# On Scalability of Two NoSQL Data Stores for Processing Interactive Social Networking Actions

Sumita Barahmand, Shahram Ghandeharizadeh, and Jia Li

Database Laboratory Technical Report 2014-11

Computer Science Department, USC

Los Angeles, California 90089-0781

{barahman, shahram, jli536}@usc.edu

March 1, 2015

#### Abstract

This paper quantifies the scalability of a document store named MongoDB and an extensible store named HBase for processing simple operations using a social networking benchmark named BG (www.bgbenchmark.org). We report on the vertical and the horizontal scalability of both data stores. We quantify speedup and scaleup characteristics of each data store's Social Action Rating (SoAR). SoAR is the highest observed throughput with a data store while satisfying a Service Level Agreement (SLA) such as 95% of requests observing a response time of 100 milliseconds or faster. While the speedup experiments maintain a fix sized social graph and vary the number of nodes, the scaleup experiments vary both the size of the social graph and the number of nodes proportionally. A system provides a superlinear SoAR speedup as a function of the number of nodes when it transitions from fully utilizing its disk bandwidth to fully utilizing either its CPU cores or network bandwidth. The speedup and scaleup of both data stores is limited with a resource such as the network interface card of one to three nodes becoming fully utilized and dictating the SoAR of a many (12) node deployment. One may configure MongoDB to use replicas of shards, enhancing its speedup characteristics and incurring a very small amount (<0.0003%) of unpredictable data with workloads consisting of writes.

## **A** Introduction

In recent years, several new systems referred to as "NoSQL" data stores have been introduced to process simple operations in a scalable manner [12]. A *simple* operation reads and writes a small amount of data from the database [33, 21]. An application specific example includes interactive social networking actions [4, 3] such as view a member profile or extend a friend invitation by popular person-to-person cloud service providers such as Facebook, Google+

and LinkedIn. A key question is how well do these novel "NoSQL" data stores scale? In this study, we investigate this question for a document store named MongoDB and an extensible data store named HBase. Their key features are as follows. First, they assume a shared-nothing [34] architecture and scale by partitioning and replicating data. Second, they assume data models with the ability to dynamically add new attributes to records. MongoDB implements a JSON-like data model and HBase implements an extensible record structure similar to Bigtable [13, 14]. MongoDB sacrifices the durability of operations by writing data produced by updates to the disk periodically, every 100 milliseconds [28]. Moreover, it supports deployments that enhance performance by using secondary replicas of data for query processing. This is at the expense of producing some stale data (termed *unpredictable* data [4, 6]). A key claim of both MongoDB and HBase is their ability to scale simple operations. Below, we describe two forms of scalability with speedup and scaleup as metrics to quantify this claim.

Vertical scaling is the ability to process a larger number of simple operations by increasing the resources of a single node. Example resources include a node's number of CPU cores, number of network interface cards, disk bandwidth using RAID, bandwidth provided by the disk host bus adapter, etc. Ideally, the throughput of a data store should increase proportionally with an increase in system resources. Throughput is defined as the number of simple operations processed per unit of time [27]. An increase in system resources may have either no or marginal impact on the average service time. Service time is defined as the amount of time elapsed from when a simple operation is issued to the data store until the data store provides either results with a read request or a confirmation of the completion of a write request. Service time is quantified with a system processing a single request [27]. The average service time may not scale vertically. To illustrate, consider a simple operation that uses an index to look up the profile information of a member of a social networking site, termed a View Profile (VP) action [4] and presented formally in Section D.1. With a high system load, the network interface card of the server may become fully utilized to define the observed throughput with VP. Doubling the number of networking cards may double the throughput of VP with a high system load, however, it may not enhance its average service time when requests are processed using one of the networking cards.

Horizontal scaling is the ability to use multiple nodes to support a higher throughput. This is realized by distributing both the data and the load of the simple operations across many servers. Ideally, each simple operation must be processed by a single node to avoid the communication overhead of parallelism associated with coordinating multiple nodes [23, 24, 12, 33]. Use of multiple nodes increases the average service time and is undesirable as it increases the response time observed by users. (Response time is defined as the service time plus queuing¹ delays [27].) Similar to the discussion of vertical scaling, one should not expect the service time of simple operations to improve with additional nodes as they perform a minimal amount of work using a single node [20, 23]. To illustrate, consider the same VP example action with 1 and 100 nodes. With one node, it utilizes an index structure to perform one or two disk I/Os to retrieve the profile of a member. Ideally, with 100 nodes, the VP action should be directed to one node to perform

<sup>&</sup>lt;sup>1</sup>Queues are formed in the presence of multiple concurrent requests competing for resources and waiting for one another. With a low system load, it is possible to have no queuing delays, causing average response time to equal average service time.

Action	Type	Frequency)
View Profile (VP)	Read	40%
List Friends (LF)	Read	10%
View Friend Requests (VFR)	Read	5%
Invite Friend (IF)	Write	4%
Accept Friend Request (AFR)	Write	2%
Reject Friend Request (RFR)	Write	2%
Thaw Friendship (TF)	Write	2%
View Top-K resources (VTR)	Read	35%

Table 1: A mix of read and write actions used throughout this study.

the same one or two disk I/O. Use of multiple nodes would increase its service time [23, 33, 12].

One may quantify both the horizontal and vertical scalability characteristics of a data store for simple operations using *speedup* and *scaleup* metrics. These two terms were originally introduced in [20] to quantify the scalability characteristics of Gamma, a parallel SQL system, by focusing on the average response time of queries. We adapt the definition of these terms for use with BG by focusing on its Social Action Rating, SoAR, of a data store. SoAR is the highest observed throughput with a data store while satisfying a Service Level Agreement (SLA) on response time. This study assumes the following SLA: 95% of requests observing a response time of 100 milliseconds <sup>2</sup> or faster. Speedup emulates a service provider with a fixed database size that becomes popular. It evaluates whether doubling the amount of resources would double system SoAR, i.e., provide the same service level agreement (SLA) on response time while processing twice as many actions per unit of time. Scaleup emulates a service provider with an increasing number of members (data set size). It quantifies whether increasing the amount of resources proportional to the size of the social graph would provide the same SoAR.

More formally, we quantify speedup by assuming a fixed size database while varying the size of the hardware platform employed by a data store. Ideally, the system SoAR should increase proportional to the increase in its hardware platform. We quantify scaleup by varying the database size proportional to the increase in the size of the hardware platform employed by the data store. Ideally, a data store should provide the same SoAR across different database and hardware platform sizes.

SoAR provides for a side-by-side comparison of the scalability of two systems, preventing the possibility of using a low throughput to quantify the scalability of a data store. This is especially true with those data stores whose throughput is a concave function of the rate of requests. To illustrate, Figure 1 shows the throughput of a single node MongoDB for a fixed workload as we increase the rate of issued requests. The values in red are the percentage of requests that satisfy our SLA requiring a response time of 100 milliseconds or faster. The SoAR of the system is established with 16 threads and is 417 actions per second. A request rate generated with 128 threads and more results in a throughput lower than that observed with one thread with less than 10% of actions satisfying the SLA. BG's SoAR

<sup>&</sup>lt;sup>2</sup>100 milliseconds is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result [29].

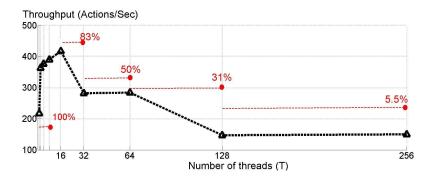


Figure 1: Throughput of a single node MongoDB with an increasing system load, i.e., higher number of threads (T), and the mix of actions shown in Table 1 using a 100K social graph with 100 friends per member.

facilitates a comparison of a data store either with itself or other data stores using a different number of nodes.

To observe a linear speedup, if a data store provides a SoAR of  $\lambda$  actions per second with one node, it should provide a SoAR of  $N\lambda$  actions per second with N nodes. To realize this ideal speedup, the data store must distribute its imposed load evenly across the N nodes, harnessing the  $\lambda$  processing capability of each node. This is challenging to realize when future requests issued by a workload are not known in advance. A random arrival time and pattern of access to the data may cause a large number of requests to collide on a single node randomly, resulting in formation of hotspots and bottlenecks. This causes the resources of a single node to become fully utilized and dictate the overall processing capability of a multi-node system. Several evaluation and applications papers have reported on this phenomena, see [23, 14] as two examples.

The primary contributions of this study are as follows. First, we use BG's SoAR to establish the speedup and scaleup characteristics of HBase and MongoDB. The main lessons are as follows. We observe a resource to become the system bottleneck and dictate the overall system performance. In this study, we assumed each node is configured with sufficient memory to prevent the disk from becoming the system bottleneck. Second, sacrificing consistency by using secondary copies of shards for query processing enhances speedup characteristics of MongoDB. With our workloads, the amount of stale data is a very small percentage of the read actions. The requirements of an application dictate if this tradeoff is acceptable. Third, a system with a superlinear speedup may not be a superior system. In our experiments, HBase exhibits a superlinear speedup by transitioning from the disk becoming fully utilized with one node to a different resource becoming the bottleneck with two or more nodes. With the same hardware and benchmark database/workload, MongoDB's performance with one node is dictated by the network bandwidth of that node. Hence, its single node SoAR is significantly higher than HBase. However, its speedup is sublinear as a function of additional nodes even though its SoAR is comparable with that of HBase for each configuration.

To further elaborate on the third lesson, we do not isolate a single variable to compare MongoDB with HBase. These two systems provide different data models with different implementations of the BG benchmark. Moreover, as detailed in Section E, we selected the operating system that maximized the SoAR of a data store with a single node.

Our primary focus is on the vertical and horizontal scalability characteristics of these two data stores as representatives of the NoSQL systems.

The rest of this paper is organized as follows. Section B presents the related work. While Section C describes the hardware platform and software architecture of HBase and MongoDB, Section D presents the logical data model used to represent BG's social graph with each data store. Section E presents the vertical and horizontal scaleup and speedup characteristics of these data stores. Brief conclusions and future research directions are outlined in Section F. To provide additional information, appendix B provides an overview of the BG benchmark.

#### B Related work

For more than two decades, different studies have reported on the performance of data stores using benchmarks such as TPC [17, 25], the Wisconsin benchmark [10], YCSB [16], OLTP-Bench [18], BG [4, 8, 5], and others [30]. [20] presents the speedup and scaleup characteristics of the Gamma system for processing SQL queries by using the service time of different queries with the Wisconsin benchmark. The TPC-C and the YCSB benchmarks use the system throughput to quantify its speedup and scaleup characteristics by varying the system size, database size, and system load. They assume a fixed imposed load with a configuration consisting of a fixed number of nodes and vary the load proportional to the configuration size.

Our use of BG's SoAR differentiates this study from others in two ways. First, we do not report on the average response time of a system as a function of either system load or the amount of resources as done by studies such as [16, 31]. In our experiments, the average response time is slightly below 100 milliseconds due to our specified SLA that requires 95% of actions to observe a response time of 100 milliseconds or faster. Second, with each configuration of a data store, BG searches the possible system loads that result in the highest throughput while satisfying the SLA. This reported SoAR is different than a throughput reading observed with a proportional increase in the system load.

Several studies report on the horizontal scalability of the throughput of NoSQL data stores as a function of the number of nodes [31, 15, 19]. The NetFlix benchmark [15] focuses on workloads dominated by write actions to show linear scalability. This is realized by performing writes that reference busy nodes on a less utilized node, writing to the destination node eventually. Our study is different because it considers workloads with a high read to write ratios. A read request must be serviced using the node with the relevant data. Thus, a load imbalance limits the horizontal scalability of a multi-node system.

In [19, 31], HBase and several other data stores are shown to exhibit a linear speedup using YCSB with different mixes of read and write actions, including one with a high read to write ratio. The results obtained with HBase [31] are consistent with ours. We speculate this is because HBase transitions from the disk being a bottleneck with a single node to the network becoming the bottleneck, providing a competitive speedup results. However, in general, the results of these studies are not comparable with ours due to our use of BG's SoAR metric with an SLA. To illustrate, the throughputs reported in [19] are established with a fixed number of threads as a function of the number of nodes N:

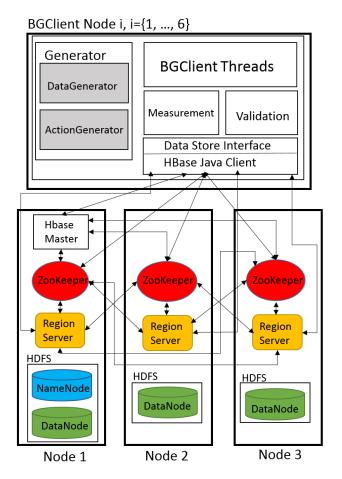


Figure 2: A 3 node deployment of HBase. Each Region Server communicates with its local HDFS instance only (edges are not shown).

 $128 \times N$ . If the resources are partially idle with 128 threads and 1 node, the system throughput will speedup linearly with additional nodes. Appendix A provides additional details on these benchmarks and their comparison with this study.

A novel feature of BG is its ability to quantify the amount of unpredictable data produced by a data store. We use this feature of BG to show how MongoDB utilizes the secondary shards for query processing to enhance its speedup characteristics while producing a very little amount of unpredictable data. This characterization is a future research direction identified in [16]. To the best of our knowledge, this is the first study to quantify this tradeoff and its impact on horizontal scalability, see discussions of Figures 9 and 10.

#### C Two Data Stores

In this section we describe the details of each data store and its deployment. Below, the term node refers to one PC. **HBase** consists of:

- 1. Hadoop 2.2.0: We deployed the Hadoop File System (HDFS) consisting of NameNode, DataNode, and a SecondaryNameNode<sup>3</sup>. The NameNode implements file system namespace operations such as opening, closing, and renaming files and directories. The DataNode manages the disk storage attached to a node, performing block creation, deletion, and replication issued by the NameNode <sup>4</sup>. This component also serves read and write requests from the file system's clients.
- 2. HBase 0.96.2 consists of a Master and one or more RegionServers. In our deployment, there is one RegionServer on each PC. There is only one Master in the system and it is responsible for balancing the load across the RegionServers. The Master also handles RegionServer failures by assigning the region to another RegionServer. A RegionServer is responsible for serving and managing Regions, where a Region (shard) is the basic element of availability and distribution for tables in HBase. We set the HDFS replication factor for a file (shard) to three. One may configure HBase to replicate data across multiple clusters. We did not exercise this feature of HBase as there is only one cluster in each of our experiments.
- 3. ZooKeeper 3.4.6 maintains the status of the functioning HBase Master and RegionServers. With a single-node deployment, there is one instance of ZooKeeper. With a multi-node deployment consisting of 3 or more nodes, three nodes host three different instances of the ZooKeeper.

We use JDK 7 and the HBase Java API version 0.96.2 client component to implement the social actions of our BG-Client, i.e., the HBase Java Client box shown in the BGClient node of Figure 2. This client component caches metadata and issues a get/put command to one or more HBase RegionServers directly.

Figure 2 shows a three node HBase deployment with an HDFS DataNode (Hadoop component) and RegionServer (HBase component) on each node. There is one NameNode (Hadoop component) and one Master (HBase component). Three of the nodes host three different instances of the ZooKeeper. A BGClient node running one BGClient (one out of twelve) issues requests to this HBase deployment using the HBase Java API. See Appendix B for a description of the components of the BGClient. Figure 3 shows HBase deployment with 6 nodes.

**MongoDB** version 2.4.9 is employed for the purposes of this evaluation. It consists of the following 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

<sup>&</sup>lt;sup>3</sup>The SecondaryNameNode is a replica of the NameNode and is constructed by HDFS. We did not configure this component for our experiments.

<sup>&</sup>lt;sup>4</sup>One DataNode resides on each PC.

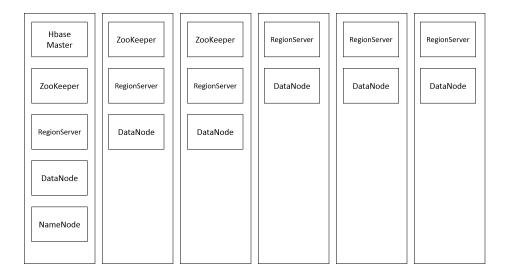


Figure 3: A 6 node configuration of HBase.

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.

We used MongoDB Java driver version 2.10.1 to implement the social actions of our BGClient, i.e., the MongoDB Java Client box shown in Figure 4.

The mongod, mongos, and config server components might be deployed in one node or across multiple nodes in a variety of ways. The performance of a single node is enhanced by constructing multiple shards and partitioning BG's social graph across the shards. This is realized by deploying multiple mongod, one mongos, and one config server on a node. Such a deployment enhances MongoDB's vertical scalability with one 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 the mix of read and write actions (see Table 1) from 4,143 with 1 mongod to 6,939 with 3 mongods. We attribute this 67% improvement to how MongoDB uses a readerswriter [29] lock per 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 actions is enhanced to improve its SoAR. There is no improvement in SoAR beyond 3 shards on a single node.

Figure 3 shows a deployment of the components of MongoDB using a 3 node shared-nothing architecture. (Appendix C describes two other possibilities, see Figures 12 and 13, that provide a lower performance.) It deploys each shard as a replica set consisting of one primary and two secondaries assigned to different nodes. This figure 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. Its corresponding secondaries are denoted as  $Rs_{i-j}$  where the value of j is either 1 or 2. Each is a mongod instance and shown as a cylinder to highlight its use of mass storage to store its assigned shard, retrieve and update documents in its shard.

With this deployment, a BGClient thread opens a connection to a mongos instance and issues all it queries to that instance. The mongos is co-located with the BGClient and each BG thread connects to the mongos hosted on its node (to minimize the impact of network communication). When the data referenced by the BGClient thread resides in a different shard, the mongos instance directs the query to the appropriate node for processing and returns the results. Using BG, we realized the configuration of Figure 4 provided a higher performance compared to the alternatives described in Appendix C. This is due to the additional overhead of message passing between the BGClients and the mongos nodes. Hence, we focus on the this configuration for the rest of this paper.

With the experimental results of Section E, 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. Section E.2 considers processing of a workload when mongos instances are configured to use either primary or secondary instances. Use of the secondary instances 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. This enhances the observed SoAR and results in a small amount of stale reads (< 0.0003%) with workloads that include write actions.

## D Two Data Models and BG's Social Graph

This section describes the logical data model of BG's database used to evaluate HBase and MongoDB in turn.

#### D.1 An Extensible Data Model of BG's Social Graph

With HBase, we analyzed two physical extensible data designs named Basic and Manual. Both store BG's social graph in 6 tables with each table consisting of a fixed number of column families. Each column family may have an arbitrary number of columns for each row which can be removed and added by an implementation of a BG action. The two designs are different in how they implement the simple analytics of BG's View Profile action. This is reflected in a different design for the Members table with the two data models. This difference enables the two designs to implement the simple analytics of BG in a different manner (see the description of the View Profile action in Appendix B).

The six tables and their descriptions are as follows:

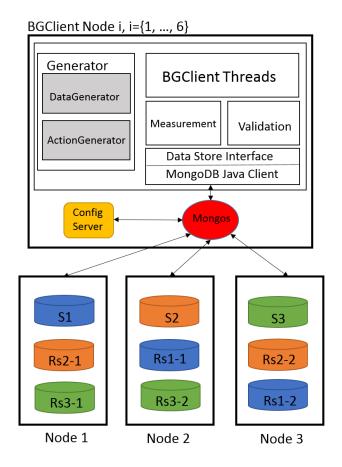


Figure 4: The MongoDB deployment used in this study.

Members	~					46							
Row Key	MemberAttributes						ThumbnailImage			ProfileImage			
userid	firstNa	me	DOB	Country		У	thumbnail		T	profile			
ConfirmedFr	ConfirmedFriends PendingFriends WallResources												
Row Key	ConfFriends		ls	Row Key Pe		PendF	endFriends		Row Key		Resources		
userid	friendid1	frie	ndid2	invit	ee ii	nviterid1	inviterid2		userid		resourceid1	reso	urceid2
Resources	Resources Manipulations												
Row Key	Resou	urceA	ttributes	Manipulat			ions		Row Key		Manipulation	nAtt	ributes
resourceid	ownerid		wallOwner	mani	oulationid1	manip	ulationid2		Resourceid manipulation	2	resourceid		body

Figure 5: Basic extensible design of BG's database.

1. Members: Maintains member's basic information. The characteristics of this table is different with Basic and Manual. With *Basic*, this table consists of three column families: MemberAttributes, ThumbnailImages and ProfileImages. MemberAttributes includes the following fields [4]: login name, password, first name, last name, gender, date of birth, join date, last login date, address, email and telephone number. The *Manual* data model extends this design with three additional columns that store the value of the simple analytics required when retrieving a member's profile, namely, number of friends (ConfFriendCount), number of pending friend invitations (PendFriendCount), and number of resources posted on this member's wall (WallResourceCount). See Figure 6.

We separated the ProfileImage and ThumbnailImage from the MemberAttributes column family as each is required by a different action. While the LF action of BG only requires the member column values along with their thumbnail image, the VP action requires the member column values along with their profile image.

The row key for this table is the memberid. This id is generated by BG when loading the data store and referenced by an action such as VP (see Appendix B). (The DataGenerator and ActionGenerator boxes in Figure 2.)

- **2. ConfirmedFriends**: Maintains the list of confirmed friends for a member. It is a separate table in order to improve concurrency. It consists of one column family named ConfFriends and for each row key (id of a member), the id of each of her friends is added as a column to this column family. Its row key is memberid.
- **3. PendingFriends**: Maintains the list of pending inviters for an invitee. Similar to ConfirmedFriends, this is a separate table to improve concurrency. It consists of one column family named PendFriends and for each row key invitee (id of the member who received an invitation), the id of each of her inviters is added as a column to this column family. Its row key is memberid.
- **4. Resources**: Maintains information about resources. It consists of two column families: ResourceAttributes and Manipulations. The ResourceAttributes maintains resource attributes such as type and date of creation (one column for each attribute) and the Manipulations column family maintains the manipulationIds for all the comments posted on a resource (id of each manipulation is inserted as a column in this column family). The row key for this table is the id of the resource. (These ids are unique and generated by BG.)
- **5.** WallResources: Maintains the mapping between a resource and the member's wall it has been posted on. It consists of one column family named Resources. For every member, the resource id for a resource which is posted on the member's wall is added as a column to the Resources column family. The row key for this table is the memberid.
- **6. Manipulations**: Maintains the information for each comment. It consists of one column family named the ManipulationAttributes. This column family contains basic information for a comment such as its date of creation and body. Each attribute is added as a column to this column family. The row key for this table is a composite consisting of its manipulationid and the resourceid for the resource the manipulation is posted on.

HBase constructs index on rows and columns in each column family (they all are sorted based on ids) automatically<sup>5</sup> [14]. In the following we describe how BG actions are implemented using the Basic and Manual designs. We present SoAR numbers using a social graph consisting of 100K members with 100 friends per member using a single

<sup>&</sup>lt;sup>5</sup>HBase does not support the concept of index structures created manually.

node deployment of HBase per discussions of Section C.

View Profile (VP): The input to this action is two member ids A and  $U_r$ . BG emulates member A visiting the profile of member  $U_r$ . A may equal  $U_r$ , emulating a socialite referencing her own profile. VP retrieves the profile information of  $U_r$ , including  $U_r$ 's attribute values and the following two simple analytics:  $U_r$ 's number of friends and number of posted resources on her wall. If the socialite is referencing her own profile (A equals  $U_r$ ) then VP retrieves a third simple analytic, A's number of pending friend invitations.

With Manual, we implement the VP action by retrieving the value of MemberAttributes and ProfileImage column families of the Members table for the row with the key equal to  $U_r$ . This reduces the simple analytics to a value lookup. With Basic, on the other hand, the implementation computes the value of simple analytics by querying the ConfirmedFriends, PendingFriends, WallResources tables using  $U_r$  as the row key. With the first two, it counts the number of member ids to compute the number of confirmed and pending friends, respectively. With the last, it counts the number of resource ids to compute the number of resources. (Note that the number of pending friends is computed only when A equals  $U_r$ .)

With a single node HBase configuration (see Section C) and a workload consisting of the VP action only, the SoAR of Manual is 35% higher than Basic (8,400 versus 6,200 actions per second). Hence, we focus on the Manual configuration in the rest of the paper. Below, we detail write actions that require the simple analytics to be maintained consistent and their impact on SoAR.

List Friends (LF): Using this action, BG emulates a socialite with member id A listing the friends of a member  $U_r$ . This action retrieves all attribute values of each friend of  $U_r$  including their thumbnail image and excluding their profile image. An implementation of this action using HBase is as follows. First, it retrieves the row key with value  $U_r$  from ConfirmedFriends table, obtaining the list of friend ids of  $U_r$ . Next, using each friendid as a row key for table Members, it retrieves the value of MemberAttributes and ThumbnailIamge of that row. This can be done in two ways with HBase. Issue one get with either each friendid or with all friendids, termed a SingleGet and a BatchedGet, respectively. BatchedGet is significantly more efficient, providing a SoAR of 320 actions per second with a single node HBase. (SingleGet provides a SoAR of 0.) We use BatchedGet for the scalability study of Section E.

View Friend Requests (VFR): This action is similar to LF with the difference that it has one input, member id A, emulating a socialite with this member id listing her pending friend requests. Its HBase implementation uses A as the key to look up the row of PendingFriends table containing all those members who extended an invitation to A, i.e., a list of inviterids. Similar to the discussion of LF, one may retrieve the profile information of each inviterid either one by one or as a set, with the latter providing the highest performance.

Invite Friend (IF): A socialite with memberid A uses this action to extend a friend invitation to a member with id  $U_r$ . Its HBase implementation inserts A as an inviter column of the column family of the PendingFriends table row with key  $U_r$ . With the Manual design, the implementation of IF must also increment the PendFriendCount column for the Members row with key  $U_r$ .

Accept Friend Request (AFR): A socialite with memberid A accepts the friendship invitation from a member with id  $U_r$  using AFR. Its HBase implementation performs two column inserts using the ConfirmedFriends table: 1) inserts  $U_r$  as a column of the row with key A, and 2) inserts A as a column of the row with key  $U_r$ . Next, it removes  $U_r$  from the inviter column of the PendingFriends row with key A. Finally, if the Manual design is employed, this implementation must also update two different rows of the Members table: 1) increments the ConfFriendCount column of the row with key  $U_r$ , and 2) increments the ConfFriendCount column and decrements PendFriendCount of the row with key A.

**Reject Friend Request (RFR)**: A socialite with memberid A rejects the friendship invitation from a member with id  $U_r$  using RFR. Our implementation of this action removes  $U_r$  from the inviterid column of the PendingFriends row with the key A. With Manual, RFR also decrements the PendFriendCount of the Members row with the key A.

Thaw Friendship (TF): A socialite with memberid A unfrineds a member with id  $U_r$  using TF. Our HBase implementation of this action issues the following two update commands to the ConfirmedFriends table: 1) deletes A from the ConfFriends column family of the row with the key  $U_r$ , and 2) deletes  $U_r$  from the ConfFriends column family of the row with the key A. With Manual, this implementation must also decrement the ConfFriendCount of two rows of the Members table corresponding to the keys A and  $U_r$ .

View Top-k Resources (VTR): This action enables a socialite with memberid A to retrieve and display the top k resources posted on her wall. An application may provide a different definition of "Top" such as highest number of "likes", posted recently, or simply their IDs. This action retrieves all attribute values for each resource. Our HBase implementation of this action first retrieves the row key with value A from the WallResources table. This retrieves the list of resource ids posted on A's wall. Next, the top k resources are selected. Finally it uses each resourceid in the list as the row key for Resources table to retrieve the attributes for each resource. Similar to the discussion of the LF action this can be done in two ways with HBase: issue one get with either each resourceid (SingleGet) or with all resourceids in the list (BatchedGet).

View Comment On Resource (VCR): A socialite with memberid A displays the comments (manipulations) posted on a resource uniquely identified by RID. Its HBase implementation is as follows. First, it obtains the list of manipulationids on the resource identified by RID by retrieving the Manipulation column family of the row with RID as its key from the Resources table. Next, it uses a composite consisting of its manipulationid and its resourceid as the row key for the Manipulations table to retrieve the attributes of the manipulations. This can be done either using the SingleGet or BatchedGet approaches as described with LF.

**Post Comment On Resource (PCR)**: A socialite with memberid A uses this action to post a comment identified by MID on a resource identified uniquely by RID. Its HBase implementation performs a row and a column insert: 1) inserts a new row with a composite key consisting of RID and MID into the Manipulations table corresponding to the new comment and its attributes, and 2) inserts MID as a column of the Manipulations column family of the row with key RID in the Resources table.

**Delete Comment From Resource (DCR)**: This action enables socialite A to delete a unique comment identified by

<sup>&</sup>lt;sup>6</sup>In BG the VTR action computes the top k by sorting the rersourceids in an descending order and selecting the top k.

Row Key			ThumbnailImage	ProfileImage					
userid	firstName	DOB		ConfFriend Count	PendFriend Count	WallResource Count	Country	thumbnail	profile

Figure 6: Manual extends the MemberAttributes of Basic's Members table with the three columns (shown in red) to store the value of simple analytics.

```
Members:{
                                       Resource:{
          "userid":""
                                                 "rid":""
          "username":""
                                                 "creatorid":""
          "pw":""
                                                  "walluserid":""
          "firstname":""
                                                  "type":""
                                                  "body":""
          "lastname":"
          "gender":""
                                                  "doc":""
          "dob":""
                                      }
          "jdate":""
          "address":""
                                       Manipulations:{
          "email":""
                                                 "mid":"
          "tel":""
                                                  "rid":""
          "pendingFriends":[]
                                                  "modifierid":""
          "confirmedFriends":[]
                                                  "type":""
          "profileImage":[]
                                                 "content":""
          "thumbnailImage":[]
                                                 "timestamp":""
          "WallResourceIds":[]
                                      }
}
```

Figure 7: The JSON data model of BG's social graph identified by [8] as providing the highest SoAR.

MID posted on one of her owned resources identified by RID. Its HBase implementation performs a row and a column delete as follows. First, it deletes the row with the composite key equal to RID-MID from the Manipulations table. Next, it removes MID from the Manipulations column family of the Resources table for the row with the key RID.

#### D.2 A JSON Data Model of BG's Social Graph

The alternative physical JSON designs of BG's social graph using MongoDB are analyzed in [8] with the same scrutiny as the discussion of HBase in Section D.1. We do not repeat this discussion. Instead, we assume the design that was identifed in [8] as providing the highest performance (see Figure 7) for our scalability study of Section E. Two key features of this design are as follows. First, it stores the profile and thumbnail images of each member as byte arrays in the JSON document that represents each member. Second, it computes the simple analytics by looking up the size of the arrays that maintain the list of friends, pending friends, and resourceids.

	1 Net	work Card	2 Network Cards			
	HBase	MongoDB	HBase	MongoDB		
VP	8,302	7,937	16,569	15,880		
LF	322	289	630	593		

Table 2: SoAR (actions/second) with a 100K member social graph and 100 friends per member. One node with 4 hyperthreaded cores.

### **E** Scalability Results

This section describes vertical and horizontal scalability of HBase and MongoDB in turn. While the discussion of horizontal scalability discusses both speedup and scaleup, we focus only on speedup with vertical scalability. This is because both HBase and MongoDB use index structures and the scaleup numbers are identical to speedup numbers when either the network cards or the CPU cores are the bottleneck. While changing the amount of memory or the number of disks is a plausible vertical scaleup experiments with results different than speedup experiments, we did not consider it as it would be easier to change the number of nodes horizontally.

For the purposes of this evaluation, our shared-nothing architecture consists of twelve Dell PCs purchased at approximately the same time. Each PC consists of an Intel i7-4770 CPU (quad core, 3.8 GHz), 16 Gigabytes of memory, a 1 Terabyte disk drive, and 1 network interface card. (We vary the number of CPU cores and network cards in Section E.1 to evaluate vertical scalability of a data store.) We configured each PC to be dual boot with either Ubuntu 13.10 or Windows Server 2012 Data Center Edition: Ubuntu is used for HBase and Windows for MongoDB <sup>7</sup>.

#### **E.1** Vertical Scalability

With *vertical* scaling, we considered an increase in the number of either the networking cards or the CPU cores of a single node <sup>8</sup>. Table 2 shows the observed SoAR with a workload consisting of either VP or LF action only using MongoDB and HBase. This workload utilizes the bandwidth of the networking cards fully, dictating the observed SoAR. Doubling the number of networking cards from one to two causes the SoAR to double with both HBase and MongoDB. Moreover, the observed SoAR with HBase and MongoDB is similar (less than 4% difference<sup>9</sup>) because the same network card(s) are used to rate each data store.

Table 3 highlights the significance of a resource being a bottleneck in order for the SoAR of a single node deployment to improve as a function of the resource size. We use the operating system to change the number of available cores from 1 to 4. The VP action observes minimal benefit as the network bandwidth is fully utilized. Similarly, with MongoDB and the LF action, the network bandwidth is the bottleneck resource and increasing the number of cores provides a negligible improvement. With the LF action using HBase, the 1 hyperthreaded CPU core is fully utilized.

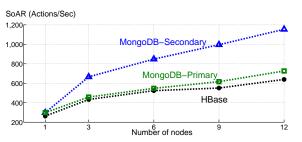
<sup>&</sup>lt;sup>7</sup>Our experiments showed that the performance of a single node MongoDB on Windows is higher than that on Ubuntu. So for our experiments we configure MongoDB to run on Windows. This decision is valid as the purpose of this study is not to compare the two data stores with one another but to understand their scalability characteristics.

<sup>&</sup>lt;sup>8</sup>We deployed both data stores in their standalone mode on a single node.

<sup>&</sup>lt;sup>9</sup>We attribute differences less than 15% to experimental noise.

	1 Hypert	hreaded Core	4 Hyperthreaded Cores			
	HBase	MongoDB	HBase	MongoDB		
VP	15,685	15,881	16,569	16,020		
LF	314	591	630	594		

Table 3: SoAR (actions/second) with a 100K member social graph and 100 friends per member. One node with 2 network interface cards.



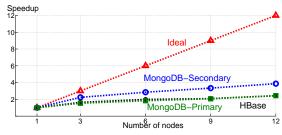


Figure 8a: SoAR.

Figure 8b: Speedup.

Figure 8: LF action with a 1M member social graph and 100 friends per member.

An increase to 4 hyperthreaded cores doubles the SoAR of HBase. This speedup with HBase is due to a transition from the 1 hyperthreaded CPU core being fully utilized to the 2 network interface cards becoming fully utilized with 4 hyperthreaded cores.

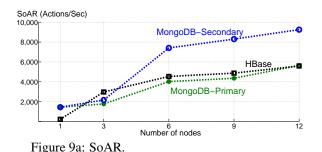
#### **E.2** Horizontal Scalability

This section characterizes the speedup and scaleup characteristics of HBase and MongoDB. In these experiments, with N nodes, each LF action is directed to a single node (instead of all the N nodes) almost always for the following reason. Both HBase and MongoDB are configured to range partition the social graph using the id of members. And, BG constructs the social graph by assigning members (i  $\pm$  j)%M as friends of member i. This means a member and her friends are almost always in one shard. A future research direction is to investigate hash partitioning and whether it enhances speedup and scaleup of system SoAR, see Section F.

#### E.2.1 Speedup

This experiment assumes a social graph consisting of 1M members with 100 friends per member with one node. Next, it increases the size of the system to 3, 6, 9, and 12 nodes. The same 1M member social graph is employed to compute SoAR of the system. We investigated speedup with workloads consisting of VP action only, LF action only, and the mix of actions shown in Table 1. The characteristics observed with VP is identical to the mix of actions. Hence, we present the results obtained with the LF action and the mix of actions in turn.

Figure 8b shows the speedup observed with HBase and MongoDB with a workload consisting of the LF action only. With the different configurations of HBase, the network interface card of a single node becomes fully utilized



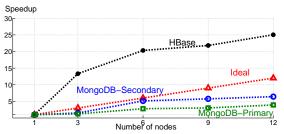


Figure 9b: Speedup.

Figure 9: Mix of actions with a 1M member social graph and 100 friends per member.

to dictate system performance. We configured MongoDB to process request using its secondary shards. This causes the network interface card of two to three nodes to become fully utilized, providing a higher SoAR and speedup when compared with HBase.

Figure 9b shows the observed speedup with HBase and MongoDB with the mix of read and write actions shown in Table 1. HBase provides a superlinear speedup as the disk has a queue with one node and this queue disappears with 3 and more nodes. Additional nodes enable HBase to assign a fraction of the social graph to each node while providing it with a larger amount of total memory. For example, with 3 nodes, the social graph assigned to each node consists of approximately 300K member with a total amount of 48 Gigabyte of memory. By eliminating disk I/Os with additional nodes, HBase observes a huge performance boost to provide a superlinear speedup.

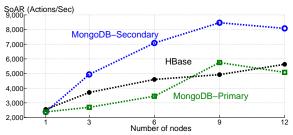
With MongoDB, the disk is not the bottleneck with one node. Instead, its SoAR is dictated by a combination of the CPU cores and network bandwidth becoming fully utilized. With additional nodes, the speedup observed with MongoDB is sublinear as the network bandwidth of one node becomes fully utilized to limit its speedup.

A superior speedup characteristic does not imply a superior overall performance. Even though HBase scales superlinearly, its SoAR with a single node is inferior to MongoDB as it observes a disk queue, see Figure 9b. With additional nodes, similar to MongoDB, the network bandwidth of a single node becomes fully utilized to dictate its SoAR. SoAR of HBase with 3, 6, and 9 nodes is is only slightly better than MongoDB. The SoAR observed with MongoDB with 12 nodes using primary shards is comparable to the SoAR observed with HBase with 12 nodes.

When MongoDB is configured to use secondary shards for query processing, its SoAR increases almost two folds while producing a very small percentage (< 0.0003%) of reads observing unpredictable data. Now, MongoDB balances the load across two secondary replicas with the network interface card of two (at times three) nodes become fully utilized to dictate the observed SoAR with MongoDB, explaining its enhanced performance.

#### E.2.2 Scaleup

This experiment establishes SoAR of a single node deployment with a social graph consisting of 100K members and 100 friends per member. Next, it increases both the size of the system and the number of members in the social graph proportionally to 3 nodes with 300K members, 6 nodes with 600K members, 9 nodes with 900K members, and 12



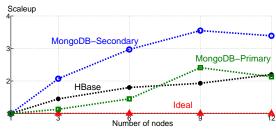


Figure 10a: SoAR. Figure 10b: Scaleup.

Figure 10: Mix of actions with a  $N \times 100$ k members social graph and 100 friends per member (N = Number of nodes).

nodes with 1.2M members. (The number of friends per member ( $\phi$ ) with each social graph is fixed at 100 friends.) We divide the SoAR observed with each multi-node configuration with the SoAR observed with one node to compute the scaleup characteristics. Ideally, the SoAR of a system should remain constant as a function of this increase in system and social graph size. Both HBase and MongoDB scale SoAR up superlinearly with workloads consisting of VP action only, LF action only, and the mix of actions shown in Table 1. Below, we present results for the mix of action only.

Figure 10b shows the scaleup with MongoDB and HBase. Both provide a scalability that is better than ideal. We consider MongoDB when configured to process the workload using either its primary or secondary shards. Use of secondary shards provides a superior scaleup characteristic and a small amount of stale data (<0.0003%). With this deployment, the network interface card of two (at times three) nodes becomes fully utilized. With HBase, the network interface card of a single node is fully utilized to dictate its scaleup.

Even though the network interface card of two or three nodes becomes fully utilized in all experiments with multiple nodes, the other nodes continue to process a fraction of system load. For example, with a 12 node MongoDB deployment using secondaries, once the network interface card of three nodes is 100% utilized, the network cards of the other nodes in the system is 15% to 45% utilized. This explains why the SoAR scaleup with 12 nodes is higher than that with three nodes. The same explanation applies to HBase and its reported SoAR scaleup being higher than two even though the network card of one node is fully utilized.

#### **F** Conclusion and Future Research Directions

This study analyzes scalability characteristics of HBase and MongoDB for processing interactive social networking actions using BG. These simple operations read and write a small amount of data. We quantified their speedup and scaleup with a multi-node deployment of each data store. While both data stores scale superlinearly, their speedup is limited by the resources of a few nodes out of many becoming fully utilized. MongoDB's use of secondary shards for query processing reduces the impact of this limitation to enhance its speedup characteristic. This is at the expense of producing a small amount of stale data.

Our future research directions are several folds. First, we plan to investigate alternative physical designs of BG's social graph with each data store and its impact on the reported speedup numbers. An example alternative may store images in a file system instead of the data store [32, 9, 8]. Second. we plan to study the scalability characteristics of different classes of data stores such as key value stores. Third, we intend to extend this study to analyze scalability of more complex social networking actions such as feed following [7] and get shortest distance between two members [22, 11, 1, 26, 2]. With these, we intend to compare the hash partitioning of BG's social graph with its range partitioning. Fourth, we plan to further analyze the speedup characteristics of MongoDB with additional secondary shards. In this study, we fixed the number of secondary shards (mongod instances) at two and varied the number of nodes. A key question is how the SoAR speedup behaves with additional secondary shards. For example, with a 12 node configuration, would 11 secondaries enable SoAR to speedup linearly relative to a one node configuration? Even though the frequency of write actions is low, there might be a point of diminishing returns where additional number of secondaries may not enhance speedup characteristics of MongoDB.

#### References

- [1] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kostev, and I. Toma. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. SIGMOD Rec., 43:27–31, March 2014.
- [2] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J. Larriba-Pey. Benchmarking Database Systems for Social Network Applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013.
- [3] S. Barahmand. Benchmarking Interactive Social Networking Actions, Ph.D. thesis, Computer Science Department, USC, 2014.
- [4] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *Proceedings of 2013 CIDR*, January 2013.
- [5] S. Barahmand and S. Ghandeharizadeh. Expedited Rating of Data Stores Using Agile Data Loading Techniques. In *Proceedings of the 22Nd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '13, pages 1637–1642, 2013.
- [6] S. Barahmand and S. Ghandeharizadeh. Benchmarking Correctness of Operations in Big Data Applications. Proceedings of IEEE MASCOTS, 2014.
- [7] S. Barahmand and S. Ghandeharizadeh. Extensions of BG for Testing and Benchmarking Alternative Implementations of Feed Following. ACM SIGMOD Workshop on Reliable Data Services and Systems (RDSS), June 2014.

- [8] S. Barahmand, S. Ghandeharizadeh, and J. Yap. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. In *CIKM*, 2013.
- [9] 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.
- [10] D. Bitton, C. Turbyfill, and D. J. Dewitt. Benchmarking Database Systems: A Systematic Approach. In *VLDB*, pages 8–19, 1983.
- [11] P. Boncz. LDBC: Benchmark for Graph and RDF Data Management. IDEAS, October 2013.
- [12] R. Cattell. Scalable SQL and NoSQL Data Stores. SIGMOD Rec., 39:12–27, May 2011.
- [13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2006.
- [14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
- [15] A. Cockcroft and D. Sheahan. Benchmarking Cassandra Scalability on AWS Over a Million Writes per Second. November, 2011, http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html.
- [16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.
- [17] Transaction Processing Performance Council. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp.
- [18] E. Difallah D, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.
- [19] DataStax and End Point Corporation. Benchmarking Top NoSQL Databases. White paper, 2013, http://www.datastax.com/resources/whitepapers/benchmarking-top-nosql-databases.
- [20] 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.
- [21] A. Floratou, N. Teletria, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the Elephants Handle the NoSQL On-slaught? In *VLDB*, 2012.
- [22] S. Ghandeharizadeh, R. Boghrati, and S. Barahmand. An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions. In *TPC Technology Conference*, 2014.
- [23] S. Ghandeharizadeh and D. J. DeWitt. A Multiuser Performance Analysis of Alternative Declustering Strategies. In *ICDE*, 1990.

- [24] 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.
- [25] J. Gray. The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann 1993, ISBN 1055860-292-5.
- [26] F. Holzschuher and R. Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, 2013.
- [27] L. Kleinrock. Queueing Systems, volume II: Computer Applications. Wiley Interscience, 1976.
- [28] MongoDB Inc. Manage Journaling, MongoDB Administration Tutorial, 2011, http://docs.mongodb.org/manual/tutorial/manage-journaling/.
- [29] J. Nielsen. Usability Engineering. Academic Press Inc., 1993.
- [30] ObjectWeb. RUBBoS: Bulletin Board Benchmark, http://jmob.ow2.org/rubbos.html.
- [31] T. Rabl, M. Sadoghi, H. Jacobsen, S. Gomez-Villamor, V. Muntes-Mulero, and S. Mankowskii. Solving Big Data Challenges for Enterprise Application Performance Management. *VLDB*, 2012.
- [32] 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.
- [33] M. Stonebraker and R. Cattell. 10 Rules for Scalable Performance in Simple Operation Datastores. *Communications of the ACM*, 54, June 2011.
- [34] M. R. Stonebraker. The Case for Shared-Nothing. In *Proceedings of the 1986 Data Engineering Conference*. IEEE, 1986.

#### **A Extended Related Work**

This section describes three benchmarks that use the throughput of NoSQL data stores to characterize their scalability in greater detail.

The NetFlix benchmark [15] is dominated by write operations, exhibiting linear scalability. This is possible because the clients pick Cassandra nodes at random to perform the write operation. If the chosen node is the wrong one, it will act as a coordinator to send replicas of the data to the correct nodes that are identified using a consistent hash of the row key being inserted. This approach cannot be used with read operations due to affinity of requests for data that resides on a subset of the nodes. The scalability characteristics presented here are novel and different because our workload is dominated by read actions that correspond to social networking actions.

The scaleup characteristics of several NoSQL data stores are reported in [31]. This study employs the YCSB benchmark with workloads consisting of different mixes of read and write actions. It reports on linear scalability of HBase with an increasing number of nodes with workload R consisting of 95% read and 5% write actions. While [31] does not explain why the throughput scales linearly, a possible speculation is that HBase transitions from the disk being the bottleneck with one disk to another resource becoming the bottleneck.

Finally, the Datastax study [19] employs YCSB to characterize scalability of Cassandra, HBase and MongoDB with a different number of nodes, ranging from 1 to 32. Similar to [31], it considers workloads with a different mix of read and write actions, including a read-mostly workload with 95% read actions. It shows different systems speedup linearly as a function of the number of nodes. This is realized by varying the number of YCSB threads as a function of the number of nodes N; T = 128\*N. The reported linear speedup appears to contradict the sublinear speedups presented in this study and attributed to the formation of bottlenecks. A possible explanation for this discrepancy is as follows. If the system is under-utilized with 128 threads and 1 node, system throughput will speedup linearly with additional nodes. Another possibility might be that the bottleneck resource is the disk with 1 node and transitions to some other resource with additional nodes. [19] lacks an explanation for its results and does not report on either a bottleneck resource or utilization of system resources.

In general, the results of [15, 31, 19] are not comparable with those reported in this study because the SoAR metric constrains the observed system throughput with a SLA. Other novel lessons reported in this study are as follows. First, we demonstrate SoAR speedup of both data stores is limited by the resources of a few (one to three) nodes becoming the bottleneck. Second, we show one may configure MongoDB to use its secondary shards to enhance its speedup characteristics while incurring a small amount of stale data. Third, we show a system that exhibits a superlinear SoAR speedup does not necessarily provide a superior SoAR. Finally, we analyzed a shared-everything (SE) deployment of both data stores and compared it with a shared-nothing (SN) deployment, showing the SE to provide a higher SoAR with a skewed patterns of data access.

#### **B** Overview of BG

This section provides an overview of BG [4, 8, 5] as it relates to this study.

Figure 11 shows the conceptual design of BG's social graph. For the purposes of this study, we simplify this design to focus on the Members entity set and its friendship relationships. Each member is identified by a unique id and has the following attributes: login name, password, first name, last name, gender, date of birth, join date, last login date, address, email, telephone number, profile image, thumbnail image. One may configure BG with different image sizes. This study assumes a 12 KB profile image and a 2 KB thumbnail image for each member. The Invite relationship set describes a member A extending a friendship invitation to another member  $U_r$ . Once  $U_r$  accepts A's invitation, this pending friend invitation becomes a confirmed friendship relationship. We refer the interested reader to [4, 3] for a description of the rest of the conceptual design. Section D.1 provides a brief description of this remainder in order to

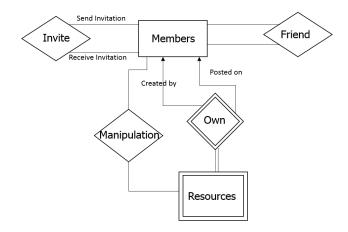


Figure 11: BG's conceptual schema.

present a physical implementation of a complete set of BG's actions using HBase.

BG employs a closed emulation model to generate a workload for a data store. It uses a thread to emulate a Member A who performs an action on another member or a resource, see the BGClient Threads box of Figure 2. This member who is actively engaged in performing a social action is termed a *socialite*. A thread does not emulate another socialite until the pending action of the current socialite is processed by the data store. BG controls the load imposed on a data store by varying the number of threads used to emulate concurrent socialites performing actions.

This study focuses on eight<sup>10</sup> actions that either read or write a small amount of data from the social graph. (These actions are equivalent to simple operations of Section A.) These include:

- 1. View Profile, VP, a read action: Socialite A previews the profile of the member with id  $U_r$ , retrieving  $U_r$ 's member attribute values including her 12 KB profile image. In addition, this action retrieves the following simple analytics:  $U_r$ 's number of friends and number of resources posted on her wall. When the socialite references her own profile  $(A=U_r)$ , this action also retrieves  $U_r$ 's number of pending friend invitations.
- 2. List Friend, LF, a read action: Socialite A previews the list of friends of the member with id  $U_r$ , retrieving each friend's member attribute values including her 2 KB thumbnail image.
- 3. View Friend Request, VFR, a read action: Socialite A lists her pending friend invitations, retrieving each inviter's member attribute values including her 2 KB thumbnail image.
- 4. Invite Friend, IF, a write action: Socialite A extends a friendship invitation to the member with id  $U_T$ .
- 5. Accept Friend Request, AFR, a write action: Socialite A accepts the friendship invitation extended by the member with id  $U_r$ .

<sup>&</sup>lt;sup>10</sup>BG consists of sixteen actions [4, 7, 22].

- 6. Reject Friend Request, RFR, a write action: Socialite A rejects the friendship invitation extended by the member with id  $U_r$ .
- 7. Thaw Friendship, TF, a write action: Socialite A unfriends the member with id  $U_r$ .
- 8. View Top-k Resources, VTR, a read action: Socialite A retrieves and displays the top k resources posted on her wall.

Section D.1 provides additional details on each of these actions in the context of their implementation using HBase. We consider workloads consisting of VP action only, LF action only, and a mix of actions as shown in Table 1.

BG is a stateful benchmark that emulates valid actions always (using its ActionGenerator component of Figure 2). For example, it does not emulate a socialite A accepting a friend invitation by Member  $U_r$  unless Member  $U_r$  has extended a friend invitation to Member A. Moreover, BG is an extensible framework and data store agnostic, exposing its schema and an implementation of its actions to be tailored to a data store, see the "Data Store Interface component" of Figure 2. This interface implements sixteen actions as detailed in Section D.1 even though we emphasize system scalability with eight BG actions. Finally, BG is a scalable benchmarking framework and may employ multiple nodes to generate a workload for the fastest data stores. In the experiments conducted with a 12 node deployment of HBase and MongoDB, we used six BGClient nodes hosting 12 BGClients to generate requests. In all reported experiments, we ensured the network and CPU cores of the BGClient nodes are not a bottleneck, i.e., their utilization is less than 100% always.

## C Various MongoDB Deployments

This section shows two alternative configurations for MongoDB. These configurations are possible when one assumes BG is emulating a web server accessing the data in the data store. With these configurations the mongos instance is co-located with a shard, see Figures 12, 13. Each shard is deployed as a replica set consisting of one rpimary and two secondaries.

Using BG, we observed the configuration of Figure 12 to provide a slightly lower SoAR with the mix of read and write actions when compared to the configuration of Figure 13.

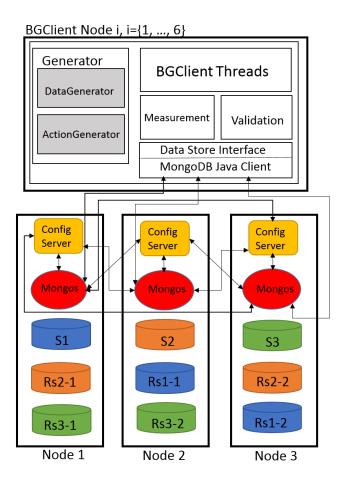


Figure 12: One mongos and config server per data store node. A mongos may communicate with all mongod instances depicted as a cylinder (edges are not shown).

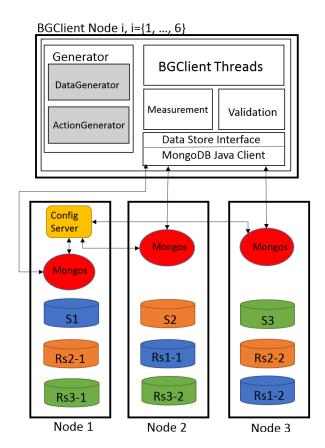


Figure 13: MongoDB with one config server for the cluster. A mongos may communicate with all mongod instances depicted as a cylinder (edges are not shown).