3.2. The "Invite" and "Manipulation"' relationship sets showed in Figure 3.1.a have been removed to make this figure readable. The Account entity set is a generalization of Members (shown in Figure 3.1) and Pages entity sets. Pages are special topics such as business, celebrities, brands and etc. that share resources with their followers who are Members. (See Table 3.2 a for list of terms and their definitions.) Members also share resources with other Members who are their friends. These two relationships are captured using the two "Share" relationship sets.

A member owns a News Feed entity. The News Feed entity for a member displays the top k events shared by Pages followed by the member or shared by her friends. This is captured using the "Displays" relationship set. Figures 3.2.b and 3.2.c show the logical design for the ER diagram shown in Figure 3.2.a with both MongoDB's JSON-like and relational data models.

With this conceptual model, an experimentalist must specify the number of pages (P), the number of followers for each page, ι, and the number of pages followed by each member, ϱ, in addition to the parameters mentioned before, see Table 3.1. BG first inserts the members, their friendship relationships and their resources into the data store. Next, it inserts the pages, creates the following relationships and inserts the page resources, see Chapter 5.

One may specify BG workloads at the granularity of an action, a session, or a mix of these two possibilities. A session is a sequence of actions with ϵ think time between actions and ψ inter-arrival time between sessions. Table 1.1 shows BG's list of actions and its compatibility with several social networking sites. We detail these in Section 3.2. Section 3.3 enumerates the different sessions supported by BG. One may extend BG with new sessions consisting of an arbitrary mix of actions.

Similar to YCSB [29], BG exposes both its schema and its actions to be implemented by an experimentalist. Thus, the experimentalist may target an arbitrary data store, specify its physical data model for the conceptual data model of Figure 3.2.a, provide an implementation of the actions of Table 1.1, and run BG to evaluate the target data store. In addition, the experimentalist may use BG to evaluate various physical data representations for a given data store, see Section 8.1. As detailed in Chapter 7, these functionalities are divided between a Coordinator, named BGCoord, and N slave processes, named BGClients.

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

3.2 Actions

This section details thirteen social networking actions that an experimentalist may use to define a workload for a data store. We present each action by describing an implementation of it using SQL-X, MongoDB, and CASQL to highlight the significance of database parameters such as number of friends per member and their impact on the performance of a data store. The three data stores are described in Chapter 1. A more in depth analysis of alternative physical data design with the different systems is provided in Section 8.1.

For the first two actions, we present SoAR numbers (see Chapter 1) using the following SLA: 95% of requests observe a response time equal to or faster than 100 msec with the amount of stale data less than 0.1%. Member ids are generated using a Zipfian distribution with exponent 0.27. Reported numbers were obtained from a dedicated hardware platform consisting of six PCs

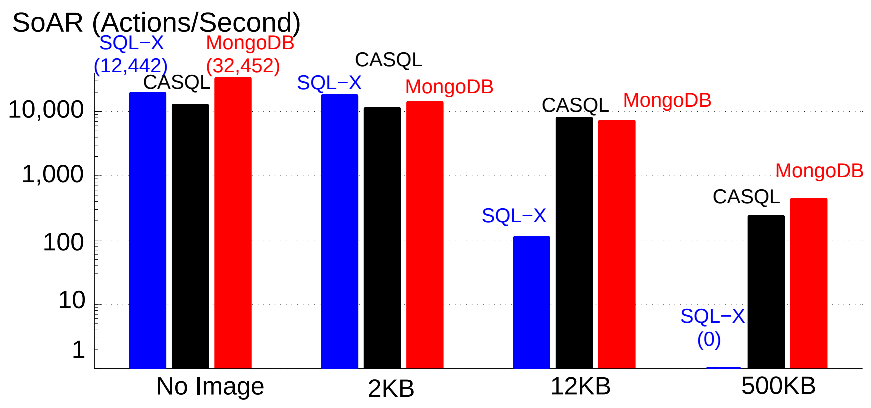


Figure 3.3: SoAR of 3 different systems with View Profile and different profile image sizes, M=10,000, β=100 msec, α=95%, ϵ=ψ=0, θ=0.27.

connected using a Gigabit Ethernet switch. Each PC consists of a 64 bit 3.4 GHz Intel Core i7-2600 processor (4 cores with 8 threads) configured with 16 GB of memory, 1.5 TB of storage, and one gigabit networking card. Even though these PCs have the same exact model and were purchased at the same time, there is some variation in their performance. To prevent this from polluting our results, the same one node hosts the different data stores for all ratings. This node hosts both memcached and SQL-X to realize CASQL. Either all or a subset of the remaining 5 nodes are used as BGClients to generate requests for this node, see Section 4.3. With all reported SoAR values greater than zero, either the disk, all cores, or the networking card of the server hosting a data store becomes fully utilized. When SoAR is zero, this means the data store failed to satisfy the SLA with one single threaded BGClient issuing requests, N=T=1.

3.2.1 View Profile, VP

View Profile (VP) emulates a socialite visiting the profile of either herself or another member. Its input include the socialite's id and the id of the referenced member, U_r. BG generates these two ids using a random number conditioned using the Zipfian distribution of access with a pre-specified[2] exponent (specified in the input configuration file by the experimentalist who is benchmarking a system), see Figure 4.4. When the Socialite's id is not equal to U_r the output includes U_r's profile attributes and the following two aggregate information: U_r's number of friends, U_r's number of resources (e.g., images). If the socialite is referencing her own profile (socialite's id equals U_r's id) then VP retrieves a third aggregate information: U_r's number of pending friend invitations.

VP retrieves all attributes of U_r except U_r's thumbnail image. This includes U_r's profile image assuming the database is created with images, see Section 3.1. An implementation of VP with the different data stores is as follows. With MongoDB (SQL-X), it retrieves the document (row) corresponding to the specified U_r userid [14]. With MongoDB, VP may compute the number of friends and pending invitations by counting the number of elements in pendingFriends and confirmedFriends arrays, respectively. It may count the number of resources posted on U_r's wall by querying the Resources collection using the predicate "walluserid = U_r's userid". With SQL-X, VP may issue different aggregate queries. With a CASQL system, VP may construct two different keys using U_r's userid: self profile when socialite's id equals U_r's userid and browse profile

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

when socialite's id does not equal U_r's userid. Section 8.1 describes other physical data designs using SQL-X and MongoDB. Presence of a profile image and its size impact SoAR of different data stores for VP dramatically [97], see Section 8.1. Figure 3.3 shows the performance of three different systems for a BG database consisting of no-images, and a 2 KB thumbnail image with different sizes for the profile image: 2 KB, 12 KB, and 500 KB. These settings constitute the x-axis of Figure 3.3. The y-axis reports SoAR of different systems.

With no images, MongoDB provides the best performance, outperforming both SQL-X and CASQL by almost a factor of two. With 12 KB images, SoAR of SQL-X drops dramatically from thousands to hundreds[3]. With 500 KB image sizes, SQL-X cannot perform even one VP action per second that satisfies the 100 msec response time (with 1 thread), producing a SoAR of zero. SoAR of MongoDB and CASQL also decrease as a function of larger image size because they must transmit a larger amount of data to the BGClient using the network. However, their decrease is not as dramatic as SQL-X.

CASQL outperforms SQL-X because these experiments are run with a warm up phase that issues 500,000 requests to populate memcached with key-value pairs pertaining to different member profiles. Most requests are serviced using memcached (instead of SQL-X). While this does not payoff[4] with small images, with 12 KB and 500 KB image sizes, it does enhance performance of SQL-X considerably.

3.2.2 List Friends, LF

List Friends (LF) emulates a socialite viewing either her list of friends or another member's list of friends. This action retrieves the profile information of each friend. In the presence of images, it retrieves only the thumbnail image of each friend. At database creation time, BG empowers an experimentalist to configure a database with a fixed number of friends per member (ϕ). This has a significant impact on the performance of a data store. To illustrate, Figure 3.4 shows SoAR of the alternative data stores for LF with three different ϕ values. per member. (The median Facebook friend count is 100 [110, 9].) A larger ϕ value lowers the rating of all data stores. Overall, CASQL provides the best overall performance with 50 and 100 friends per member. With SQL-X and CASQL the network on the data store and the node hosting the cache respectively become the bottleneck. With MongoDB the CPU on the node hosting the data store becomes fully utilized. Below, we describe implementation details of each system in order to explain the presented results.

SQL-X must join the Friend table with the Members table (see Figure 3.1.c) to compute the socialite's list of friends. We assume the friendship relationship between two members is represented as 1 record[5] in Friend table, see Figure 3.1.c. CASQL caches the final results of the LF action and enhances SoAR of SQL-X by less than 10% with ϕ values of 50 and 100. With ϕ=1000, SQL-X slows down considerably and can no longer satisfy the 100 msec response time requirement. The CASQL alternative is also unable to meet this SLA because each key-value is larger than 1 MB, the maximum key-value size supported by memcached. This renders memcached idle, redirecting all requests issued by CASQL to SQL-X, producing zero for system SoAR. One may modify memcached to support key-value pairs larger than 2 MB (ϕ=1000 and each thumbnail is 2 KB) to realize an enhanced SoAR with CASQL.

[3]We use SQL-X with the physical data design shown in Figure 3.1.c. This design can be enhanced to improve performance of SQL-X by ten folds or more [14]. See Section 8.1 for details.

[4]There are several suggested optimization to the source code of memcached to improve its performance [86, 7]. Their evaluation is a digression from our main focus. Instead, we focus on the standard open source version 2.5.1 [77].

[5]See Section 8.1 for a discussion of representing friendship as 2 records and its impact on SoAR.

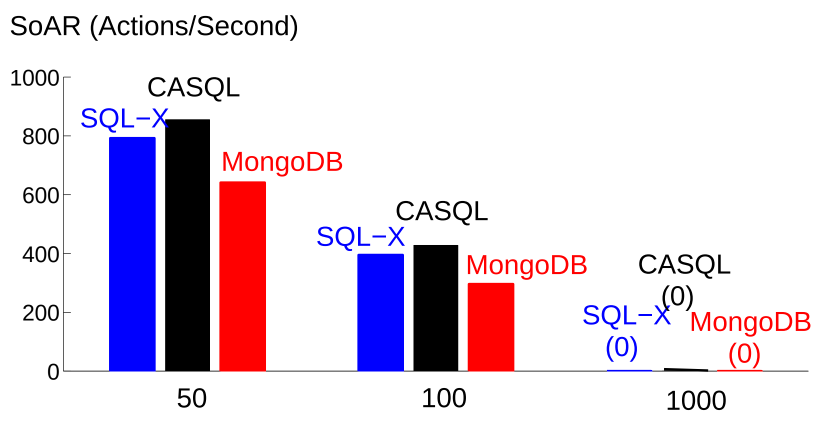


Figure 3.4: SoAR of List Friends with 3 different data stores as a function of number of friends per member (ϕ), M=10,000, β=100 msec, α=95%, ϵ=ψ=0, θ=0.27.

With MongoDB, an implementation of LF may retrieve the confirmed friends either one document at a time or as a set of documents. With both approaches, the BG client starts by retrieving the confirmedFriends array of the referenced member, see Figure 3.1.b. With one document at time, the client processes the array and for each userid, retrieves the profile document of that member. With a set at a time, the client provides MongoDB with the array of userids to retrieve a set containing their profile documents. With both, SoAR of MongoDB is inferior to the join operator of SQL-X.

3.2.3 Other Actions

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

Invite Friend, IV: This action enables a socialite, say B, to invite another member, say A, of the social network to become her friend. With MongoDB, this action inserts B's userid into A's array of pendingFriends, see Figure 3.1.b. With both SQL-X and CASQL, this operation inserts a row in the Friend table with status set to "pending", see Figure 3.1.c. CASQL invalidates the memcached key-value pairs corresponding to A's self profile (with a count of pending invitations) and A's list of pending invitation. A subsequent VP invocation that references these key-value pairs observes a cache miss, computes the latest key-value pairs, and inserts them in the cache.

Accept Friend Request, AFR: Socialite A uses this action to accept a pending friend request from Member B of the social network. With MongoDB, this action inserts (a) A's userid in B's array of confirmedFriends, and (b) B's userid in A's arrays of confirmedFriends, see Figure 3.1.b. Moreover, it removes B's userid from A's array of pendingFriends. With both SQL-X and CASQL, this operation updates the "status" attribute value of the row corresponding to B's friend request to A to "confirmed", see Figure 3.1.c. CASQL invalidates the memcached key-value pairs corresponding to self profiles of members A and B, profiles of members A and B as visited by others, list of friends for members A and B, list of pending invitations for member A.

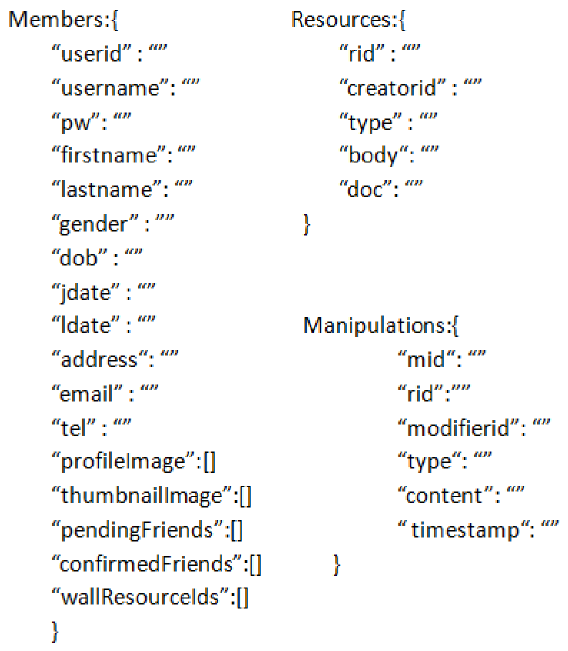
Reject Friend Request, RFR: Socialite A uses RFR to reject a pending friend request from a Member B. BG assumes the system does not notify Member B of this event. With MongoDB, we implement RFR by simply removing B's userid from the A's array of pendingFriends, see Figure 3.1.b. With both SQL-X and CASQL, RFR deletes the friend request row corresponding to B's friend request to A, see Figure 3.1.c. CASQL invalidates the key-value pairs corresponding to A's self profile and pending friend invitations from memcached.

Thaw Friendship, TF: This action enables Socialite A to remove Member B as a friend. With MongoDB, TF removes A's userid from B's array of confirmedFriends and vice versa, see Figure 3.1.b. With both SQL-X and CASQL, TF deletes the row corresponding to the friendship of user A and B (with status equal to "confirmed") from Friends table, see Figure 3.1.c. CASQL invalidates the key-value pairs corresponding to the list of friends for users A and B, self profile of users A and B, and profiles of users A and B as visited by other users (because their number of friends has changed).

View Top-K Resources, VTR: When BG populates a database, it requires each member to create a fixed number of resources. Each resource is posted on the wall of a randomly chosen member, including oneself's wall. View Top-K Resources (VTR) enables a socialite to retrieve and display her top k resources posted on her wall. Both the value of k and the definition of "top" are configurable. Top may correspond to those resources with the highest number of "likes", date of last view/comment (recency), or simply its id. At the time of this writing, BG supports the last one. With MongoDB, we analyzed two implementations. With the first, VTR queries the Resources collection in a sorted order to retrieve top k resources posted on a socialite's profile. With the second, a sorted array of resource ids for the resources posted on each member's wall is stored with the member's information; VTR queries this array to retrieve the top k resource ids for a socialite and then queries the Resources collection to retrieve the resources, see Figure 3.5.a. The latter required issuing two queries and resulted in a performance lower than that observed with the former. With SQL-X and CASQL, VTR queries the Resources table and uses top k ordered using their rid. CASQL constructs a unique key using the action and socialite userid, serializes the results as a value, and inserts the key-value pair in memcached for future reference.

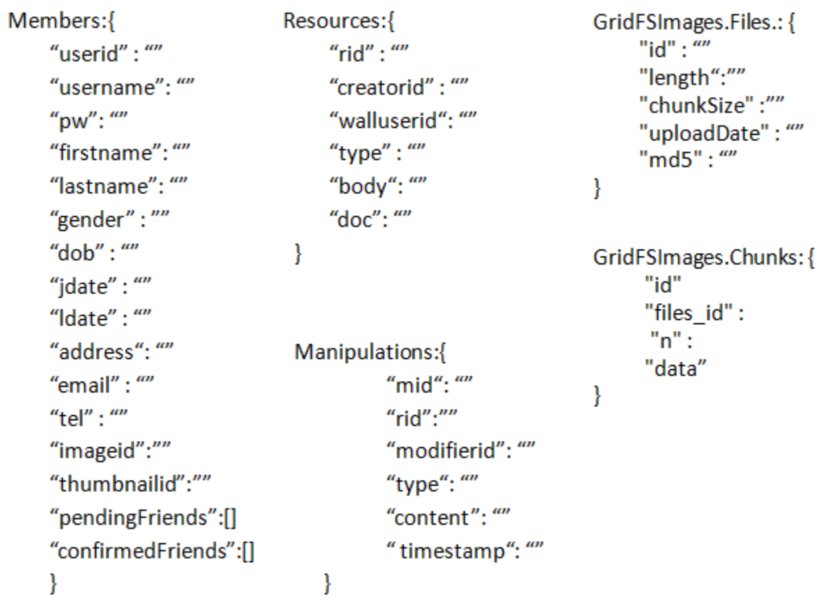
View Comments on Resource, VCR: A socialite displays the comments posted on a resource with a unique id (rid) using VCR action. BG generates rids for this action by randomly selecting a resource owned by a member (selected using a Zipfian distribution). With MongoDB, we analyzed two different implementations. The first implementation supported the schema shown in Figure 3.1.b where the comments for every resource are stored within the manipulation array attribute for that resource. With this implementation, VCR retrieves the elements of manipulation array of the referenced resource, see Figure 3.1.b. The second implementation creates a separate collection for the comments named Manipulations, see Figure 3.5. With this implementation, VCR queries the Manipulations collection for all those documents whose rid equals the referenced resource'id. With SQL-X, VCR employs the specified identifier of a resource to query the Manipulation table and retrieve all attributes of the qualifying rows, see Figure 3.1.c. CASQL constructs a unique key using rid to look up the cache for a value. If it observes a miss, it invokes the procedure for SQL-X to construct a value. The resulting key-value pair is stored in memcached for future reference.

Post Comment on a Resource, PCR: A socialite uses PCR to comment on a resource with a unique rid. BG generates rids by randomly selecting a resource owned by a member selected using a Zipfian distribution. It generates a random array of characters as the comment for a user. The number of characters is a configurable parameter. With MongoDB, PCR is implemented by either generating an element for the manipulation array attribute of the selected resource, see Figure 3.1.b or generating a document, setting its rid to the unique identifier of the referenced resource and inserting it into the Manipulations collection, see Figure 3.5. With SQL-X and CASQL, PCR inserts a row in the Manipulation table. CASQL invalidates the key-value pair corresponding to comments on the specified resource id.



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

Resources:{
    "rid" : ""
    "creatorid" : ""
    "type" : ""
    "body": ""
    "doc": ""
 }

Manipulations:{
        "mid": ""
        "rid":""
        "modifierid": ""
        "type": ""
        "content": ""
        " timestamp": ""
    }
```

3.5.a. Data Model 1



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

Resources:{
    "rid" : ""
    "creatorid" : ""
    "walluserid": ""
    "type" : ""
    "body": ""
    "doc": ""
 }

Manipulations:{
        "mid": ""
        "rid":""
        "modifierid": ""
        "type": ""
        "content": ""
        " timestamp": ""
    }

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

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

3.5.b. Data Model 2

Figure 3.5: Two alternative JSON-Like data models of BG's database.

Delete Comment from a Resource, DCR: This action enables a socialite to delete a unique comment posted on one of her owned resources chosen randomly. With MongoDB, an implementation of DCR either removes the element corresponding to the comment from the manipulation array attribute of the identified resource, see Figure 3.1.b or removes the document corresponding to the comment posted on the referenced resource from the Manipulations collection, see Figure 3.5. With SQL-X and CASQL, DCR deletes a row of the Manipulation table. CASQL invalidates the key-value pair corresponding to comments on the specified resource id.

Share Resource, SR: Each member (page) in BG's social graph can share a resource she (it) owns either publicly or with a list of specific members. If shared publicly, then the resource will be available for all the members who are friends with the owner of the resource (are following the page). If shared specifically with a list of members, then the resource is made available to those members only.

When an experimentalist creates a BG database with pages, BG requires each member to follow a fixed number of pages (ϱ) and each page to consist of a fixed number of resources (ρ). These resources are created on the page's own wall and can be shared publicly with all the followers of the page or specifically with a list of them. Share Resource (SR) action enables a socialite to share resources she owns with all her followers or a subset of them. We analyze two implementations of this action using MongoDB and SQL-X. With the first implementation, Pull, every time a resource is shared, a new record for the shared item is created [100]. This record maintains the resource information as well as some meta information such as a list of followers allowed to see the shared resource (recipients in Figure 3.2.b). The second implementation, Push, extends Pull as follows. It maintains a News Feed entity for each follower and inserts the shared resource in this entity [100], see Section 8.4 for details.

The SR action requires the memberid of either the member or the page who emulates the action, the resourceid for the resource owned by the member which is going to be shared with followers (the resource is owned by the socialite performing the action) and a list of followers allowed to see the shared item. If this list is set to -1 then the resource is shared with all the followers of the resource owner. This is the only action that can be emulated by pages. BG can also be configured to issue $r\%$ of SR actions by pages and $(1 - r\%)$ of them by members. The value of r can be given to BG as an input parameter in the configuration file.

Both the number of followers per page (ι) and the number of friends per member (ϕ) may impact the throughput observed with different data stores using workloads consisting of SR actions. For example with the Push approach, every time a celebrity (page with more than 1,000,000 followers) shares a resource, the News Feed entities for all its followers need to be updated, see Section 8.4.2 for more details.

View News Feed, VNF: Each member of BG owns a News Feed entity. The VNF action enables a socialite to retrieve her news feed and display the top k resources shared with the member. These resources may be shared publicly or privately with the member by other members she is following. The definition of top k is configurable. This action requires the memberid of the Member emulating the action, and the value for k and returns a list of k resources satisfying the order required by the application, as the events in the member's news feed.

An implementation of this technique may use the design of the Share Resource action as follows. With the first one, Pull, upon a VNF action, the pages followed by the member and her friends are queried. Next, all the resources shared by these members/pages which are either shared publicly or specifically shared with the member are retrieved. Finally the top k criteria is applied to limit the number of shared resources to k and the final k resources are returned as the events displayed on the member's News Feed. With the second implementation, Push, the events shared with a member (publicly or privately) are maintained in a structure (News Feed entity) and updated upon an SR action. So a VNF action only retrieves this News Feed structure for each member emulating the action.

The performance of VNF is impacted by the number of friends per member (ϕ) and number of pages followed by each member (ϱ). With a larger value of ϕ and ϱ for a member, VNF will require retrieving a larger amount of data that results in slower service times.

Computing the SoAR for a data store with a workload consisting of VNF and SR actions depends on the duration of the experiment, see Section 7.3. This is because with an increase in the experiment duration, a larger number of SR actions are emulated and the News Feed for each member will consist of larger number of events resulting in slower response times and a lower throughput.

3.3 Sessions

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

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

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

2. ViewFrdProfileSession, $\{$VP(m_i), VTR(m_i), LF(m_i), VP(m_j), VTR(m_j) $\mid m_j \in$ LF(m_i))$\}$: After viewing self profile and top k resources, Member m_i lists her friends and picks one friend randomly, m_j. Next, m_i views m_j's profile and m_j's top k resources. If m_i has no friends, the session terminates without performing the two actions on m_j.

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

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

5. InviteFrdSession, $\{$VP(m_i), VTR(m_i), LF(m_i), IF(m_j), VFR(m_j) $\mid m_j \cap$ LF(m_i) = $\emptyset\}$: After viewing self profile and top k resources, Member m_i lists her friends, and selects a random member m_j who has no pending or confirmed relationship[6] with m_i. (If all members of the

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

database are m_i's friend then the remaining two actions are not performed.) She invites m_j to be friends and concludes by listing her own pending friend requests.

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

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

8. ThawFrdshipSession, $\{$VP(m_i), VTR(m_i), LF(m_i), TF(m_j) $\mid m_j \in$ LF(m_i), LF(m_i) $\}$: After viewing self profile and top k resources, Member m_i lists her friends and select a friend m_j randomly. Next, m_i thaws friendship with m_j. This session concludes with m_i listing her friends. If m_i has no friends, she skips the remaining actions and the session terminates.

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

3.4 Summary

This chapter described the conceptual data model of BG and its reduction to two logical data models, JSON and relational. In addition, we described the different actions and sessions supported by BG. Conceptually, BG is a stateful benchmark. It generates actions and session requests that are meaningful. For example, it does not issue an Invite Friend action on behalf of Socialite A to Member B if they are friends. Similarly, with a session such as DeleteCmtOnResSession that enables Socialite A to delete a comment created on one of the resources posted on her wall, BG generates a comment that is guaranteed to exist prior to invocation of the DCR command of this session. Section 5.2 describes how BG implements these concepts physically.

Chapter 4

An Extensible and Scalable Benchmarking Framework

BG is a scalable benchmark that utilizes multiple nodes to generate requests for highly scalable data stores. It is an extensible framework with the following two architectural components: one or more BGClients and one BGCoord. The BGCoord is a coordinator that starts and stops the BGClients, gathers performance statistics from the BGClients and aggregates them together, and rates a data store per discussions of Chapter 7. Each BGClient is an extensible component that exposes BG's conceptual database schema and its social networking actions, and data store initialization and shut down for implementation by an experimentalist, see Chapter 3. This makes BG data store agnostic by empowering an experimentalist to tailor BG to any data store and use its core functionality to initialize the data store, create a database, populate the database with data, generate a pre-specified mix of actions using a fixed number of threads, and rate a data store. The core of each BGClient is a plug-and-play infrastructure that is modular and configurable. Its thirteen actions are modules and one may extend BG with new actions by authoring new modules. A BGClient reads a configuration file that specifies a mix of actions and sessions, the degree of skew that should be used to reference members, and the duration of an experiment specified either as a fixed amount of time or a fixed number of requests issued to the data store.

Next section describes BG's input configuration file and how it can be used to invoke a valid workload. Section 4.2 describes the the software architecture of BGClient, its core components, its extensibility and how an experimentalist may introduce new commands as modules. Finally, Section 4.3 describes how multiple BGClients partition a social graph in order to generate requests without synchronizing with one another to scale to a large number of nodes. This discussion includes a novel decentralized implementation of Zipfian named D-Zipfian [13]. D-Zipfian ensures the distribution of generated requests is not impacted by the degree of parallelism, i.e., the number of BGClients used to generate requests in a scalable manner. We detail BG's rating mechanism in Chapter 7.

4.1 Mix of Actions

One may evaluate a data store by specifying a workload consisting of a mix of actions. Three example workloads are shown in Table 4.1. Note that it is acceptable to specify zero as the frequency of an action as it causes BG to not issue that action. An action may reference one or more entities (e.g., members, resources, comments used by the actions). BG selects the identity of the entity us-

BG Social Actions	Type	Very Low (0.1%) Write	Low (1%) Write	High (10%) Write
View Profile	Read	40%	40%	35%
List Friends	Read	5%	5%	5%
View Friend Requests	Read	5%	5%	5%
Invite Friend	Write	0.04%	0.4%	4%
Accept Friend Request	Write	0.02%	0.2%	2%
Reject Friend Request	Write	0.02%	0.2%	2%
Thaw Friendship	Write	0.02%	0.2%	2%
View Top-K Resources	Read	49.9%	49%	45%
View Comments on a Resource	Read	0%	0%	0%
Post Comment on a Resource	Write	0%	0%	0%
Delete Comment from a Resource	Write	0%	0%	0%
Share Resource	Write	0%	0%	0%
View News Feed	Read	0%	0%	0%

Table 4.1: Three mixes of social networking actions.

ing either a random or a Zipfian distribution as specified by the configuration file, see Section 4.3.1. For example, with the View Profile action, the configuration file may specify the use of the Zipfian distribution to select a member performing this action and the use of a random distribution to select the member whose profile is being viewed. Similarly, the Thaw Friendship action, a Zipfian distribution generates the identity of a member to emulate the action and a random distribution to select one of this member's friends to thaw friendship[1].

In addition, each workload is symmetric. A symmetric workload is one that does not change the state of the database characterized by its size and the structure of its social graph. With a workload involving write actions, the database size remains constant only if total size of the records inserted equals the total size of records deleted from the database. The structure of a social graph is determined by the number of friends per member. In order for it to remain unchanged, workload involving updates should maintain the number of friends per member and the number of comments posted per resource constant (The number of friends per member should remain the same as the initial number of friends inserted per member and the number of comments posted per resource should remain the same as the initial number of comments posted for each resource). The workload mixes of Table 4.1 ensure a constant database size and a fixed social graph structure by ensuring the following two conditions:

- As the size of each comment posted on a resource is constant, the total number of new comments inserted should be equal to the total number of comments deleted from the system. This is possible by specifying an equal percentage for the Post Comment on Resource and Delete Comment From Resource actions.

- The rate at which friendships are generated should be equal to the rate at which friendships are being thawed from the system. This is possible if the percentages specified for the friend-

[1]BG is developed using YCSB's core modules and inherits its Uniform and Latest distributions that can be used to select a member for emulating an action.

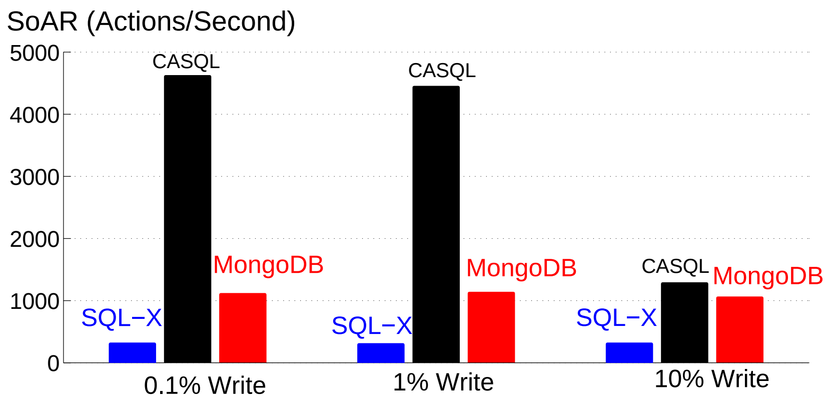


Figure 4.1: SoAR for 3 mixes of read and write actions for three different data stores, M=10,000, 12 KB image size, ϕ=100, $\rho = 100$, β=100 msec, α=95%, τ=0.01%, ϵ=ψ=0, θ=0.27.

ship actions (Accept Friend Request, Reject Friend Request, Thaw Friendship and Invite Friend) satisfy the following constraints:

1. Percentage of Invite Friend = percentage of Reject Friend Request + percentage of Accept Friend Request
2. Percentage of Thaw Friendship = percentage of Accept Friend Request

An experimentalist may specify an arbitrary mix of actions as a workload by defining new set of parameters (percentage for actions and distribution). Figure 4.1 shows SoAR of the different systems with the 3 mixes for a database with 10,000 members and 100 friends per member. MongoDB[2] outperforms SQL-X for the different mixes by almost a factor of 3. The CASQL is sensitive to the percentage of write actions as they invalidate cached key-value pairs, causing read actions to be processed by the RDBMS. With a Very Low (0.1%) write mix, CASQL outperform MongoDB by more than a factor of 3. With a high percentage (10%) of write actions, SoAR of CASQL is slightly higher than MongoDB.

The observed trends with SQL-X and MongoDB change depending on the mix of VP and LF actions. In Figure 4.1, MongoDB outperforms SQL-X because the frequency of VP is significantly higher than LF. If this was switched such that LF is more frequent than VP then SQL-X would outperform MongoDB. A system evaluator should decide the mix of actions based on the characteristics of a target application.

4.2 Extensibility and Modularity

In addition to being configurable, BG is modular and extensible. Hence, one may adapt BG by modifying or updating it to support new data stores and new application requirements. BG consists of loosely-coupled modules/components allowing each component to be modified with minimal impact on the rest of the system[3]. Figure 4.2 shows BG's components. The Generator component,

[2]We use MongoDB with its strict write concern which requires each write to wait for a response from the server [58]. Without this option, MongoDB produces stale data (less than 0.01%).

[3]As BG was developed on top of YCSB's core functionality it inherits some of YCSB's modules [29].

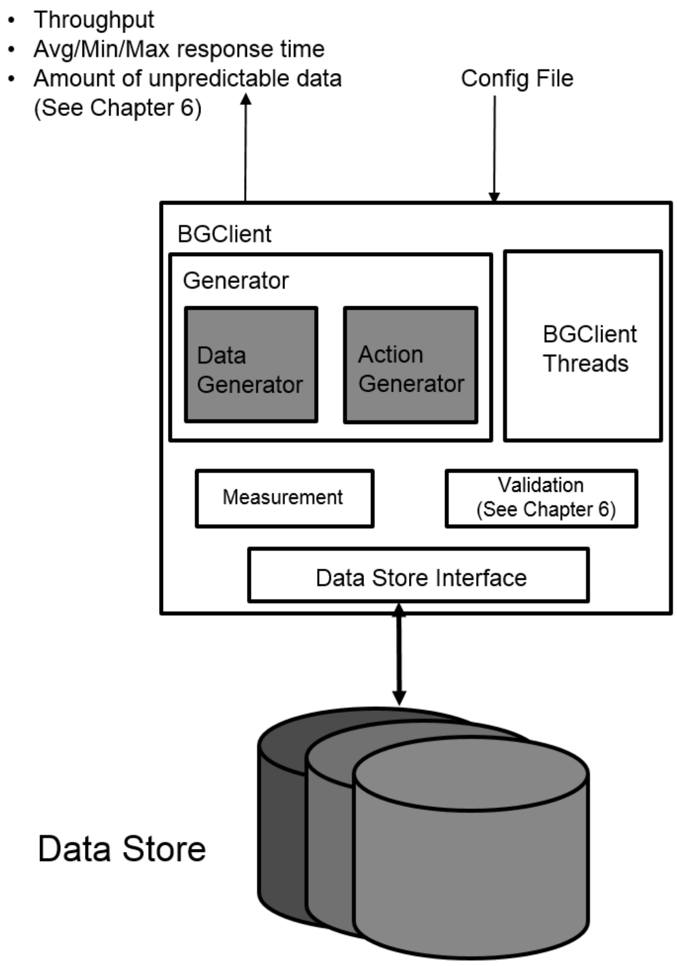


Figure 4.2: BGClient architecture and its components.

produces both the data loaded into the data store during the load phase and the workload (see Section 4.1) issued to the data store during the benchmarking phase. This component consists of two sub-components: ActionGenerator and DataGenerator. The ActionGenerator module is responsible for generating the actions that need to be issued against the data store. These actions depend on the mix of workload specified (by the configuration file) for the benchmarking phase. The DataGenerator is responsible for populating the data store with data during the load and identifying member ids that participate in an action. The member distribution controls the activity level for the different members (see Section 8.4). A member with a higher activity level is picked more frequently to issue actions to the data store. These components allow for introducing new social actions or member distributions easily.

BG's Measurement component provides methods to quantify the time used to execute different actions. It can be easily extended to measure, summarize and report new performance metrics. The Validation component is responsible for computing the amount of unpredictable data as detailed in Chapter 6. One may modify and extend the algorithms in this module with minimal impact on other components. And finally, the Data Store Interface component is fully exposed an available to be tailored to other data store back-ends to evaluate new data stores. We now discuss how BG can be extended in more details.

- New Actions: Introducing new actions within BG is simple and can be done in three steps. The first step is to define the interface for the action, this is the name of the action, the value it returns and the parameters required for the action, for example the Thaw Friendship action requires two memberids, one identifies the socialite performing the action and the second is the target member of this action. Next the ActionGenerator is extended with the body of the action to identify the memberids. In our example, this is picking socialite A using a distribution, picking the second member from the friends of Member A[4] , issuing the action, updating BG's internal data structures and logging the appropriate information required for validation. The final step is to measure the duration of the action by adding the action to the measurement component.

- New Distributions: BG can also easily be extended to support different distributions to generate memberids. For this purpose, the DataGenerator component can be extended to create the new distribution by assigning probability of references for each of the members. Next, BG can be configured to use the new distribution to select members when emulating actions.

- New Data Store Back-ends: BG can be used to evaluate new data stores. This can be done by introducing a new implementation of the Data Store Interface class for the target data store. This class should implement all the social actions that have an interface in the ActionGenerator.

BG is extensively being used all around the world and its highly modular design minimizes the effort required for using and extending it. We conducted a survey on 25 of BG's users to understand its usability, extensibility with a variety of NoSQL data stores and ease of software update, see Appendix A for the surveys. These members were familiar with Java and Java IDE Tools and had a good understanding of their data store's functionalities. 88% of the members found BG easy to install, and claimed to be comfortable using BG and extending it for a new data store. Figure 4.3 shows the average number of hours spent by these users to extend BG for their data stores.

4.3 Scalable Request Generation

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

To address this issue, BG implements a scalable benchmarking framework using a shared-nothing architecture, see Figure 4.4. Its software components are as follows:

[4]BG is stateful and maintains the information about members' friends and pending friendship relationships in its internal data structures. This is required in order for BG to perform valid actions. For example if Member A and Member B are already friends BG should not try to generate a friend request from Member A to Member B.

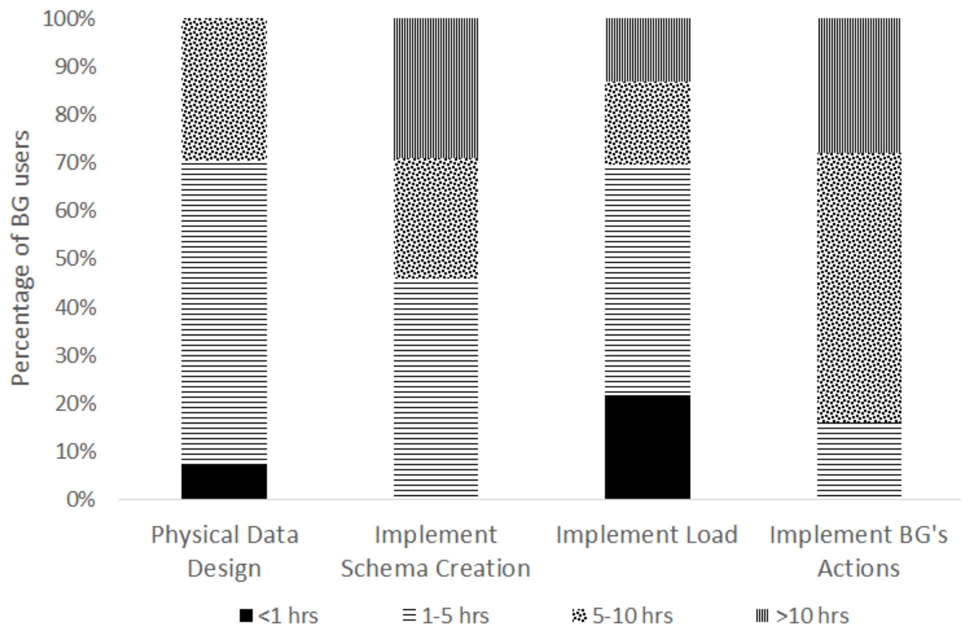


Figure 4.3: Average number of hours spent to extend BG for a new data store.

1. A coordinator, BGCoord, computes SoAR and Socialites rating of a data store by implementing both an exhaustive and a heuristic search technique. Its input are the SLA specifications and parameters of an experiment, see Table 7.2. It computes the fraction of workload that should be issued by each worker process, named BGClient, and communicates it with that BGClient. BGCoord monitors the progress of each BGClient periodically, aggregates their current response time and throughput, and reports these metrics to BG's visualization deck for display, see Item 3. Once all BGClients terminate, BGCoord aggregates the final results for display by BG's visualization deck.

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

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

Once BGCoord activates N BGClients, each BGClient generates its workload independently to enable the benchmarking framework to scale to a large number of nodes. We realize this by constructing the physical database of Section 3.1 to consist of N logical self-contained fragments. Each fragment consists of a unique collection of members, resources, and their relationships. BG can realize this because it generates the benchmark database. BGCoord assigns a logical fragment to one BGClient to generate its workload. This partitioning enables BG to implement uniqueness of concurrent socialites, i.e., the same member does not manipulate the database simultaneously. Note that construction of logical fragments has no impact on the size of the physical database and its parameter settings such as number of friendships.

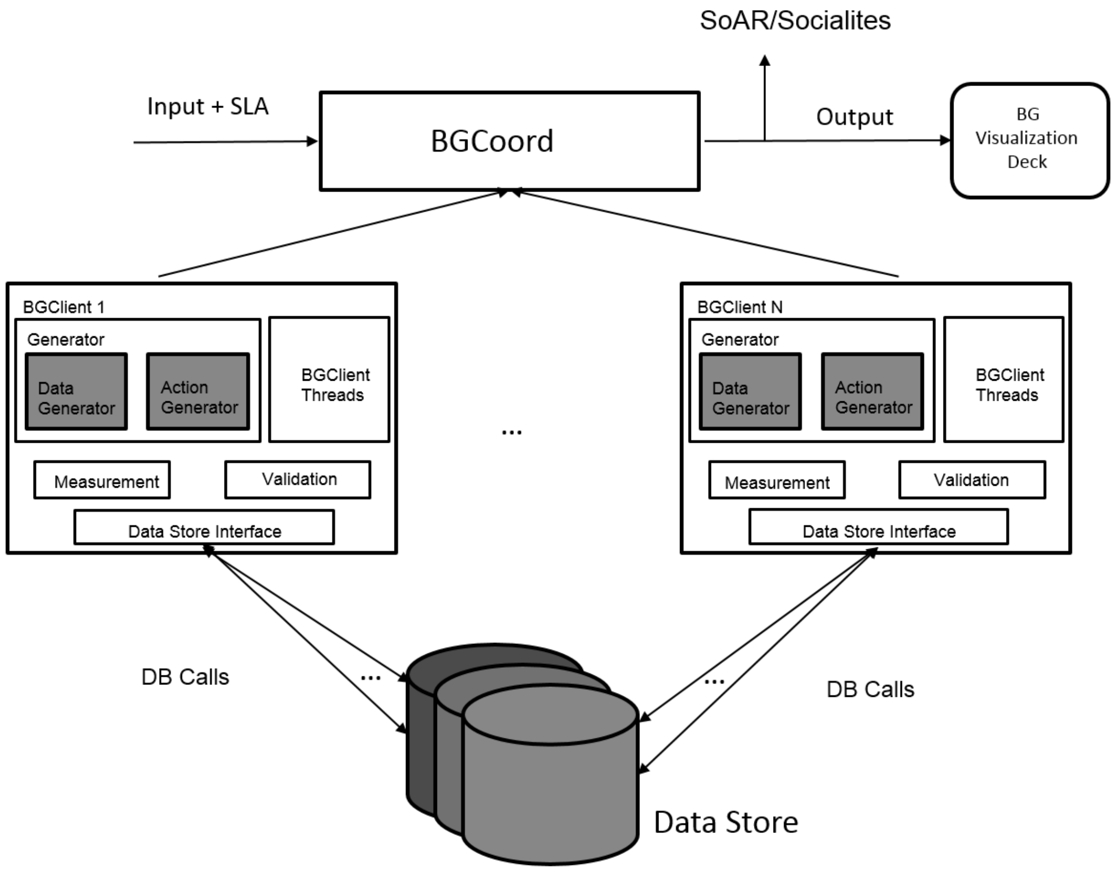


Figure 4.4: BG's shared nothing architecture.

With BG, an experiment may specify a Zipfian distribution with a fixed exponent and vary the number of BGClients, value of N. BGClients implement a decentralized Zipfian, D-Zipfian [13], that produces the same distribution of references with different values of N. This enables us to compare results obtained with different number of BGClients with one another. We implement D-Zipfian to incorporate heterogeneity of nodes (hosting BGClients) where one node produces requests at a rate faster than the other nodes. D-Zipfian assigns more load to the fastest node by assigning a larger logical fragment to it and requiring it to produce more requests. Hence, the N BGClients complete issuing requests at approximately the same time. For details of D-Zipfian, see Section 4.3.1.

Figure 4.5 shows the throughput of MongoDB as a function of threads (T) that emulate concurrent socialites. Presented results pertain to different number of BGClients performing View Profile (VP) action with D-Zipfian and exponent 0.27. The Socialites rating is the length of each curve along the x-axis. While it is 317 with 1 BGClient, it increases 3.2 folds to 1024 with 8 (16) BGClients. A solid rectangular box denotes the SoAR rating with a given number of BGClients. It also increases as a function of N; from 15,800 with 1 BGClient to 33,200 with 16 BGClients. With 1 BGClient, the client component of MongoDB used by the BGClient to communicate with its server component is limiting the observed ratings. We know it is not the hardware platform because we can run multiple BGClients on one node to observe higher ratings. Four physical nodes are used

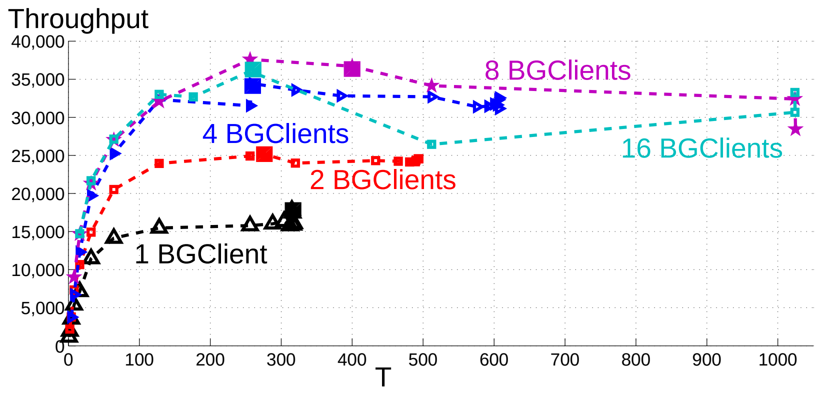


Figure 4.5: MongoDB's throughput as a function of T with View Profile (VP) action and different number of BGClients, N. M=10,000, No image, β=100 msec, α=95%, ϵ=ψ=0, θ=0.27.

in the experiments of Figure 4.5. Both SoAR and Socialites rating remain unchanged from 8 to 16 BGClients. D-Zipfian ensures the same distribution of requests is generated with 1 to 16 BGClients.

4.3.1 D-Zipfian: A Decentralized Implementation of Zipfian

With most applications, a uniform random distribution of access to data items is typically not realistic due to Zipf's law [122]. This law states that given some collection of data items, the frequency of any data item is inversely proportional to its rank in its frequency table. This means the data item with the lowest rank in the frequency table will occur more often than the data item with the second lowest rank, the data item with the second lowest rank in the frequency table will occur more often than the one with the third lowest rank, and so on and so forth. By manipulating the exponent[5] θ that characterizes the Zipfian distribution one may emulate different rules of thumb such as: 80% of requests (ticket sales [33], frequency of words [122], profile look-ups) reference 20% of data items (movies opening on a weekend, words uttered in natural language, members of a social networking site).

Use of multiple BGClients raises the following research question: How do BGClients produce requests such that their overall distribution conforms to a pre-specified Zipfian distribution? One solution, named Replicated Zipfian (*R-Zipfian*), requires each BGClient to employ the specified Zipfian distribution with the entire population independently. R-Zipfian is effective when BG produces workloads with read only references. It also accommodates heterogeneous nodes where each node produces requests at a different rate as each BGClient uses the entire population to generate the Zipfian distribution.

However, with BG, R-Zipfian introduces additional complexity in two cases. First, different BGClients might be required to reference a unique data item at an instance in time in order to model reality. For example, they might be required to emulate a unique user of a social networking site performing an action such as accepting friend request. R-Zipfian would require additional software to coordinate multiple BGClients to guarantee uniqueness of the referenced data items. Second, BG measures the amount of unpredictable data produced by a data store using workloads that are a mix of read and write actions. It time stamps these to detect unpredictable reads. R-Zipfian would

[5]See Equation 4.1 in Section 4.3.1.

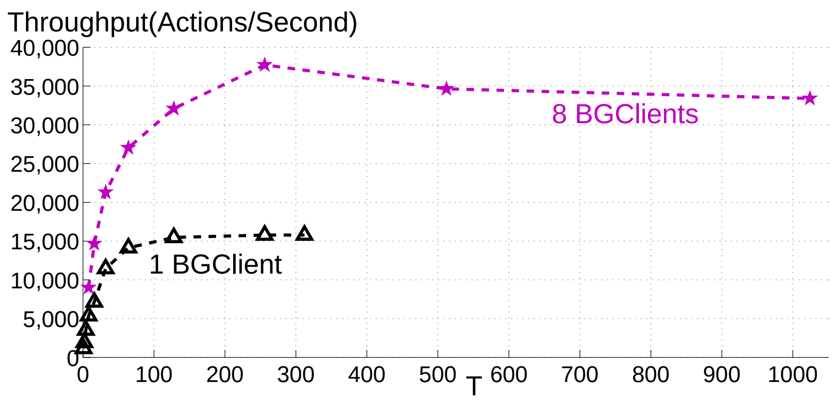


Figure 4.6: Performance of MongoDB with two different number of BGClients.

require BG to utilize synchronized clocks [70, 78, 43, 61, 93] to detect unpredictable reads. Both complexities are avoided by partitioning data items across BGClients.

With partitioning, BGCoord assigns a disjoint set of data items to each BGClient. A BGClient issues requests that reference its assigned data items only. This ensures BGClients reference unique data items simultaneously. Moreover, the potential read-write and write-write conflicts are localized to each BGClient and its partition, enabling it to quantify its observed amount of unpredictable data using its own system clock and independent of the other BGClients.

With N BGClients, each BGClient must reference data items such that the overall distribution of references conforms to a Zipfian distribution with a pre-specified θ. Moreover, the resulting distribution must remain constant as a function of N, i.e., the degree of parallelism employed by BG. This property is not trivial to realize because each BGClient has a subset of the original population and issues requests independently. As discussed in Section 4.3.1, if each BGClient uses the original θ with a subset of the population, the resulting distribution becomes more uniform as we increase the value of N. This is not desirable because it produces experimental results that are erratic and difficult to explain. For example, one may quantify the processing capability of a cache augmented SQL (CASQL) data store [48, 86, 7] with n_1 and n_2 BGClients ($n_1 < n_2$) and observe a lower processing capability with n_2 because its distribution pattern is more uniform (which reduces the cache hit rate with a limited cache size). This is avoided by making the Zipfian distribution independent of N.

Problem Statement

With a Zipfian distribution, assuming M is the number of data items, the probability of data item i is:

$$p_i(M, \theta) = \frac{\frac{1}{i^{(1-\theta)}}}{\sum_{m=1}^{M}(\frac{1}{m^{(1-\theta)}})} \tag{4.1}$$

where θ characterizes the Zipfian distribution.

Assuming data items are numbered 1 to M, a centralized implementation of Zipfian is as follows:

1. Compute the probability of each data item using Equation 4.1.

2. Compute array A consisting of M elements where the value of the first element is set to the probability of the first item, $A[1] = p_1(M, \theta)$, and the value of each remaining element m is the sum of its assigned probability and the probabilities assigned to the previous $m-1$ elements, $A[i] = \sum_{j=1}^{i} p_j(M, \theta)$, $1 \leq i \leq M$. The last element of the array, $A[M]$, should equal 1 because sum of the M probabilities equals one. If this value is slightly lower than 1 then set it to 1.

3. Generate a random value r between 0 and 1. Identify the k^{th} element of the array that satisfies the following two conditions: a) A[k] is greater than or equal to r, and b) Either A[k-1] has a value lower than r or is non existent (because k is the first element of A). Produce k as the referenced data item, $1 \leq k \leq M$.

For an example, see discussions of Table 4.2 in Section 4.3.1.

The challenge is how to parallelize this simple algorithm such that N BGClients reference data items and produce a distribution almost identical to that of one BGClient referencing data items. Below, we differentiate between local and global probability of a data item to provide a mathematical formulation of the problem.

Each data item i has a local and a global probability of reference. Its local probability specifies its likelihood of reference by its assigned BGClient k with m_k data items. One possible definition of the local probability of an object i is provided by Equation 4.1, $p_i(m_k, \theta)$. An algorithm may either use this definition or provide a new one, see Crude in Section 4.3.1 and D-Zipfian in Section 4.3.1. The global probability of data item i assigned to BGClient k is a function of its local probability and the ratio of the number of references performed by BGClient k (O_k) relative to the total number of references (O) by N BGClients:

$$q_i(M, \theta, N) = \frac{O_k}{O} \times p_i(m_k, \theta) \tag{4.2}$$

With 1 BGClient, $N = 1$, local and global probability of a data item are identical, $q_i(M, \theta, 1) = p_i(m_k, \theta)$, because all data items are assigned to one BGClient, $m_k = M$, and that BGClient issues all requests, i.e., $\frac{O_1}{O} = 1$. With 2 or more BGClients, the global probability of a data item is lower than its local probability, $q_i(M, \theta, 1) \leq p_i(m_k, \theta)$. See discussions of Table 4.2 in Section 4.3.1.

In sum, a parallel implementation of Zipfian with N BGClients may manipulate either the number of data items (m_k) assigned to each BGClient k and their identity, the definition of the local probability of an object i, the number of references (O_k) made by BGClient k, or all three. Note that by manipulating O_k, we are not shortening the execution time of one BGClient relative to the others, see Section 4.3.1. To the contrary, as detailed in Section 4.3.1, D-Zipfian manipulates O_k to require a mix of fast and slow BGClients to complete at approximately the same time. This is important because if one BGClient finishes considerably sooner than the others then the degree of parallelism is no longer N.

A mathematical formulation imposes the following constraint on a parallel implementation of Zipfian: $q_i(M, \theta, N) \approx q_i(M, \theta, 1)$ for all i and $N > 1$. It states the computed global probability of each data item i with two or more BGClients should be approximately the same as its computed probability with one BGClient.

The concepts presented in this section are demonstrated with an example in the next section using two naïve and intuitive ways to parallelize the centralized implementation of the Zipfian. They pave the way for the correct parallel implementation, D-Zipfian of Section 4.3.1. The reader may skip to Section 4.3.1 for the final solution.

Data item	Zipfian/Crude with $N = 1$		Crude with $N = 3$	
i	$p_i(12, 0.01) = q_i(12, 0.01, 1)$	A[i]	$p_i(4, 0.01)$	$q_i(4, 0.01, 3)$
1	0.319014588	0.319014588	0.477558748	0.159186249
2	0.160616755	0.479631343	0.240440216	0.080146739
3	0.107512881	0.587144224	0.160944731	0.053648244
4	0.080866966	0.668011191	0.121056305	0.040352102
5	0.064838094	0.732849284	0.477558748	0.159186249
6	0.054130346	0.786979631	0.240440216	0.080146739
7	0.046469017	0.833448647	0.160944731	0.053648244
8	0.04071472	0.874163368	0.121056305	0.040352102
9	0.036233514	0.910396882	0.477558748	0.159186249
10	0.032644539	0.943041421	0.240440216	0.080146739
11	0.029705152	0.972746574	0.160944731	0.053648244
12	0.027253426	1	0.121056305	0.040352102

Table 4.2: Example with 12 data items and θ=0.01.

Example and Two Naïve Approaches

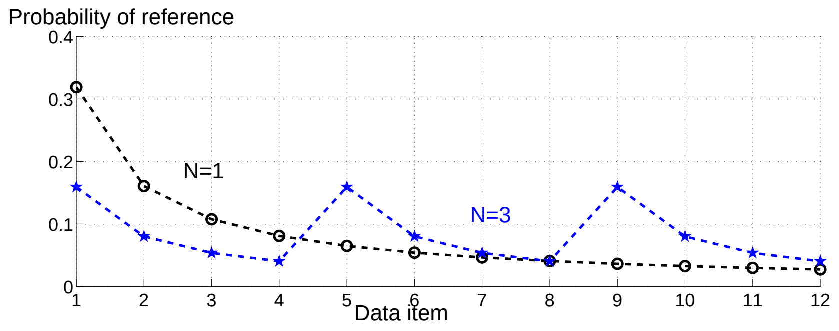
This section uses a small population consisting of twelve data items (M=12) to demonstrate the concepts presented in Section 4.3.1. In addition, it describes two naïve techniques to parallelize Zipfian and their limitations.

Table 4.2 shows the local and global properties of the individual data items with 1 and 3 nodes, N=1 and N=3. Its first column shows the individual data items numbered from 1 to 12. Its second and third columns correspond to one node ($N = 1$) and show the local and global probabilities of each data item with the exponent 0.01, θ=0.01, and the values of Array A used by a centralized implementation to generate the Zipfian distribution, respectively. To implement Zipfian, an implementation generates a random value r between 0 and 1, say r=0.5. It produces data item 3 as its output because A[3] exceeds 0.5 and A[2] is less than 0.5. (See Step 2 of the pseudo-code to generate data items in Section 4.3.1 for a precise definition of selecting A[i].)

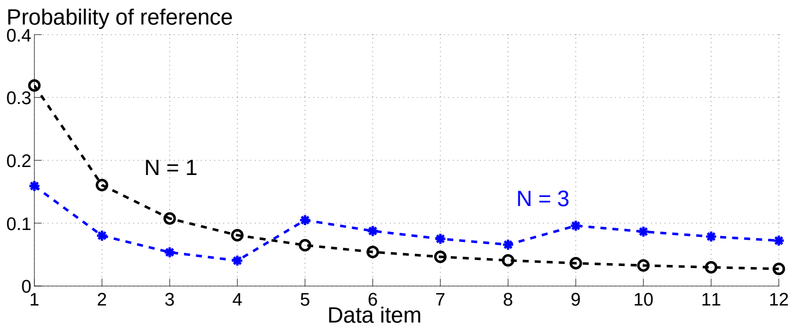
With N BGClients, say N=3, a technique named *Crude* range partitions data items across the BGClients as follows: BGClient 1 is assigned data items number 1 to 4, BGClient 2 is assigned data items number 5 to 8, and BGClient 3 is assigned data items number 9 to 12. It uses Equation 4.1 with $m_i = 4$ and the original θ value (0.01) to compute the local probability of each data item, see the fourth column of Table 4.2. The fifth column of Table 4.2 shows the global probability of each data item with Crude using Equation 4.2 assuming each BGClient produces $\frac{1}{3}$ of references, i.e., $O = 3 \times O_k$. These are significantly different than those with 1 BGClient, compare 2nd and 5th columns, and do not satisfy the mathematical constraint presented in Section 4.3.1.

Crude may assign data items to N BGClients in several other ways including:

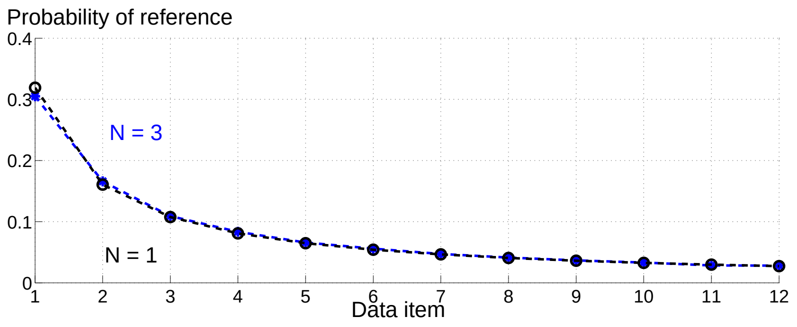
- Hash (instead of range) partition data items using their id i to assign m_k data items to BGClient k.
- Provide BGClient k with m_k assigned data items. Next, each BGClient would use the centralized implementation of Zipfian (see Section 4.3.1) with the entire population to reference a data item. If the referenced data item is not one of the m_k data items then BGClient k discards this request and generates a new one.



4.7.a : Crude



4.7.b : Normalized-Crude



4.7.c : D-Zipfian

Figure 4.7: $q_i(M, \theta, N)$ of data items with three different techniques, M=12, θ=0.01, and N={1, 3}.

While these enable each BGClient to generate a Zipfian distribution independently, the resulting distribution (across all N BGClients) is dependent on the value of N. As we increase the value of N, the resulting distribution becomes more uniform, see Figure 4.7.a. Note that with N=3, the same distribution is repeated 3 times because each BGClient generates its distribution independently with m_k=4 and θ=0.01. Hence, a data item that was referenced infrequently with N=1 is now accessed more frequently. Unless Crude manipulates either its definition of local probability of a data item (p_i) or the number of references issued by a BGClient, the results of Table 4.2 remain unchanged.

A variant of Crude, named *Normalized-Crude*, defines the local probability of a data item i as $p_i = \frac{p_i(M,\theta)}{\sum_{k=1}^{m_k} p_k(M,\theta)}$. This definition utilizes M (instead of m_k) to normalize the probability of data items assigned to each BGClient. With one node, it is identical to the centralized Zipfian because its denominator equals 1 ($m_i = M$ and the sum of the probability of data items equals 1). With more than one node, the global probabilities produced by Normalized-Crude are more uniform than Crude, see Figure 4.7.b assuming $O = 3 \times O_k$. Note that the most popular data item with $N = 1$ has a global probability that is almost twice that with $N = 3$. Section 4.3.1 shows that with a minor adjustment, Normalized-Crude is transformed into the final solution.

D-Zipfian

We present D-Zipfian assuming BGClients are homogeneous and produce requests at approximately the same rate. Subsequently, Section 4.3.1 extends the discussion to heterogeneous BGClients that produce requests at different rates.

Homogeneous BGClients

With N BGClients, D-Zipfian constructs N clusters such that the sum of the probability of data items assigned to each cluster is $\frac{1}{N}$. Given a cluster k consisting of m_k elements and assigned to BGClient k, D-Zipfian overrides the local probability of each data item i as follows:

$$p_i = \frac{p_i(M, \theta)}{\sum_{m=1}^{m_k} p_i(M, \theta)} \tag{4.3}$$

This definition of local probability is identical to that used by Normalized-Crude. D-Zipfian is different because it constructs clusters by requiring the sum of probability of data items assigned to one cluster to approximate $\frac{1}{N}$. Thus, denominator of Equation 6.1 approximates $\frac{1}{N}$. Details of D-Zipfian can be summarized in two steps.

In this first step, BGCoord computes the probability of access to the M data items using Equation 4.1. Next, it constructs N clusters of data items such that the sum of the probability of the m_k data items assigned to cluster k is $\frac{1}{N}$, $\sum_{i=1}^{m_k} p_i(M, \theta) = \frac{1}{N}$. Finally, it assigns cluster k to BGClient k by transmitting[6] the identity of its data items to BGClient k. (A heuristic to construct clusters is described in the following paragraphs.)

In the second step, each BGClient adjusts the probability of its assigned data items using Equation 6.1. Note that the denominator of Equation 6.1 approximates $\frac{1}{N}$ because BGCoord assigned objects to each BGClient with the objective to approximate $\frac{1}{N}$. Finally, each BGClient uses its computed probabilities to generate array A to produce data items, see Section 4.3.1. Generation of the requests by each BGClient is independent of the other BGClients.

One may construct clusters of Step 1 using a variety of heuristics. We use the following simple heuristic. After BGCoord computes the quota for each BGClient k, $Q_k = \frac{1}{N}$, it assigns data items to the BGClients in a round-robin manner starting with the data item that has the highest probability. Once it encounters a BGClient whose Q_k is exhausted, BGCoord attempts to assign the data item with the lowest probability to this BGClient as long as its Q_k is not exceeded. Otherwise, it removes this BGClient from the list of candidates for data item assignment. It proceeds to repeat this process until it either assigns all data items to BGClients or runs out of BGClients. If the later, the coordinator assigns the remaining data items to one of the BGClients[7].

Figure 4.7.c shows D-Zipfian's produced probability with 1 and 3 BGClients and 12 data items. When compared with Figures 4.7.a and 4.7.b, D-Zipfian approximates the original distribution closely.

[6]Alternatively, with a deterministic technique to partition data items into clusters, each BGClient may execute the same technique independently to compute its m_k assigned objects.

[7]With the discussions of Section 4.3.1, this is the fastest BGClient always.

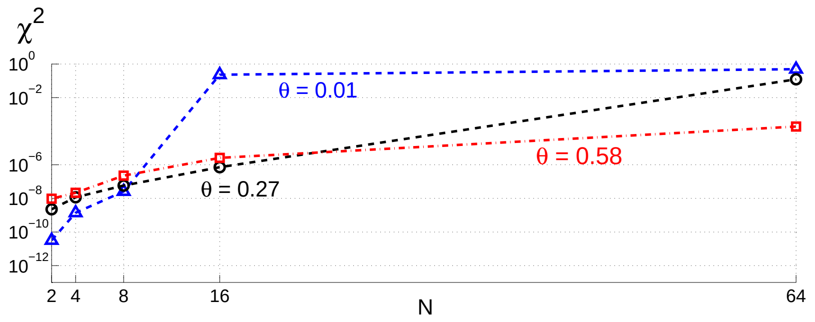


Figure 4.8: χ^2 analysis of centralized Zipfian with D-Zipfian as a function of N, M=10K.

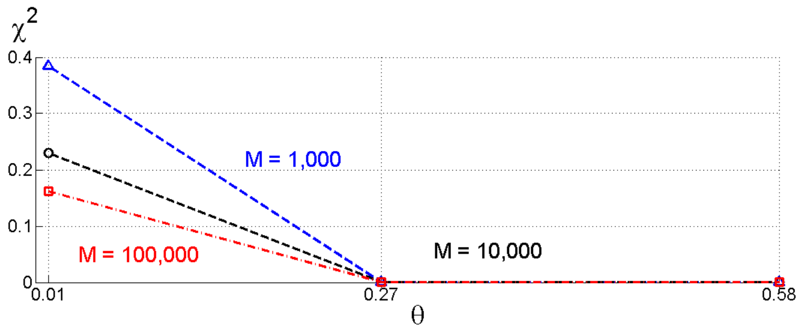


Figure 4.9: χ^2 analysis of centralized Zipfian with D-Zipfian as a function of θ with different number of data items, M.

We use chi-square statistic to compare the distributions obtained with $N = 1$ with those obtained using $N > 1$. The chi-square statistic with $N > 1$ is computed as follows: $\chi^2 = \sum_{i=1}^{M} \frac{(q_i(M,\theta,N)-q_i(M,\theta,1))^2}{q_i(M,\theta,1)}$. A smaller value of χ^2 is more desirable. When $\chi^2 = 0$, it means the probability distribution with $N > 0$ is identical to that with $N = 1$.

Figure 4.8 shows the χ^2 statistic as a function of N BGClients with 10,000 data items and three different θ values. A smaller θ value results in a more skewed distribution. Obtained results show distributions with a handful of BGClients ($N \leq 8$) are almost identical to $N = 1$ as χ^2 value is extremely small. With tens of BGClients, the χ^2 value is higher because there is a higher chance of the sum of probabilities assigned to each BGClient to deviate from $\frac{1}{N}$. This is specially true with a more skewed distribution, θ=0.01. One way to enable D-Zipfian to better approximate a probability of $\frac{1}{N}$ for each BGClient is to increase the number of data items, M. This is shown in Figure 4.9 with three different values of M and θ=0.01. As we increase the value of M, the χ^2 statistic becomes smaller and approaches zero.

Heterogeneous BGClients

It is rare for one to purchase PCs that provide identical performance. As an example, on January 24, 2012, we purchased four identical Desktop computers configured with Intel i7-2600 processors, 16 Gigabyte of memory, and 1 TB of disk storage. When using them as BGClients, we observed one node to be considerably faster than the others. This fast node is almost twice faster than the slowest node. This discrepancy violates the assumption of Section 4.3.1 that with N BGClients, each BG-

R_1	R_2	R_3	R_4	χ^2
1	1	2	2	0.11
1	1.25	1.5	2	0.07
1	2	2	2	0.06
1	1	1	2	0.12
1	4	4	4	0.16

Table 4.3: Processing rate of four BGClients and their impact on the χ^2 statistic, N=4, M=10K, θ=0.27.

R_1	R_2	R_3	R_4	χ^2
1	1	2	2	1.91E-08
1	1.25	1.5	2	1.49E-10
1	2	2	2	1.08E-09
1	1	1	2	1.19E-10
1	4	4	4	6.13E-09

Table 4.4: χ^2 improves dramatically with the refined D-Zipfian, N=4, M=10K, θ=0.27.

Client issues $\frac{1}{N}$ of requests. This increases the error (χ^2) between the distributions observed with $N > 1$ and $N = 1$. As an example, Table 4.3 shows χ^2 observed with five different configurations of four heterogeneous BGClients. R_i denotes the rate at which a BGClient issues requests, see the first four columns of Table 4.3. The last column shows the χ^2 value when θ=0.27, comparing the observed theoretical[8] probabilities with 1 BGClient, i.e., $N = 1$. Each row corresponds to a different configuration of BGClients. For example, the first corresponds to a mix of 4 BGClients where two BGClients are twice faster than the other two BGClients. This results in errors (χ^2 values) significantly higher than those shown in Figure 4.8.

To address this limitation, we change the first step of D-Zipfian (see Section 4.3.1) to construct clusters for each BGClient such that their total assigned probability is proportional to the rate at which they can issue requests. Its details are as follows. Step 1 assigns objects to BGClient k with the objective to approximate a total probability of $\frac{R_k}{\sum_{j=1}^{N} R_j}$ for this BGClient (instead of $\frac{1}{N}$). With this change, the distribution with N BGClients becomes almost identical to that of one BGClient, see Table 4.4.

Discussion

Section 4.3.1 used the observed theoretical probabilities by considering the local probability of a data item in combination with the number of requests, $\frac{O_k \times p_i(M,\theta)}{O \times \sum_{j=1}^{m_k} p_j(M,\theta)}$. This study does not consider the actual generation of requests using a random number generator because it would require a too long a diversion from our main topic. We do wish to note that the considered probabilities are the foundation of generating requests and, without them, it is difficult (if not impossible) to generate

[8]We compute the observed theoretical probabilities by requiring each BGClient k to multiply its computed probabilities for a data item with its number of issued requests divided by the total number of requests issued by all the BGClients, $\frac{O_k \times p_i(M,\theta)}{O \times \sum_{j=1}^{m_k} p_j(M,\theta)}$.

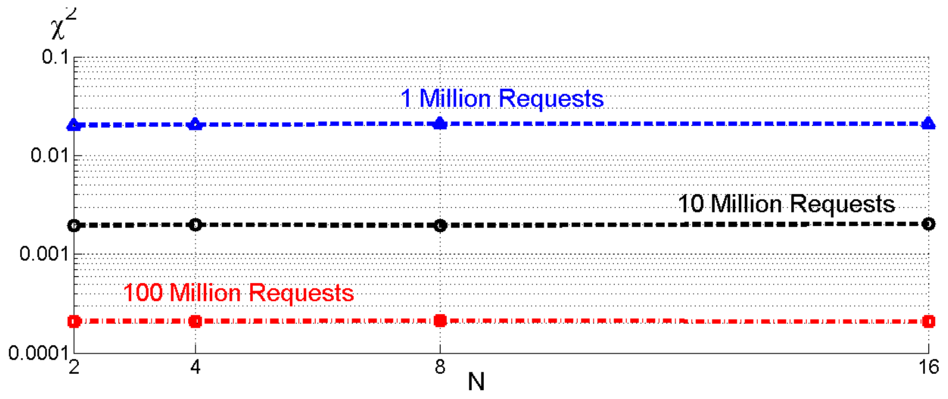


Figure 4.10: χ^2 analysis of an implementation of D-Zipfian generating requests. This analysis compares centralized Zipfian's probability for different data items with D-Zipfian as a function of different degrees of parallelism (x-axis). M=10,000, θ=0.27.

references that produce a Zipfian distribution. An implementation of D-Zipfian with actual request generation is analyzed in Figure 4.10. The y-axis of this figure shows χ^2 statistic, quantifying the difference in observed probabilities with a centralized Zipfian when compared with D-Zipfian and different degrees of parallelism (x-axis). As we increase the number of issued requests, D-Zipfian resembles its centralized counterpart more closely.

With Section 4.3.1, one may apply the concepts of Section 4.3.1 to reduce the observed χ^2 values by several orders of magnitude and very close to zero. The idea is as follows. Once objects are assigned to the different BGClients, the number of references issued by a BGClient k is normalized relative to the total probability of its assigned objects. Thus, assuming the benchmark issues a total of O requests, each BGClient k would issue O_k requests:

$$O_k = O \times \frac{\sum_{i=1}^{m_k} p_i(M,\theta)}{\sum_{j=1}^{N}\sum_{i=1}^{m_j} p_i(M,\theta)} \tag{4.4}$$

While this enhances the χ^2 statistic dramatically, its potential usefulness is application specific. For example, a benchmarking framework may consist of a ramp-up, a ramp-down, and a steady state. Such a framework collects its observations during its steady state. The steady state might be defined as either a duration identified by conditions that mark the ramp-up and the ramp-down phases or a fixed number of requests. With the former, O is not known in advance and the system may not use Equation 4.4. Even when O exists, different values of O_k might be undesirable because different BGClients finish at different times. This is because participating nodes are assumed to be identical and those BGClients with the lowest O_k finish sooner, reducing the degree of parallelism.

We considered constructing V virtual BGClients ($V \geq N$) with several such BGClients mapped to one physical BGClient [45, 102, 95]. This is beneficial as long as it better approximates the quota assigned to each physical BGClient. In our experiments, we observed negligible improvement because approximating the appropriate quota for each virtual BGClient becomes more challenging as we increase the value of V, see discussions of Figure 4.8 in Section 4.3.1.

For an analysis of how BG generates requests while preserving the intended distribution refer to [63]

Chapter 5

BG's Physical Data Design

One may implement how BG creates a social graph in different ways. These choices constitute alternative physical data designs for BG. A physical design in turn impacts how BG generates its actions. This is specially true for the write actions because BG strives to produce valid ones, e.g., BG emulates Member A thawing friendship with Member B only if they are friends. This chapter describes one physical design that we have implemented and released to the public domain (see http://bgbenchmark.org for details). We provide its details in the next section. Subsequently, we describe how BG uses this design to generate its actions.

5.1 Social Graph

Prior to generating a workload for a data store, it must be populated with a social graph corresponding to the social actions emulated by the workload [18]. BG's DataGenerator component is executed to create and insert the data for the social graph into the data store, see Section 4.2. A social graph is identified by its number of Accounts that are specialized into either individual members or pages. It consist of a fixed number of members (M), pages (P), number of friends per member (ϕ), number of followers per page (ι), number of pages followed by each member (ϱ) and number of resources owned by each member or page (ρ), see Table 3.1. While the number of members for a social graph must be non-zero, its number of pages may be set as 0. With $P = 0$, BG will not insert pages in the data store and will not create following relationships between members and pages. There are multiple ways of creating the friendship and following relationships in a social graph. Here, we describe our design and implementation. To simplify discussion and without loss of generality, we use the term BG to refer to this physical data design.

BG creates a social graph with the same number of friends per member. It also assumes the same number of followers for each page and the same number of pages followed by each member. In addition, BG creates an equal number of resources owned by each member and page. Each resource is posted on a randomly selected member's wall which may be the resource owner's own wall, see Section 3.1. BG creates following relationship between members and pages only if the following conditions are satisfied (see Table 3.1):

- $\varrho \times \iota \leq M$
- $(P \times \iota)/\varrho \leq M$
- $P \leq M$ (This condition is required for BG to create deterministic following relationships.)

- $\varrho \leq P$
- $\iota \leq M$

Using these assumptions, BG generates the social graph using two deterministic functions that result in deterministic friendship and following relationships between various accounts. For every Member i with $\varrho > 0$, BG generates the following relationships with pages $(i\%P), ((i+1)\%P), ..., ((i+\varrho)\%P)$. With $\varrho = 0$ it does not create any following relationships between members and pages. The maximum value that can be assigned to ϱ is P. For the same Member i, BG also generates friendship relationships with members $((i-\frac{\phi}{2}+M)\%M), ..., i-1, i+1, ..., ((i+\frac{\phi}{2})\%M)$ where the maximum value of ϕ is $M-1$. The limitation of generating friendships in this manner is that only an even number of friends can be inserted for each member, i.e., $\phi\%2 = 0$. This is trivial to resolve by assigning friendships in one direction where Member i has the following members as friends: $(i+1)\%M, ..., ((i+\phi)\%M)$

BG can also use multiple threads, T, to load the data for a social graph. In this case the social graph is divided into T dis-joint social graphs, each with an equal number of members and pages. This division occurs if the following four conditions hold true:

- $\frac{M}{T} > \phi$
- $\frac{P}{T} > \varrho$
- $\frac{M}{T} > \iota$
- $\frac{M}{T} \times \varrho = \frac{P}{T} \times \iota$

Otherwise, BG does not create the social graph. BG uses these deterministic functions during the benchmarking phase to identify the existing relationships in the social graph in order to issue valid actions, see Section 5.2. In this case, in addition to the social graph parameters, the benchmark will need the number of threads used for the loading phase, *numloadthreads* (T), as one of its input parameters.

Once the sub social graph for each thread is decided, each thread creates relationships between the members and pages of its sub social graph using the aforementioned functions. Each thread also inserts ρ resources for each member and page. If the T assigned by the experimentalist for the load phase violates one of the conditions described above then the BG framework invokes a deterministic function and selects the greatest thread count less than T that satisfies these conditions and uses that to construct the sub social graphs and load the data. This same function is used in the benchmarking phase to manipulate the numloadthreads parameter given as an input parameter to BG by the experimentalist and set it to the actual thread count value that was used to construct the sub social graphs and load the data store.

With N BGClients, the D-Zipfian distribution (see Section 4.3) divides the social graph into N sub social graphs. When $P \neq 0$, the number of followers per page (ι) and the number of pages followed by each member (ϱ) are utilized to manipulate the sub-social graphs generated by D-Zipfian for each BGClient. In this case once D-Zipfian creates the sub social graph, in a post processing step the members for the sub social graphs are shuffled such that:

1. Total probability of members assigned to each BGClient remains almost equal.
2. The total number of members assigned to each BGClient is equal.

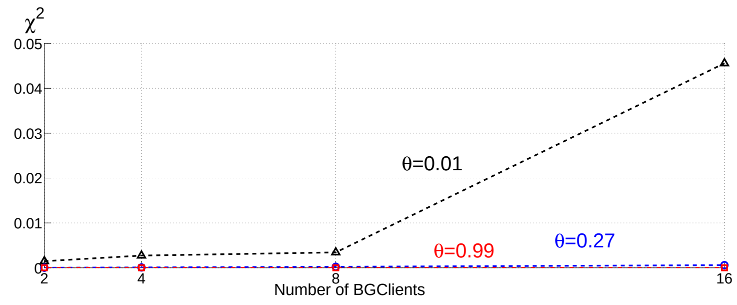


Figure 5.1: χ^2 analysis of the new D-Zipfian distribution used for feed following actions for $M = 1,000,000$ as a function of number of BGClients with three different values of θ for D-Zipfian.

The second condition is required in order to load an equal number of pages per member (ϱ) and can be achieved by shuffling the members with the least probability across the various sub social graphs for BGClients. BG also ensures that the the number of pages assigned to each BGClient is equal. For example a social graph with $M = 100,000$, $P = 100$, $\phi = 100$, $\rho = 10$, $\varrho = 10$ and $\iota = 10,000$ is divided to four social graphs with 4 BGClients where each social graph has the following characteristics: $M = 25,000$, $P = 25$, $\phi = 100$, $\rho = 10$, $\varrho = 10$, $\iota = 10,000$. This allows us to compare the evaluation results gathered using different number of BGClients for the same social graph characteristics. For evaluation of extreme cases such as scenarios for which all the members in a social network are following a single page, breaking the social graph into multiple sub-graphs may not be possible.

The changes made to the D-Zipfian algorithm impact the request distribution. And the impact increases as the number of BGClients increase. Figure 5.1 shows the amount of error introduced using the χ^2 analysis described in Section 4.3.1. As shows in this figure, the amount of error introduced is negligible and decreases as the skewness in the distribution decreases (D-Zipfian exponent, θ).

Next, each BGClient loads its own sub social graph into the data store and generates the relationships between members and pages within its own sub graph. In addition, each BGClient may use multiple threads during loading the data into the data store. This can only happen if the previously mentioned conditions hold true for the BGClient's sub social graph and the number of load threads the BGClient is using during its loading. Once the social graphs are loaded into the data store, BG's ActionGenerator uses the distribution of actions provided as an input to emulate social actions, see Section 4.2. This generator uses the D-Zipfian distribution to decide the socialite performing each action.

5.2 Stateful Request Generation

BG generates meaningful requests by maintaining in-memory data structures that maintain the state of the database. As an example, consider the Accept Friend Request (AFR) action. To ensure this command is meaningful, BG maintains data structures that track each member and their pending friend invitations. BG selects a Member A with a list of pending friend invitations. Next, it selects a member from this list randomly for use with the AFR action. If there are no members with a list of

Action	Lock Member A	Lock Member B
A Thaws Friendship with B	Yes	Yes
A Accepts B's Friend Request	Yes	No
A Invites B to be Friends	Yes	Yes
A Rejects B's Friend Request	Yes	No

Table 5.1: Four of BG's write actions and how they lock members.

pending friend invitations then BG increments a no operation count. At the end of an experiment, BG reports this number to the experimentalist. An experiment with a high no operation count does not reflect the workload intended by the experimentalist and its metrics should not be associated with the workload.

BG is thread safe and employs synchronization primitives such as semaphores. In addition, it locks the members referenced by the write actions to ensure multiple threads do not issue simultaneous actions that cause one to become meaningless. As an example, consider the Thaw Friendship action. When using one thread to emulate Socialite A who thaws friendship with Member B, BG locks both Members A and B prior to issuing the command. This prevents another thread from emulating Socialite A performing the same action simultaneously. It also prevents a concurrent thread from emulating Socialite B who thaws friendship with Member A. Table 5.1 shows the four write actions that impact friendship and whether they lock either one or both members.

One may turn the locking feature of BG off. This may cause concurrent threads to issue concurrent actions that may cause one or more to become a no operation. This may also cause a data store to detect integrity constraint violations and throw exceptions.

5.3 Uniqueness of Simultaneous Socialites

BG supports a closed emulation model consisting of a fixed number of simultaneous threads issuing actions against a data store. Each thread emulates a socialite and BG ensures concurrent socialites are unique at any point in time. In other words, the members selected to emulate simultaneous actions against the data store are unique. This is required in order to emulate reality in which a social network's member accesses or manipulates data using one active login session.

With BG, the uniqueness of the socialites is implemented using a local data structure that maintains if an account (member or page) is busy or not. Once a BG thread decides to emulate an action and decides the member/page to use, it looks up the status of the member/page in this data structure. If the member/page's status is busy, which means it has been picked by another thread to emulate a simultaneous action, BG finds the next non-busy member/page to use for its action by linearly searching through the data structure. If it fails to find a non-busy member, it does not issue an action against the data store resulting in a no operation, see Section 5.2. If it finds a non-busy member/page, it changes its status to busy, issues the action against the data store, waits for the action to complete and then changes the status of the member/page to non-busy. This is performed in thread-safe manner using latches. Once the status of a member is set to non-busy, it can be selected by other threads to emulate actions against the data store.

With multiple BGClients, each BGClient issues request by selecting members from its own sub social graph, so checking a BGClient's local data structure is sufficient and no coordination between the BGClients is required to ensure the uniqueness of the simultaneous emulated members.

Chapter 6

Unpredictable Data

Today's data stores strive to be scalable, highly available, and fast. To realize these objectives, a system may employ techniques introducing delayed propagation of updates and undesirable race conditions, that result in stale, inconsistent or erroneous output, collectively termed as unpredictable data. Example techniques that produce unpredictable data include use of a weak consistency technique such as eventual [112, 103] and use of a cache [47] in a manner that results in dirty reads [56] and inconsistent cache states [46, 86].

With an increase in this kind of design approaches, it is becoming increasingly important to understand the implications of to what extent provision of unpredictable data improves the performance. This motivates the need for a framework quantifying the amount of unpredictable data [94]. There are many metrics that can be used in this framework to quantify data staleness for an architecture. Some of these are as follows:

- Probability of observing an accurate value a fixed amount of time, say t seconds, after an update occurs [114], termed as freshness confidence.
- Amount of time required for an updated value to be visible by all subsequent reads. This is termed inconsistency window [114].
- Percentage of reads that observe a value other than what is expected, quantified as the percentage of unpredictable data [12].
- Probability of a read observing a value fresher than the previous read for a specific data item, termed monotonic read consistency [114].
- Age of a value read from the updated data item. This might be quantified in terms of versions or time [10, 11].
- How different is the value of a data item from its actual value? For example, with a member who has 10 friends, a solution may return 9 friends for the member whereas a different solution may return 20 friends. An application may prefer the first [10, 11].

Each of these metrics may provide a new insight into a system's behavior and its design decisions. A variant of the first two metrics are quantified by benchmarking frameworks such as [94, 88]. BG [12] is the only benchmark to evaluate the correct execution of operations in support of interactive social networking operations. Today's BG supports two of the aforementioned metrics: percentage of unpredictable data and freshness confidence.

Many of today's applications may tolerate some amount of unpredictable data observed by data stores [99]. For example, with social networks, once a member posts a status message, it can be visible to others after 2 minutes and the members may find this acceptable. On the other hand, some applications such as banking applications may not tolerate unpredictable data. Thus the amount of unpredictable data (stale, inconsistent or erroneous data) produced by data stores has a significant impact on its possible use by an application. A novel feature of BG is its ability to quantify the amount of unpredictable data produced by a data store for an application. It does so by computing a list of acceptable values for each read using the timestamp for both read and write actions, and comparing the acceptable values with the observed values. One may use this feature to characterize the trade-offs associated with different architectures. It can also be used to specify an SLA when rating different data store solutions, see Chapter 7.

BG also uses the timestamp for read and write actions to quantify the freshness confidence (%) which is the probability of observing an accurate value a fixed amount of time after an update occurs. The amount of unpredictable data and freshness confidence are both workload and data store dependent.

In this chapter, we describe BG's validation mechanism which is used to quantify the amount of unpredictable data. This is presented in two parts. First, we describe how validation is performed for all actions excluding the feed following actions, namely, View News Feed action and Share Resource Action. Subsequently, we include feed following actions. We conclude this chapter by describing the scenarios for which the current validation algorithm fails to compute the amount of unpredictable data accurately.

6.1 Validation

This section describe *validation* as the process of quantifying the amount of unpredictable data produced by a data store and presents BG's modular and configurable validation component.

Conceptually, BG is aware of the initial state of data items in the database (by creating them using deterministic functions supporting an analytical model) and the change of value applied by each write action[1]. There is a finite number of ways for a read of a data item to overlap with concurrent actions that write it. BG enumerates these to compute a range of acceptable values that should be observed by the read operation. If a data store produces a different value then it has produced unpredictable data. This process is named *validation* and BG's physical implementation is as follows.

Validation might be an online or an offline technique. While an online technique reports the correctness of a value produced for a read operation immediately after it completes, an offline technique would quantify the amount of unpredictable data after a benchmark completes generating its specified workload. The latter does not quantify the amount of unpredictable data while the experiment is running. A validation component may implement either extremes, or a hybrid design that reports the amount of unpredictable data after some delay based on its allocated resources. BG decouples generation of requests to quantify the performance of a data store from the validation phase, performing validation in an off-line manner. By doing so, BG prevents the validation phase from exhausting the resources of a BGClient and reduces the number of nodes required to generate requests to evaluate a data store.

[1]For example consider the Accept Friend Request (AFR) action of BG. When the AFR operation is issued by Member A to confirm friendship with Member B, then the write operation increments the number of friends of each member by one. Or when a write operation is issued by Member A to unfriend B then Bs news must not appear in As feed retrieved by the View News Feed (VNF) action of BG.

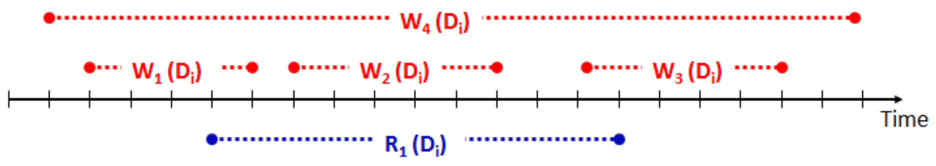


Figure 6.1: There are four ways for a write of D_i to overlap a read of D_i: It may either start before the read and end in the middle of the read operation (W_1), start in middle of the read and end before the read ends (W_2), start in the middle of the read and end after the read operation (W_3), or start before the read and end after the read (W_4).

During the benchmarking phase, each thread of a BGClient invokes an action. All actions except for the Share Resource action generate one log record. There are two types of log records, a read and a write log record corresponding to either a read or a write action. These log records are written to separate files. One file for the read log records and a second file for the write log records. During the validation phase, BG processes these log records to quantify the amount of unpredictable data produced by a data store [12].

A log record consists of a unique identifier, the action that produced it, the data item referenced by the action, its socialite session id, and start and end time stamp of the action. The read log record contains its observed value from the data store. The write log record contains either the new value (named *Absolute Write Log*, AWL, records) or change (named *Delta Write Log*, DWL, records) to existing value of its referenced data item. The log records generated for the Share Resource action contain information about the referenced data item (e.g. the resource being shared by the member) and the members the resource is being shared with. The start and end time stamps of each log record identify when an action that either reads or writes a data item begins and finishes. They enable BG to compute the 4 possible ways that a write operation may overlap a read operation, see Figure 6.1. During validation phase, for each read log record that references data item D_i, BG enumerates all completed and overlapping write log records that reference D_i to compute a range of possible values for this data item. If the read log record contains a value outside of this range then its corresponding action has observed unpredictable data.

To elaborate, BG uses the set of q DWL records to compute all serializable schedules that a data store may generate. The theoretical upper bound on the number of schedules is $q!$. However, BG computes fewer schedules because it does not consider the non-overlapping DWL records. BG identifies these by detecting when the end time stamp of one is prior to the start of the second. This produces an accurate range of possible values for the read operation. This is best illustrated with an example. Consider the four log records of Table 6.2 where 3 DWL records overlap 1 read log record. Theoretically, there is a maximum of six (3!) possible ways for the updates to overlap one another. However, the actual number of possibilities is two, {{DWL1,DWL2,DWL3}, {DWL2,DWL1,DWL3}}, because DWL3 has a start time stamp after both DWL1 and DWL2. Thus, assuming the value of D_1 is zero at time zero, acceptable values for the read are {-1, 0, 1, 2}, flagging the observed value 3 as unpredictable. If one had assumed 3! possible schedules incorrectly then value 3 would have appeared in the acceptable set. This would have confirmed Read1 as valid incorrectly.

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

Figure 6.2: Example log records.

Log records produced by one BGClient are independent of those produced by the remaining $N - 1$ BGClients because BGCoord partitions members and resources among the BGClients logically. Thus, there are no conflicts across BGClients and each BGClient may perform validation independently to compute number of actions (sessions) that observe unpredictable data. During rating, BGCoord collects these numbers from all BGClients to compute the overall percentage of actions (sessions) that observed unpredictable data, see Chapter 7.

Depending on the duration of the experiment, a BGClient may produce a large number of log records. These records are scattered across multiple files.

6.1.1 Validation of Actions Excluding Feed Following Actions

We now discuss the implementation details of the validation technique used for validating BG's non feed following actions. Currently, there are two centralized implementations of this validation phase using interval-trees [30] as in-memory data structures and a persistent store using a relational database. The latter is more appropriate when the total size of the write log records exceeds the available memory of a BGClient. The former is fast when there is a sufficient amount of memory to stage the write log records in interval trees. This enables the validation phase to read the log files once to process both read and write log records in one pass. The interval tree maintains the start and end time stamp of the write log records for each data item, D_i. The validation phase assumes three different kind of interval trees for: (1) each member and write actions that impact her friendships, (2) each member and write actions that impact her pending friend invitations, and (3) each resource that is annotated with a write action. It constructs interval trees for a member/resource on demand as it reads the write log records in memory, creating them when one does not exist for a member id (resource id) for the corresponding action. Once the write log records are staged in memory, the validation phase retrieves the read log records. It employs the member id (resource id) and the action to identify the interval tree with the relevant write log records. Next, it uses the start and end time stamp of each read log record to enumerate the number of ways it overlaps with the different write actions by querying the interval trees.

Both in-memory and persistent implementations of validation are optimized for workloads dominated with actions that read data items [3]. These optimizations are as follows. First, if there are no update log records then there is no need for a validation phase; the validation phase terminates by deleting the read log file(s) and reporting 0% unpredictable reads. Second, write log records are processed first to construct a main memory data structure (independent of interval-trees or the RDBMS) that maintains each updated data item and its value prior to the first write log record and after the last write log record on that item, start time stamp of the first write log record, and the end time stamp of the last write log record on that item. This enables BG to quickly process (prune) read log records that either reference data items that were never updated (do not exist in the main memory data structure), or were issued before the first or after the last writer (there is only one pos-

Workload	Number of read logs	Number of pruned read logs	Number of write logs	In-memory Structure Creation duration (msec)	Validation duration (msec)
0.1% Write Actions	2,414,569	2,194,486	3,343	47	10,047
1% Write Actions	2,327,587	1,446,998	39,922	281	10,281
10% Write Actions	1,592,701	481,317	343,224	1172	22,735

Table 6.1: Validation duration for three workloads of Table 4.1

sible value for these and available in the main memory data structure). Third, multiple threads may process the read log records by accessing the aforementioned data structure with no semaphores as they are simply looking up data. This makes the validation phase suitable for multi-core CPUs as it employs multiple threads to process the read log files simultaneously.

Table 6.1 shows the duration of the validation phase for the three workloads described in Table 4.1 using the interval tree approach. These workloads emulate 200 threads issuing actions against the data store for 900 seconds. As shown in this table, as we increase the percentage of writes, the number of write actions and the amount of time it takes to construct the in-memory structures increases. The increase in the duration of processing write files as well as the reduction in the number of read logs that can be pruned[2] increase the overall rating duration.

A key limitation of in-memory implementation using interval trees is that it may exhaust the available physical memory, causing the operating system to exhibit a thrashing behavior that results in an unacceptably long validation process. This is specially true with high throughput multi-node data stores and cache augmented data stores such as KOSAR that process requests in the order of millions of actions per second. In such cases, one should employ the alternative using an RDBMS.

We have also examined a preliminary implementation of the validation phase using MapReduce [37] that requires the log files to be scattered across a distributed file system. Such a deployment is warranted once BG is deployed at a large scale to evaluate many different data stores.

6.1.2 Validation for Feed Following Actions

The View News Feed (VNF) action of BG retrieves the top k most recent resources shared with a member by those she follows. A socialite, Member A, shares resources using the Share Resource (SR) action. Member A may share a resource either publicly with all the members following A or with a select list of members following A. Conceptually, BG is aware of the initial state of feed for every member and changes the value of each member's news feed upon any related SR action by adding the shared resource to it. A related SR action for Member i's feed is one that is issued by Member j followed by Member i. Note that this discussion applies to pages as well. For each VNF action, BG enumerates all the relevant SR actions and computes a list of acceptable feed values (list of acceptable resources). If the data store returns a value other than what is expected then unpredictable data is observed.

During the benchmarking phase BG generates two write log records for each SR action and one read log record for each VNF action. The first write log record contains the start and end time stamp of the action, the memberid of the member issuing the action and the resourceid of the resource being shared. The second log record also contains the start and end time stamp for the action, the

[2]With an increased number of updates, the probability of a reading a data item before an update on the data item reduces.

resourceid of the resource being shared and the list of followers the resource is shared with. If the list of followers is set to -1 the resource is shared publicly with all members following this member. The read log record contains start and end time stamp for the VNF action, the memberid for the member retrieving her news feed and the list of resourceids that are displayed on it. BG is also aware of the initial friendship/following relationships for members and pages. In addition, as discussed in section 6.1 it logs the changes made to the friends of a member[3]. During validation BG maintains two additional interval trees. One for the shared resources and the members they are being shared with and another for the members and the resources they are sharing with other members. Upon encountering a read log record for a VNF action, BG computes the list of members followed by the member performing the VNF action at the time of the read using the start and end time stamp logged for it by querying the interval trees described in Section 6.1. Next, it finds all the resources shared by these members until that point of time. It only retrieves the list of resources that are either shared publicly with all followers or are specifically shared with this member. It then compares this list with the list of resources retrieved from the data store for the member's feed. As some write actions may overlap with the VNF action (write actions modifying friendship relationships or performing SR actions), BG computes a super-set of acceptable lists containing resourceids[4]. If the News Feed retrieved form the data store is not a subset of any of the lists in the computed set, then the data store has produced unpredictable data. This is best illustrated with an example. Consider the five log records in Table 6.1.2. Assume Member B is only following Member A and Member A owns Resources 1 and 2.

Operation id	Type	Data item	Start	End	Value
AWL1-1	Write	Member A	0	3	1
AWL1-2	Write	Resource1	0	3	-1
AWL2-1	Write	Member A	0	5	2
AWL2-2	Write	Resource2	0	5	{B,C}
Read1	Read	Member B	4	6	{1,2}

AWL1-1 and AWL1-2 belong to the same SR action. AWL1-1 indicates that Member A shares Resource1 and AWL1-2 indicates that Resource1 is shared publicly (identified with value = -1). Similarly AWL2-1 and AWL2-2 are created by the same SR action. AWL2-1 indicates that Member A shares Resource2 and AWL2-2 indicates that Resource2 is shared with Member B and Member C. Read1 indicates that Member B retrieves her news feed and observes Resource1 and Resource2 in it. BG's validation for Read1, finds all the members followed by Member B (in this example only Member A). Next, it retrieves all the SR actions initiated by the members Member B is following and either completed before the read or overlapped with it. In this example AWL1-1 and AWL1-2 completed before the read and AWL2-1 and AWL2-2 overlap with it. From these, it retrieves those that either share a resource publicly (AWL1) or share it specifically with Member B (AWL2) and

[3]In BG, once a following relationship between a member and a page is created, it can not be modified. This is because in BG pages can only participate in feed following actions and all other BG actions involving modification of friendship/following relationships, are only applicable to members and the relationships between them.

[4]As the list of friends for a member depends on the completed and overlapping write actions, BG computes a set of friend lists for the member, S_f. The same holds true for the resources shared by members and BG computes a set of acceptable shared resources by each member in the friends list, S_r. The final set of acceptable resources displayed on a member's news feed is a Cartesian product of S_f and S_r. Computing this product is expensive so BG computes the union of S_r for each list in S_f as the list of acceptable resources to be displayed on a member's feed. By doing so BG computes an approximation of the member's news feed.

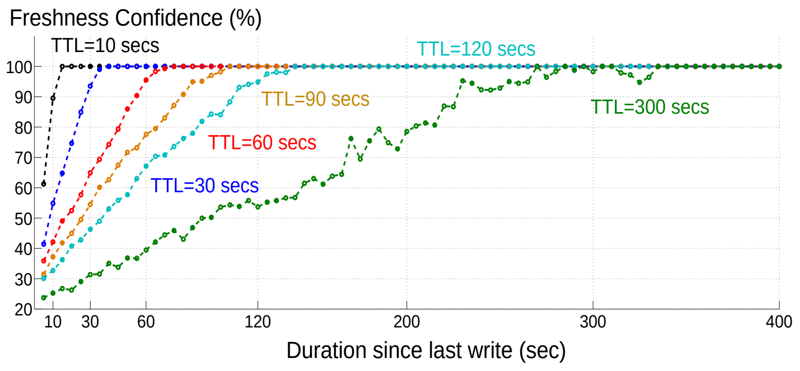


Figure 6.3: Freshness confidence for CASQL with TTL invalidation mechanisms for a workload consisting of 10% write actions. $b = 120$ buckets, $\Delta = 600$ seconds.

uses them to compute a set of acceptable resource lists for Member B's feed. In this example these are the two acceptable sets: $\{1\}$ and $\{$1,2$\}$. If the VNF Read1 observed $\{$1,3$\}$ then it had observed unpredictable data. Hence, in this example it did not observe unpredictable data.

6.1.3 Freshness Confidence

The validation phase also computes the probability (p_i) of a read observing the freshest value at most t units of time after the update on the same data item was completed, freshness confidence. The log records are processed only once to compute both the amount of unpredictable data and freshness confidence. The experiment duration, Δ, is divided into b time buckets each with a duration of $\frac{\Delta}{b}$ units of time. Each bucket b_i maintains the total number of read operations (R_{b_i}) between $\frac{\Delta}{b} \times i$ to $\frac{\Delta}{b} \times (i + 1)$ units of time (where $0 \leq i \leq b$) after the last write on the relevant data item, i.e., the last write is retrieved by querying the relevant interval tree. They also maintain the number of valid reads that did not observe unpredictable data for the same period (V_{b_i}). The probability of observing the freshest value for a read at most t units after the write is computed by finding i for which $t = \frac{\Delta}{b} \times (i + 1)$ and computing the following (see Figure 6.3):

$$p_i = \frac{\sum_{j=1}^{j=i} V_{b_j}}{\sum_{j=1}^{j=i} R_{b_j}} \tag{6.1}$$

This metrics is important for applications such as news feed which are tolerant for missing content but are timely. For example, let us assume Member A follows 100 members and each member produces at least 1 event every three minutes. Member A may tolerate not seeing the last content produced by each of the producers in the last three minutes in her feed. At the same time, it will be undesirable if she does not see events produced more than 3 minutes ago, say 1 hour ago. This means even though Member A encounters an unpredictable read, the application can tolerate the missing events. Thus computing the amount of unpredictable reads is not a good metric when evaluating alternative solutions of news feed. A better metric is to compute the probability of observing the freshest value at most t units of time after the write on the same data item is completed.

Figure 6.3 shows the freshness confidence for CASQL systems with six different time to live (TTL) values: 10, 30, 60, 90, 120 and 300 seconds. Use of TTL is an alternative to using an

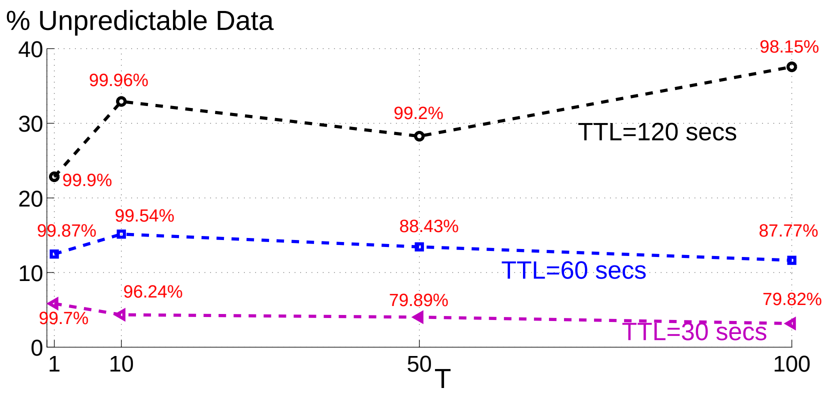


Figure 6.4: Percentage of unpredictable data (τ) as a function of the number of threads with memcached (CASQL). Mixed workload with 10% write actions, see Table 4.1. M=10,000, 12 KB image size, ϕ=ρ=100, β=100 msec, θ=0.27, α is in red.

invalidation technique [47, 86]. It is a simple technique that invalidates a cached entry once its life time expires. While it reduces software complexity, it produces unpredictable data. Figure 6.3 shows the freshness confidence for a fixed t decreases with higher TTL values which means it takes a longer time for the value of updates to be available for all reads occurring after the update.

6.1.4 Evaluation

We used BG's ability to compute the amount of unpredictable data to analyze the trade-offs for a CASQL system employing a time to live (TTL).

Figure 6.4 shows the behavior of the system with three different TTL values, 30, 60, and 120 seconds, as a function of the number of BG threads, T. We assume 10% of actions are writes (see Table 4.1). As expected, obtained results show a higher TTL value results in a higher percentage of unpredictable data. A higher TTL value also enhances performance of CASQL by increasing the number of references that observe a cache hit. This is shown with a higher percentage of request that observe a response time faster than 100 msec (α) with T=100: α increases from 79.8% with a 30 second TTL to 98.15% with a 2 minute TTL. In essence, a higher TTL value enhances performance of CASQL at the expense of producing a higher amount of stale data.

6.2 Discussion

The current implementation of the validation technique is an implementation of a subset of design choices. This implementation targets a common use case scenario and will not work in all cases:

1. With high throughput data stores that process millions of operations per second an experimentalist may not be willing to wait for hours to obtain an accurate measure of the amount of unpredictable data. They may be interested in a sampling approach that provides an inaccurate and quick measure of the amount of unpredictable data.

2. The consistency requirements of one application might be different than another. For example with a social networking system, apart from the data value, the order of display may also

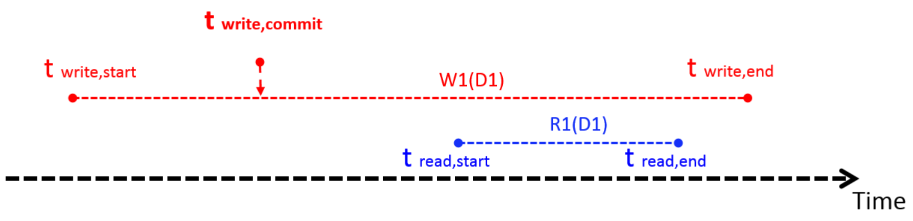


Figure 6.5: An example showing the limitation of BG's validation mechanisms with an overlapping read and write log record for data item D_i.

matter. We may have a solution that retrieves the correct comments for a photo but displays them in the wrong order resulting in miss-interpretations and miss-communications. The current implementation of BG's validation, does not emphasize on the order of observed values but can be extended to maintain the order of data items to be displayed (i.e., based on recency) and use that in its evaluation of unpredictable data. The validation mechanism can also be further extended to evaluate the solutions based on the accuracy of results returned. For example, if the desired order for the commentids for a photo is 1,2,3,4,5, a solution that returns 2,1,3,4,5 is more accurate than one that returns 1,5,4,3,2.

3. BG's validation mechanism consumes the start and end timestamps for invoked write actions to compute a list of acceptable values for every read. This may lead to scenarios for which the amount of unpredictable data reported by BG may not be accurate. With Figure 6.5, the write action modifying the value of data item D_i starts at time $t_{write,start}$ and ends at $t_{write,end}$, but the actual update is committed to the data store at time $t_{write,commit}$, between $t_{write,start}$ and $t_{write,end}$[5]. The read action for the value of data item D_i, starts at time $t_{read,start}$ and ends at time $t_{read,end} > t_{read,start}$ such that $t_{write,commit} < t_{start,read} < t_{write,end}$ and $t_{read,end} < t_{write,end}$. Ideally, the read should observe the updated value as it started after the update had been committed to the data store. BG's current validation mechanism will not report any stale data, if the read observes the old value rather than the updated value. For such a scenario, one may need to extend BG's validation mechanism with more complex models.

Below, we describe an extension to today's validation component and discuss a host of design choices for it. We organize these in a taxonomy consisting of three interdependent steps: Runtime configuration, Log processing, and Validating. Below, we describe each step and its possible design choices in turn.

As suggested by its name, the Runtime configuration component configures the modules employed by the validation phase. This component is used once. Its output specifies the components used by the validator at runtime. This output is based on the following input parameters:

- How much memory should be allocated to the validation phase? The provided value controls how much of the write log records are staged in memory prior to processing read log records, preventing the validation phase from exhausting the available memory.

[5]This can happen with CASQL solutions with invalidation consistency mechanisms, for which once the update is committed to the underlying data store, the corresponding cached key-value pair needs to be deleted as a part of the write action.

- How long should the validation process run for? The specified duration dictates the number of processed read and write log records. A duration shorter than that required to process all log records may produce inaccurate measures of the amount of unpredictable reads.
- What is the degree of parallelism used by the validation process? It is trivial to parallelize the validation process by partitioning the log records using the identity of their referenced data items. With a RAID disk subsystems and multi-core CPUs, it is sensible for the component that streams the log records to partition them (this is the map function of a MapReduce job). This facilitates independent processing of each partition using a different thread/core to quantify the amount of unpredictable data.
- What sampling technique should be used by the validation phase? The validation process may use a host of sampling techniques that should be identified at configuration time. These are listed under the Log Processing step and detailed below.

The Log Processing phase reads the log records from the log file and processes them to create the data structures needed for the validation process. It may use a sampling technique to decide which log records are processed in order to reduce the duration of the validation phase. A disadvantage of sampling is that the amount of reported unpredictable data may not be accurate. A sampling technique may be based on:

- Data item/operation type: The validation process may focus on a subset of data items by processing their read and write log records. It may process those data items with either the highest frequency of reference, largest number of write log records, most balanced mix of read and write operations, and others. Alternatively, it may process the log records produced by a subset of operations.
- Time: A second possibility may require the validation process to sample a fixed interval of time relative to the start of the benchmarking phase, e.g., process one minute of log files ten minutes into the benchmarking phase or process the log records generated during the peak system load, e.g., shoppers during Christmas time.
- Number of processed log records: A third possibility requires the validation process to sample a fixed number of log records, e.g., 10K log records, 20% of log records relative to the start of the benchmarking phase, 2 out of every 10 log records, and others.
- Random/Custom: A final possibility is for one to come up with a custom sampling approach where log records satisfying a specific condition are processed by the validation process, e.g., log records generated by members is a specific geographical locations.

The Validating phase can execute at the same time as the Log Processing phase and is responsible for computing a list of acceptable values for each read and comparing the observed value with the computed list. One can implement custom validation approaches which deal with different data types such as primitive, array type or user-defined and apply different kinds of manipulations to identify the list of acceptable values for a read.

Chapter 7

Rating a Data Store

To *rate* a data store is to compute a value that describes the performance of the data store for a workload. BG computes two possible ratings for a data store, named SoAR and Socialites. The Social Action Rating, SoAR, of a data store is its highest number of simultaneous social actions completed while satisfying the pre-specified SLA. The Socialites rating quantifies the maximum number of simultaneous members that may access the data store while satisfying the pre-specified SLA. Figure 7.1 illustrates these two ratings using MongoDB as an example. It shows the SoAR of MongoDB is approximately 35,000 while its Socialites rating is 1025.

Figure 7.2 shows BG's software components that rate a data store. These include multiple BG Clients (**BGClient**), one BG Coordinator (**BGCoord**), and one delta analyzer. 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 BGClient 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). Each **BGClient** implements the 13 actions of Table 1.1 for the target data store. In addition, they include the design elements outlined in Chapter 4 to realize a scalable

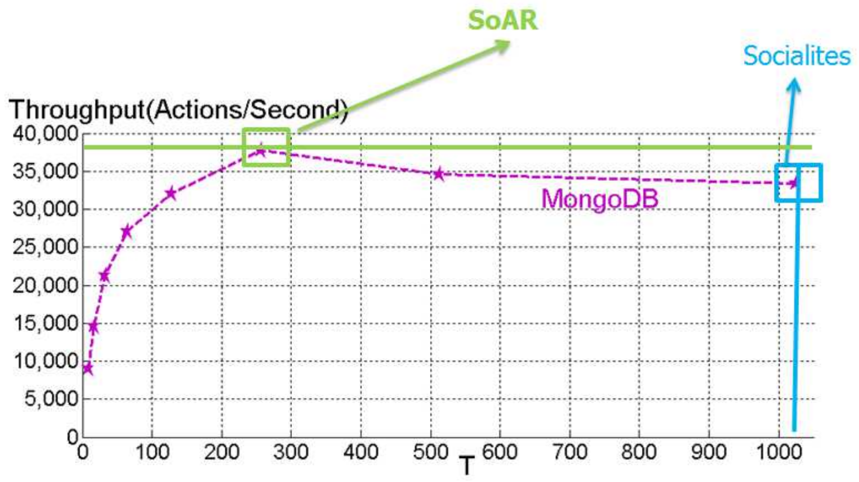


Figure 7.1: BG's SoAR and Socialites rating.

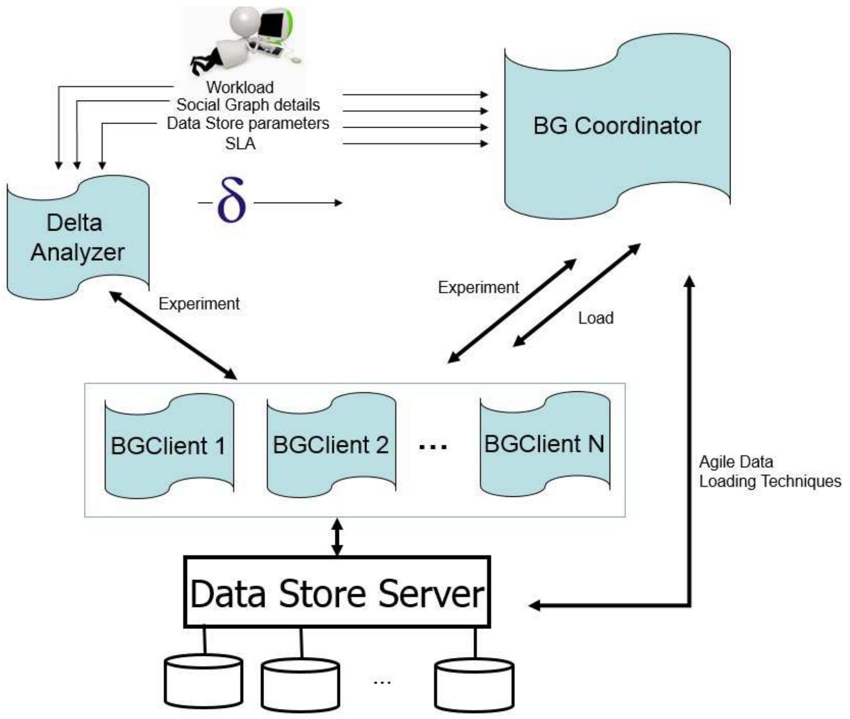


Figure 7.2: Software components of BG's rating mechanism.

request generation framework. The **BGCoord** employs a heuristic search technique that conducts experiments, each with a fixed number of threads T, using the target data store. 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[1] experiments. When the value is in the order of one hundred million, BGCoord may conduct between 50 to 60 unique experiments. It may load the database in between experiments and use agile data loading technique to expedite the rating process. This chapter describes these two components in Section 7.1 and 7.2, respectively.

The duration of each experiment (δ) conducted by BGCoord is dictated by the **Delta Analyzer**. The maximum value of δ is the duration specified by the SLA, Δ. A smaller value of δ is desirable because it expedites the rating process. Ideally, δ should be the smallest possible value that reflects the behavior of a data store as if the experiment was running for Δ. We discuss the computation of δ in Section 7.3.

This chapter concludes with an analysis of the agile delta loading techniques and the delta analyzer. It quantifies the observed speedup when compared with a rating mechanism that does not utilize these techniques. Obtained results show a factor of 4 to more than 10 speedup with the proposed techniques.

[1]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.

7.1 Heuristic Search

BGCoord conducts several experiments, each with a fixed number of threads T. It employs a heuristic algorithm to vary the value of T to impose a different amount of load on the data store. These threads are spread across the N BGClients. At the end of each experiment, each BGClient reports its observed number of unpredictable reads, and the percentage of requests that observed a response time equal to or faster than that required by the SLA, β. This experiment is *successful* as long as all of the following hold true: 1) observed average percentage of unpredictable reads across all N BGClients is less than or equal to the SLA specified tolerable amount of unpredictable reads, and 2) the percentage of requests that observe a response time less than or equal to the SLA specified response time (β) is greater than or equal to the SLA specified percentage (α). Otherwise, this experiment has *failed* to meet the specified SLA.

One approach to compute SoAR and Socialites rating of a data store is to conduct experiments starting with T=1 and increment T by one every time an experiment succeeds. It would maintain the highest observed throughput and the highest T value. And, it terminates once an experiment fails (see Assumption 1 below) to satisfy the SLA, reporting the highest observed throughput as SoAR and the largest T as Socialites rating of the data store. A limitation of this strategy is that it requires a substantial amount of time. For example, in Figure 7.1, MongoDB supports a Socialites rating of 1025, T=1025. An exhaustive search starting with 1 thread and assuming Δ=10 minutes would require more than 7 days.

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

1. Throughput of a data store is either a square root function or a concave inverse parabola of the number of threads, see Figure 7.9.a.

2. Average response time of a workload either remains constant or increases as a function of the number of threads, see Figure 7.9.b.

3. Percentage of stale data produced by a data store either remains constant or increases as a function of the number of threads, see Figure 7.9.c.

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

Figure 7.9.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 [20, 64], (b) queuing delays attributed to fully utilized server resources where $\bar{RT}$=$\bar{S}$+$\bar{Q}$ and $\bar{Q}$ is the average queuing delay, or both. In the absence of (a), the throughput of the data store is a square root function of T, see Figure 7.9.a. In scenario (b), $\bar{Q}$ is bounded with a fixed number of threads since BG emulates a closed simulation model where a thread may not issue another request until its pending request is serviced. Moreover, as $\bar{RT}$ increases, the percentage of requests observing an $\bar{RT}$ lower than or equal to β decrease, see Figure 7.9.d.

The heuristic search technique to compute Socialites rating of a data store starts with an experiment using one thread, T=1. If the experiment succeeds, it doubles the value of T. It repeats this process until an experiment fails, establishing an interval for the value of T. The minimum value of this interval is the previous value of T that succeeded and its maximum is the value of T that failed. The heuristic performs a binary search of T in this interval to compute the highest T value that enables an experiment to succeed. This is the Socialites rating of the data store. It is accurate as long as Assumption 1 is satisfied, see Figure 7.9.a.

The heuristic to compute SoAR is similar to Socialites with several key differences. First, BGCoord maintains the highest observed throughput with each T value, λ_T. It stops doubling T once an experiment produces a throughput lower than λ_T (or fails to satisfy the pre-specified SLA as is the case with the square root curve of Figure 7.9.a). This is the point denoted as $2T$ in Figure 7.9.e. Next, it searches the interval ($\frac{T}{2}$, $2T$). It may not simply focus on the interval (T,2T) because the peak throughput might be in the interval ($\frac{T}{2}$,T), see Figure 7.9.e. We now describe the SoAR rating approach in more detail.

Computing the SoAR of a system, the global maxima, is a well-known problem in mathematical optimization. One may compare it as a search space consisting of nodes where each node corresponds to the observed throughput with an imposed system load (T). The node with the highest throughput that satisfies the pre-specified SLA identifies the SoAR of the system. To identify this node, the BGCoord implements two techniques to navigate the search space. Both techniques assume the aforementioned assumptions and are detailed in this section. This section also discusses scenarios that violate our assumptions and suggests a possible approach to navigate the search space.

The first technique, named *Optimal,* is guaranteed to compute the SoAR for the system. The second technique, named *Approximate*, is a heuristic search technique that computes the SoAR of a system with $\pm 10\%$ margin of error. 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 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 Optimal and Approximate navigate the search space by changing the value of T, imposed load. Both techniques traverse the search space in two distinct phases: a hill climbing phase and a local search phase. The local search phase differentiates Optimal from Approximate. Approximate conducts fewer experiments during this phase and is faster. However, its SoAR rating incurs a margin of error and is not as accurate as Optimal. Below, we describe the hill climbing phase that is common to both techniques. Subsequently, we describe the local search of Optimal and Approximate in turn.

BGCoord implements the hill climbing phase by maintaining the thread count (T_{max}) that results in the maximum observed throughput (λ_T) 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 SoAR 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 BGCoord. (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 (λ_T), establishing an interval for the value of T that yields the SoAR of the system. Once this interval is identified, the hill climbing phase terminates, providing the local search space with the identified interval. Below, we describe how Optimal and Approximate navigate this interval in turn.

The local search phase inputs the limited interval identified by a starting and ending thread count. The starting thread count is $\frac{T}{r^2}$ and the ending thread count is the current T. The peak throughput, SoAR, may reside in either the interval $(\frac{T}{r^2}, \frac{T}{r})$ or $(\frac{T}{r}, T)$. Optimal identifies the peak throughput as follows. It focuses on $\frac{T}{r}$ and conducts experiments with $\frac{T}{r^2} \pm \eta$ threads to determine the slope of the curve in each direction. If both slopes are negative then the point T is the peak and the throughput is reported as the SoAR of the data store[2]. Otherwise, it focuses on the interval that contains the peak (the interval which has the increasing slope), selects the mid-point of this interval and continues to compute the slope on either of its sides to decide the interval the peak is in and this continues until the SoAR of the system is established.

Approximate navigates the interval identified by the hill climbing phase differently. It treats the start of the interval as the point with the highest observed throughput among all points that have been executed and its end as the point with the lowest thread count that failed. It then executes an experiment with the mid-point in this interval. If this experiment succeeds and observes a throughput higher than λ_T, then the heuristic changes the start of the interval to focus on to this point and continues the process. Otherwise, it changes the end point of the interval to be this point and continues the process until the interval shrinks to consist of no more points. The Approximate approach is not guaranteed to find the peak throughput (SoAR) for a system. Its margin of error depends on the behavior of the data store and the climbing factor r. With our synthetic experiment (see below), it produces a result that is within $\pm 10\%$ of the true SoAR for the system. Below, we describe an example to illustrate why Approximate incurs a margin of error.

Consider a scenario where the experiment succeeds with $\frac{T}{2}$ threads and increases the thread count to T. With T the experiments succeeds again and observes a throughput higher than the max throughput observed with $\frac{T}{2}$. Thus, the hill climbing phase increases the thread count to $2T$ (assuming a climbing factor of 2, r=2). With $2T$, the experiment produces a throughput lower than the maximum throughput observed before. This causes the hill climbing phase to terminate and establishes the interval (T, $2T$) for the local search. While the local search phase of Optimal considers an interval ranging from $\frac{T}{2}$ to $2T$, Approximate considers only the interval T to $2T$. With Figure 7.9.e, Approximate eventually selects T as the thread count for the peak throughput.

Both these techniques visit tens and hundreds of states although the SoAR for the system may be in orders of thousands and millions. We used a 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, SoAR, and is computed by solving the first derivative of the quadratic function: $T = \frac{b}{2}$. The Optimal solution and the Approximate must compute this value as the SoAR of a system modeled using the function. We select different values of b and c to model diverse systems whose SoAR varies from 100 to 100 million.

Every time the BGCoord executes an experiment with a given value of T, it maintains the observed throughput in a HashMap. When exploring points in an interval, it uses this HashMap to identify repeated experiments. It does not repeat them and simply looks up their observed throughput using the HashMap. This is significantly faster than executing an experiment. Figure 7.3 shows the number of visited states. When SoAR is 100 million, the Optimal technique conducts 54 exper-

[2]If the slope on both sides is positive, value of η is increased and two new points in the intervals are explored. This continues until the slope on one side is positive and the slope on the other side is negative, or until there are no more points in the two intervals to be explored. With the latter the Optimal solution fails to compute the SoAR for the system. Based on our assumptions the latter is not possible.

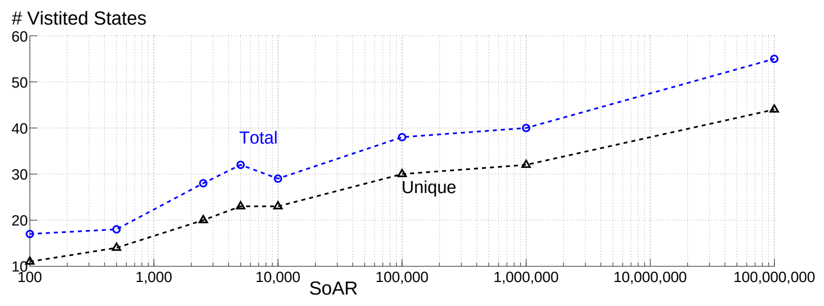


Figure 7.3: Number of experiments conducted to compute SoAR using the Optimal technique.

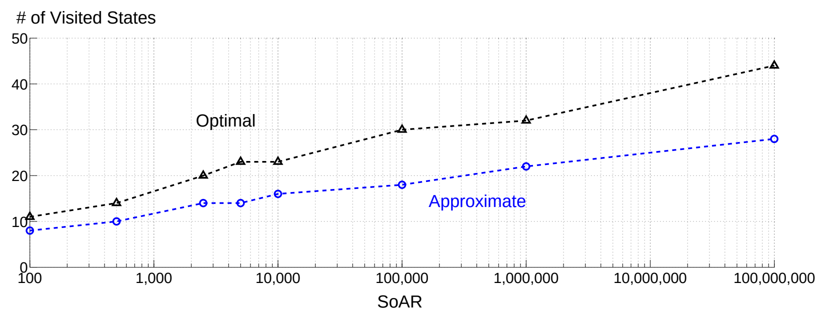


Figure 7.4: Number of experiments conducted to compute SoAR with the Optimal and the Approximate techniques.

iments to compute the value of T that maximizes the output of the function. Ten states are repeated from previous iterations with the same value of T. To eliminate these, the heuristic maintains the observed results for the different values of T and performs a look up of the results prior to conducting the experiment. This reduces the number of unique experiments to 44. This is 3.14 times the number of experiments conducted with a system modeled to have a SoAR of 500 (which is several orders of magnitude lower than 100 million).

Figure 7.4 shows the number of unique experiments executed with each of the techniques. Figure 7.5 shows the ideal SoAR as well as the computed SoARs by the two techniques for the different curves. As shown in Figure 7.5 the Optimal solution always computes the expected SoAR for the system and the Approximate approach reports a value which is within $\pm 10\%$ of the expected SoAR value. However, in some cases the Optimal approach conducts more experiments and visits more states in order to find the solution, see Figure 7.4. This is because for these cases the Optimal solution, executes a larger number of experiments before it discards intervals that do not contain the peak.

The total number of visited states is computed by summing the number of experiments executed in the hill climbing phase and the number of experiments executed in the limited search phase. For both techniques this value is dependent on the value of the climbing factor. 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 limited search phase as it may result in a larger search interval, see Figure 7.6.

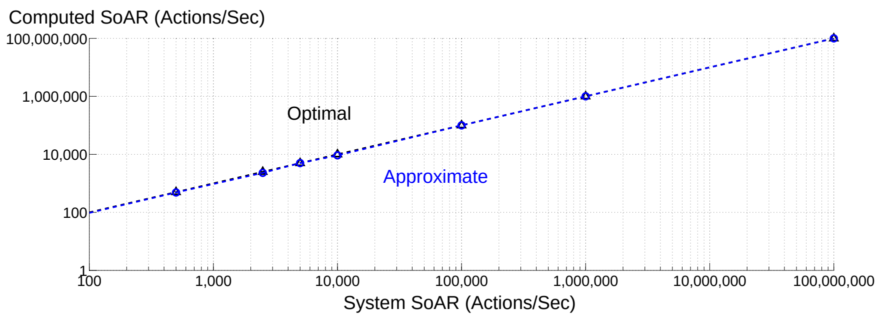


Figure 7.5: Computed SoAR using Optimal and Approximate techniques.

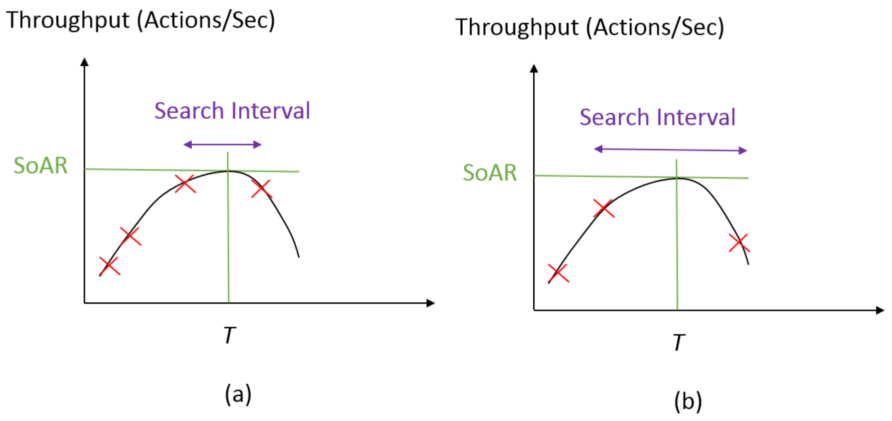
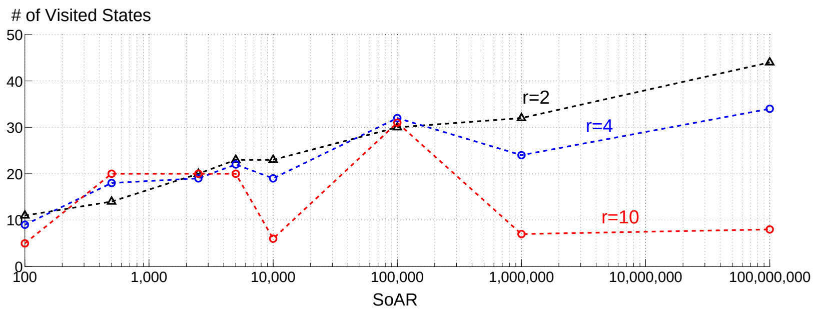


Figure 7.6: The impact of the value of climbing factor on the rating, (a) small climbing factor (b) large climbing factor.

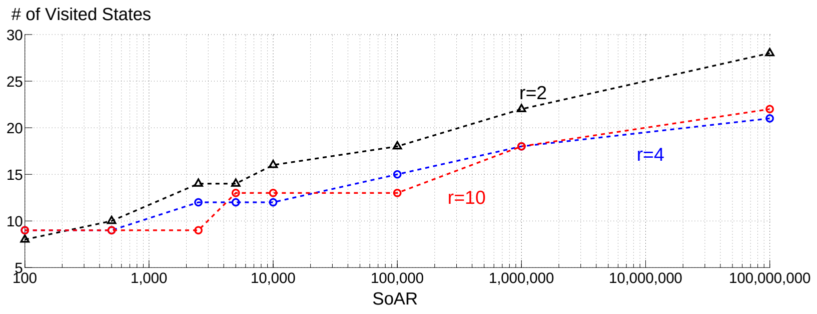
Figure 7.7.a and 7.7.b show the impact of three different climbing factors on the number of states visited by the Optimal and the Approximate techniques as a function of the SoAR for the system. With these results, with a higher SoAR for the system, a larger climbing factor visitis fewer states for both techniques.

Deciding the appropriate value for the climbing factor depends on the workload issued and the behavior of the data store as a function of T. One approach is to start with $r = 2$ and adjust its value at the end of every experiment in the hill climbing phase. The first experiment runs with $T = 1$ and observes λT_1 as its throughput. Now we increase the number of threads to $2T$ as the initial value for r is 2 and note down the observed throughput as λT_2. If $\frac{\lambda T_2}{\lambda T_1} > r$ then we increase the value of r by a factor of two else we do not change its value.

Finally, for a system that violates our assumptions, both Optimal and Approximate may fail to identify system SoAR. For example, Figure 7.8 shows a system where the observed throughput is not a increasing function of system load. In such a case, both Optimal and Approximate may become trapped in a local minima and fail to identify the peak throughput of the system. A possible approach may resemble simulated annealing that performs (random) jumps to escape local minima. We do not discuss this possibility further as we have not observed a system that violates the stated assumptions.



7.7.a) Optimal



7.7.b) Approximate

Figure 7.7: The impact of the value of climbing factor on the number of states visited.

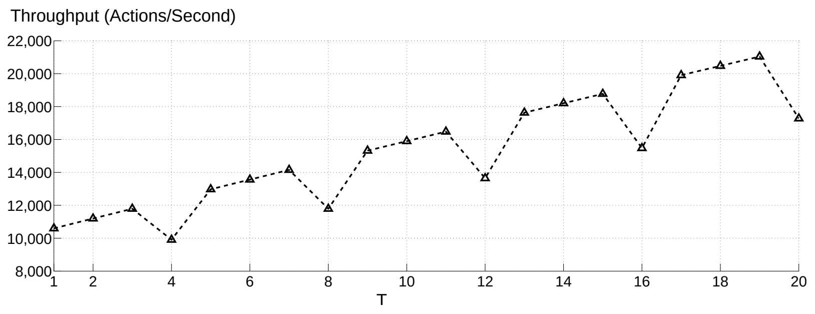


Figure 7.8: Behavior of a system violating the SoAR assumptions.

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

The value of δ is an input to BGCoord. An experimentalist computes this value using the Delta Analyzer, see Section 7.3. If it is left unspecified, BG uses Δ for the rating process.

7.2 Agile Loading Techniques

With those workloads that change the state of the database (its size, characteristics, or storage space[3]), one may be required to destroy and reconstruct the database at the beginning of each experiment to obtain meaningful ratings. To illustrate, consider an asymmetric BG workload that generates more friendships than thawing them, resulting in a larger number of friendships among members. Use of this kind of a workload across different back to back experiments results in each experiment imposing an action such as List Friend for a member with a larger number of friends. If the data store becomes slower [4] 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 the introduction of agile data load techniques to expedite the rating mechanism.

This section presents 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.

[3]Using workloads that insert/delete data to/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.

[4]An example from YCSB is Workload D that 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 which may slow down a data store.

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 repeated experiments, see Section 4.1. This prevents both an increased database size and the possible depletion of the benchmark database from friendship relationships to thaw. See Section 7.2.3 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 7.4 shows the hybrid approaches provide a factor of five to twelve speedup in rating a data store.

The primary contribution of this section is several agile data loading techniques for use with the cloud benchmarks. Note that the overhead of loading a benchmark database is a recognized topic by practitioners dating back to Wisconsin Benchmark [19, 39] and 007 [23, 116, 117], and by YCSB [29] and YCSB++ [88] more recently. YCSB++ [88] 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 three 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. Third, DBIL does not require knowledge about load balancing mechanisms implemented within a multi-shard data store and can be used to load the appropriate data on each shard[5]. 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.

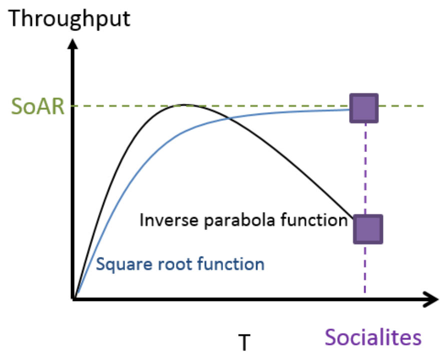
The following 3 sections describe an implementation of DBIL, RepairDB, and LoadFree with BG. As shown in Figure 7.2, BG assumes BGCoord issues commands to BGClients to either create BG's schema, construct a database and load it, or generate a workload for the data store. We now describe DBIL, RepairDB, and LoadFree in turn.

7.2.1 Database Image Loading

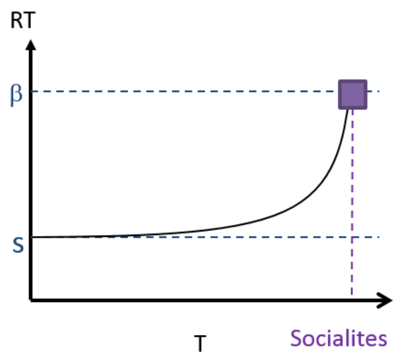
Various data stores provide specialized interfaces to create a "disk image" of the database [85]. Ideally, the data store should provide a high-throughput external tool [88] that the benchmarking framework employs to generate the disk image. Our target data stores (MongoDB, MySQL, an industrial strength RDBMS named[6] SQL-X) do not provide such a tool. Hence, our proposed tech-

[5]With multi-shard MongoDB, the data is initially loaded onto the primary shard and then chunks are dispersed across all the shards for better load balancing. With DBIL once the disk image for each shard is created after the load balancing is completed, the image can be used repetitively to avoid the load balancing process for subsequent loads which may be slow. For example with an 18 shard MongoDB and two secondaries for each shard, it takes almost 24 hours to load a social graph with 100,000 members and wait for chunk migration/balancing to complete. with DBIL the subsequent loads were reduced to 9 minutes.

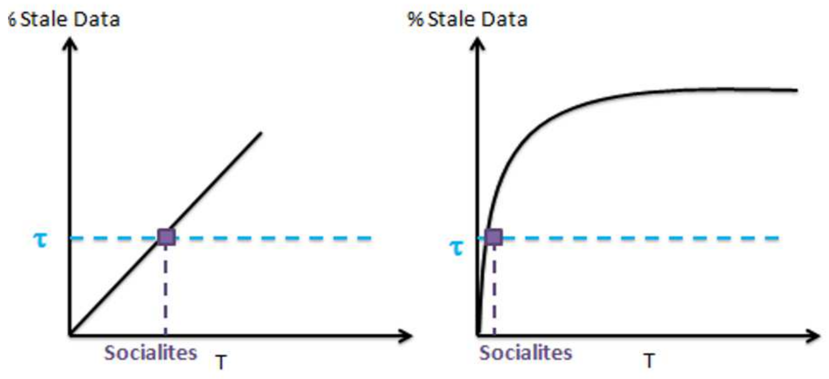
[6]Due to licensing restrictions, we cannot disclose the name of this system.



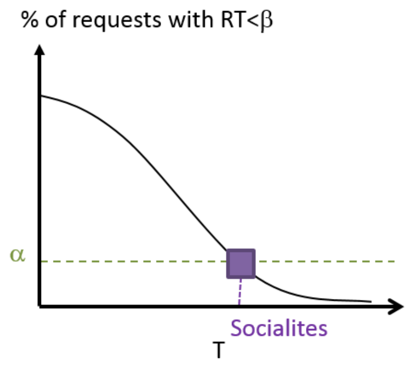
7.9.a : Throughput as a function of T



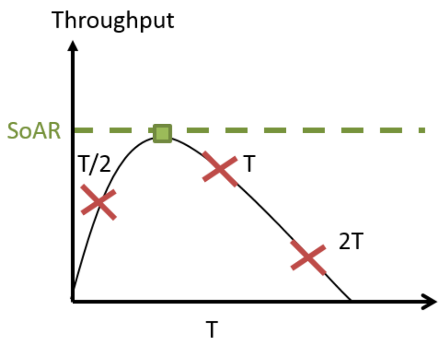
7.9.b : Average response time as a function of T



7.9.c : Percentage of stale data as a function of T



7.9.d : α as a function of T



7.9.e : SoAR search space

Figure 7.9: Assumptions of BG's rating technique.

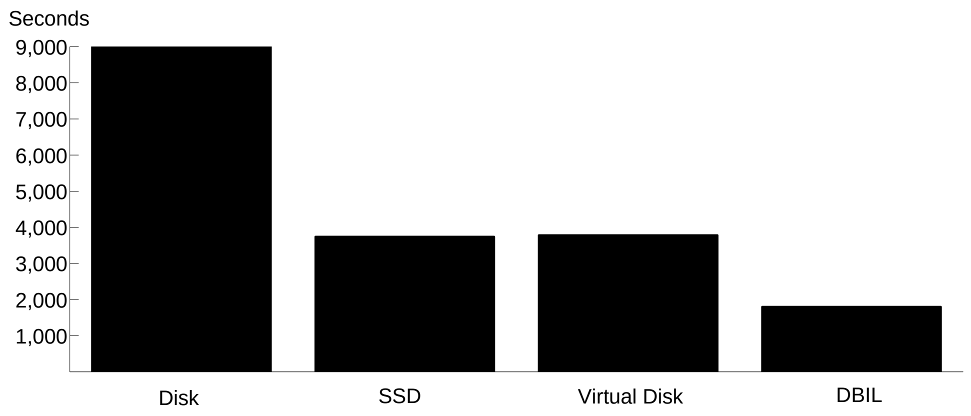


Figure 7.10: Comparing the loading duration for DBIL with loading duration for using BG to construct a social graph of 100,000 members using different storage mechanisms for the data store.

nique 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. Figure 7.10 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[7]. 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 members and friendships dynamically[8]. This may cause a data store to read and write the same page (corresponding to a member) many times in order to update a piece of information (a member's JSON object) repeatedly (modify friends). In addition, it also must construct index structure that is time consuming[9].

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[10]. Subsequently, it creates the disk image of the database for future use. Its implementation details are specific to a data store. Below,

[7]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.

[8]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.

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

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

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 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 [84]. 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.

7.2.2 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 [74, 73] 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 7.4), 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

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

Table 7.1: Factor of improvement in load times with RepairDB when compared with recreating the entire database, target data store is MongoDB, M=100K, ρ=100.

unpredictable[11] 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 7.1 shows RepairDB improves the load time of MongoDB[12] 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.

7.2.3 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 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, see Section 5.2. 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

[11]See Chapter 6 for the definition of unpredictable data.

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

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 (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 7.2, 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 [56] or suffer from race conditions that leave the cache and the database in an inconsistent state [46], a data store may employ an eventual consistency [112, 103] technique that produces either stale or inconsistent data for some time [88], 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 7.4) 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 7.4. 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[13] amount of time or has issued a pre-specified number of requests [29, 88]. 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.

[13]Described by the workload parameters.

We extend BGCoord to issue the following additional[14] 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 [12] for details.

7.3 Ideal δ

BG computes the SoAR and Socialite rating for a data store given a pre-specified SLA requirement. The SLA consists of four parameters: tolerable response time, β, percentage of actions to observe the given response time, α, amount of unpredictable data, τ and duration for which these requirements should hold true, Δ. The duration of the rating process is a function of the number of conducted experiments and the duration of each experiment (δ). The number of conducted experiments depends on the heuristic search process and whether the target is SoAR or Socialites rating.

Below, we focus on SoAR and how to determine the duration of each experiment, δ. It is undesirable to select a δ value equal to Δ, i.e., $\delta = \Delta$, because Δ is an input parameter and its value might be large, resulting in a long rating process. The challenge is to identify a value for δ such that it reflects the behavior of the system with $\delta = \Delta$ and is the smallest possible value that satisfies the following: 1) Computes the same SoAR as Δ. 2) It is sufficiently long in duration to generate the pre-specified workload. The workload generated with the ideal δ should resemble the workload that was given to the benchmark as an input (same distribution for actions). Otherwise, the numbers observed may be incorrect because different actions may have different service times. For example, consider a mixed workload of four actions with the following probabilities: 0.99, 0.002, 0.003, 0.005 where the first action has a service time of 1 second. With a small value of δ (i.e., $\delta = 1$ second) and with 1 thread ($T = 1$), BG may only generate the first action and never have a chance to generate other actions. This is because BG supports a closed simulation model, all other actions have very low probabilities of reference and the service time for the first action is equal to the ideal δ. In this case the workload generated by BG will not resemble the actual intended workload and the results gathered may not be accurate.

3) To estimate the ideal δ, we should observe the system processing the workload in a *steady-state*. This state is one whose resource utilization and observed throughput do not change in time. Hence, the observed performance will continue to hold into the future. Note that a system reaches the steady-state after its warm-up phase. For example, with data stores such as CASQLs, requests observe a higher response time during the warm up phase when the cache is cold. Once the cache

[14]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.

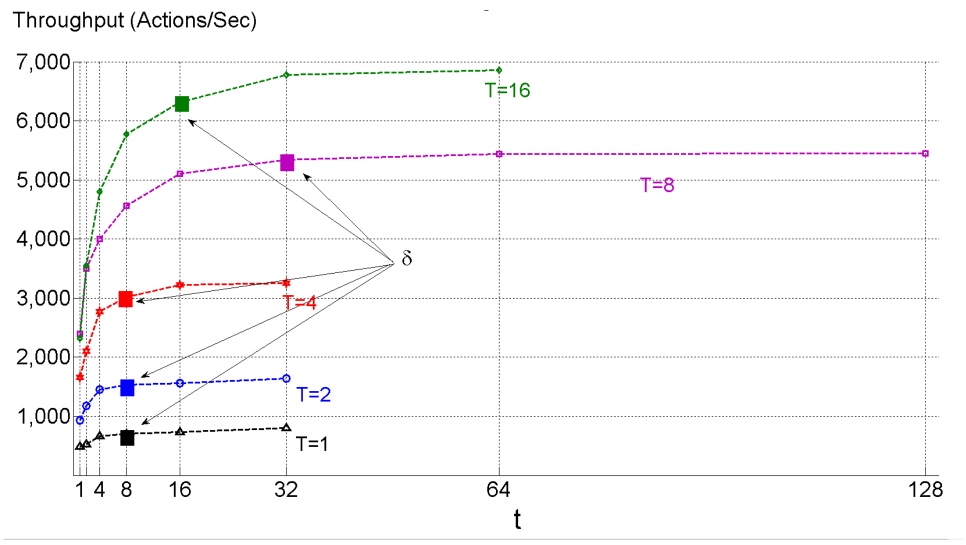


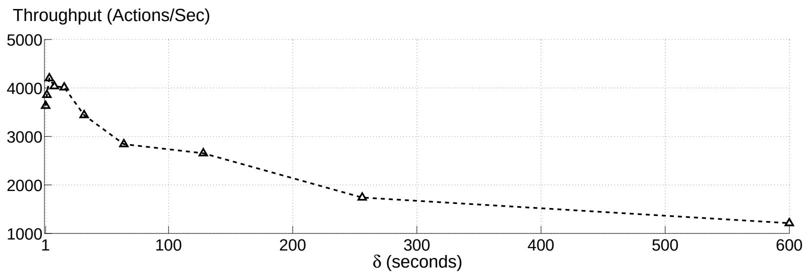
Figure 7.11: System throughput as a function of experiment duration with different imposed load against the system for a fixed workload.

reaches a state that produces a constant hit rate for a workload, requests start to observe the average response time. Thus, a steady-state is reached by requiring a ”warm-up” period that is supported by BG. In the following discussion, we separate a discussion of δ from the warm-up period, assuming BG is configured with the correct duration for this period. The appropriate value for δ is both workload and data store dependent. This value can be decided in a pre-processing phase (using the *Delta Analyzer*) and then given to the BGCoord as an input parameter.

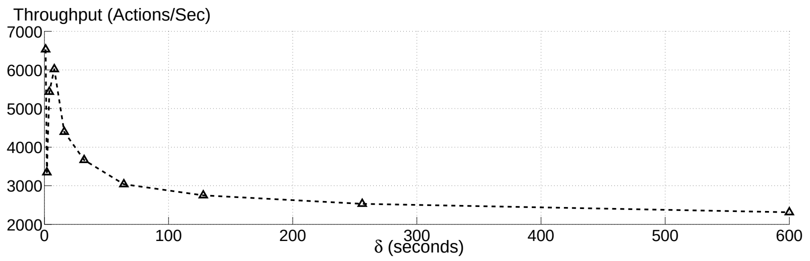
Figure 7.11 shows the observed throughput for a data store as a function of the experiment duration with different amount of load (T) imposed against the data store. As shown in this figure, an increase in the thread count (T) results in a higher observed throughput. For this workload, with $T = 16$, the network on the node hosting the data store gets fully utilized resulting in the maximum observed throughput for the system[15]. For all values of T, smaller experiment durations result in either an incorrect workload issued against the data store or and unsteady data store. However, as we increase the experiment duration, the observed throughput from the data store and the resource utilization for the node hosting the data store stabilize identifying a range of acceptable values for δ.

Next the BGCoord will use δ to conduct the rating experiments which compute the SoAR and Socialites rating for a data store for a given workload. If the pre-processing phase fails to compute a value for δ ($\delta < \Delta$) which satisfies the conditions above, it will pick $\delta = \Delta$ as the duration for the rating experiments. For example, with a workload consisting of feed following actions, Share Resource action (SR) and View News Feed action (VNF), the data set gets larger as more SR actions are issued against the data store. The complexity of the VNF action is a function of the database size, causing the observed throughput to change as a function of the experiment duration. This prevents the Delta Analyzer from computing an ideal δ value, see Figure 7.12. However, this is not important for this class of experiments because one may use BG to quantify the behavior of different

[15]Increasing the thread count to a value greater than 16 will not improve the observed throughout for the system suggesting the SoAR of the system to be observed with a thread count equal or lower than 16.



7.12.a : Mixed High (11%) Write workload of Table 8.9
T=4



7.12.b : Mixed Very Low (0.2%) Write workload of Table 8.9
T=24

Figure 7.12: Importance of experiment duration (δ) on the throughput of MongoDB for two different workloads of Table 8.9. With both workloads, the Delta Analyzer fails to compute the ideal δ. T is the thread count picked by the Delta Analyzer for Delta Analysis.

algorithms with one another. With these, the SoAR rating is not relevant and the Delta Analyzer does not apply.

In Figure 7.12, with small values of δ, the duration is not sufficiently long, the pre-specified workload is not generated and the numbers observed in these experiments are not accurate. As the value of δ increases the observed throughput decreases. This is because the number of issued SR actions increases the size of the database and this increase impacts the response time for different actions.

7.3.1 Delta Analyzer

The Delta Analyzer uses an iterative process to compute the ideal value of δ. It consists of two steps: 1) identifying the amount of load to impose on the data store which results in the data store becoming 100% utilized (this load is defined by the number of threads (T) emulating concurrent socialites), and 2) computing the ideal value of δ using the T computed in Step 1.

The analyzer first emulates 1 thread issuing the pre-specified workload against the data store for duration of t seconds. t is determined as the amount of time required for the data store to reach a steady-sate. The steady-state is identified by monitoring the data store's behavior. This includes monitoring both the resource utilization on the data store and its observed throughput for the given workload in discrete windows of t' seconds. Once the resource utilization and the observed throughput for m consecutive windows varies by less than π%, the experiment halts. Next, the final resource utilization (average disk queue, network utilization, memory utilization and

CPU utilization) on the data store is used to predict the appropriate thread count (T) that may result in the data store utilizing at least one of its resources completely. Next the analyzer uses BG with $T = c \times T$ threads to impose the specified workload on the data store for t seconds (c is an inflation factor which is used to speed up the process of finding the thread count that utilizes the data store resources completely).

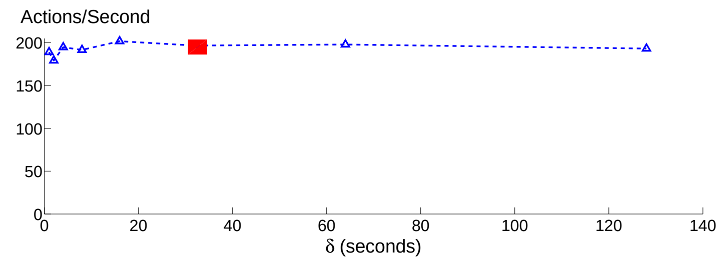
It also monitors the resource utilization on the data store to identify if its predictions were correct and if T results in the data store becoming fully utilized. If it does then T is selected for the second phase of the analysis, else the resource utilization with the current T and the inflation factor are used to predict the new thread count. This process continues until either the resources on the node hosting the data store become fully utilized or the data store node's utilization for the various thread counts does not vary by more than $\pi\%$. Once the value for T is decided, Delta Analyzer conducts multiple rounds of experiments imposing a workload generated by T threads with various durations. It starts by assigning $\delta = 1$ second and increases δ by a factor of two in each iteration. The workload issued against the data store (mix of actions) and the overall throughput for each experiment is monitored. The issued workload is compared with the pre-specified workload and the observed throughput is compared with the observed throughput from the previous experiments. The minimum δ value which results in a constant overall throughput (+/- $\pi\%$ marginal error) for m consecutive experiments (δ values) and generates a workload similar to the pre- specified workload will be selected as the ideal δ value for the given workload and data store. For example, if the experiments with $\delta = 2$, 4 and 8 seconds all result in almost the same throughput and generate the appropriate workload, then $\delta = 2$ second is selected as the ideal δ value. Next, the BGCoord of BG employs this value to rate the data store. Figure 7.13 and Figure 7.14 we show the behavior of MongoDB and SQL-X in terms of observed throughput (actions/second), as a function of the duration of the experiment (δ). The T used by the Delta Analyzer for each experiment is available in the image caption. This T is predicted to result in full resource utilization for the data store. The ideal value for δ is highlighted in each graph. Figure 7.13.a shows 8 experiments are conducted with different durations shown on the x-axis. In these experiments, the variation between the observed throughput with the first six experiments was higher than the tolerable threshold chosen at 5%. The next 3 experiments met the requirement and the one with the smallest δ (32 seconds) was chosen. The workload in Figures 7.13.b and 7.13.c are different than those in Figure 7.13.a resulting in a different thread count that utilizes resources completely.

In addition, the number of conducted experiments to find three consecutive experiments that meet the requirements for computing the ideal δ is fewer. Similar observations are shown with SQL-X in Figure 7.14. However, with SQL-X, the number of conducted experiments with workloads involving writes is higher.

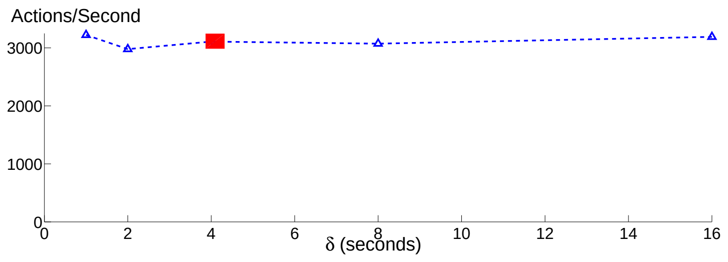
7.4 An Evaluation

This section quantifies the speedup observed with the 3 proposed loading techniques and the Delta Analyzer using the 10% Write workload of Table 7.3. 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 Delta Analyzer, we focus on both a naïve data loading technique and DBIL to quantify the observed speedup.

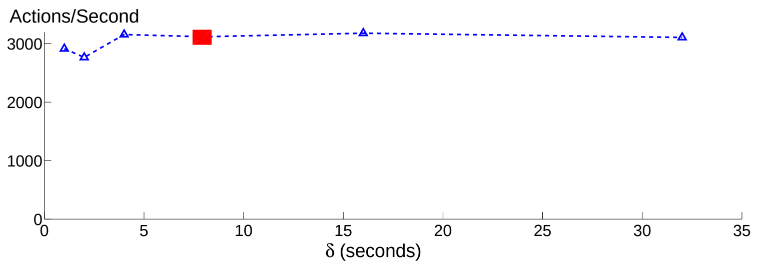
In the following, we start with an analytical model that describes the total time required by BG to rate a data store. Next, we describe how this model is instantiated by the data loading techniques. Subsequently, we describe how the ideal δ impacts the overall rating duration. We conclude by



7.13.a : List Friends, T=2

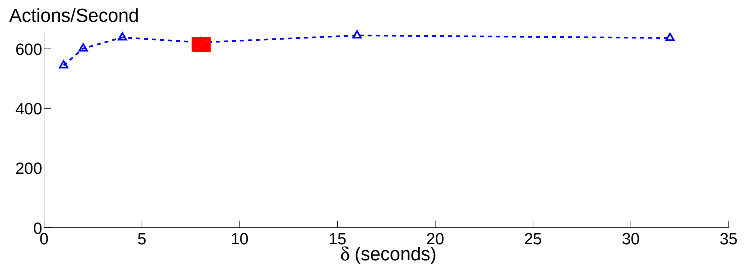


7.13.b : Very Low (0.1%) Write, T=4

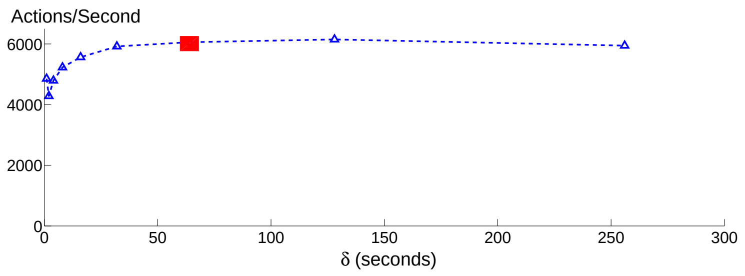


7.13.c : High (10%) Write, T=4

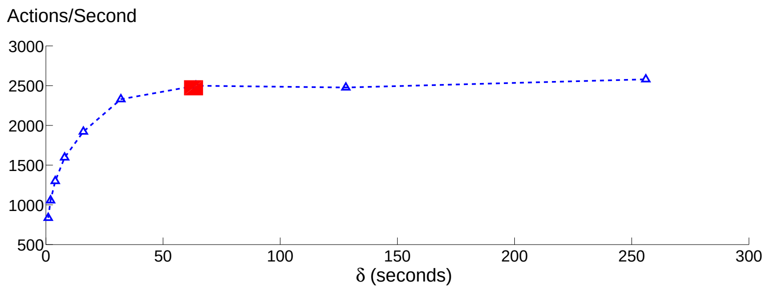
Figure 7.13: Computation of ideal δ for MongoDB for three different workloads of Table 7.3. With the List Friends workload, the Delta Analyzer picks $T = 2$, with the Very Low workload, it picks $T = 4$ and with High it picks $T = 4$ as the thread count for Delta analysis. The load imposed by T results in the network on the data store becoming 100% utilized.



7.14.a : List Friends, T=4



7.14.b : Very Low (0.1%) Write, T=4



7.14.c : High (10%) Write, T=2

Figure 7.14: Computation of ideal δ for SQL-X for three different workloads of Table 7.3. With the List Friends workload, the Delta Analyzer picks $T = 4$, with the Very Low workload, it picks $T = 4$ and with High it picks $T = 2$ as the thread count for Delta analysis. The load imposed by T results in the network on the data store becoming 100% utilized.

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 7.2: BG's rating parameters and their definitions.

BG Social Actions	Type	List Friends	Very Low (0.1%) Write	High (10%) Write
View Profile	Read	0%	40%	35%
List Friends	Read	100%	5%	5%
View Friends Requests	Read	0%	5%	5%
Invite Friend	Write	0%	0.04%	4%
Accept Friend Request	Write	0%	0.02%	2%
Reject Friend Request	Write	0%	0.02%	2%
Thaw Friendship	Write	0%	0.02%	2%
View Top-K Resources	Read	0%	40%	35%
View Comments on Resource	Read	0%	0%	9.9%
Post Comment on a Resource	Write	0%	0%	0%
Delete Comment from a Resource	Write	0%	0%	0%

Table 7.3: BG workloads consisting of a mix of social networking actions used for ideal δ experiments.

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 7.4: BG's rating of MongoDB with 1 BGClient using High workload of Table 7.3, 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 7.5: Observed speedup (S) when rating MongoDB using agile loading techniques.

presenting the observed enhancements and quantifying the observed speedup relative to not using the proposed techniques.

7.4.1 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 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, 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 7.4.
- 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.

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

The total rating duration is:

$$\Lambda = \zeta + (\omega \times \nu) + (\eta \times \delta) + \Delta \tag{7.1}$$

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 7.4. Its value is zero with LoadFree[16].

By setting ν equal to ζ, Equation 7.1 models a naïve use of BG that does not employ the agile data loading techniques described in this chapter. Such a naïve technique would require 1927 minutes (1 day and eight hours) to rate MongoDB with 100K members. The third row of Table 7.4 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 7.4. 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 7.4 show the observed trends continue to hold true with databases consisting of 500K and 1 million members. In addition, when rating a data store 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 naïve use of BG (ν=ζ) and when $\delta = \Delta$? Amdahl's Law [2] provides the following answer:

$$S = \frac{1}{(1-f) + f/k} \tag{7.2}$$

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 these two factors are computed for the speedup results shown for various agile data loading techniques and the ideal δ rating duration.

With only focusing on the data loading techniques, 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. Figure 7.15 illustrates this by showing the value of S as the value of f with 0.92 (1 million member database) using different k values.

When only changing the ideal δ from $\delta = \Delta$ to ideal δ, the fraction of work done in the faster mode is computed as $f = \frac{\eta \times \Delta}{\Lambda}$. With a fixed rating duration, e.q. $\delta = 180$ seconds, f will be computed as $f = \frac{\eta \times 180}{\Lambda}$. Similarly, k is computed as $k = \frac{\Delta}{\delta}$ and $k = \frac{180}{\delta}$ respectively.

When we use both an agile data loading technique and the ideal δ computed by the Delta Analyzer in our rating experiments, the following are used to compute the overall speedup compared to the naïve usage of BG with $\delta = \Delta$ snd $\delta = 180$ seconds respectively: $f = \frac{(\omega \times \zeta) + (\eta \times \Delta)}{\Lambda}$ and $k = \frac{(\omega \times \zeta) + (\eta \times \Delta)}{(\omega \times \nu) + (\eta \times \delta)}$, and $f = \frac{(\omega \times \zeta) + (\eta \times 180)}{\Lambda}$ and $k = \frac{(\omega \times \zeta) + (\eta \times 180)}{((\omega \times \nu) + (\eta \times \delta)}$.

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

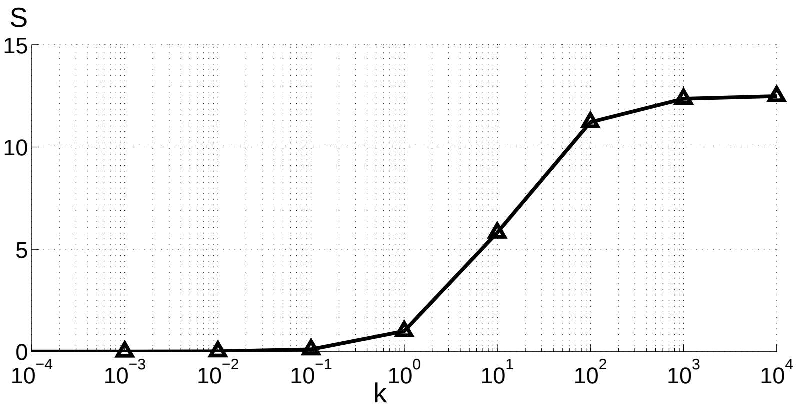


Figure 7.15: S as a function of k.

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 7.6: BG's rating of MongoDB, MySQL and SQL-X with 1 BGClient using High workload of Table 7.3, M=100K, ϕ=100, ρ=100, ω=11, $\delta = 3$ minutes, $\Delta = 10$ 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.

7.4.2 Observed Speedup with Load Techniques

Table 7.5 shows the observed speedup (S) for the experiments reported in Table 7.4. 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 7.5 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 7.4. 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 [17] SQL-X. The time to load these data stores and rate them with 100K member database is

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

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 7.7: Observed speedup (S) when rating MongoDB, MySQL and SQL-X with M=100K for High workload of Table 7.3.

shown in Table 7.6. 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 7.7.

7.4.3 Observed Speedup with Load Techniques and Ideal δ

This section analyzes the observed speedup with the δ value computed using the Delta Analyzer, highlighting its usefulness to expedite the rating process. We assume that 11 rating experiments are required to compute the SoAR of the systems. We compare this with two alternative choices of δ. First, when δ is set to the SLA duration specified by the experimentalist, i.e., δ=Δ. Second, when an experimentalist sets the value of δ based on experience. In particular, when we first started using BG's rating mechanism, we quickly converged on 180 seconds as sufficiently long to rate different data store effectively, i.e., δ=180 seconds. The following considers these alternative choices of δ values with both the DBIL technique and a naïve technique that re-loads the database each time. Table 7.8 shows the parameters used for the analytical model for both MongoDB and SQL-X with a social graph of 10,000 members.

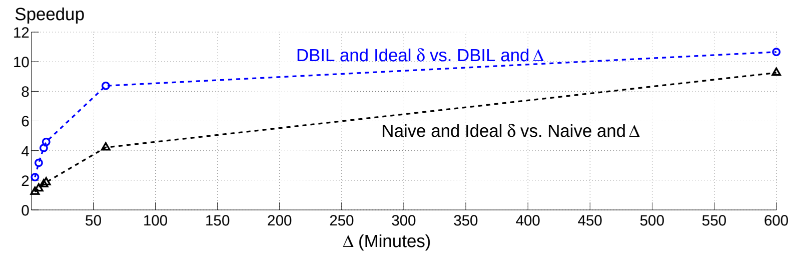
Figure 7.16 shows the observed speedup observed with MongoDB using the analytical models of Section 7.4.1. Its x-axis shows different SLA durations (Δ) ranging from 3 minutes to 10 hours. The y-axis shows the observed speedup. The speedup is most dramatic with the $delta$ computed by the Delta analyzer because it computes a value of 4 seconds for δ, see Figure 7.16.a. In this figure, as the value of Δ increase on the x-axis, the duration of each experiment becomes longer with both DBIL and naïve. With the Delta Analyzer the duration of each experiment is kept constant at 4 seconds, enhancing the observed speedup. The gains are more significant with DBIL because the time to recreate the database at the beginning of each experiment is significantly faster than that with naïve, 50 seconds versus 11 minutes.

Figure 7.16.b shows the scenario where the value of δ is kept constant at 180 seconds. As the duration specified by the SLA increases, the observed speedup drops because the portion of work that does not benefit from a shorter experiment time dominates, see Equation 7.2. These results highlight the importance of using Delta Analyzer to compute the duration of each experiment instead of either estimating it or defaulting to the SLA duration.

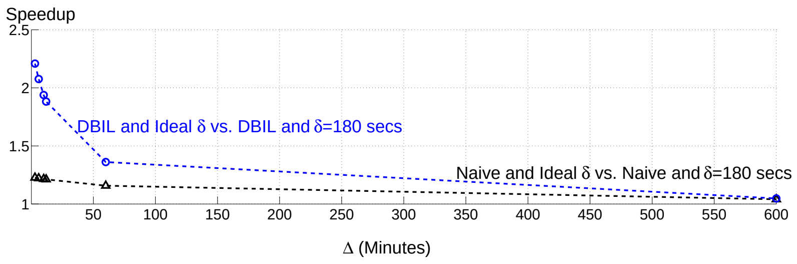
Figure 7.17 shows the observations made with MongoDB hold true with the SQL-X system. Figure 7.18 compares the speedup observed with both SQL-X and MongoDB as a function of the duration specified by SLA and used as the duration of each experiment, δ=Δ. The observed speedup is higher with MongoDB because its database creation time at the beginning of each experiment is faster. Thus, the fraction of work that benefits from the use of δ computed using the Delta Analyzer is more dominant, see Equation 7.2.

Data store	ζ (secs)	ν (secs) using DBIL	ideal δ (secs) for Very Low (0.1%) Writes workload of Table 7.3
MongoDB	686	50	4
SQL-X	491	300	64

Table 7.8: Parameters used for speed up evaluation for a social graph with 10,000 members.

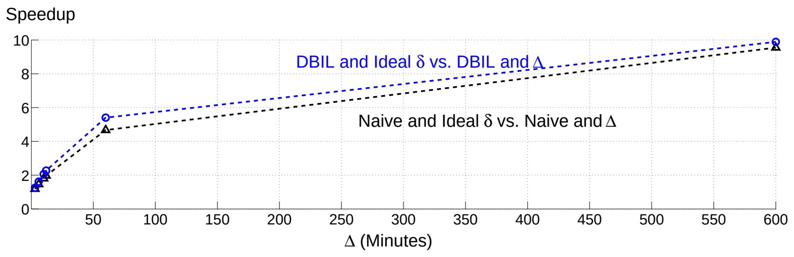


7.16.a : $\delta = \Delta$.

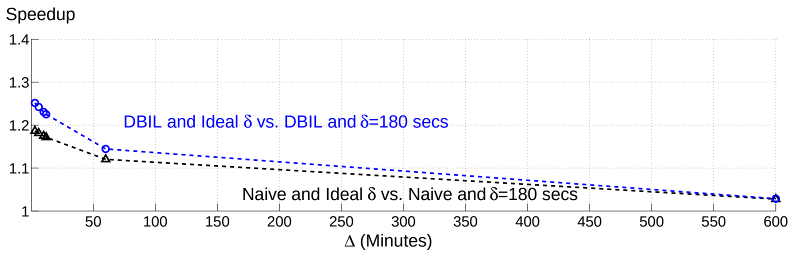


7.16.b : $\delta = 180$ seconds.

Figure 7.16: Speedup computed with MongoDB when comparing the use of ideal δ for rating vs. the use of two δ values for two loading techniques: DBIL and naïve. Table 7.8 shows the parameters used for this computation. To generate the graphs the following values were assigned to Δ: 3, 6, 10, 12, 60, 600 minutes. The Very Low (0.1%) Write workload of Table 7.3 was used for these experiments.



7.17.a $\delta = \Delta$.



7.17.b : $\delta = 180$ seconds.

Figure 7.17: Speedup computed with SQL-X when comparing the use of ideal δ for rating vs. the use of two δ values for two loading techniques: DBIL and naïve. Table 7.8 shows the parameters used for this computation. To generate the graphs the following values were assigned to Δ: 3, 6, 10, 12, 60, 600 minutes. The Very Low (0.1%) Write workload of Table 7.3 was used for these experiments.

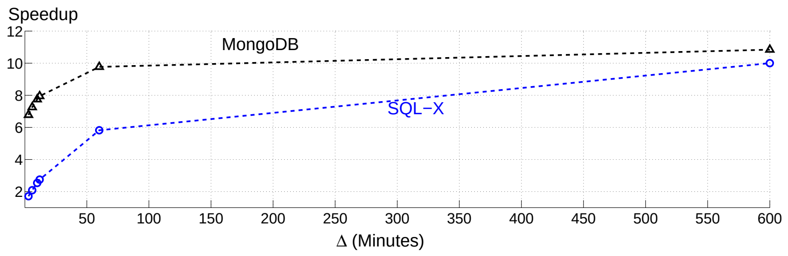


Figure 7.18: Speedup computed with SQL-X and MongoDB when comparing the use of the DBIL loading technique and ideal δ for rating vs. the use of the naïve loading technique and $\delta = \Delta$. The following values were assigned to Δ to generate this graph: 3, 6, 10, 12, 60, 600 minutes. The Very Low (0.1%) Write workload of Table 7.3 was used for these experiments.

Chapter 8

BG's Alternative Use Cases

One may use BG for a variety of purposes. This includes comparing the performance of different data stores with one another, quantifying the tradeoffs associated with alternative design decisions such as weak consistency techniques, characterizing the behavior of data mining algorithms in the context of biological databases and programming paradigms and others. This chapter focuses on the first two use case of BG. BG can be used to both compare various solutions by computing their SoAR and Socialites rating, see Chapter 7, and study their behavior by understanding observed trends. Comparing the performance of alternative solutions is important as it provides the experimentalist with possible inefficiencies and bugs. This will allow an application developer to pick the solution which is better for her application and also help data store vendors improve their solutions. In addition, understanding the behavior of a solution by analyzing trends in its observed behavior will help an application developer to make predictions about the system behavior in various scenarios and take the appropriate steps to improve the performance of her application. It will also help data store vendors improve the functionality and behavior of their system for different scenarios. Section 8.1 highlights the importance of physical data representation on the performance of a single node data store. It describes the use of BG to compare and contrast various physical data models for the data stores introduced in Chapter 1. The results gathered provide insights about how the data model for a single node data store can be modified to improve its performance. In addition, it uses the obtained rating results to identify trade-offs between the different data stores for its various workloads.

Section 8.3, emphasizes on the use of BG to study the scalability claims of different architectures. We first identify the factors impacting the scalability of a data store and then use those factors to investigate their impacts, understand tradeoffs between the performances of the alternative architectures and learn about the formed bottlenecks limiting their behavior. We use the results of our first study from Section 8.1 to select the architecture which results in the best single node performance for BGs workloads and utilize BG's rating mechanism to characterize the impact of sharding, replication, processing capability and data set size on a multi-node clustered MongoDB's performance.

Finally Section 8.4 uses BG to explore two alternative architectures enabling feed following actions. Real-time computing of news feed for users is now a key feature of many popular social networking systems. This computation is not trivial due to the combination of dense connection networks, low latency requirements and high throughput for most recent and relevant activities. Hence, developing an architecture that considers all these factors and results in the best performance is one of the challenging research topics. We describe each of the architectures, namely termed as Push and Pull, in detail, and demonstrate the tradeoffs between them. Our studies focus on using

BG to understand the factors impacting the performance of these architectures as well as the trends in their observed behaviors.

8.1 Use of BG to Study Performance of a Single Node Data Store

In this section, we use BG to investigate alternative physical data organization techniques to enhance the performance of an industrial strength relational database management system (RDBMS) and a document store named MongoDB. First, we highlight the importance of physical data design and its impact on the performance of a data store. Second, we illustrate the use of the BG benchmark to evaluate alternative physical representations for a given data store. We report SoAR ratings using one SLA: 95% of actions observing a response time equal to or faster than 100 milliseconds. Given several data organization techniques, the one with the highest SoAR (see Chapter 7) is superior.

The RDBMS represents the so-called SQL solutions that employ the join operator and implement the concept of transactions that support ACID properties. MongoDB represents systems that employ a JSON representation of data to eliminate the join operator and scale horizontally. While horizontal scaling is important and discussed in Section 8.3, the performance of a single node is equally important. For example, Section 8.1.1 shows that a change in how thumbnail images (used to display friends of a member) are managed by MongoDB enhances its SoAR from zero to more than seven thousand. Horizontal scalability is not a substitute for such physical data design decisions (and vice versa).

In general, with a scalable system, the faster the performance of a single node, the fewer nodes one needs [105]. Consider two solutions that utilize the same system. With an enhanced data design, say A, a single node provides 10 times the performance of a basic data design B. This means a service provider with a 1,000 node deployment using design B may provide the same performance with 100 nodes using design A. This lowers hardware cost, rack space, cooling requirement, and power consumption. More significantly, if each node fails on average every three years, then design B will see a failure every day, while design A will see a failure less than once every 2 weeks.

The data design techniques that we investigate are as follows. First, we analyze alternative designs to store and retrieve images. Many social networking sites store and retrieve the profile image of a user and the thumbnail image of their friends. With the RDBMS, we analyze whether these images should be stored as files in the file system, as a BLOB in the RDBMS, or a hybrid of the two. With MongoDB, we investigate the use of its Grid File System (GridFS), thumbnails as an array of bytes in the member document, and as files in the file system. These decisions have a profound impact on the observed system performance. With SQL-X, BG's Social Action Rating (SoAR) of the right design is more than forty times higher than a basic design without the optimizations. With MongoDB, SoAR of the right design is seven thousand while the basic design has a SoAR of zero.

Second, with a relational data design, we consider whether pending friend invitations and confirmed invitations should be stored in one table or two different tables. With each, we consider whether a friendship should be represented as one or two rows. We observe the two table design enhances SoAR of SQL-X by 33% when write actions occur frequently (10%). The observed difference between one and two row representation of a friendship is negligible. In addition, we consider migrating the workload of simple analytics performed by a read action (such as showing the number of friends for a member) to write actions. A key negative finding here is that materialized views of our industrial strength RDBMS do not support this concept effectively. We consider an alternative that requires a developer to implement the analytics as columns of a row and show that it enhances SoAR by more than a factor of two when write actions are infrequent.

Third, we analyze the use of two middle-tier caches, memcached and Ehcache, to look up the results of social actions instead of computing by issuing queries to a data store. Both memcached and Ehcache are in-memory Key-Value Stores (KVSs) that augment a data store to implement two different architectures. A central premise of our study is that there is sufficient memory to accommodate the entire database [66, 90]. This renders the results obtained from the cache augmented data stores comparable with the data store in stand alone mode. Our experimental results show Ehcache provides the highest SoAR rating. It enhances the performance of our data stores by more than a factor of 5 when a workload utilizes the CPU of the server hosting our data store fully. It enhances SoAR by more than a factor of 50 when the workload exhausts the network bandwidth link to the server, utilizing it fully, see Section 8.1.7 for details.

BG models a database consisting of a fixed number of members (M) with a registered profile. Each member profile may consist of either zero or 2 images. With the latter, one image is a thumbnail and the second is a higher resolution image. While thumbnails are displayed when listing friends of a member, the higher resolution image is displayed when a member visits a member's profile. An experiment starts with a fixed number of friends (ϕ) and resources per member. This section assumes a database of 10,000 members ($M = 10,000$) with no pages ($P = 0$), 2 KByte thumbnail images and 12 KByte profile images. We also consider larger databases with larger number of members and databases with no images. All experiments start with 100 friends and resources per member.

An ideal physical data design is one that maximizes SoAR of a system. All SoAR ratings in this section are established with the following SLA: 95% of requests observe a response time of 100 milliseconds or faster with unpredictable (stale) data lower than 0.1%. Data designs using materialized views and cache augmented RDBMSs may produce stale data. The former is because the RDBMS may propagate updates to the materialized view asynchronously. The latter is due to write-write race conditions between the RDBMS and the cache [46].

Figure 3.1.c shows the relational design of BG's database. Index structures are constructed on the appropriate attributes to facilitate efficient processing of read actions. For example, with View Profile action referencing a member with a specific userid, say 5, a hash index facilitates efficient retrieval of the member corresponding to this userid. Members table may store images as BLOBs. Alternatives are discussed in Section 8.1.1. Computing either list of friends or pending friends requires a join between Members and Friends table. Section 8.1.5 explores use of materialized views and its alternatives to migrate the work of read actions to write actions for computing simple analytics. We report SoAR of these designs with an industrial strength relational data store named[1] SQL-X.

Figure 3.5.a shows the JSON design of BG's database tailored for use with MongoDB. For each member M_i, this design maintains three different arrays:

- pendingFriends maintains the id of members who have extended a friend invitation to M_i.
- confirmedFriends maintains the id of members who are friends with M_i.
- wallResourceIds maintains the id of resources (e.g., images) posted on M_i's profile.

One may store profile and thumbnail image of each member either in the file system, MongoDB's GridFS, or as an array of bytes. Figure 3.5 shows the last two choices. When images are stored in the GridFS, the imageid and thumbnailid store the profileimageid and thumbnailimageid as attributes of

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

the Members collection (instead of the array of bytes shown in Figure 3.5.a). Section 8.1.1 discusses these alternatives and shows one design provides a SoAR significantly higher than the other two.

In the next 3 sections, we provide additional details about BG's actions and their implementation using both the relational and JSON representations. We discuss changes to the physical organization of data and their impact on the SoAR of SQL-X and MongoDB. We analyze SoAR of SQL-X with different mixes of actions, see Table 4.1. Post Comment and Delete Comment actions are eliminated because we have no improved designs to offer for this action.

To simplify discussion, we classify BG's actions into those that either read or write data. A read action is one that queries data and retrieves rows without updating them. A write action is one that either inserts, deletes, or updates rows of the RDBMS. Column 2 of Table 4.1 identifies different read and write actions.

All reported SoAR numbers are based on a dedicated hardware platform consisting of six PCs connected using a Gigabit Ethernet switch. Each PC consists of a 64 bit 3.4 GHz Intel Core i7-2600 processor (4 cores with 8 threads) configured with 16 GB of memory, 1.5 TB of storage, and one Gigabit networking card. One node hosts SQL-X at all times. All other nodes are used as BGClients to generate workload for this node. With all reported SoAR values greater than zero, either the disk, all cores, or the networking card of the server hosting a data store become fully utilized. We report on use of two networking cards to eliminate the network as a limiting resource. When SoAR is reported as zero, this means a design failed to satisfy the SLA.

8.1.1 Manage Images Effectively

There is folklore that an RDMBS efficiently handles a large number of small images, while file systems are more efficient for storage and retrieval of large images [97]. With BG, we show physical organization of profile and thumbnail images in a data store impacts its SoAR rating dramatically[2]. For example, if thumbnail images are not stored as a part of the profile structure representing a member then the performance of the system for processing the List Friend (LF) action is degraded significantly. This holds true with both MongoDB and SQL-X. Performance of SQL-X is further enhanced when profile images are stored in the file system. The same does not hold true with MongoDB. Below, we provide experimental results to demonstrate these observations.

The LF action of BG retrieves the thumbnail image and profile information of friends of a member, the attributes of Member table shown in Figure 3.1.c. Figure 8.1 shows the SoAR rating of LF with SQL-X and MongoDB with 100 friends per member. While SQL-X performs a join between two tables (Members and Friends of Figure 3.1.c) to perform this action, MongoDB looks up an array of member identifiers (confirmedFriends of Figure 3.1.b for the referenced Member JSON instance). With SQL-X, we consider thumbnails stored in either the file system or the record representing the member. With MongoDB, we consider thumbnails stored in its Grid File System (GridFS) or as an array of bytes in the JSON-like representation of a member. With both systems, storing the thumbnail image as a part of the member profile enhances SoAR rating of the system from zero to a few hundred. In these experiments, the CPU of the data store becomes 100% utilized. It is interesting to note that, with a single node, the join operation of SQL-X is not necessarily slower than MongoDB's processing of cofirmedFriends array to retrieve documents corresponding to the friends of the member.

The performance of SQL-X for processing View Profile (VP) action of BG is enhanced when profile images are *not* stored in the RDBMS. An alternative is to store them in the file system with

[2]BG can be used to evaluate various physical organization of images such as storing them in an RDBMS, in the file system or other alternative approaches like those implemented within Haystack [17].

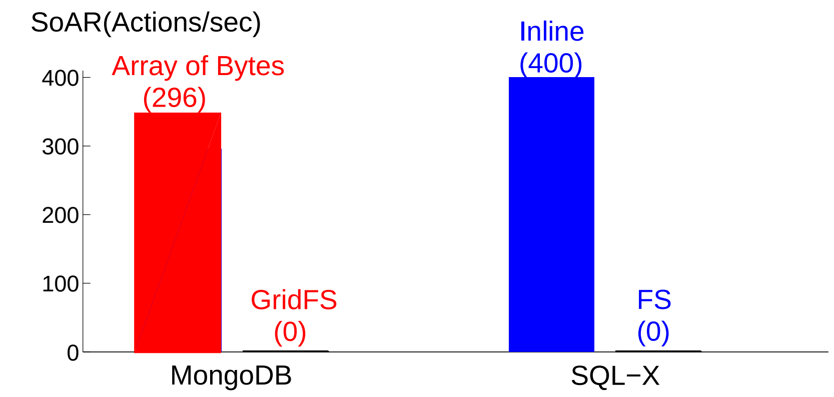


Figure 8.1: SoAR of LF with different organization of 2 KB thumbnail image, M=10K, ϕ=100.

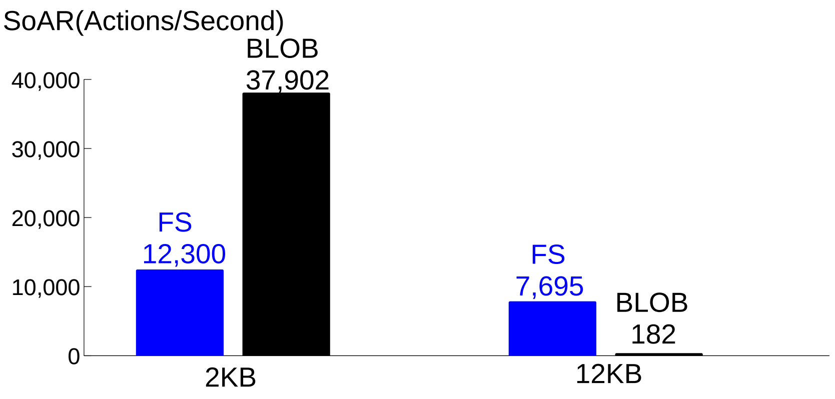


Figure 8.2: SoAR of SQL-X for processing a workload consisting of 100% View Profile action with images stored as either BLOBs or in the FS, M=10K, ϕ=100.

a member record maintaining the name of the file containing the corresponding profile image [97, 16]. Figure 8.2 shows the SoAR of SQL-X with these two alternatives for two different image sizes: 2 KB and 12 KB. (As a comparison, with no images, SoAR of SQL-X is 119,746 for this workload.) A small image size, 2 KB, enables SQL-X to store the image inline with the member record, outperforming the file system by a factor of 3. SQL-X limit on storing images inline is 4 KB BLOB sizes. Beyond this, for example with 12 KB image sizes, its performance diminishes dramatically, enabling the file system to outperform it by more than 40 folds.

MongoDB's GridFS provides effective support for images and its SoAR is comparable to the use of the file system with profile images equal to or smaller than 12 KB. It outperforms the file system by more than a factor of two with very large profile images, e.g., 500 KB. It is worth noting that SQL-X outperforms MongoDB with image sizes smaller than 4 KB by inlining them in profile records. Beyond this limit, MongoDB outperforms SQL-X. Similar to the thumbnail discussions, if profile image sizes are known to be small in advance then one may inline them with MongoDB by representing them as an array of bytes in the Members collection, see Figure 3.5.a. Key considerations include MongoDB's limit of 16 Megabyte for the size of a document and the impact of large

Social Action	One Record per Friendship	Two Records per Friendship
Member 1's number of friends	SELECT count(*) FROM Friends WHERE (inviterID=1 OR inviteeID=1) AND status='C'	SELECT count(*) FROM Friends WHERE inviterID = 1 AND status = 'C'
Member 1's list of friends	SELECT m.* FROM Member m, Friends f WHERE ((f.inviterID=1 and m.MemberID=f.inviteeID) OR (f.inviteeID=1 and m.MemberID=f.inviterID)) and f.status = 'C'	SELECT m.* FROM Member m, Friends f WHERE f.inviteeID=1 and f.status='C' and m.MemberID=f.inviterID
Member 1 invites Member 2	INSERT INTO Friends values (1, 2, 'P')	
Member 2 accepts Member 1's invitation	UPDATE Friends SET status = 'C' WHERE inviterID=1 and inviteeID=2	1. UPDATE Friends SET status = 'C' WHERE inviterID=1 and inviteeID=2 2. INSERT into Friends (inviteeID, inviterID, status) values (1, 2, 'C')
Member 2 rejects Member 1's Invitation	DELETE FROM Friends WHERE inviterID=1 and inviteeID=2 and status='P'	
Member 1 thaws friendship with Member 2	DELETE FROM Friends WHERE (inviterID=1 and inviteeID=2) OR (inviterID=2 and inviteeID=1) and status='C'	

Table 8.1: One record and two record representation of a friendship with one table, Friends table of Figure3.1.c.

documents on actions that do not require the retrieval of the profile image. For example, the List Friend (LF) action does not require the profile image. MongoDB provides an interface to remove some attribute values of a document while constructing a query. For example, one may query the Members collection for a document with userid 1 and not retrieve the profile image of the qualifying document by issuing the following expression: db.member.find({"userid":1,"profileimage":false}).

8.1.2 Friendship

The concept of friendship between two members is central to a social networking site. The first column of Table 8.1 shows most of BG actions that exercise this concept. This section evaluates the alternative design of data with both a relational and a JSON representation. An important consideration is how to represent the thumbnail image of each member displayed when listed as a friend of another member. This was discussed in Section 8.1.1. Hence, all SoARs presented in this section use a BG database configured with no images.

Social Action	One Record per Friendship	Two Records per Friendships

Member 1's number of friends	SELECT count(*)
	FROM Frds
	WHERE frdID=1 OR frdID2=1
		SELECT count(*)
		FROM Frds
		WHERE frdID = 1

Member 1's list of friends	SELECT m.* FROM Member m, Frds f
	WHERE ((f.frdID=1 and
	m.MemberID=f.frdID2) OR
	(f.frdID2=1 and
	m.MemberID=f.inviterID))
		SELECT m.*
		FROM Member m, Frds f
		WHERE f.frdID1=1 and
		m.MemberID=f.frdID2

Member 1 invites Member 2	INSERT INTO PdgFrds values (1, 2)	

Member 2 accepts Member 1's invitation	1. DELETE FROM PdgFrds WHERE
	inviterID=1 and inviteeID=2
	2. INSERT into Frds (frdID1, frdID2)
	values (1, 2)
		1. DELETE FROM PdgFrds WHERE
		inviterID=1 and inviteeID=2
		2. INSERT into Frds (frdID1, frdID2)
		values {(1, 2), (2,1)}

Member 2 Rejects Member 1's Invitation	DELETE FROM PdgFrds
	WHERE inviterID=1 and inviteeID=2	

Member 1 thaws friendship with Member 2	DELETE FROM Frds WHERE (frdID1=1 and frdID2=2) OR (frdID1=2 and frdID2=1)	

Table 8.2: One record and two record representation of a friendship with two tables, Frds and PdgFrds.

8.1.3 Relational Design: A Tale of One or Two

With a relational design, one may represent pending and confirmed friendships as either one or two tables. With each alternative, a friendship might be represented as either one or two rows. We elaborate on these designs below. Subsequently, we establish their SoAR rating. Obtained results show that a two table design is superior to a one table design.

Figure 3.1.c shows a one table design that employs an attribute named "status" to differentiate between pending and confirmed friendships: A 'C' value denotes a confirmed friendship while a 'P' value denotes a pending friendship. The second column of Table 8.1 shows the SQL commands issued to implement the alternative BG actions with this design. Note the use of disjunctive ("OR") predicates in the qualification list of the SQL queries. A designer may simplify these queries and eliminate their use of disjuncts by representing a friendship with two records. The resulting queries are shown in the third column of Table 8.1. The design changes the implementation of accept friendship (fourth row of Table 8.1) into a two SQL statement transaction. In our implementation, all transactions are implemented as stored procedures in SQL-X.

An alternative to the one table design is to employ two different tables and separate pending friend invitations from confirmed invitations, see Table 8.2. This eliminates the "status" attribute

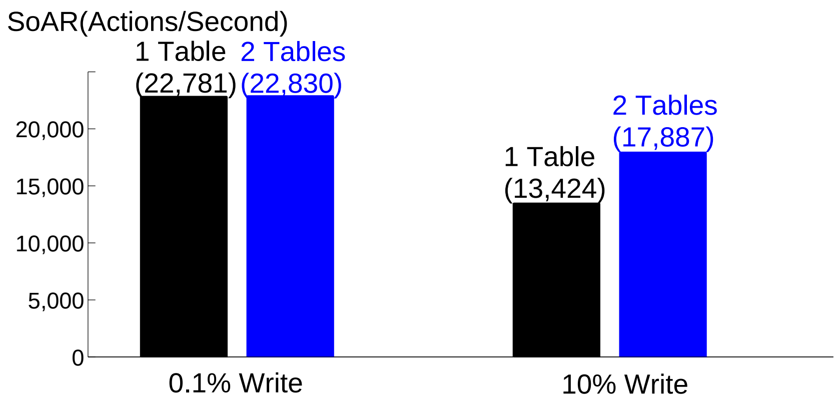


Figure 8.3: SoAR of SQL-X with either one or two tables for pending and confirmed friendships with two workloads, M=10K and ϕ=100. Each friendship is represented as two records.

used with the one table design. However, the data designer is still faced with the dilemma to represent a friendship either as one row or two rows in the table corresponding to the confirmed friends. The second and third row of Table 8.2 shows the SQL commands with these two possibilities. A key difference is that SQL queries are simpler with the two record design.

When comparing the alternative designs, the two record design requires more storage space than the one record design. However, its resulting SQL queries are simpler to author and reason about. With one user issuing requests (single threaded BG), the larger number of records does not impact the service time of issued queries and update commands because index structures facilitate retrieval and manipulation of the relevant records. In a multi-user setting with a mix of read and write actions, see Table 4.1, the two table design outperforms the one table design when the frequency of write action is high enough to result in conflicts. Figure 8.3 shows SoAR of these two alternatives with each friendship represented as two records. Observed SoAR with a mix of very low (0.1%) write actions is almost identical for the two designs due to the use of index structures and a low conflict rate. With a mix of high (10%) write actions, the two table design outperforms the one table design by more than 30%. We speculate this is due to ACID property of transactions slowing down the one table design as it is used concurrently to process both pending and confirmed friendship transactions. The two table design reduces this contention among concurrently executing actions. For example, the query to compute the number of pending friend invitations for a member no longer competes for the same data as a transaction that thaws friendship between two members.

8.1.4 MongoDB: List Friends

With MongoDB, BG's List Friend (LF) action is most interesting because it must retrieve the documents pertaining to the friends of a referenced member. These can be retrieved either one document at a time or all documents at once. With the former, LF is implemented by issuing a query to retrieve the basic profile information for each confirmed friend. With the latter, the entire list is used with the $in operator to construct the query issued to MongoDB. This operator selects all the documents whose identifiers match the values provided in the list. With an under utilized system (a few BG threads), the second approach provides a response that is approximately 1.5 times faster than the first. This is because the first approach incurs the overhead of issuing multiple queries across the network for each document. The SoAR of these two alternatives is almost identical because the CPU of the server hosting MongoDB becomes 100% utilized.

MongoDB supports a host of write concerns, see [80] for details. We investigate two, termed *normal* and *safe* in MongoDB's documentation. Both are implemented by MongoDB's java client. The normal write concern returns the control once the write is issued to the driver of the client. The safe write concern returns control once it receives an acknowledgment from the server. With a low system load (BG with one thread), the normal write concern improves the average response time of MongoDB by 13%. It does not, however, improve the processing capability of the MongoDB server and has no impact on its SoAR when compared with the safe write concern. Moreover, in our experiments, it produced a very low ($< 0.1\%$) amount of unpredictable reads.

8.1.5 Migrate Work of Reads to Writes

Due to a high read to write ratio of social networking sites, one may enhance the average service time of the system by migrating the workload of reads to writes. With RDBMSs, one way to realize this is by using materialized views, MVs. Section 8.1.6 discusses this approach and shows that it slows down write actions so dramatically that it is difficult to argue they are interactive. It presents an alternative named *Manual* that does not suffer this limitation. However, Manual requires additional software and incurs the overhead of a development life cycle (design, implementation, debugging and testing).

8.1.6 Read Mostly Aggregates as Attributes

Social networking sites present their members with individualized "small analytics" [104], aggregate information such as count of friends. BG models these using its View Profile (VP) action that provides each member with her count of resources, friends, and pending friend invitations. One may implement these in two ways: 1) Compute the aggregates each time the VP action is invoked, 2) Store the value of aggregates, look them up to process VP, and maintain them up to date in the presence of write actions that impact their value. An example SQL query that implements the former, termed *Basic*, is illustrated in the first row of Table 8.1. The latter migrates the workload of read actions to write actions. It is appropriate when write actions are infrequent, Below, we present two alternatives to implement the second approach.

One may use Materialized Views (MVs) of SQL-X to store the value of BG's simple analytics and require the RDBMS to maintain their value up to date. This was implemented as follows. First, we define one MV for each aggregate of the VP action. The resulting 3 views have two columns: userid and the corresponding aggregate attribute value. Next, we author an MV that joins these three views with the original Member table (using the userid attribute value), implementing a table that consists of each member's attributes along with 3 additional attribute values representing each aggregate for that member. This table is queried by the VP action to look up the value of its simple analytic instead of computing it.

One may configure SQL-X to refresh MVs either synchronously or asynchronously in the presence of updates. The asynchronous refresh is in the order of hours, causing the MV to contain stale data. BG quantifies these as *unpredictable* reads. Below, we discuss this in combination with the observed SoAR.

With no profile image and a read workload that invokes the VP action only, the authored MV improves SoAR of SQL-X by more than factor of 6 from 19,020 to 119,746 actions per second. With a workload the performs infrequent (0.1%) writes, asynchronous mode of processing updates

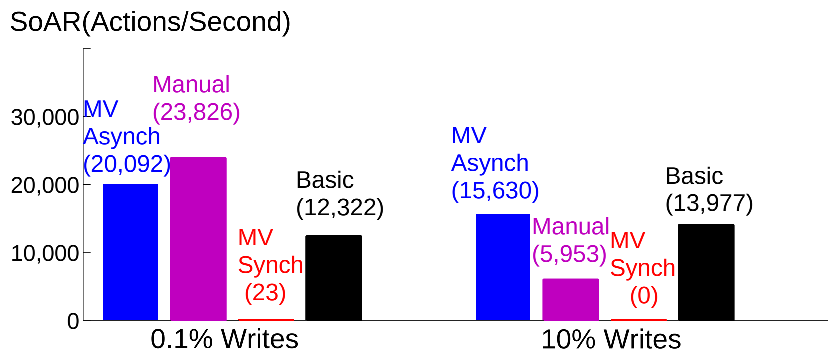


Figure 8.4: SoAR with *Basic* database design of Figure3.1.c, materialized views (*MV*) for aggregates as attributes with both synchronous and asynchronous mode of refresh, and developer maintained (*Manual*) aggregates as attributes with no images.

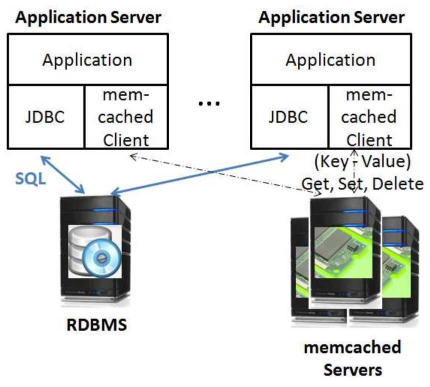
enables MVs to enhance SoAR of SQL-X by almost a factor of two, see Figure 8.4. However, this causes 31% of reads actions to observe unpredictable (stale) data. The amount of unpredictable data increases to 72% with a high frequency (10%) of write actions, enhancing SoAR of SQL-X by a modest 11%.

The synchronous refresh mode of MVs eliminates unpredictable data. However, as shown in Figure 8.4, it degrades SoAR of SQL-X significantly. This is because it slows down write actions dramatically. As an example, the service time of the Accept Friend Request write action is slowed down from 1.7 millisecond to[3] 1.94 seconds with an under-utilized system, i.e., one BG thread. These service times are not interactive, rendering MVs inappropriate for BG's workload.

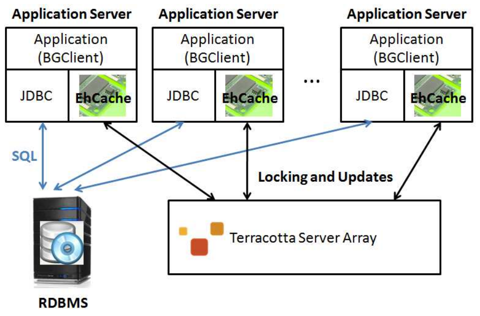
An alternative to MVs, named *Manual*, is for a software developer to implement aggregates as attributes by extending the Member table with 3 additional columns, one for each aggregate. When a member registers a profile, these attribute values are initialized to zero. The developer authors additional software (either in the application software or in the RDBMS in the form of stored procedures and triggers) for the write actions that impact these attribute values to update them by either incrementing or decrementing their values with one. For example, the developer extends a write action that invites Member 1 to be friends with Member 2 to increment the number of pending friends for Member 1 by one as a part of transaction that updates the Friends table, see Section 8.1.2.

Manual speeds up the VP action by transforming 4 SQL queries into one. The four queries include retrieval of the referenced member's profile attribute values, count of friends, count of pending friend invitations, and count of resources. In our experiments, Manual enhanced SoAR of SQL-X for processing the VP action by the same amount as MVs. When write actions are infrequent (0.1%), Manual enhances SoAR of SQL-X by almost a factor of two and outperforms MVs, see Figure 8.4. With frequent (10%) write actions, Manual continues to outperform MV. However, its SoAR is two times lower than Basic due to the overhead of write actions updating attributes in a transactional manner with ACID properties. Note that response time of write actions remains interactive with Manual, faster than 2 milliseconds with an underutilized system.

[3]A 1,141 fold slow down.



8.5.a) Client-Server (CS)



8.5.b) Shared Address Space (SAS)

Figure 8.5: Alternative cache augmented SQL architectures.

		Basic SQL-X	Basic MongoDB	Boosted SQL-X	Boosted MongoDB	memcached	Ehcache
No Image	0.1% Write	12,322	12,097	33,694	11,434	55,634	271,760
	10% Write	13,976	8,492	28,503	8,222	49,006	286,260
12 KB Profile Image	0.1% Write	305	0	11,820	8,451	11,888	147,845
	10% Write	300	0	10,977	6,385	10,271	144,672

Table 8.3: SoAR of alternative designs for two write workload of Table 7.3 for 10,000 members.

A draw back of Manual is the additional software and its associated software development life cycle (design, implementation, testing and debugging, maintenance). Its key advantages include interactive response times for both the read and write actions with no unpredictable reads.

8.1.7 Cache Augmented Database Management Systems, CADBMS

With both MongoDB and SQL-X, a developer may avoid issuing queries to the data store by caching its output, *value*, given its unique input, *key*. This is the main motivation for middle tier caches [62, 27, 118, 38, 36, 68, 5, 6, 91, 56]. This section focuses on a specific subclass that employs in-memory Key-Value Stores (KVS) with a simple put, get, delete interface. Its use case is as follows. The developer modifies each read action to start by converting its input to a key. Next, it looks up the KVS for a value. If the KVS returns a value then the value is produced as the output of the action without executing the main body of the read action which issues data store queries. Otherwise, the body of the read action executes, issues data store queries to compute a value (i.e., output of the read action), stores the resulting key-value pair in the KVS for future use, and returns the output to BG.

The developer must modify each write action to invalidate key-value pairs that are impacted by its insert, delete, update command to the data store. For example, the write action that enables Member 1 to accept Member 2's friendship request must invalidate 5 key-value pairs. These correspond to Member 1's profile, list of friends and list of pending friends, and Member 2's profile and list of friends.

		Basic SQL-X	Basic MongoDB	Boosted SQL-X	Boosted MongoDB
No Image	0.1% Write	15,593	11,715	22,512	11,312
	10% Write	3,477	8,541	7,388	8,913
12 KB Profile Image	0.1% Write	201	0	5,487	8,137
	10% Write	198	0	3,509	6,574

Table 8.4: SoAR of alternative designs for two write workload of Table 7.3 for 100,000 members.

The maximum number of unique key-value pairs is a function of the number of members and read actions. With a database of 10,000 members, the View Profile action of BG may populate the KVS with 10,000 unique key-value pairs. With all six read actions of BG, see Table 4.1, the KVS may consist of a maximum of 60,000 unique key-value pairs. The KVS may consist of fewer key-value pairs because BG may not reference some members due to our use of the Zipfian distribution [13] of access to pick userids.

There are two categories of in-memory KVSs: Client-Server (CS) and Shared Address Space (SAS), see Figure 8.5. With CS, the application server communicates with the cache via message passing. A popular CS KVS is memcached [77]. With SAS, the KVS runs in the address space of the application. Examples include Terracotta's Ehcache [108] and JBoss Cache [22]. SAS KVSs implement the concept of a transaction to atomically update all replicas of a key-value in different application instances. Both CS and SAS architectures may support replication of key-value pairs and implement consistent hashing to enhance availability of data and implement elasticity. A discussion of these topics is a digression from our focus. Instead, we focus on the performance of a single cache instance. With memcached, the cache server is a process hosted on a different server than the one hosting the data store. With Ehcache, the cache instance executes in the address space of the BGClient.

In the following, we focus on the impact of the KVS with a very low (0.1%) and a high (10%) frequency of writes. With these workloads, both MongoDB and SQL-X provide comparable SoARs as either the CPU or network bandwidth of the server hosting the KVS becomes 100% utilized. Hence, without loss of generality, we present SoARs observed with SQL-X using either memcached or Ehcache.

Table 8.3 presents SoAR of the alternative designs when the database is configured with either no images or 12 KB profile image sizes with two different mixes of workloads. These results show Ehcache provides the highest SoAR, outperforming memcached by more than a factor of 13 (5) with images (no images). This is because it runs in the same address space as the BGClient, avoiding the overhead of transmitting key-value pairs across the network and deserializing them. In these experiments, the four core CPU of the server hosting BGClient (and the Ehcache) becomes 100% utilized, dictating the overall system performance. (This bottleneck explains why there is no difference between SQL-X and MongoDB once extended with Ehcache.) It is interesting to note that the SoAR of Ehcache with 12 KB images is almost twice lower than that with no images. This is due to network transmission of images for invalidated key-value pairs, increasing network utilization from 30% to 88%.

With memcached, the four core CPU of its server becomes 100% utilized when there are no images, dictating its SoAR rating. With 12 KB profile images, the network bandwidth becomes

100% utilized dictating SoAR of memcached. In these experiments, memcached could produce key-value pairs at a rate of 2 Gbps as its server was configured with two Gbps networking cards.

8.1.8 A Comparison of Alternative Designs

In addition to presenting SoAR of memcached and Ehcache, Table 8.3 shows SoAR of the Basic SQL-X and MongoDB data designs when compared with their Boosted alternatives. Boosted incorporates all of the best practices presented in the previous sections except for the use of caches[4]. With both SQL-X and MongoDB, the Basic data design is inferior to the Boosted alternative because it is inefficient and utilizes its 4 core CPU fully.

With Boosted and no images, the CPU of the server hosting the data store becomes 100% utilized, dictating its SoAR. This is true with both SQL-X and MongoDB and the two workloads, 0.1% and 10% frequency of writes. These results suggest SQL-X processes BG's workload more efficiently than MongoDB because its SoAR rating is two folds higher.

With 12 KB profile images, both SQL-X and MongoDB continue to utilize their CPU fully with the Basic data design. With Boosted, the network becomes 100% utilized and dictates their SoAR rating. Table 8.4 compares the SoAR rating for the Basic and Boosted designs of MongoDB and SQL-X for a social graph with 100,000 members ($M = 100,000$). With the Basic design and no images, for both SQL-X and MongoDB the CPU of the node hosting the data store becomes the bottleneck. With images, for MongoDB the CPU continues to be fully utilized but with SQL-X the disk becomes the bottleneck. With the Boosted design and without images, network of the SQL-X server becomes the bottleneck, whereas with MongoDB the CPU of the data store node becomes 100% utilized. With images, for both Boosted SQL-X and Boosted MongoDB the network becomes the bottleneck.

[4]The presented SoAR for memcached and Ehcache use the Boosted data design.

8.2 Use of BG to Study Scalability of a Data Store

In recent years a number of new systems have been designed which provide scalability for simple read/write operations such as the operations in social networks. The design of these systems is optimized for different workloads and is impacted by the following tradeoffs [29] that may impact the scalability of data stores.

- Read vs. write performance: In a social networking application, it is difficult to predict which data will be read or written next. A higher read throughput can be achieved by either moving the work of reads to writes or by using cache augmented architectures [47].

- Performance vs. durability: Writes may either be synchronized to disk before the system returns success to the user or stored in memory and flushed to disk at a later time [29]. The advantage of the latter is enhanced system performance. Its disadvantage is possible data loss (non flushed writes) in the presence of failures. If writes are performed asynchronously, they may produce unpredictable data [112].

- Performance vs. consistency: According to the CAP theorem, distributed systems cannot satisfy consistency and availability in the presence of network partitions. Many of today's data stores utilize weaker consistency techniques such as eventual consistency to synchronize replicas. Replication is used to improve availability, prevent data loss, and enhance performance. Eventual consistency mode avoids high write latency by allowing replicas to be out of synch, resulting in users observing unpredictable data.

- Data model: Flexible data models used in NoSQL solutions simplify upgrading the application to support new entities and enhance scalability of a data store by providing a simple schema and a put/get interface with less overhead.

- Row-based vs. column-based representation: In row-based storage, all of a record's fields are stored contiguously on disk. With column-based storage, different columns can be stored separately on different servers. Row-based storage supports efficient access to an entire record and is ideal if we typically access a few records in their entirety. Column-based storage is more efficient for accessing a subset of the columns from multiple records together.

Rick Cattell in his paper [25] does a survey of more than 20 scalable data stores and their characteristics, and claims that although the performance on a single multicore node is important, a key feature of these systems is their shared nothing horizontal scaling architecture which enables them to complete a large number of simple read/write operations per second. He also points out at the scarcity of benchmarks to evaluate and compare the scalability of these data stores with one another.

BG is a data store agnostic benchmark which can be used to evaluate the scalability claims of various data stores and compare them with each other in a fair manner. BG's shared nothing architecture, see Section 4.3, makes it a perfect choice for evaluating the performance of various scalable data stores with increased capabilities. In this section we use BG to examine the scalability characteristics of MongoDB for the simple operations of Web 2.0 [79] applications such as social networking systems. By simple operations, we refer to reads or writes that access or modify a small amount of data from big data and do not contain complex queries, complex joins or large table scans. A cloud service provider such as a social networking site may start on a small set of servers. As the number of members and their request rates increase, every tier in the stack must scale to support additional load. This is why evaluating the scalability of data stores used in social networks is becoming more and more important. Understanding the scalability behavior of these systems provide data store vendors with insights to address the limitations of their solution. This can be helpful as it allows them to add features and functionality to their products to become more competitive in their market segment and advocates different software and hardware architectures opening several research directions that will benefit the community. In addition, application developers should consider the scalability behavior of various data stores and the factors that impact them for their applications in order to predict the performance of their system in different scenarios and make decisions about how to improve it.

8.3 Scalability

In contrast to traditional RDBMSs such as SQL-X, NoSQL and NewSQL [90] data stores are designed to scale to thousands and millions of users performing read and write actions. BG's SoAR rating is ideal to evaluate the scalability claims of these data stores. One may characterize scalability as a function of the database size, size of a hardware platform hosting the data store, or both. Below, we describe each in turn.

Table 8.5 shows the SoAR of a single node with different database sizes. As we increase the size of the social graph from 100K to 500K, the impact on a workload consisting of either the View Profile or List Friends action is minimal. With 500K, MongoDB utilizes the available 16 GB of memory fully. With an increase to a 1 million (1M) member social graph, the SoAR of View Profile drops several folds due to formation of a transient disk queue and a high ($> 50\%$) CPU utilization. With workloads consisting of a mix of write actions, the disk queue becomes permanent, causing the SoAR of MongoDB to drop to zero. This also holds true with the 500K social graph. Though, the impact is characterized by examining the percentage of actions that satisfy the SLA. For example, with 500K and a Very Low (0.1%) write mix, BG reports 75% of actions observe a response time faster than 100 milliseconds. This percentage drops to 50% with the 1M member social graph.

One may increase the size of a hardware platform in two ways, vertically or horizontally. The term *vertical* scaling refers to increasing the resources of a single node to improve its performance. These resources might be CPU cores, mass storage devices, amount of memory, and the number/capacity of networking cards. Table 8.6 shows the observed SoAR[5] with a 64-bit Dell PC configured with 16GB RAM and an Intel(R) Core(TM) i7-4770 CPU @3.40GHz processor, and either one or two 1Gbps networking cards. With a read only workload that issues either the View Profile or List Friends action and a mixed workload consisting of Very Low (0.1%) Writes, the available network bandwidth is the limiting resource. Thus, by increasing the number of networking cards from one to two, MongoDB scales vertically to double its SoAR with the same SLA. (We discuss the 10% mix of write actions and why its performance does not improve as dramatically below in the context of multiple shards for one node.)

Horizontal scaling refers to distributing both the data and the load of an application across many servers. Its typical hardware platform is based on a *shared-nothing* architecture [106] consisting of many nodes where each node has its own memory, CPU cores, and mass storage devices. One may realize such a hardware platform using commodity off-the-shelf PCs. In this section, we report on

[5]The reported SoAR values in this section are with MongoDB version 2.4.9. Those reported in Section 8.1 are using MongoDB version 2.0.8. These two sections were written at different times and 2.4.9 was not available when conducting the experiments reported in Section 8.1. With MongoDB version 2.0.8 using the Dell PC, the reported SoAR with High (10%) Write would increase from 4,143 to 5,730. With the same workload using MongoDB version 2.4.9 and the ZT PC of Section 8.1, the reported SoAR with the same workload reduces from 6,574 to 4,856. This is comparable to the numbers shown in Table 8.3.

	View Profile	List Friends	Very Low (0.1%) Write	High (10%) Write
$M = 100K$	7,699	295	3,866	3,069
$M = 500K$	7,586	246	0	0
$M = 1M$	1,381	281	0	0

Table 8.5: SoAR Rating for single node MongoDB with three different database sizes: 100k member, 500k member and 1M member social graphs with $\phi = 100$ and $\rho = 100$.

	View Profile	List Friends	Very Low (0.1%) Write	High (10%) Write
1 Network card	7,699	295	3,866	3,069
2 Network cards	15,684	581	7,514	4,143

Table 8.6: Vertical scalability for a single node MongoDB for a fixed social graph consisting of 100k members, $\phi = 100$ and $\rho = 100$.

the horizontal scalability characteristics of MongoDB using a cluster of 12 64-bit Dell PCs with the same aforementioned specifications. See Table 8.5 for SoAR with one node of this cluster.

MongoDB partitions BG's social graph across the nodes of a shared-nothing architecture in order to scale horizontally. Its software architecture consists of three components:

1. A shard is a *mongod* instance that contains a subset of the database. It might be deployed as either a standalone or a replica set. A standalone mongod instance is the primary daemon process for the MongoDB system that processes data requests, manages data format, and performs background management operations. A replica set consists of one *primary* mongod instance and one or more *secondary* mongod instances. These instances are deployed on different nodes of a shared-nothing hardware platform. They enable multiple nodes to have a copy of the same data, thereby ensuring redundancy and facilitating load balancing.

2. A *mongos* routing component processes queries from the application layer, determines the nodes (shards) with the relevant fragment of data, and routes the requests to the corresponding mongod instances to process these operations. A mongos instance returns results to the application directly.

3. A *config server* component stores the cluster metadata. This metadata includes details about which fragment (shard) holds which ranges of documents/chunks of data. mongos instances communicate with the config servers and maintain a cache of the metadata for the sharded cluster. MongoDB supports deployment of either one or three config servers. With three config servers, the metadata across all config servers should be identical. In a production deployment, one config server may act as a central point of failure. Hence, one may deploy exactly three config server instances to improve data availability.

These components might be deployed in one server or across different servers. We elaborate on these in turn.

With one server, one may deploy multiple shards and one mongos instance and partition BG benchmark's social graph across the shards. This is most useful when a node is configured with two or more mass storage devices. In our experiments with one mass storage device and two networking cards, we observed the SoAR of MongoDB to improve with a High (10%) mix of write actions from 4,143 with 1 shard to 6,939 with 3 shards. We attribute this 67% improvement to how MongoDB uses a readers-writer [81] lock per shard (mongod instance). These locks enable concurrent read actions to access the database simultaneously and grant exclusive access to a single write action. With 3 mongod instances, the concurrency of the system is enhanced to improve its SoAR. We observed no improvements in SoAR beyond 3 shards on a single node. Note that in Table 8.5 with 1 shard and the 10% mix of write actions, no resource becomes the bottleneck as threads wait for one another. With 3 shards, the network of the MongoDB server becomes fully utilized.

Figure 8.6.a and 8.6.b show two different deployments of the components of MongoDB. Both deploy each shard as a replica set consisting of one primary and two secondaries assigned to different nodes of a 3 node shared-nothing architecture. These figures show 3 replica sets in different colors. The primary of a replica set is denoted as S_i where i is the identity of the replica set. The corresponding secondaries are denoted as Rs_{i-j} where the value of j is either 1 or 2.

With both architectures, a BGClient thread opens a connection to a mongos instance and issues all it queries to that instance. This mongos instance is co-located with a shard. When the data referenced by the BGClient resides in a different shard, the mongos instance directs the query to the appropriate node for processing and returns the results[6]. Using BG, we observed the configuration of Figure 8.6.a to provide a slightly lower SoAR with a High (10%) mix of write actions. We attribute this to (a) the message passing overhead between config server instances, and (b) a fully utilized network bandwidth of each server configured with 1 Gbps networking card. Hence, we focus on the second configuration for the rest of this section.

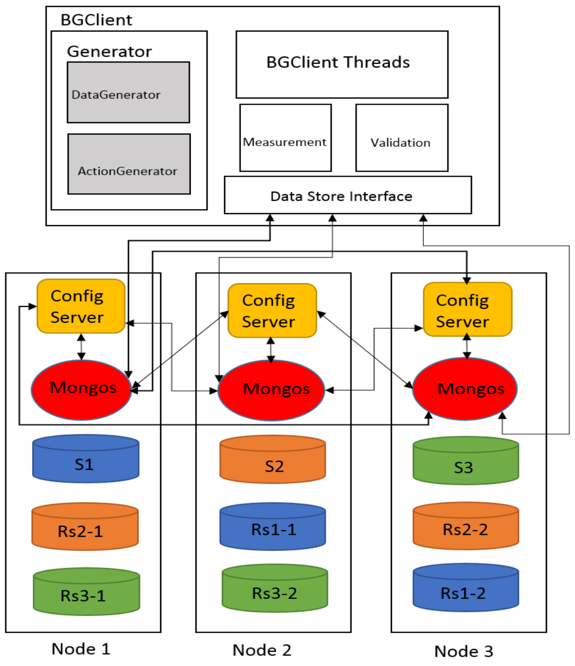
We analyze the impact of varying the number of nodes by analyzing the speedup and scaleup of SoAR. These terms are defined as follows:

- SoAR Speedup: With a fix sized social graph and a workload, this metric quantifies the improvement in SoAR as we increase the number of nodes in the hardware platform. Ideally, with twice as many nodes, SoAR should double. This is termed linear speedup. Speedup emulates a service provider with a fixed database size that becomes popular with an increasing number of simultaneous socialites. It evaluates whether doubling the number of nodes would provide the same SLA with twice the number of socialites, i.e., SoAR.

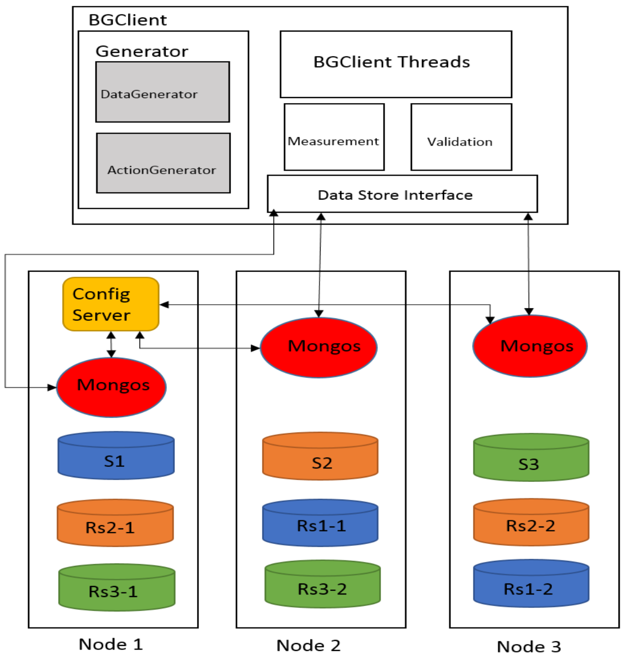
- SoAR Scaleup: To quantify this metric, one increases both the size of the social graph and the number of nodes in the hardware platform proportionally, quantifying SoAR with each configuration. Ideally, the SoAR should either remain a constant or improve. Scaleup emulates a service provider with a fixed number of socialites (members accessing the service at the same time) and an increasing number of members (data set size). It quantifies whether increasing the number of nodes proportional to the size of the social graph would provide the same SLA.

Both metrics use a base hardware platform consisting of a fixed number of nodes that is increased in size. The choice of the hardware platform is arbitrary. Below, we illustrate the flexibility of BG by using a base hardware platform consisting of one node to quantify SoAR speedup and a base hardware platform consisting of 3 nodes for SoAR scaleup.

[6]To minimize the impact of network communication, one may host a mongos on the same node as the BGClient.



8.6.a Three Config Servers per deployment



8.6.b One Config Server per deployment

Figure 8.6: Two alternative deployments for a multi-node MongoDB.

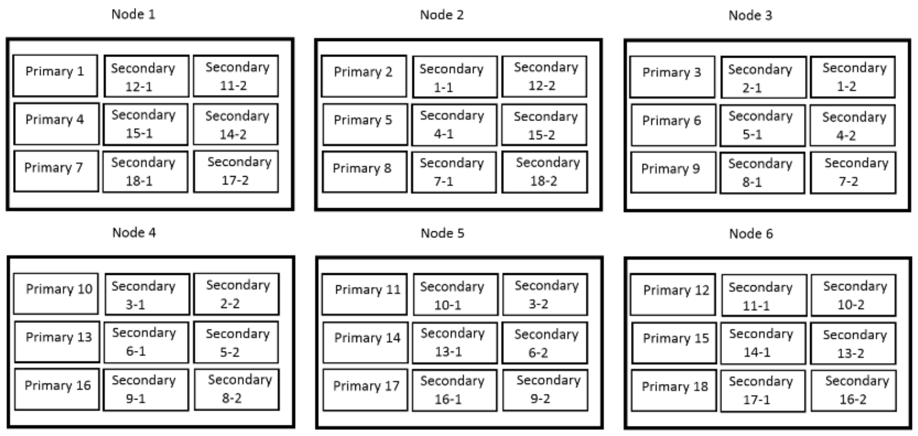


Figure 8.7: Implementing 18 MongoDB replica sets on 6 nodes.

With the experimental results of the next two sections, the mongod instances are deployed as a replica set consisting of one primary and two secondary instances. There are three times as many shards as nodes since the SoAR of a single node is enhanced with 3 shards. For example, with six nodes, there are 18 replica sets as shown in Figure 8.7. We configured mongos instances to use the secondary instances for processing read actions [83]. This results in a more balanced distribution of workload across the nodes as a mongos instance has a choice of two nodes to process a read action. Below, we describe SoAR speedup and scaleup in turn.

SoAR Speedup

The database size used for the base hardware platform has a significant impact on the observed speedup. This is because the amount of data per node decreases as we increase the number of nodes. With a sufficiently high number of nodes, the data assigned to each node fits in the memory of each node, boosting the performance of each node dramatically. This would result in a super-linear speedup where with N nodes, the speedup relative to one node is higher than N. To illustrate, assume a base hardware platform consisting of one node with a 1M social graph and a workload consisting of the View Profile action. Table 8.5 shows a SoAR of 1,381 with one node. With 10 nodes and beyond, the size of the social graph per node drops below 100K members. Table 8.5 shows the SoAR of each node increases to 7,699. This is more than a 5 fold increase in SoAR of a single node, resulting in a super linear speedup.

We decided to sidestep the impact of memory by using a social graph small enough to fit in the memory of our base configuration consisting of one node. This is the 100K social graph of Table 8.5. Next, we increased the size of the base configuration to 3 and 6 nodes, quantifying the SoAR of MongoDB with each configuration. Figure 8.8 shows the speedup as function of the number of nodes with three different workloads consisting of 100% List Friend action, 100% View Profile action, and the High (10%) Write mix of actions. With all workloads, the network bandwidth of the nodes of MongoDB becomes 100% utilized, dictating its SoAR. None of the workloads observe a linear speedup (labeled as ”‘Linear”’ in figure 8.8).

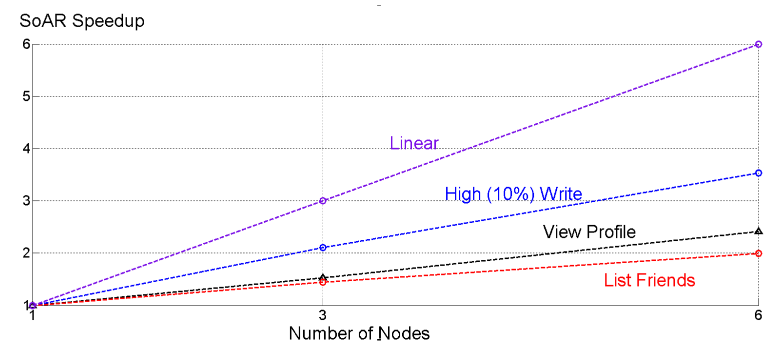


Figure 8.8: MongoDB's speedup as a function of the number of nodes for three workloads for a social graph consisting of 100k members, $\phi = 100$ and $\rho = 100$.

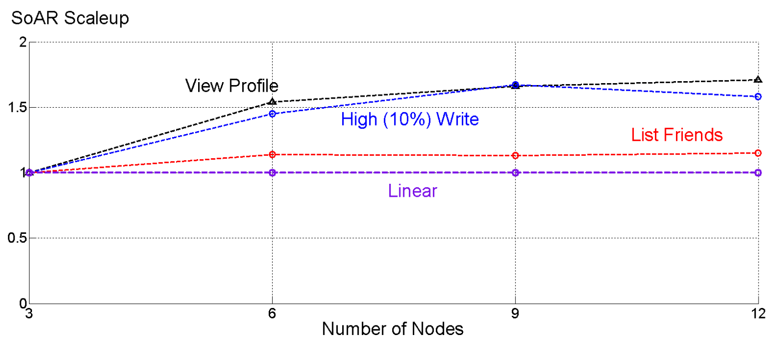


Figure 8.9: MongiDB's scale up as a function of the number of nodes and size of social graph. For each member in the social graph, $\phi = 100$ and $\rho = 100$.

SoAR Scaleup

Figure 8.9 shows the observed SoAR scaleup as we increase the number of nodes of a base configuration consisting of 3 nodes to 6, 9, and 12 nodes. Similar to the discussions of Section 8.3, we used a 100K social graph with the 3 node base configuration. The size of the social graph is increased to 200K with 6 nodes, 300K with 9 nodes, and 400K with 12 nodes. The number of friends and resources per member is fixed at 100. Figure 8.9 shows the scaleup characteristics of MongoDB is better than linear. This is because the network card of the nodes is 100% utilized with the base configuration. As we increase the number of nodes, the number of networking cards increases to enhance the performance better than linear. With all configurations, the networking cards of all nodes were 100% utilized.

8.3.1 Discussion

Today, there is no explicit metric to compare the alternative scalable data stores with one another. And for two systems with improved scalability one cannot decide if a system scales better than another for a given workload. One approach to solve this problem is to develop a scalability metric with a score (*Scalability Score*) that is both data store and workload dependent. With a workload, one may use this score to reason about the scalability characteristics of a data store and compare different data stores with one another. A data store with the highest single node SoAR/Socialites rating and the highest Scalability Score is most desirable.

Scalability Score Model

The Scalability Score is a numeric measurement of horizontal and vertical scalability for a system. Consider an initial load or data set size, D_1, defined by the social network characteristics such as the number of members, number of friends per member and number of resources per member loaded onto an N_1 node data store. The peak throughput of this data store is the SoAR rating for it. Let's denote this rating as S_{D_1,N_1}. This system scales up if $S_{D_1,N_1} \leq S_{D_2,N_2}$. In other words, if we increase the data set size by a factor of k and increase the number of physical nodes hosting the data store by a factor of k, and if the new configuration results in an equal or higher SoAR rating, then that system scales up. We can use this information to decide if a system is more scalable compared to the other by computing the Scalability Score (SS) of the systems for a given workload as follows:

$$SS = \frac{S_{D_2,N_2}}{S_{D_1,N_1}} \tag{8.1}$$

A system with a higher SS has better scaleup characteristics and may scale to a larger number of nodes when compared to another system with a lower SS score. For example, assume single node data stores A and B provide the same SoAR rating on the same hardware platform for a given workload and data set size. Increasing the data set size and the number of nodes hosting the data store by a factor of two for A results in a SoAR 1.2 times higher than before. On the other hand, increasing the data set size and the number of nodes hosting the data store by a factor of two for B results in a SoAR 2.5 times higher than before. For these two systems, B results in a higher Scalability Score and has better scale up characteristics.

The same discussion can be extended and applied for speedup analysis and vertical scalability. For example, assume single node data stores A and B provide the same SoAR for a fixed workload and data set size. When we double the amount of memory on the node hosting the data stores, A's SoAR rating improves by a factor of two but B's SoAR rating improves by a factor of 1.5. In this scenario, the Scalability Score computed for vertical scalability analysis of A is higher than that of B, so A provides better vertical scalability characteristics. In addition, this metric can be used to identify the cost effectiveness of a solution.

8.4 Feed Following

Social networks are highly dynamic. They may grow and evolve quickly with additional members, communication edges and appearance of new social interactions in the underlying social graph. These services are valuable as long as they are popular among their members. This motivates developers to constantly improve the overall user experience by introducing new social networking actions. Examples include “Personalized Recommendations”, “Top and Hot” and “News Feed” as described below.

- Personalized Recommendations: Personalized recommender systems help members identify items of interest. “People You May Know” feature of Facebook suggests people on Facebook that a user is likely to know. These may be chosen based on mutual friends, work and education information, networks a user is a part of, contacts they have imported and many other factors [42]. “Suggested Communities” feature of Google+ recommends communities that a user might want to participate in [119]. These are decided based on various factors such as communities that are related to the ones a user has created or previously joined. YouTube’s “Recommended for You” videos suggests videos a member might like based on her previous viewing history and its related videos [120].

  These recommendations are generally made in two ways. First, by computing the similarity between items (friendships, videos, communities, etc.) and recommending items related to what the user has expressed interest or interacted with. Second, by calculating the similarity between members in the system and recommending items that are interesting for similar members [50].

- Top and Hot: The “Top and Hot” feature usually shows selected exemplary and interesting content that are spreading across the social networking system. These may be breaking news, beautiful photos, unexpected videos, etc. Examples of this feature are YouTube’s Trends displaying latest trending topics and videos on YouTube and a resource for daily insight into what’s happening in web video [121], Google+’s Hot and Recommended feature which helps members find and interact with popular content outside their circles shared within Google+ [52] and Twitter’s Top Tweets which selects and re-tweets some of the most interesting tweets spreading across Twitter [109]. These contents are generated algorithmically derived from common metadata (similar keywords in the title, tags, description, comments and etc.) within a set of items that are currently rising in popularity because a significant number of people view or interact with them.

- Feed Following: In a social networking system such as Facebook, Google+ and Twitter, a member’s “News Feed” captures the most recent events(activities) of her friends or those she is following. In Facebook, news feed displays the latest headlines generated by the events of a user’s friends and the pages she follows [26]. The home activity tab of Twitter [21] is similar to Facebook’s news feed. It contains a list of the recent activity by those members she follows, including their tweets and whom they have chosen to follow recently. Similarly, the Home Stream of Google+ [51] displays posts that have been shared with a member. The displayed content might be shared specifically with the member, shared with the circle the member is in, or shared publicly.

Some of these actions require time consuming computations and, in some cases, use of AI techniques while processing a large amount of data.

We have maintained BG up to date by extending it with additional social networking actions as they become popular. This section focuses on “Feed Following” a popular social action offered in social networking systems. This action seems radically different compared to the others offered in a social networking system and is considered as a challenging big data application [99].

Many social networking systems allow their users to follow the events produced by other members or entities (e.g. pages) in social networks and produce a personalized feed consisting of these events for them. The “follow” relationship may be symmetric such as the friendship relationship in Facebook where if Member A and Member B are friends, Member A (consumer) follows/observes the events produced by Member B (producer), e.g., status messages and comments posted by Member B. Similarly, Member B follows and consumes the events produced by Member A[7]. This relationship is asymmetric with Twitter where Member A may follow the tweets produced by Member B whereas Member B may not follow the tweets produced by Member A [99]. The events displayed in the news feed for Member A may be classified into two categories: First, events produced by Member B such as Member B becoming friends with Member C, Member B posting a comment on a picture uploaded by Member D, and others. Second, events (say a comment) produced on a content (say an image), owned by a Member B. For example, Member D posting a comment on a picture uploaded by Member B.

News Feed is personalized for each member, and there are three key factors that must be considered to produce it: What to show, in what order and when? Social networking systems must ensure that the events available in a member’s feed are relevant, interesting and timely. For this purpose, algorithms are designed [60, 59, 34, 35] that process thousands of potential events to identify those that a member is most likely to engage with by viewing the event or interacting with it by liking, sharing or commenting on it. One approach to realize this is to assign an engagement metric to each event related to a member’s personalized feed displaying the top k most recent events with the highest overall engagement value for the member in some order specified by the member. Computing a single engagement value is challenging as different types of events may have different levels of importance. For example, posted comments may be more important compared to member likes. In addition, members may also be more interested in events by some of the members they follow more than those produced by others. For example, Member A may be more interested in events produced by her top friends. Thus, if one of her non top friend’s event production rate is significantly higher than that of her top friends (say for the last one hour), the calculation of the engagement metrics should ensure Member A’s feed displays her top friends’ events and not only her non top friend’s events.

The term “Social Graph” usually refers to the connections between people who participate in a social networking service. The social graphs pertaining to social networks are massive in scale[8]. In every system, there are a group of members considered as *Social* members who have a high consumption rate and retrieve their news feed frequently. On the other hand, there are a group of members who are considered as *Power* members [57]. These members contribute significantly more events than a typical member (higher production rate). For example, they may produce more friendships/follower relationships, post more comments and upload more photos. Typically the Power members constitute 20% of the members and produce approximately 80% of the events in a social networking system [57, 55]. Thus, the Power members skew the average level of event production by members. As Power members are involved in more friendship events they are associated with a

[7]In Facebook users can customize their feed to not show events produced by a specific user [40], they can also edit their privacy settings to stop the display of events generated by them on a specific user’s feed [41].

[8]Facebook, for example, boasts (as of May 2011) 500 million unique users, each with an average of 130 friends [99].

higher level of social support from the social network. This increases the probability of members interacting with the events produced by them and increases the likelihood of their events appearing on other's news feeds[9].

The performance of feed following actions (producing an event by a producer and consuming news feed by a consumer) depends on the producer fan-in, consumer fan-out,producers' production rates and consumers' consumption rates, see Section 8.4.1. And due to combination of dense connection networks, skewed number of producers followed by a member[10], low-latency requirements (tens of milliseconds) for the feed following actions, extremely high event production and news feed query rates and skewed consumption and production rates for members[11], computing the news feed for members is complicated and deciding to pre-compute or re-compute it for each member requires complicated strategies. For example, studies show the median number of friends for a member in Facebook is 100, the average number of photos uploaded or liked by a member is 300 [101], the average number of comments per post is 9 [65] and the average number of comments on posts in fan pages is 300 [107]. Hence, the average size for the news feed for every member is large. These characteristics argue that re-computing the news feed query every time a member accesses it may not always be efficient, motivating use of materialized views, caches and novel feed computing architectures [99].

The news feed is tolerant for missing content [99]. For example, lets assume Member A follows 100 Power members. Each Power member produces at least 1 event every three minutes. Member A may tolerate not seeing the last event produced by each of the producers in the last three minutes in her feed. This means even though Member A encounters an unpredictable read (see Chapter 6) which can be mapped to 100 unpredictable reads (one for the event produced by each Power producer), the application can tolerate the missing events. Thus computing the percentage of unpredictable reads may not be a good metrics when evaluating alternative implementations of news feed with solutions that result in unpredictable data. A better metric might be the elapsed time from when an event is produced by a producer till the event is available for display by the consumers (inconsistency window), see Chapter 6. When considering different members, one may aggregate this metric as the probability of reading the freshest event (freshness confidence) t units of time after the event was produced. This probability can be added as a new SLA requirement for the applications and used while rating architectures, see Chapter 7. An example SLA requirement can be as follows: 95% ($\alpha = 95\%$) of requests observe a response time equal to or faster than 100 ($\beta = 100$ msecs) milliseconds while at least 80% (freshness confidence) of reads observe the freshest value at most 1 minute ($\Delta = 1$ minute) after the update.

The rest of this section describes the social graph used by BG to model the feed following actions discussed in Section 4.1. The objective of this section is to use BG to explore different paradigms for feed following actions. We will mainly look into two different approaches named

[9]One challenge here is that studies show that a member's friends always have more friends than the member herself and only 10% of members have friends who on average have smaller network than theirs arguing that member's tend to be friends with members who are as active as them or more active in event production (i.e. friendship) [57]. So Non-Power members who don't produce much events are most likely to have friends similar to themselves making the computation of engagement metrics more challenging. And similarly, Power members are most likely friends with other Power members and as event production by members increases, there is more competition for what makes it into news feed and that makes the computation of the engagement metrics even more challenging

[10]On Facebook some consumers follow over 1000 producers; others follow a few.

[11]Some consumers visit the social networking site constantly, triggering requests for their feed query each time, while others visit it once a week or less often.

Push and *Pull* with SQL and NoSQL systems, and use BG to understand their unique characteristics and tradeoffs in terms of throughput and response time.

8.4.1 Social Graph Characteristics and News Feed Analytical Model

Benchmarking graph-oriented applications are increasingly becoming relevant in Big Data applications, such as social networks. These applications can be represented using a *Social Graph*. The term Social Graph is an abstract representation of the relationships between people who participate in a social networking system. It can be illustrated by drawing a graph $G(V, E)$, where members are represented by vertices (V), and their relationships are represented by directed edges (E) drawn between the vertices which determine the fan-out and fan-ins for the vertices. In Facebook an edge may imply a friendship (both friends consume the events produced by one another) or a following relationship (members follow pages and consume the events produced by those pages). With the first the consumer-producer relationship is symmetric. With the second, the relationship is asymmetric. Twitter's follow relationship is another example of the asymmetric relationships.

The news feed for every member only displays the k most recent events related to those a member is following. Ideally, these k are most relevant to the member. The news feed can be modeled as an abstraction that assumes every news feed request for Member A (R_A), retrieves all the recent[12] events produced (C) by those members followed by Member A in the related social graph and then applies other filters to decide the final events displayed in Member A's news feed. Assuming a directed edge from vertex A to vertex B, $E_{A,B}$, indicates that Member A follows Member B, news feed of Member A is defined as:

$$R_A = \bigcup_{\forall E_{A,B} \in G} C_B \tag{8.2}$$

Where C_B is the recent events produced by Member B.

So when Member A retrieves her feed all events produced by each producer that Member$A$ is following is gathered and a subset of relevant events is presented to the member. With feeds, members don't expect to see all events from their followed producers, so a feed retrieval paradigm can use this to drop some events and return feed query results with lower latency. In addition, as shown in Figure 8.10, member behavior in a social networking system can be divided based on four criteria that further impact the development of a feed retrieval architecture. These criteria are as follows (see Table 8.7):

- Consumption rate: which is computed as the number of times a member retrieves her feed in t units of time. A member who accesses her feed infrequently is referred to as a Non-Social member and one who retrieves her feed very frequently is referred to as a Social member.

- Production rate: which is computed as the number of events produced by a member in t units of time i.e., frequency at which friendships are formed and thawed by a member. A member who produces events at a high rate is termed as a Power member and one who does not produce events frequently is termed as a Non-Power member.

- Number of producers followed (fan-out): which is computed by summing the number of member's friends (in a symmetric relationship) and the number of producers such as pages

[12]Recency may have different definitions, i.e. the last 10 events generated by a producer.

Term	Definition
Consumer	Members who retrieve their news feed.
Producer	Members or Pages who share resources with their followers. These resources will be posted on their followers' news feed.
Social	Members with a high consumption rate who access their news feed frequently.
Non-Social	Members with a low consumption rate who do not access their news feed frequently.
Power	Members and Pages with a high production rate who share resources with their followers frequently.
Non-Power	Members and Pages with a low production rate who do not share resources with their followers frequently.
Fan-out	Number of producers followed by a member.
Fan-in	Number of members following a member or a page.
Super-Follower	Members following more than 1,000 producers (members or pages).
Celebrity	Pages with more than 1,000,000 followers.
Active	Members with both a high production and high consumption rate.

Table 8.7: Terms describing members of a social network.

that she is following(in an asymmetric relationship). Members following a large number of other members/pages ($> 1,000$) are referred to as Super-Followers[13].

- Number of followers (fan-in): which is computed by summing the number of member's friends (in a symmetric relationship) and the number of members which are following this member (in an asymmetric relationship). Members followed by a large number of members ($> 1,000,000$) are considered as Celebrities[14].

So for a consumer with a low consumption rate the feed paradigm may predict when the member may retrieve her feed and delay delivering events to her feed for all other times.

8.4.2 Two Feed Following Architectures

In this section we emphasize on two alternative architectures for feed following actions (View News Feed action and Share Resource action of BG). We demonstrate that the structure of the social graph, member activity level (exponent θ in D-Zipfian) and percentage of updates (i.e., Share Resource action) impact the performance results. One can use the results provided in this section to develop a new architecture which targets each of these dimensions for an improved performance. For example an architecture may retrieve the resources shared by a producer with a high production rate at View News Feed query time, while the resources shared by a producer with a low production rate are pushed to members in advance [100].

[13]The average Facebook consumer follows 130 producers [99].

[14]The most popular producer on Twitter as of May 2011 is the singer Lady Gaga, with over 10 million followers [99].

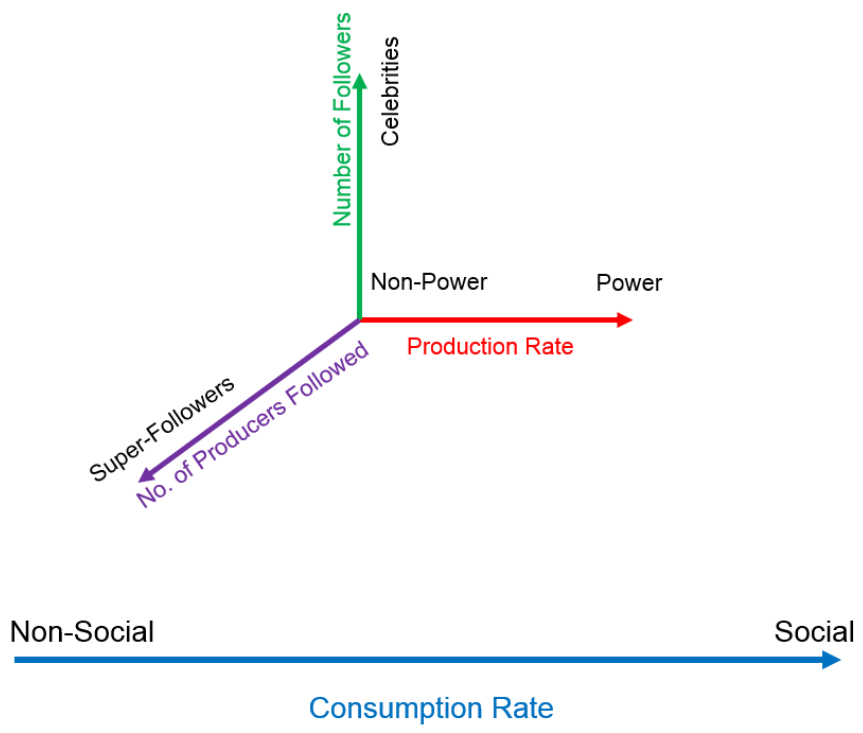


Figure 8.10: Dimensions used for characterizing member behavior in a social networking system.

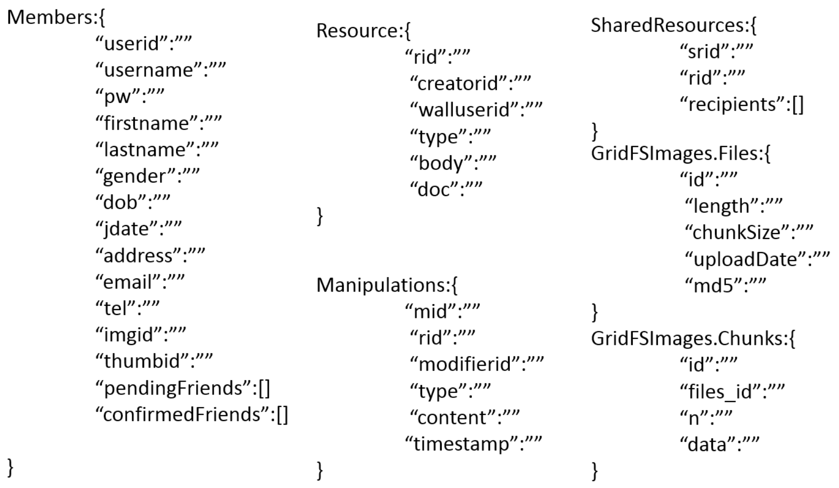
Pull Approach

A trivial implementation of displaying a feed is to re-compute it every time a member invokes a View News Feed (VNF) action. Every time a producer generates an event using the Share Resource (SR) action of BG, its attributes such as the time of creation and list of recipients (with public sharing, this is -1, and for private sharing the list contains the memberids of those members the resource is shared with) are stored into the data store, see Figure 8.11.

When a member retrieves her feed using BG's View News Feed action, first a list of all producers followed by her is queried, next the top k resources shared by them (either publicly or specifically with this member) are retrieved and displayed in the member's news feed (top k may be the k recent events, or the k most relevant events and etc.).

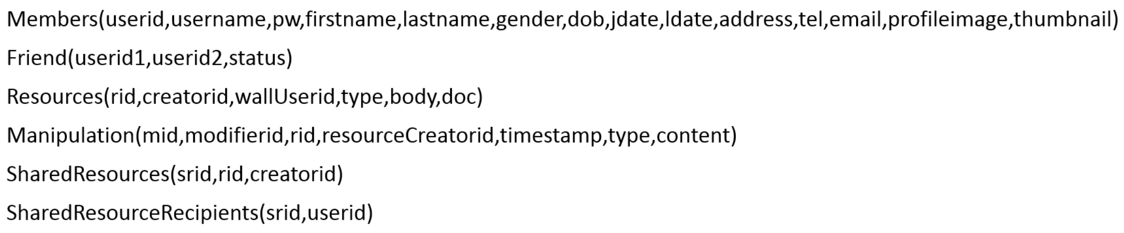
With the pull approach, modifications to friendship relationships incur no additional overhead. For example, assume Member A stops following Member B. With Pull the next time Member A retrieves her news feed, a list of producers followed by her is retrieved. This list will no longer contain Member B. Next all resources shared by these producers, either publicly or specifically with Member A, are queried. As Member B is no longer in the list of producers followed by Member A, the resources shared by Member B will not be displayed in Member A's feed.

In addition, with social networking applications that allow modification to generated events such as shared resources, once a producer updates an event produced previously (e.g. edit her status message), the next time her followers access their feed, the updated version of the event will be retrieved and displayed on their feed.



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

Resource:{
        "rid":""
        "creatorid":""
        "walluserid":""
        "type":""
        "body":""
        "doc":""
}

Manipulations:{
        "mid":""
        "rid":""
        "modifierid":""
        "type":""
        "content":""
        "timestamp":""
}

SharedResources:{
        "srid":""
        "rid":""
        "recipients":[]
}
GridFSImages.Files:{
        "id":""
        "length":""
        "chunkSize":""
        "uploadDate":""
        "md5":""
}
GridFSImages.Chunks:{
        "id":""
        "files_id":""
        "n":""
        "data":""
}
```

8.11.a : MongoDB's JSON-like data model



```
Members(userid,username,pw,firstname,lastname,gender,dob,jdate,ldate,address,tel,email,profileimage,thumbnail)
Friend(userid1,userid2,status)
Resources(rid,creatorid,wallUserid,type,body,doc)
Manipulation(mid,modifierid,rid,resourceCreatorid,timestamp,type,content)
SharedResources(srid,rid,creatorid)
SharedResourceRecipients(srid,userid)
```

8.11.b : SQL-X's relational data model

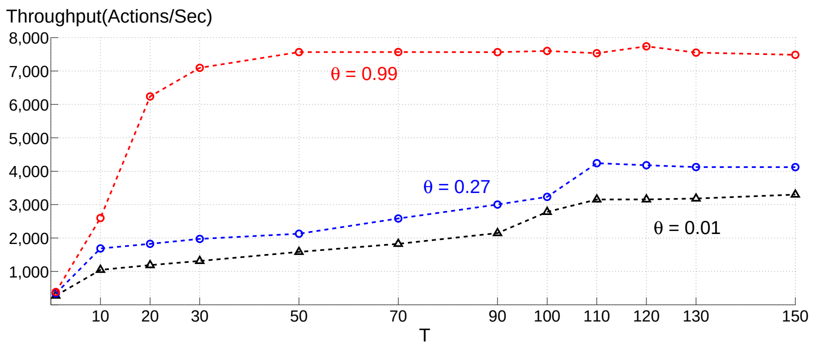
Figure 8.11: BG's logical data model for feed following actions with a pull approach.

With a Pull approach, for a workload consisting of only feed following actions (SR and VNF actions), a higher percentage of SR action, reduces the performance of the data store. This is because higher percentage of SR actions (writes) results in a larger database as the workload executes. This may result in a higher response time for View News Feed actions, as now the related queries are issued on a larger data set size, see Figure 8.12. In addition, the number of members/pages followed by a member impacts the performance of the View News Feed action with the Pull paradigm. For members with a large fan-out (following a large number of members/pages), the VNF action will observe a higher service time as it will retrieve a larger number of shared resources.

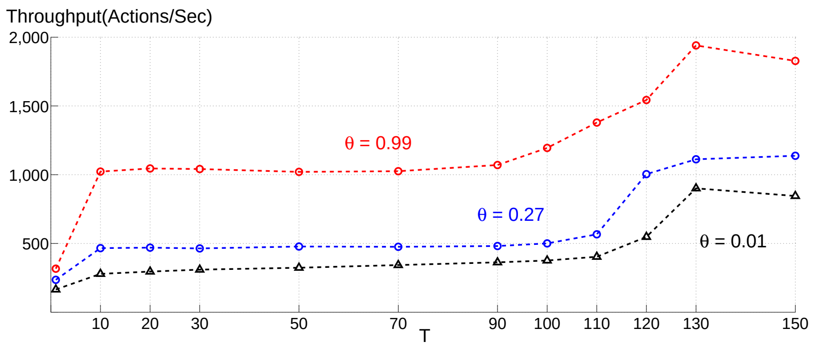
Figure 8.12, shows that the member activity distribution (identified by exponent θ of the D-Zipfian distribution, see Section 4.3.1) also has an impact on the performance of Pull. With a more skewed distribution, the feed for the follower's of Power producers will consist of a larger number of shared resources. As the VNF action retrieves the top k most recent shared resources for a member, sorting a larger number of events for these members becomes more expensive and reduces the performance of the system.

Push Approach

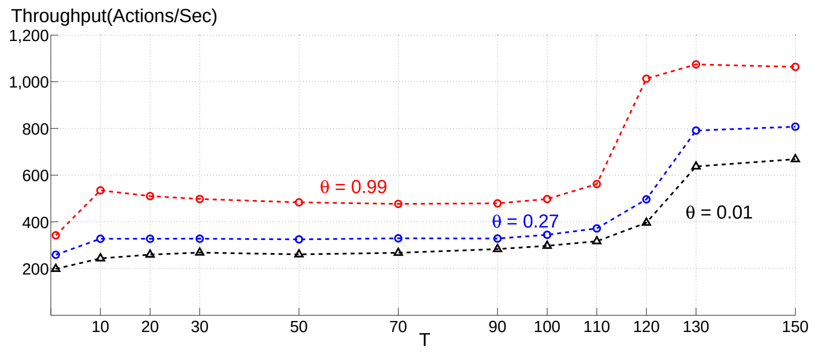
An alternative to the Pull is the Push approach. In this approach the feed for every member is pre-computed and stored in the data store and maintained up to date upon all Share Resource actions. So every time a producer generates an event by invoking BG's Share Resource action, the list of



8.12.a : 1% Share Resource Action, 99% View News Feed Action

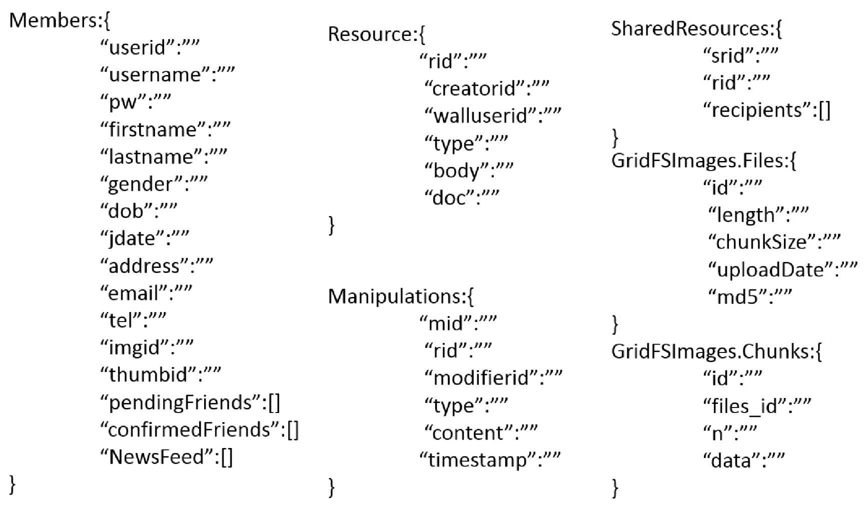


8.12.b : 10% Share Resource Action, 90% View News Feed Action



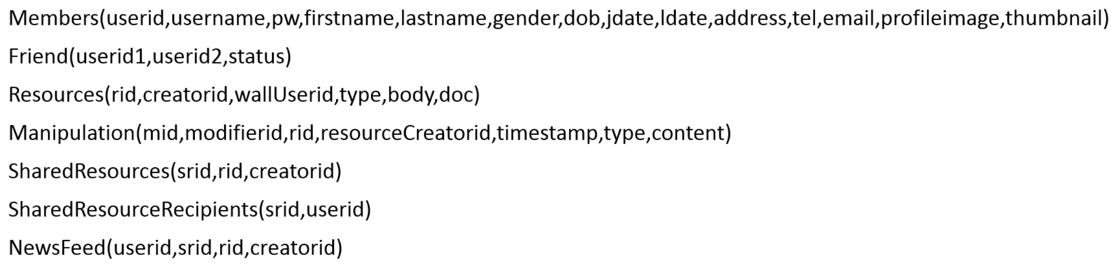
8.12.c : 80% Share Resource Action, 20% View News Feed Action

Figure 8.12: Impact of member activity distribution (θ in D-Zipfian) on MongoDB's performance for three different workloads using a Pull architecture when $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads 1% of the SR actions are issued by pages.



```
Members:{
        "userid":""
        "username":""
        "pw":""
        "firstname":""
        "lastname":""
        "gender":""
        "dob":""
        "jdate":""
        "address":""
        "email":""
        "tel":""
        "imgid":""
        "thumbid":""
        "pendingFriends":[]
        "confirmedFriends":[]
        "NewsFeed":[]
}

Resource:{
        "rid":""
        "creatorid":""
        "walluserid":""
        "type":""
        "body":""
        "doc":""
}

Manipulations:{
        "mid":""
        "rid":""
        "modifierid":""
        "type":""
        "content":""
        "timestamp":""
}

SharedResources:{
        "srid":""
        "rid":""
        "recipients":[]
}
GridFSImages.Files:{
        "id":""
        "length":""
        "chunkSize":""
        "uploadDate":""
        "md5":""
}
GridFSImages.Chunks:{
        "id":""
        "files_id":""
        "n":""
        "data":""
}
```

8.13.a : MongoDB's JSON-like data model



```
Members(userid,username,pw,firstname,lastname,gender,dob,jdate,ldate,address,tel,email,profileimage,thumbnail)
Friend(userid1,userid2,status)
Resources(rid,creatorid,wallUserid,type,body,doc)
Manipulation(mid,modifierid,rid,resourceCreatorid,timestamp,type,content)
SharedResources(srid,rid,creatorid)
SharedResourceRecipients(srid,userid)
NewsFeed(userid,srid,rid,creatorid)
```

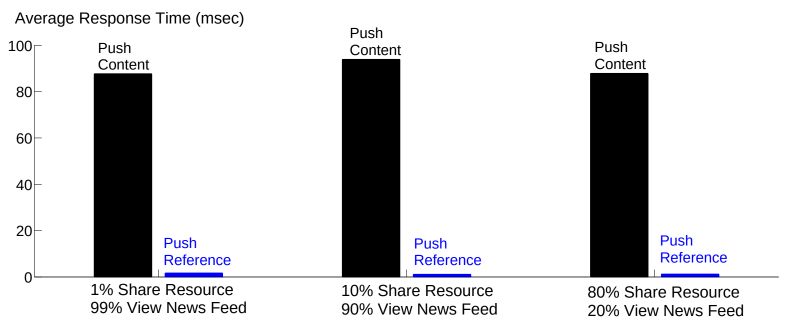
8.13.b : SQL-X's relational data model

Figure 8.13: BG's logical model for feed following actions with a Push approach.

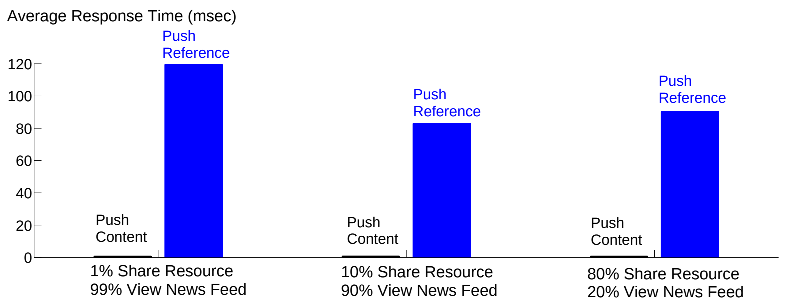
members following that producer is queried, and the shared resource is pushed (added) to these members' feed. Now the feed for a member is always constructed, up to date and available for her upon request without any additional queries. The disadvantage of this approach is obvious when the relationships between consumers and producers change. Now if a consumer decides to stop following a producer, all the events generated by that producer need to be removed from the consumer's feed. On the other hand if the consumer decides to follow a new producer, the events generated by that producer need to be retrieved and merged with the events already present in the member's news feed. For example, if the friendship between Member A and Member B is thawed, all resources shared by B should be removed from A's feed and vice versa. And if Member B and Member C become friends, all resources shared by B should be added to C's feed and vice versa.

The Push approach is more effective for Social members who constantly retrieve their feed by invoking VNF actions. For these members, using a Pull approach which recomputes the member's feed upon every request is not ideal. However, for Non-Social members who do not follow many producers, reconstructing the feed upon every Share Resource action may be wasteful work and reduce the performance of the system.

There are two alternatives for the Push approach (Figure 8.13 shows the data model used for both alternatives with MongoDB and SQL-X). In the first alternative, the resource shared by the producer along with all its attributes is pushed into the consumers' feed (*Push Content*). In the second approach, the reference for the shared resource by the producer is added to the consumers'



8.14.a : Average Response Time for BG's Share Resource (SR) Action



8.14.b : Average Response Time for BG's View News Feed(VNF) Action

Figure 8.14: Comparing the average response time of MongoDB for BG's feed following actions with two Push alternatives: Push Content and Push Reference. $T = 1$, $M = 10,000$, $P = 100$, $\phi = 100$, $\rho = 10$,$\iota = 1,000$, $\varrho = 10$, $\theta = 0.99$.

feed (*Push Reference*). As shown in Figure 8.14 the average response time for generating events (BG's Share Resource action) with the Push Content is higher than the that with Push Reference with MongoDB. On the other hand the average response time for retrieving feed for the Push Content is lower than that of the Push Reference. This is because with Push Reference, upon a View News Feed action a list of references to all shared resources that need to be displayed on the consumer's feed is retrieved. But then the actual content of each of the resources also needs to be queried which increases the average response time for the View News Feed action.

With social networking workloads that allow modifications to the events generated, the implementation that pushes the entire event to the member's feed will perform worse as it will need to update the event in every following consumer's feed. In order to support applications which provide this functionality we will use Push Reference in the rest of this document and for our evaluations and would refer to it as Push for simplicity.

With the Push approach and a workload that has a higher percentage of SR actions, the size of the database as well as the size of the feed for every member increase with time as the workload executes. This increase depends on the exponent θ of D-Zipfian distribution which decides members' activity. For social graphs with very skewed distribution, where some members have both a very high production and consumption rate, *Active members*, consumers following the Active members will have a larger feed size compared to consumers not following the Active members (as the Active members will share more resources and produce more data).

Workload	$\theta = 0.01$	$\theta = 0.27$	$\theta = 0.99$
1% Share Resource, 99% View News Feed	347	209	17
10% Share Resource, 90% View News Feed	762	272	37
80% Share Resource, 20% View News Feed	805	679	50

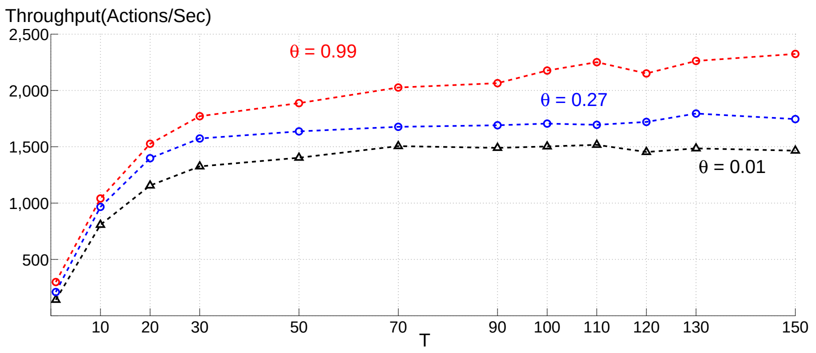
Table 8.8: Impact of skewness in member activity level (exponent θ in D-Zipfian) on the maximum number of events pushed to a member's feed for three workloads and three different values of θ: 0.01, 0.27 and 0.99, where θ=0.01 is a very skewed distribution and θ=0.99 is considered as uniform distribution. $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads 1% of the SR actions are issued by pages.

In BG, each member's activity is characterized by the D-Zipfian distribution's exponent, θ. The member with the highest probability of reference has both the highest production and consumption rate. With a single BG client, BG constructs friendships such that Active members are friends with other Active members and Non-Active members are mostly friends with other Non-Active members, see Chapter 5. Table 8.8 shows the impact of D-Zipfian's exponent θ on the number of shared resources added to the feed for the most Active member (this member is more likely to be friends with other active members because of the way BG constructs friendships, so is one of the members with the largest number of shared resources in her feed).

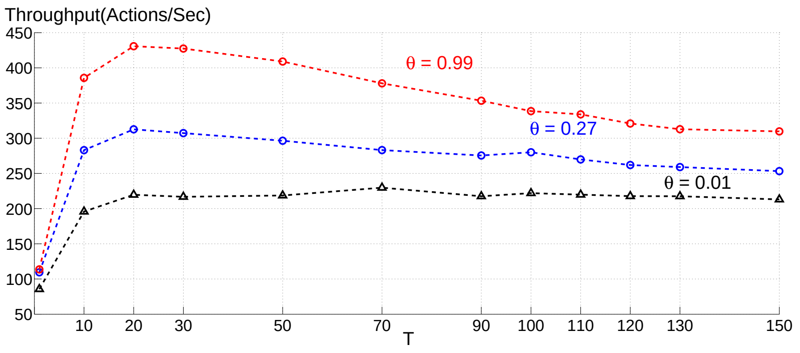
As shown in Table 8.8 for all workloads, the more skewed the distribution is the higher is the maximum number of shared resources pushed into the most Active member's feed. This is because with a fixed number of members (i.e. $M = 100$) and a fixed number of Share Resource actions (i.e. 100), with a skewed distribution, Active members will generate more events (assume one Active member issues 80 Share Resource actions) compared to the Non-Active members (20 Share Resource actions will be issued by the remaining Non-Active members). So those members following the Active members will have larger number of shared resources in their feed (maximum of 80 events). But with a uniform (less skewed) distribution, and the same number of members and Share Resource actions, the Share Resource actions will be divided evenly across all the members and the number of events pushed into each member's feed will be fewer than the skewed distribution scenario. The figure also shows that with a higher percentage of Share Resource actions the number of shared resources pushed into the most Active member's feed will increase. This is because a higher percentage of Share Resource action results in higher number of total shared resources.

With BG and MongoDB, there are two alternative implementations for the Push client referred to as *Push* and *Sorted Insert Push*. With the former, every time a Share Resource action is invoked by a producer, the shared event is pushed to the news feed for all the followers of that producer. When a View News Feed action is invoked by a member, the entire news feed for the member is retrieved by the application and the top k is computed and displayed to the member. With the latter, every time a Share Resource action is invoked by a producer, the shared resource is pushed into the news feed for all her followers in a sorted manner (depending on the ordering required by the View News Feed action). Now when a member tries to view her news feed, the top k shared resources from her feed are retrieved and displayed in her news feed. When compared with Push, Sorted Insert Push results in a higher average response time for BG's Share Resource action and a lower average response time for BG's View News Feed action.

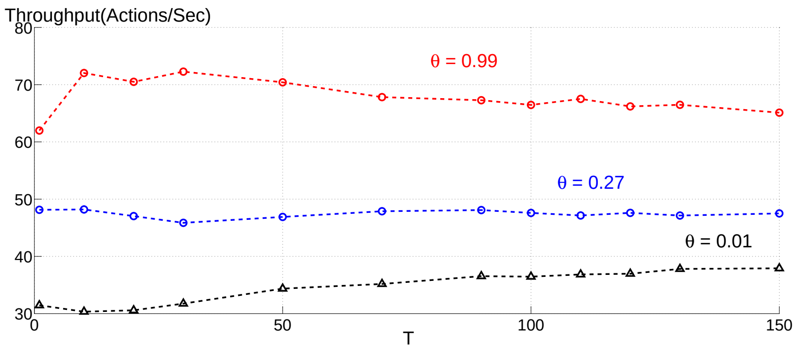
The skewness of member activity distribution has an impact on the performance of a data store for various workloads for both the Push alternatives. Figure 8.15 and 8.16 show the performance of MongoDB as a function of number of simultaneous threads (T) issuing requests against MongoDB,



8.15.a : 1% Share Resource Action, 99% View News Feed Action



8.15.b : 10% Share Resource Action, 90% View News Feed Action



8.15.c : 80% Share Resource Action, 20% View News Feed Action

Figure 8.15: Impact of member activity distribution (exponent θ in D-Zipfian) on MongoDB's performance for three different workloads using a Push architecture when $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads 1% of the SR actions are issued by pages.

BG Social Actions	Type	Mixed Very Low (0.2%) Write	Mixed High (11%) Write
View Profile	Read	0%	0%
List Friends	Read	0%	0%
View Friend Requests	Read	0%	0%
Invite Friend	Write	0.04%	4%
Accept Friend Request	Write	0.02%	2%
Reject Friend Request	Write	0.02%	2%
Thaw Friendship	Write	0.02%	2%
View Top-K Resources	Read	0%	0%
View Comments on a Resource	Read	0%	0%
Post Comment on a Resource	Write	0%	0%
Delete Comment from a Resource	Write	0%	0%
Share Resource	Write	0.1%	1%
View News Feed	Read	99.8%	89%

Table 8.9: Two mixes of social networking actions including feed following actions.

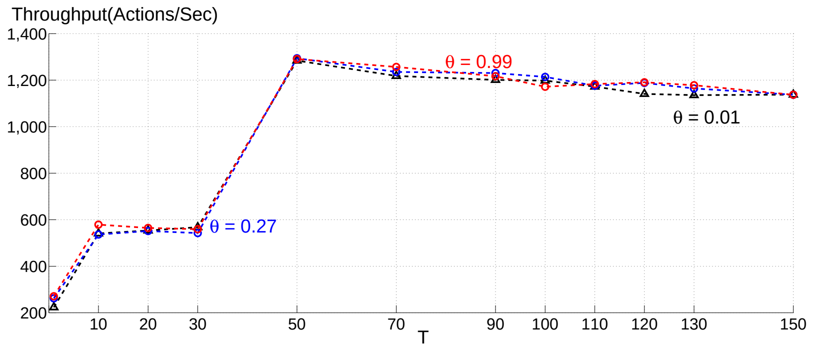
with different member activity distributions, for three different workloads. For Push, with all three workloads, the increase in the skewness of the distribution, decreases the performance of the system and the percentage of reduction increases as the percentage of Share Resource action increases. This is because with a more skewed distribution constructing the feed for the consumers' following Active producers becomes more expensive.

For Sorted Insert Push, with lower than 10% write actions, the skewness of member activity distribution does not impact the performance of the system. This is because with lower percentage of updates, the news feed for members, including those following Active members, consists of a smaller number of items and the insertion sort is performed quickly with minimal overhead. But as we increase the percentage of writes to 80%, the observed performance decreases with a more skewed member activity distribution. This is because the news feed for members following Active members contains a larger number of shared resources which slows down the insertion sort performed while Share Resource action is invoked.

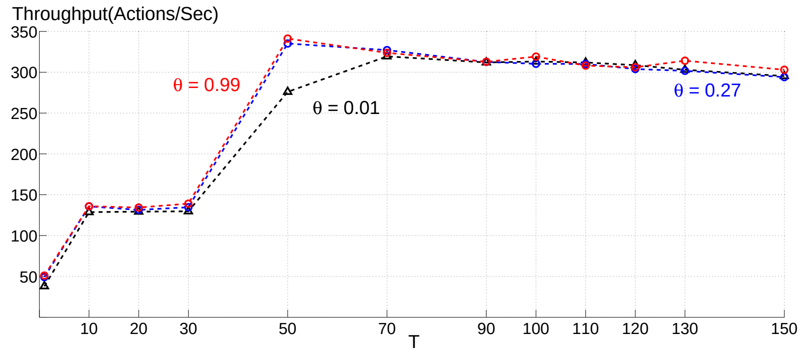
Evaluation

We now describe the experimental results we gathered by comparing the Pull, Push and Sorted Insert Push approaches described in Section 8.4.2. We examine the behavior of these models by modifying the workload characteristics as well as the member activity distribution (exponent θ for D-Zipfian). The metric we focus on is the throughput (actions/second) instead of the SoAR for the system. This is because with workloads involving feed following actions the size of the database and the SoAR of the system change depending on the duration of the rating experiment, see Section 7.3. For this kind of workloads understanding the trends for the data store for a given workload seems more relevant and appropriate. All experiments are executed using one BGClient and is ensured that the benchmarking framework does not become the bottleneck.

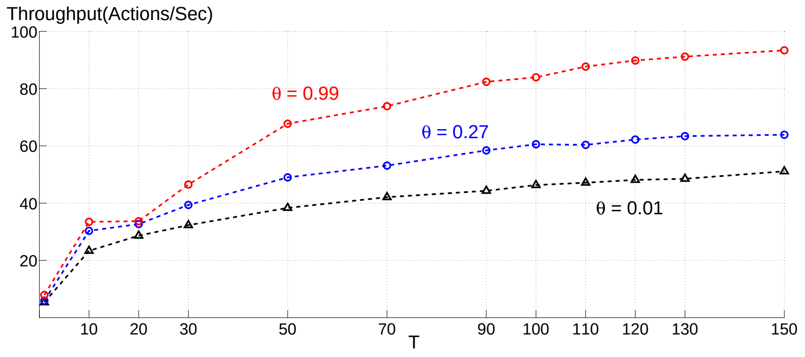
In the first experiment we studied the performance of Pull versus Push and Sorted Insert Push with MongoDB for two workloads consisting of both feed following actions and other BG actions such as actions that modify friendship relationships: IF, AFR, RFR, TF, see Table 8.9. As mentioned



8.15.a : 1% Share Resource Action, 99% View News Feed Action

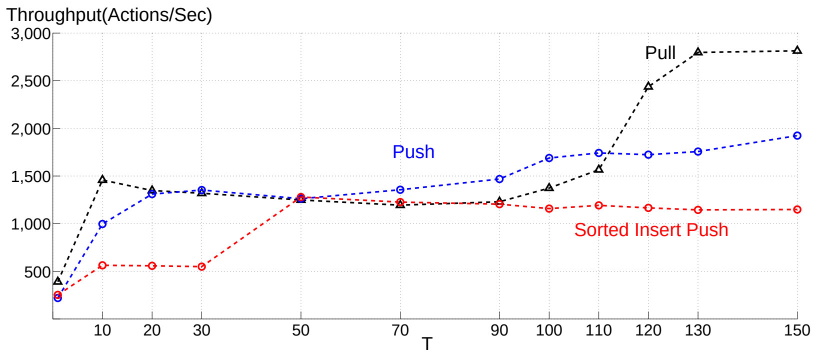


8.15.b : 10% Share Resource Action, 90% View News Feed Action

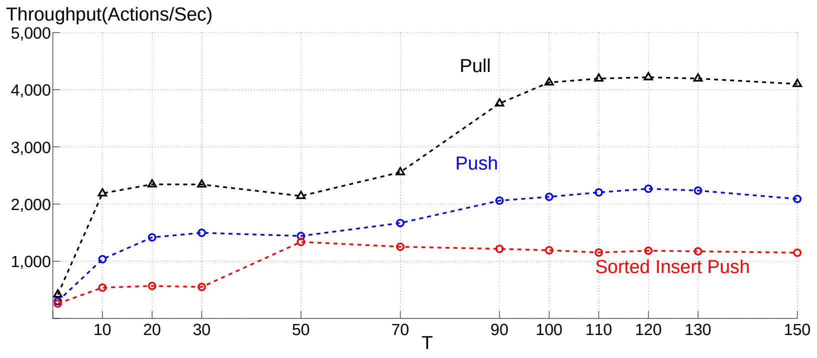


8.15.c : 80% Share Resource Action, 20% View News Feed Action

Figure 8.16: Impact of member activity distribution (exponent θ in D-Zipfian) on MongoDB's performance for three different workloads using a Sorted Insert Push architecture when $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads 1% of the SR actions are issued by pages.



8.17.a : θ=0.27



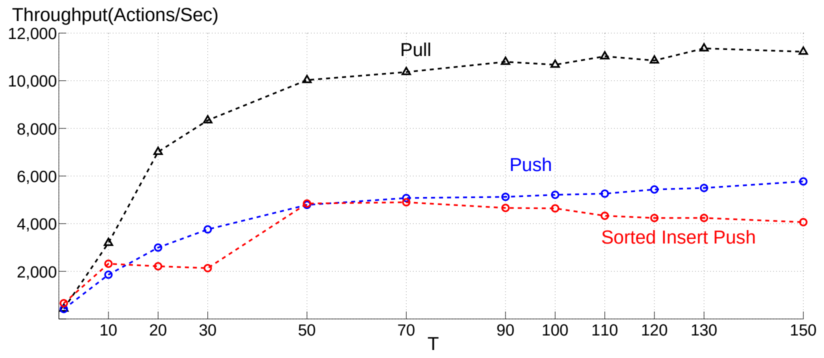
8.17.b : θ=0.99

Figure 8.17: Performance of Pull vs. Push and Sorted Insert Push with MongoDB for a High (11% Write) Mixed workload of Table 8.9 for two different member activity distributions (exponent θ of D-Zipfian). $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads 1% of the SR actions are issued by pages.

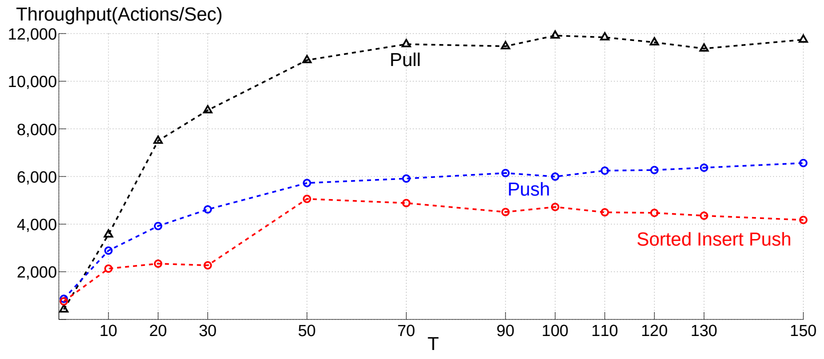
in Section 8.4.2 changes in friendship impact the feed for members. The first workload we look into consists of 10% friendship modification actions, 1% Share Resource action and 89% View News Feed action.

For this workload with $\theta = 0.27$ and a low load ($T < 50$) Pull performs better than Push, see Figure 8.17.a. This is because Push needs to construct the feed for every member upon every update which introduces an additional overhead. With medium load ($50 < T < 110$) Push performs better than Pull. This is because Push constructs the feed for every member and as the percentage of View News Feed action is higher than the Share Resource action, Push results in a better response time for retrieving feed for consumer's following Active producers and a better overall performance compared to Pull. With a high load ($T > 110$), once again Pull performs better than Push as with an increased number of updates the overhead of constructing member feed increases (as Push retrieves the entire feed and then computes the top k). With the same workload and a uniform distribution ($\theta = 0.99$) Pull performs better than Push. This is because now all members have the same activity level and pre-computing the news feed for members will be less efficient compared to re-computing it, see Figure 8.17.b.

As shown in Figure 8.17, the performance of Sorted Insert Push is both lower than Pull and Push for different member activity distributions. This is because every time an update occurs (SR



8.18.a : θ=0.27



8.18.b : θ=0.99

Figure 8.18: Performance of Pull vs. Push with MongoDB for a Very Low (0.2% Writes) Mixed workload of Table 8.9 for two different member activity distributions (exponent θ of D-Zipfian). $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\phi = 100$, $\rho = 10$. For all workloads 1% of the SR actions are issued by pages.

action is invoked or BG's friendship modification actions are invoked), the resources displayed for a member's feed need to be recomputed and sorted which reduces its performance.

We also compared the performance of Pull, Push and Sorted Insert Push with MongoDB for a workload consisting of a low percentage of BG's friendship modification actions, see Table 8.9. This workload consists of 0.1% BG's friendship modification actions (IF, AFR, RFR, TF), 0.1% Share Resource action and 99.8% View News Feed action. As shown in Figure 8.18, for this workload both with a skewed and a uniform member activity distribution, Pull results in a better performance compared to Push and Sorted Insert Push. This is because the percentage of Share Resource action is so low that the database size and feed size for members do not increase quickly. Also, Push and Sorted Insert Push need to reconstruct member feeds when friendships are modified which introduces additional overhead and reduces their performance compared to Pull.

In our next set of experiments we set the number of friends per member to 10, 100 and 1,000 and evaluated the performance of Pull, Push and Sorted Insert Push architectures for a fixed workload with MongoDB. $\phi = 1000$ emulates Super-Follower members, see Table 8.7. For all values of ϕ, Pull performs better than Push and Sorted Insert Push as it eliminates the overhead of pre-computing member's feed. But as shown in Figure 8.19 the difference between the alternatives decreases as the value of ϕ increases. This is because with a larger value for ϕ, a member follows a larger number of

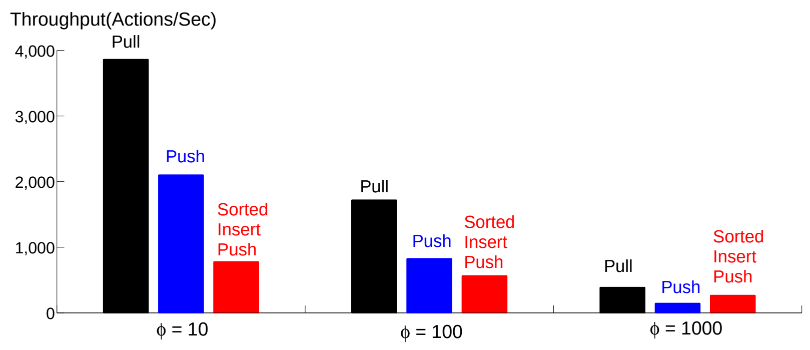


Figure 8.19: Impact of modifying the number of friends per member (ϕ) on the performance of Pull, Push and Sorted Insert Push with MongoDB for a workload consisting of 1% Share Resource action and 99% View News Feed Action. $M = 10,000$, $P = 100$, $\iota = 1,000$, $\varrho = 10$, $\rho = 10$ and $\theta = 0.27$. For all workloads 1% of the SR actions are issued by pages.

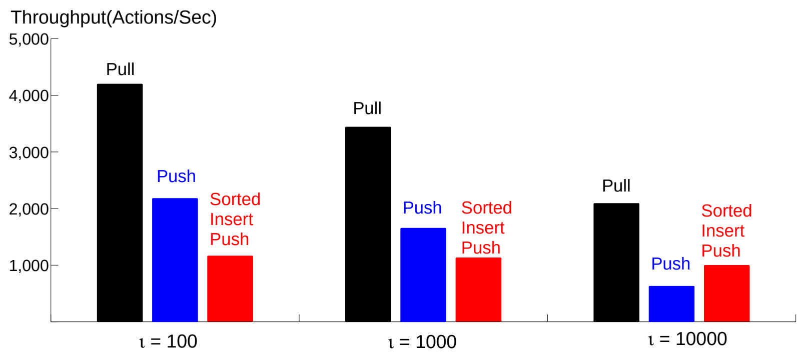
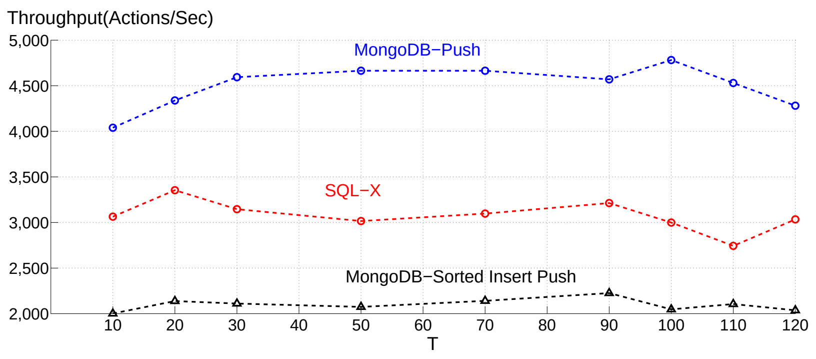


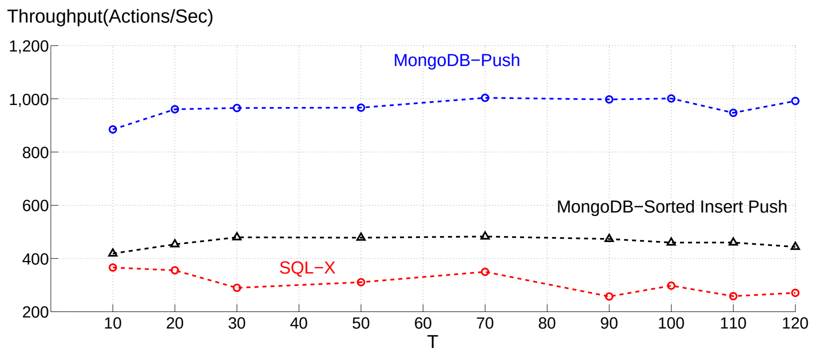
Figure 8.20: Impact of modifying the number of followers per page (ι) on the performance of Pull and Push with MongoDB for a workload consisting of 1% Share Resource action and 99% View News Feed Action. $M = 10,000$, $P = 100$, $\phi = 100$, $\rho = 10$ and $\theta = 0.27$. For all workloads 1% of the SR actions are issued by pages.

producers and the resources shared by them need to be retrieved and sorted to compute the member's feed which reduces Pull's performance.

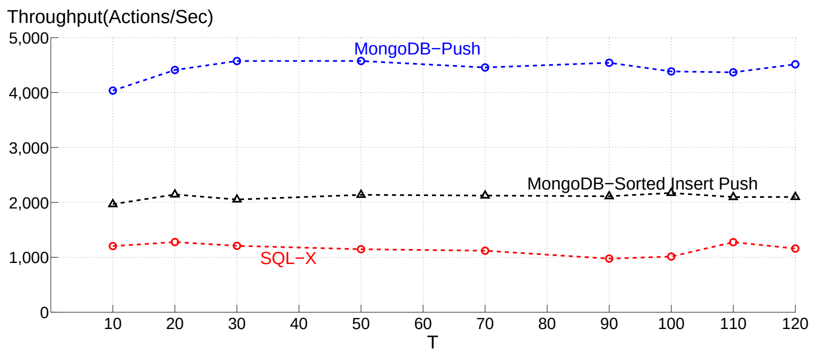
In addition, we looked at the impact of having 100, 1,000 and 10,000 followers per page (ι) on the performance of Pull and Push architectures for a fixed workload with MongoDB. In BG with a fixed value of M and P, an increase in the value of ι results in members following 1, 10 and 100 pages (values of ϱ) respectively. As shown in Figure 8.20, as we increase the value of ι, the performance of all three approaches decreases. With Push and Sorted Insert Push, this is because every time a Share Resource action by a page is issued, the shared resource needs to be pushed into the news feed for a larger number of followers. In addition each member follows a larger number of members/pages, so her feed will contain a larger number of events. Computing top k for a larger number of events is more time consuming, reducing the performance of the View News Feed actions for the members.



8.21.a : 1% Share Resource, 99% View News Feed

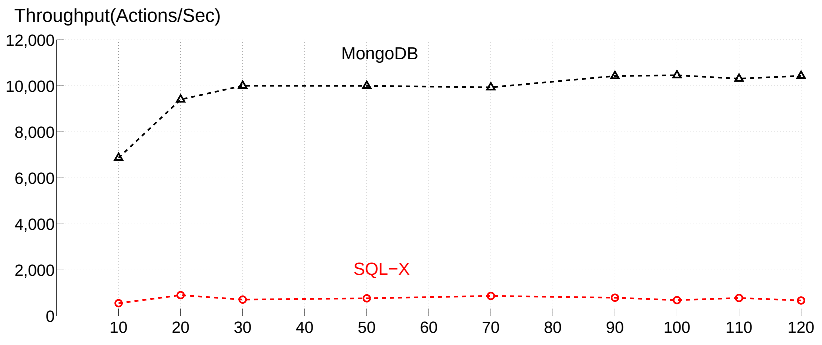


8.21.b : 10% Share Resource, 90% View News Feed

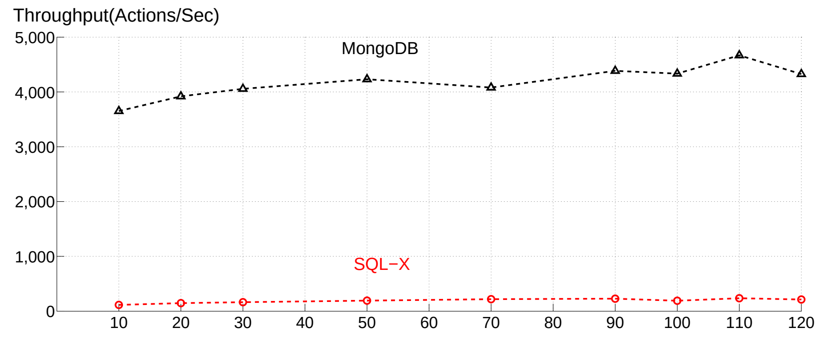


8.21.c : High Mixed Workload of Table 8.9

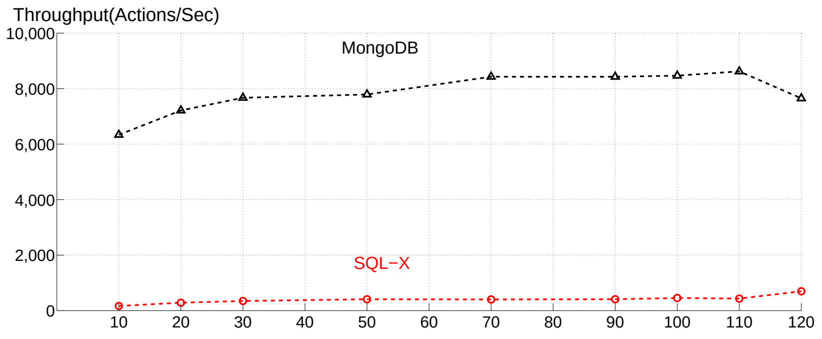
Figure 8.21: Performance of MongoDB and SQL-X with Push and Sorted Insert Push for three different workloads. $M = 100,000$, $P = 100$, $\phi = 100$, $\rho = 10$, $\iota = 10,000$, $\varrho = 10$ and $\theta = 0.99$. For all workloads 1% of the SR actions are issued by pages.



8.22.a : 1% Share Resource, 99% View News Feed



8.22.b : 10% Share Resource, 90% View News Feed



8.22.c : High Mixed Workload of Table 8.9

Figure 8.22: Performance of MongoDB and SQL-X with Pull for three different workloads. $M = 100,000$, $P = 100$, $\phi = 100$, $\rho = 10$, $\iota = 10,000$, $\varrho = 10$ and $\theta = 0.99$. For all workloads 1% of the SR actions are issued by pages.

The same trend holds true for the Pull architecture as with a fixed number of members, increase in the number of followers per page will result in an increase in the number of pages followed by each member. Hence, a larger list of producers is queried to construct a member's feed.

As show in Figure 8.20, as we increase ι from 1,000 to 10,000, the performance of Sorted Insert Push becomes superior to that of Push. This is because with an increase in the value of ι the number of producers followed by each member and the size of the member's news feed increases. With Push every time a VNF action is invoked the entire feed is retrieved by the application, sorted and displayed to the member. With a low percentage of updates, this introduces an overhead which is larger than the overhead introduced by sorted insertion required for Sorted Insert Push.

Finally in the last set of experiments we studied the behavior of MongoDB and SQL-X, a relational data store for three different workloads with Push, Sorted Insert Push (Figure 8.21) and Pull (Figure 8.22) architectures. As shown in these figures for almost all experiments, the performance of MongoDB is superior to that of SQL-X. With SQL-X and both alternatives, the disk of the node hosting the data store becomes the bottleneck. For Push, Sorted Insert Push and Pull, with MongoDB the CPU of the node hosting the data store becomes the bottleneck.

Chapter 9

Future Work

Social networks are emerging in diverse applications that strive to provide a sense of community for their users. These diverse applications range from financial web sites such as online trading system to academic institutions [1]. BG is the foundation of a benchmark to evaluate the performance of a data store for processing social networking actions such as viewing a member's profile, extending a friendship request to a member, accepting a friendship request, and others as shown in Table 1.1. BG's current implementation is used on a daily basis to evaluate the performance of novel architectures that enable high throughput, low latency data stores and provides us with insights that enable introduction of novel designs and implementations.

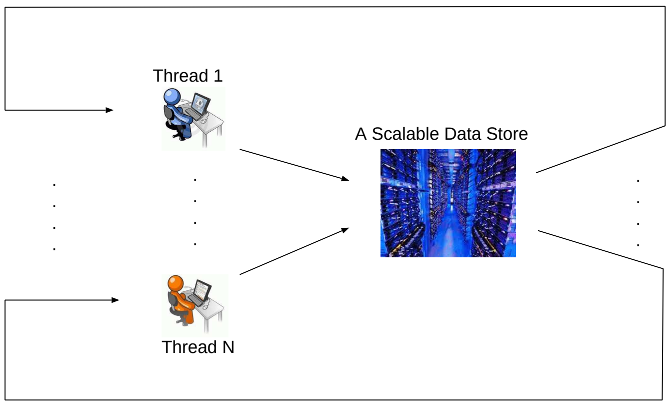
We plan to maintain BG as a state of the art benchmark. This chapter focuses on future research directions that shape our activities towards this end. We categorize these into two: those that impact the core design decisions of BG and those that extend it for use with other actions, applications, and systems.

9.1 BG's Design Decisions

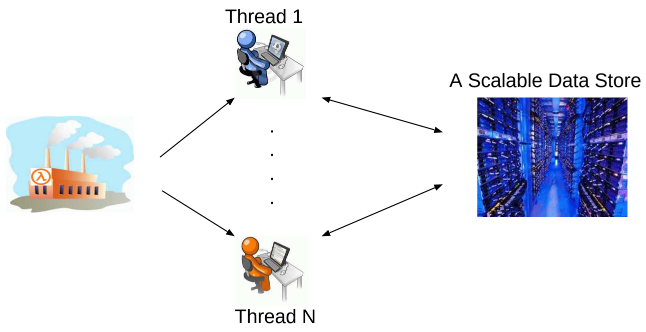
9.1.1 Closed versus Open Simulation Model

BGClients generate requests using a fixed number of threads T. Each thread emulates a random member of a social networking site performing one of BG's thirteen actions. The randomly selected member is conditioned using the D-Zipfian distribution. This is termed a closed emulation model because a thread does not emulate a new member generating a new action until its emulation of a current member completes. This model may include a think time between emulation of different members issuing actions. Historically, this is a model of a financial institution with a fixed number of tellers (ATM machines) with T concurrent customers (threads) performing financial transactions simultaneously [54]. Example transactions might include checking account balance, withdrawing and depositing money into an account, transferring funds between accounts, and others. These and others are the standard OnLine Transaction Processing (OLTP) workloads emulated by the TPC-C benchmark [54].

An open emulator is a more realistic model of a social networking site [96] (and web sites in general). With this model, the emulator generates requests based on a pre-specified arrival rate, λ. This model is depicted in Figure 9.1 where a factory generates members who issue a social networking action independently. (A member who is performing an action is termed a *socialite*.) The factory does not wait for the data store to service a request issued by a socialite. Instead, it



9.1.a) Closed



9.1.b) Open

Figure 9.1: Closed and open emulation of socialites issuing actions to a data store.

generates λ requests per unit of time using a distribution such as random, uniform, or Poisson. A Poisson distribution results in a pattern of requests that is bursty. This means λ is an average and the number of simultaneous requests at an instance in time might be higher than λ.

While the open emulator is more realistic, its design and implementation requires a careful study. This is because today's data stores service requests at such a high rate that the emulator must support λ values in the order of a million without exhausting its CPU resources. In addition the emulator must generate requests in a burst consistent with the Poisson distribution. Evaluating the feasibility of such an open emulator and its implementation in BG is one other future research direction for BG.

9.1.2 Decentralized BG

BG employs a shared-nothing architecture and scales to a large number of nodes, preventing either the CPU, network, or memory resources of a single node from limiting its request generation rate. Its software architecture consists of one coordinator and N clients, termed BGCoord and BGClient, respectively. In our experiments with an 8 core CPU, a multi-threaded BGClient is able to utilize all cores fully as long as the client component of a data store does not suffer from the convoy phenomena [20]. When the client component of a data store limits vertical scalability, as long as

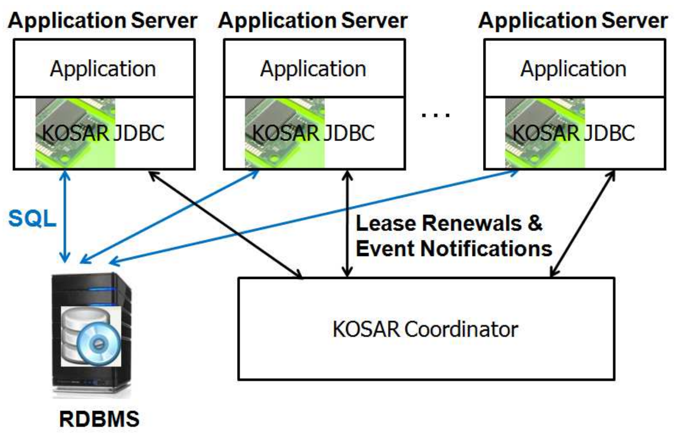


Figure 9.2: A data intensive architecture using KOSAR.

there is a sufficient amount of memory, one may execute multiple instances of BGClients on a single node to utilize all cores. BG scales horizontally by executing multiple BGClients across different nodes. BGCoord is responsible for initiating the BGClients, monitoring their progress, gathering their results at the end of an experiment, and aggregating the obtained results to compute the SoAR of a data store.

Once the BGClient instances are started, they generate requests independently with no synchronization. This is made possible using the following two concepts. First, a BGClient implements a decentralized partitioning strategy that declusters a benchmark social graph into N disjoint sub-graphs where N is the number of BGClients. A BGClient is assigned a sub-graph to generate requests referencing members of its assigned sub-graph only. While the data store is not aware of this partitioning, the data generated and stored in the data store does correspond to the N disjoint graphs. One may conceptualize each sub-graph as a province whose citizens may perform BG's actions with one another only. This means citizens of different provinces may not view one another's profile or perform friendship actions with one another.

Second, BG employs a novel decentralized implementation of the Zipfian distribution, named D-Zipfian [13, 63], that ensures the distribution of requests to the different members and resources is independent of N. Thus, the distribution of access with one node is the same as that with several nodes. D-Zipfian in combination with partitioning of the social graph enables BG to utilize N nodes to generate requests without requiring coordination until the end of the experiment, see [12, 15, 13] for details.

While BG scales to a large numbers of nodes, its two concepts may fail to evaluate some data stores objectively. As an example, consider the architecture of Figure 9.2 where an application is extended with a cache such as KOSAR or EhCache [47]. This caching framework consists of a coordinator that maintains which application server has cached a copy of a data item in its KOSAR JDBC. When one application server updates a copy of the data item, its KOSAR JDBC informs the coordinator of the impacted data item. In turn, the coordinator invalidates a copy of this data item that resides in the KOSAR JDBC of other application servers. With a skewed pattern of access to members and a workload that exhibits a low read to write ratio, a centralized coordinator may become the bottleneck and dictate the overall system performance. The aforementioned two concepts employed by BG fail to cause the formation of such a bottleneck. To elaborate, each application server references data items that are unique to itself since its assigned sub-graph is unique and independent of the other sub-graphs. Hence, once an application server updates a cached data item, BG does not exercise the coordinator informing KOSAR JDBC of another application server.

To address the above limitation, we are extending BG to employ N BGClients with one social graph. The key concept is to hash partition pages, members and resources across the N BGClients. Each BGClient is aware of the hash function and employs the original Zipfian distribution (instead of D-Zipfian) to generate member ids/page ids. When a BGClient BGC_i references a data item that does not belong to its assigned partition, it contacts the BGClient that owns the referenced data (say BGC_j) to lock that data item for exclusive use by BGC_i and to determine if its intended action is possible. BGC_j grants the lock request if there is no existing lock on the referenced data item and the action is possible, enabling BGC_i to proceed to generate a request with the identified data item to the data store. Once the request is serviced, BGC_i contacts BGC_j to release the exclusive lock on the referenced data item to make it available for use by other BGClients. This design raises the following interesting questions:

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

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

- How different are the obtained results with N disjoint social graphs (current version of BG) and one social graph (the proposed change)? This question applies to those systems that may use the current version of BG. We intend to repeat our published experiments such as those reported in [14] to quantify differences if any.

An investigation of these questions shape another research direction for BG.

9.2 Extensions And Use Cases

9.2.1 Actions

A social networking system is identified by two kind of workloads:

1. Interactive workloads that read or write a small amount of data from big data, and

2. decision support and business intelligence workloads consisting of analytical queries that read a large amount of data from big data and involve complex and resource intensive queries such as those analyzing online behavior of users for marketing purposes [76, 92].

Thus far, BG has captured the essential features of the first workloads by focusing on thirteen interactive social actions. These are an abstraction of actions performed by a member in a social networking system, see Table 1.1. A similar study must be conducted for the second workload. This would extend BG with new actions such as friend/follower suggestions, product recommendations, sentiment analysis and etc. A challenge is to identify the appropriate performance metric to measure and report. The metric should be objectively measurable and allow for meaningful trend or statistical analysis of the performance of alternative systems and design decisions. We envision generating social graphs with a set of well known results for a complex action, prioritized based on their quality. Given an experimental data store, BG would analyze the quality of responses provided for an action in this workload. It may compute the amount of time required for a data store to approximate the different responses based on their quality[1]. A challenge is to generate social graphs that are realistic, model analytics that are a relevant abstraction of those performed by different social networking sites, and reduce meaningful data to compare alternative choices with one another. With the second workload, another metric that might be of interest is the amount of time required to load database [49]. An option is to aggregate (weighted aggregate) the amount of time to load a fixed amount of data into a data store with BG's other performance metrics (i.e. SoAR and Socialites rating) and introduce a new combined metric indicating a data store's performance.

9.2.2 Applications

Most social networking systems also provide media access and retrieval services for their members. Examples include, displaying images and streaming audio/video. For such applications, apart from the timeliness of the action, the quality of the response is also important. An obvious future research direction is to extend BG to evaluate these metrics and provide insights for developing new algorithms for multimedia access and retrieval.

In addition, as BG is an extensible benchmark (see Chapter 4), it can easily be adapted to evaluate the performance of data stores used for other big data applications such as healthcare or space related scientific applications. Identifying these applications and their characteristics, and extending BG to support them is an interesting future research direction. This motivates the benchmark generator detailed in the next section.

9.2.3 Benchmark Generator

Although BG is an extensible benchmarking framework and a new action can easily be added as a module, yet some actions require accessing multiple entities with a specific relationship and logging relevant information. This is important in order for BG to create valid actions and perform an error-free validation. For example, BG's Accept Friend Request action (AFR), requires two member entities, invitor and invitee, and there should exist a pending friend request initiated from the invitor to the invitee. So in order for BG to emulate a valid AFR action by Member A, it needs to find a Member B who has initiated a friend request to Member A and accept that invitation. In addition, the appropriate log records for the change in the two members' friendship and pending friend requests need to be generated for the action.

BG maintains all relationships between different entities (friendship: members-members, pendingRequests: members-members, following: members-pages, own: members-resources and post: comments-resource) in its internal in-memory data structures. Currently when extending BG with a new action, the developer must author software to select appropriate entities (i.e. members), access the appropriate data structures (pendingRequests:members-members) and modify them accordingly.

[1]This effort would be different than simply extracting a social graph from a site such as Twitter and using it for evaluating an algorithm in that the social graph is created with a pre-specified set of known answers.

For example with the Accept Friend Request action by Member A, the pendingRequests:members-members structure is queried to find a Member B that has initiated a friend request to Member A. Next, the AFR action is issued against the data store, the entry related to the pending request is removed from the data structure maintaining the pending requests, and the new friendship is added to the data structure maintaining the friendship relationships. The code for all of this is provided by the developer. The developer also needs to provide code to generate the appropriate log records. With the previous example, this includes two log records for the two members' friendships: one indicating that A has been added as B's new friend and the other indicating that B has been added as A's new friend. And one log record for the invitee's pending request relationships indicating that A no longer has a pending invitation from B.

A future research direction for BG is to convert it to a Benchmark Generator framework [67]. The main objective of this research will be to develop a general framework that inputs the actions and their meta data and extends BG with the appropriate modules for the new actions with minimal (hopefully no) additional software from the developer. This framework may be provided with a specification file from the developer that provides a high-level description of the new action. For each action the description is split into four distinct parts:

1. The particular entity sets and their number involved in an action. For example, Thaw Friendship action involves two entities of the Member entity set.

2. The dependencies between the entities for an action. With Thaw Friendship, the second member must be friends of the first member).

3. The action behavior. For example, the second member is selected using a random distribution from among the friends of the first members.

4. Log record information. For example, a log record indicating that the second member's invitation is removed from the first member's pending friend requests, a log record indicating the the first member is added as the second member's friend and a log record indicating that the second member is added as the first member's friend.

This framework will consist of interfaces which allow developers to introduce new entity types, dependencies, behaviors and log record types. And it will use the descriptions provided by the developers in the configuration files to create and add the new action module to BG automatically. This effort is justified by the fact that there are an increasing number of big data applications with many different set of actions, see Section 9.2.2 .

9.2.4 End System Design

Another future research direction is to use BG to explore topics that are of interest for today's social networking applications. There are several topics that warrant further exploration such as elasticity, and availability and fault tolerance.

Scalability has always been an important aspect of distributed data store designs (see Section 8.3); and with the continuous drop at the cost of computers, scaling data stores to larger number of nodes continues to rise. Today one of the main challenges with distributed data stores is to provide scalability and fault tolerance without sacrificing performance. And as with a larger number of data store nodes, the probability of failure is higher, affecting the performance of the system, different distributed data stores strive to impalement a quick and efficient technique for failure detection and recovery.

Designing a system with a desired degree of fault tolerance is difficult and requires understanding factors affecting data store's availability characteristics. In this study we will use BG to study the different fault tolerance techniques used in distributed data stores, their recovery mechanisms, the amount of time it takes for the systems to recover and the system parameters impacting these. The results of this study can:

- help a designer to understand important trade-offs and choose an appropriate system architecture and fault tolerance technique which allows for greater scalability and robustness without significantly sacrificing functionality and performance,

- help an application developer to pick the data store which results in a higher performance and availability,

- provide researchers from both industry and academia with interesting challenges in this area and a way to objectively compare the effectiveness and efficiency of new and existing techniques in this area.

The challenge with performing this study is that there are many different kind of faults and introducing all of them is almost impossible [29]. The simplest way to perform this study is to start a BG workload, kill one or more data store nodes while BG is issuing the workload against the data store, and observe any resulting errors and performance impact. killing the data store can represent various data store failure scenarios including software, hardware and environmental failures. A similar approach is to use tools such as NISTNet [24], ModelNet [111] or Emulab [115] to emulate network layer faults.

9.2.5 Introducing Skewness in Social Graph

With today's BG, the structure of the social graph is dictated by a uniform distribution. For instance, all pages have the same number of followers, all members follow the same number of pages, the profile and thumbnail image sizes for all members are the same and all created resources and comments posted on the resources have the same size in bytes.

Data skewness may change the performance of the system especially in extreme cases. For example the performance of the system when a set of larger size resources are retrieved will be different from when a set of smaller size resources are retrieved. Or retrieving the news feed for a member following a large number of pages may be slower than retrieving it for one following a smaller number of pages[2].

In addition while evaluating a data store using BG, the change in the size of the data store may impact its performance. For this kind of data stores one may come up with symmetric workloads which try to maintain the data store state as constant, see Section refsec:mix. Introducing skewness into BG's data results in identifying symmetric workloads more challenging.

A research direction for BG is to modify its design to address the described data skewness.

[2]A simple approach is to use a random number generator to decide the size of the data to be inserted.

Appendix A

Survey's Used to Evaluate BG's Extensibility

We used the following three surveys to evaluate BG's usability and extensibility. These surveys were filled anonymously.

A.1 Survey 1

Data Store: Date:

1. How would you rate yourself on a scale of 1-5, with 5 being the most familiar and 1 being the least familiar with each of the following topics?

	1	2	3	4	5
Performance benchmarking	☐	☐	☐	☐	☐
ER-Diagrams and conceptual schema	☐	☐	☐	☐	☐
Social networks	☐	☐	☐	☐	☐
Assigned data store	☐	☐	☐	☐	☐

2. On a scale of 1-5 how comfortable are you with the following:

	1	2	3	4	5
Understanding benchmarking	☐	☐	☐	☐	☐
Understanding BG's actions	☐	☐	☐	☐	☐
Working with you assigned data store	☐	☐	☐	☐	☐

3. How many hours did you spend on the following:

	$<$1 hr	1-5 hrs	5-10 hrs	$>$10 hrs
Understanding BG's conceptual schema	☐	☐	☐	☐
Understanding BG's actions	☐	☐	☐	☐
Learning about your assigned data store	☐	☐	☐	☐
Developing a logical schema with your data store for BG	☐	☐	☐	☐
Modifying the logical schema you designed in support of BG's actions	☐	☐	☐	☐

4. Which of the following did you use to do your homework?
   ☐ BGBenchmark.org
   ☐ BG's Google forum
   ☐ BG Presentation slides
   ☐ BG's Paper
   ☐ BG's source code
   ☐ Other:

A.2 Survey 2

Data store: Date:

Please take a few moments to complete our BG usability survey. Your responses will help us address any issues that you may have and improve our software.

1. How would you rate yourself on a scale of 1-5, with 5 being the most familiar and 1 being the least familiar with each of the following topics?

	1	2	3	4	5
Java	☐	☐	☐	☐	☐
Any Java IDE Tool	☐	☐	☐	☐	☐
BG	☐	☐	☐	☐	☐
Databases (client/server architecture)	☐	☐	☐	☐	☐
Assigned data store	☐	☐	☐	☐	☐

2. Based on completing this homework on a scale of 1-5 how comfortable are you with the following (5 being the most comfortable and 1 being the least comfortable):

	1	2	3	4	5
Understanding BG's objective	☐	☐	☐	☐	☐
Installing BG	☐	☐	☐	☐	☐
Resolving installation issues	☐	☐	☐	☐	☐
Using BG with sample clients	☐	☐	☐	☐	☐
Using BG to connect to your data store	☐	☐	☐	☐	☐
Using BG to create schema for your data store	☐	☐	☐	☐	☐
Using BG to load your data store	☐	☐	☐	☐	☐
Understanding BG's input parameters	☐	☐	☐	☐	☐
Resolving BG related runtime errors and exceptions	☐	☐	☐	☐	☐

3. How many hours did you spend on the following:

	<1 hr	1-5 hrs	5-10 hrs	>10 hrs
Designing the logical model for your data store (overall hours including hours spent for part1)	☐	☐	☐	☐
Implementing creation of connection to the data store	☐	☐	☐	☐
Testing creation of connection to your data store (-testdb)	☐	☐	☐	☐
Implementing the schema creation phase for your data store	☐	☐	☐	☐
Understanding the parameters required for the schema creation	☐	☐	☐	☐
Testing the schema creation code for your data store	☐	☐	☐	☐
Setting up BG to create the schema for your data store	☐	☐	☐	☐
Implementing the load phase for your data store	☐	☐	☐	☐
Understanding the parameters required for the loading phase	☐	☐	☐	☐
Testing the load phase for your data store	☐	☐	☐	☐
Setting up BG to load your data store	☐	☐	☐	☐
Understanding BG's output parameters	☐	☐	☐	☐

4. Please respond to the following:

	Not At All	Not Very Often	Often	Very Often	N/A
How often did you access the bgbenchmark.org when using/extending BG?	☐	☐	☐	☐	☐
How often did you read the postings on the BG Google Group?	☐	☐	☐	☐	☐
How often did you post in the BG Google group?	☐	☐	☐	☐	☐
How often were your problems solved using the bgbenchmark.org or the BG Google group?	☐	☐	☐	☐	☐
How often did you contact the TA for help?	☐	☐	☐	☐	☐

5. Please rate the following criteria for BG.

	Poor	Fair	Neutral	Good	Excellent
Ease of installation	☐	☐	☐	☐	☐
Software dependency	☐	☐	☐	☐	☐
Installation or first use experience	☐	☐	☐	☐	☐
Repeated usage experience	☐	☐	☐	☐	☐
Timeliness of the installation	☐	☐	☐	☐	☐
Compatibility with hardware/software	☐	☐	☐	☐	☐
Quality of documentation for schema creation and load phase	☐	☐	☐	☐	☐
Appropriateness of the documentation for schema creation and load phase	☐	☐	☐	☐	☐
Usability of the documentation for schema creation and load phase	☐	☐	☐	☐	☐
Usability of BG to create schema for your data store	☐	☐	☐	☐	☐
Timeliness of schema creation	☐	☐	☐	☐	☐
Usability of BG to load your data store	☐	☐	☐	☐	☐
Timeliness of loading	☐	☐	☐	☐	☐
Feedback provided by BG during load	☐	☐	☐	☐	☐
Informative errors	☐	☐	☐	☐	☐
Overall Usage/Experience	☐	☐	☐	☐	☐

6. How long have you used BG for?
7. What do you dislike about BG so far?
8. What suggestions do you have to improve BG?

A.3 Survey 3

Data store: Date:

Please take a few moments to complete our BG usage survey. Your responses will help us address any issues that you may have as well to improve our software.

1. Based on completing this homework on a scale of 1-5 how comfortable are you with the following (5 being the most comfortable and 1 being the least comfortable):

	1	2	3	4	5
Understanding BG's objective	☐	☐	☐	☐	☐
Using BG's command line interface	☐	☐	☐	☐	☐
Implementing BG's actions for your data store	☐	☐	☐	☐	☐
Understanding BG's input parameters	☐	☐	☐	☐	☐
Resolving BG related runtime errors and exceptions	☐	☐	☐	☐	☐
Understanding BG's output	☐	☐	☐	☐	☐

2. How many hours did you spend on the following:

	<1 hr	1-5 hrs	5-10 hrs	>10 hrs
Designing the logical model for your data store (overall hours including hours spent for part1 and part2)	☐	☐	☐	☐
Understanding BG’s actions (requirements and functionality)	☐	☐	☐	☐
Implementing BG’s actions	☐	☐	☐	☐
Testing all the actions implemented	☐	☐	☐	☐
Understanding how to use BG’s command line interface	☐	☐	☐	☐
Setting up BG to issue a workload against the data store	☐	☐	☐	☐
Understanding BG’s output parameters	☐	☐	☐	☐

3. Please respond to the following:

	Not At All	Not Very Often	Often	Very Often	N/A
How often did you access the bgbenchmark.org when using/extending BG?	☐	☐	☐	☐	☐
How often did you read the postings on the BG Google Group?	☐	☐	☐	☐	☐
How often did you post in the BG Google group?	☐	☐	☐	☐	☐
How often were your problems solved using the bgbenchmark.org or the BG Google group?	☐	☐	☐	☐	☐
How often did you contact the TA for help?	☐	☐	☐	☐	☐

4. Please rate the following criteria for BG.

	Poor	Fair	Neutral	Good	Excellent
Repeated usage experience	☐	☐	☐	☐	☐
Quality of documentation for BG’s actions	☐	☐	☐	☐	☐
Appropriateness of the documentation for BG’s actions	☐	☐	☐	☐	☐
Usability of the documentation for implementing BG’s actions	☐	☐	☐	☐	☐
Usability of BG to issue actions against the data store	☐	☐	☐	☐	☐
Feedback provided by BG during issuing actions against the data store	☐	☐	☐	☐	☐
Informative errors	☐	☐	☐	☐	☐
Intuitive output	☐	☐	☐	☐	☐
Overall Usage/Experience	☐	☐	☐	☐	☐

Appendix B

BG's Visualization Deck

One may obtain invalid ratings from a data store for a variety of reasons ranging from invalid parameter settings to BGClients becoming fully utilized. We use the term *bottleneck node* to refer to a node of the system with either a fully utilized CPU, network bandwidth, or mass storage device (disk or flash). BG's interface empowers its users to visualize each participating node and utilization of its resources to detect bottlenecks, see Figure B.1. In addition, it enables its users to perform the following tasks: specify values for parameters used to populate a data store, start loading the data store, specify values for parameters used to initiate a multi-node BGClient experiment, start rating a data store and monitor its progress. This tool is useful for analyzing the SoAR and Socialites rating of a data store and detecting when the obtained ratings are invalid.

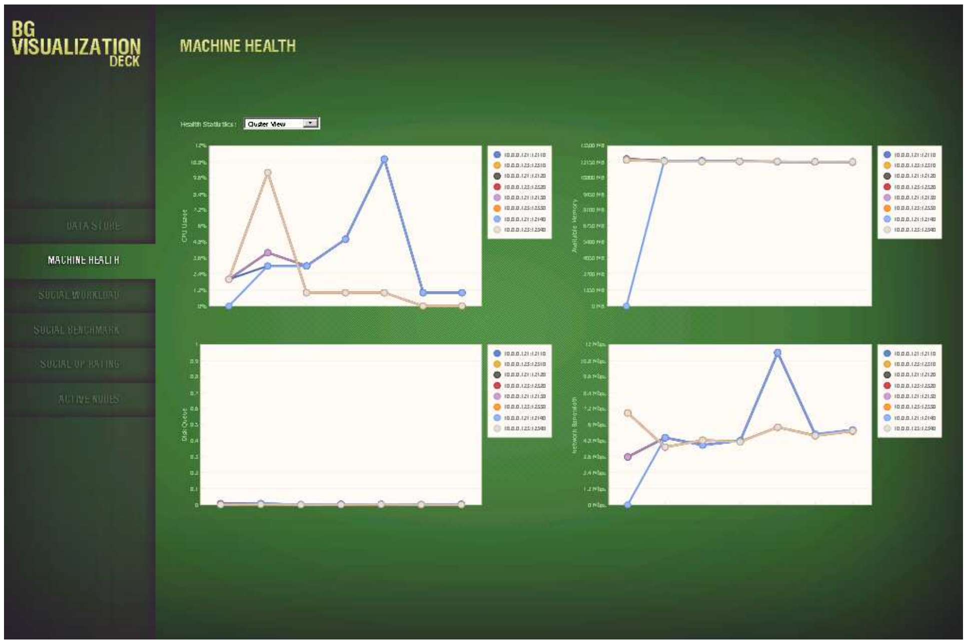


Figure B.1: BG's Visualization Deck.

Appendix C

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 our experiments.

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.

Bibliography

[1] Special report: Top 11 Technologies of the Decade. *IEEE Spectr.*, 47(12), 2010.

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

[3] 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.

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

[5] C. Amza, A. L. Cox, and W. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *ICDE*, 2005.

[6] C. Amza, G. Soundararajan, and E. Cecchet. Transparent Caching with Strong Consistency in Dynamic Content Web Sites. In *Supercomputing*, ICS '05, pages 264–273, New York, NY, USA, 2005. ACM.

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

[8] 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.

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

[10] P. Bailis and A. Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Communications of the ACM*, May 2013.

[11] P. Bailis, Sh. Venkataraman, M.J. Franklin, J.M. Hellerstein, and I. Stoica. Quantifying Eventual Consistency with PBS. *The VLDB Journal*, pages 1–24.

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

[13] S. Barahmand and S. Ghandeharizadeh. D-zipfian: A Decentralized Implementation of Zipfian. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, DBTest '13, 2013.

[14] S. Barahmand, Sh. 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.

[15] Sumita Barahmand and Shahram 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.

[16] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*. USENIX, October 2010.

[17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[18] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer zsu. Qagen: Generating query-aware test databases. Technical report, In SIGMOD, 2007.

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

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

[21] Twitter Blog. Show Me More, http://blog.twitter.com/2011/08/show-me-more.html.

[22] JBoss Cache. JBoss Cache, http://www.jboss.org/jbosscache.

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

[24] M. Carson and D. Santay. Nist net: A Linux-based Network Emulation Tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, July 2003.

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

[26] Facebook Help Center. How News Feed Works, https://www.facebook.com/help/327131014036297/.

[27] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[28] P. P. Chen. The entity-relationship model&mdash;toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

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

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

[31] Linked Data Benchmark Council. Linked Data Benchmark Council, http://www.ldbc.eu/.

[32] Transaction Processing Performance Council. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp.

[33] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *2nd ACM Multimedia Conference*, October 1994.

[34] Brittany Darwell. EdgeRank and Graph Rank Defined, http://www.insidefacebook.com/2011/12/27/edgerank-and-graph-rank-defined/.

[35] Brittany Darwell. News Feed, EdgeRank and Page Posts: What's Really Going on with Facebook?, http://www.insidefacebook.com/2012/11/08/news-feed-edgerank-and-page-posts-whats-really-going-on-with-facebook/.

[36] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, D. VanderMeer, K. Ramamritham, and D. Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *VLDB*, pages 667–670, 2001.

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

[38] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[39] 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.

[40] Facebook. Controlling What You See in News Feed, https://www.facebook.com/help/335291769884272/.

[41] Facebook. News Feed Privacy, https://www.facebook.com/help/420576494648116/.

[42] Facebook. People You May Know, https://www.facebook.com/help/501283333222485/.

[43] R. Fan and N. Lynch. Gradient Clock Synchronization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 320–327, 2004.

[44] S. Ghandeharizadeh, S. Barahmand, A. Ojha, and J. Yap. Recall All You See, http://rays.shorturl.com, 2010.

[45] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *16th International Conference on Very Large Data Bases*, pages 481–492, 1990.

[46] 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.

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

[48] S. Ghandeharizadeh, J. Yap, and S. Barahmand. COSAR-CQN: An Application Transparent Approach to Cache Consistency. In *Twenty First International Conference On Software Engineering and Data Engineering*, Los Angeles, CA, Best Paper Award, 2012.

[49] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD*, pages 1197–1208, 2013.

[50] Jennifer Golbeck. Semantic Web Interaction through Trust Network Recommender Systems . In *End User Semantic Web Interaction*, 2005.

[51] Google+. Content That Appears on Your Home Page, http://support.google.com/plus/answer/1269165?hl=en.

[52] Google. Google, http://googleblog.blogspot.com/2011/10/google-popular-posts-eye-catching.html.

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

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

[55] L. Guo, E. Tan, S. Chen, X. Zhang, and Y. (E.) Zhao. Analyzing Patterns of User Content Generation in Online Social Networks. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

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

[57] K. Hampton, L.S. Goulet, C. Marlow, and L. Rainie. Why Most Facebook Users Get More Than They Give. In *Pew Internet and American Life Project*, 2012.

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

[59] H. He and A. K. Singh. GraphRank: Statistical Modeling and Mining of Significant Subgraphs in the Feature Space. In *IEEE International Conference on Data Mining*, 2006.

[60] Awareness Inc. Gaining an Edge with EdgeRank, http://www.vetanswers.com.au/assets/gaining-an-edge-with-edgerank-facebook-white-paper.pdf.

[61] K. Iwanicki, M. van Steen, and S. Voulgaris. Gossip-based Clock Synchronization for Large Decentralized Systems. In *Proceedings of the Second IEEE International Conference on Self-Managed Networks, Systems, and Services*, pages 28–42, 2006.

[62] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 49–60, 1997.

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

[64] 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.

[65] M. Kagan. 12 Essential Facebook Stats [Data], http://blog.hubspot.com/blog/tabid/6307/bid/14715/12-Essential-Facebook-Stats-Data.aspx.

[66] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *VLDB*, 1(2), 2008.

[67] J. Kriegel, F. Broekaert, A. Pegatoquet, and M. Auguin. Waveperf: A Benchmark Generator for Performance Evaluation. *SIGBED Review*, 9(2):7–11, 2012.

[68] A. Labrinidis and N. Roussopoulos. Exploring the Tradeoff Between Performance and Data Freshness in Database-Driven Web Servers. *The VLDB Journal*, 2004.

[69] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[70] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, Jul 1978.

[71] S. T. Leutenegger and D. Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*, pages 22–31, 1993.

[72] LinkedIn. Voldemort, http://data.linkedin.com/opensource/voldemort.

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

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

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

[76] H. Ma, J. Wei, W. Qian, Ch. Yu, F. Xia, and A. Zhou. On Benchmarking Online Social Media Analytical Queries. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES ’13, pages 10:1–10:7, New York, NY, USA, 2013. ACM.

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

[78] D. L. Mills. On the Accuracy and Stablility of Clocks Synchronized by the Network Time Protocol in the Internet System. *SIGCOMM Comput. Commun. Rev.*, 20(1), December 1989.

[79] C. Mohan. History repeats itself: Sensible and nonsensql aspects of the nosql hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT ’13, pages 11–16, New York, NY, USA, 2013. ACM.

[80] MongoDB. Class WriteConcern, http://api.mongodb.org/java/2.10.1/com/mongodb/WriteConcern.html.

[81] MongoDB. FAQ: Concurrency, http://docs.mongodb.org/manual/faq/concurrency/.

[82] MongoDB. How Foursquare is Using MongoDB, http://www.mongodb.com/presentations/how-foursquare-using-mongodb.

[83] MongoDB. Read Preference, http://docs.mongodb.org/manual/core/read-preference/.

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

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

[86] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, Harry C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 385–398, Berkeley, CA, 2013. USENIX.

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

[88] 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.

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

[90] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012.

[91] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI*. USENIX, October 2010.

[92] LDBC Cooperative Project. Social Network Benchmark (SNB) Task Force Progress Report, http://www.ldbc.eu:8090/download/attachments/4325436/LDBC_SNB_Report_Nov2013.pdf.

[93] Ratzel R and R. Greenstreet. Toward Higher Precision. *Commun. ACM*, 55(10):38–47, October 2012.

[94] M.R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J.J. Wylie. Toward a Principled Framework for Benchmarking Consistency. In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[95] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, August 2001.

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

[97] R. Sears, C. V. Ingen, and J. Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem. Technical Report MSR-TR-2006-45, Microsoft Research, 2006.

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

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

[100] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: Selectively materializing users' event feeds. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 831–842, New York, NY, USA, 2010. ACM.

[101] Statista. Number of Photos Per Facebook User, http://www.statista.com/statistics/181756/number-of-photos-uploaded-and-linked-by-facebook-users/.

[102] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, San Diego, California, August 2001.

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

[104] M. Stonebraker. What Does 'Big Data' Mean? *Communications of the ACM, BLOG@ACM*, September 2012.

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

[106] M. R. Stonebraker. The Case for Shared-Nothing. In *Proceedings of the 1986 Data Engineering Conference*. IEEE, 1986.

[107] Francis Tan. 1 million Facebook fans brings in an average of 826 likes and 309 comments per post, http://thenextweb.com/facebook/2011/05/17/1-million-facebook-fans-brings-in-an-average-of-826-likes-and-309-comments-per-post/.

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

[109] Twitter. Twitter, https://twitter.com/toptweets.

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

[111] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-scale Network Emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, December 2002.

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

[113] Project Voldemort. Project Voldemort, A Distributed Database, http://www.project-voldemort.com/voldemort/.

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

[115] B. White, J. Lepreau, L. Stoller, R. Ricci, Sh. Guruprasad, M. Newbold, M. Hibler, Ch. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. pages 255–270, Boston, MA, December 2002.

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

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

[118] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *VLDB*, pages 188–199, 2000.

[119] K. Yeung. Google's Suggested Communities Feature for Google+ Aims to Make it Easier to Discover New Interests, http://thenextweb.com/google/2012/12/08/google-communities-suggested-feature-launches/.

[120] YouTube. YouTube, http://www.youtube.com/t/about_essentials.

[121] YouTube. YouTube, http://youtube-trends.blogspot.com/p/about-youtube-trends.html.

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