Benchmarking Correctness of Operations in Big Data Applications[*]

Sumita Barahmand and Shahram Ghandeharizadeh

Database Laboratory Technical Report 2014-05

Computer Science Department, USC

Los Angeles, California 90089-0781

{barahman, shahram}@usc.edu

May 22, 2014

Abstract

With a wide variety of big data applications, the past few years have witnessed an increasing number of data stores with novel design decisions that may sacrifice the correctness of an application's operations to enhance performance. This paper presents our work-in-progress on a framework that generates a validation component. The input to the framework is the characteristics of an application. Its output is a validation module that plugs-in to either an application or a benchmark to measure the amount of unpredictable data produced by a data store.

A 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 big data applications. 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 [1] and we are aware of 10 more[1] since that writing. Many of these new systems are referred to as "NoSQL" data stores. While there is no consensus on one definition of NoSQL, the NoSQL systems surveyed in [1] generally do not provide strong consistency guarantees, i.e., ACID transactional properties. Instead, they may opt for Basically Available, Soft state and Eventually consistent (BASE) properties [1, 12, 13]. It is important for a benchmark to quantify the amount of stale, erroneous, and incorrect

[*]This research was supported in part by a 2013 Google Ph.D. Fellowship. A shorter version of this paper appeared in IEEE MASCOTS, Paris, France, 2014.

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

data (termed *unpredictable* data) produced by a data store in addition to traditional performance metrics such as response time and throughput.

Consistency of values reflected by all replicas of a data item is different than the *correct* execution of operations that read and write these values. While the latter is the responsibility of a programmer, a data store may employ design decisions that introduce delayed propagation of updates and undesirable race conditions, impacting the correctness of operations. Hence, a system may have N replicas of a data item with one unpredictable value. For example, cache augmented data stores incur undesirable race conditions that produce stale data [2, 4] and result in dirty reads [3].

There are many metrics to quantify the consistency of data item replicas with a data store. Some are as follows:

1. Probability of observing an accurate value a fixed amount of time, say t seconds, after a write occurs [5], termed as freshness confidence.

2. Percentage of reads that observe a value other than what is expected, quantified as the percentage of unpredictable data [6, 12].

3. Amount of time required for an updated value to be visible by all subsequent reads. This is termed inconsistency window [5, 14].

4. Probability of a read observing a value fresher than the previous read for a specific data item, termed monotonic read consistency [5].

5. Age of a value read from the updated data item. This might be quantified in terms of versions or time [7, 8].

6. How different is the value of a data item from its actual value? For example, with a member with 1000 friends, a solution may return 998 friends for her whereas a different solution may return 20 friends. An application may prefer the first [7, 8].

A variant of the first two metrics are quantified by benchmarking frameworks such as [9, 10, 6, 12]. BG [6, 12] is the only benchmark to evaluate the correct execution of operations in support of interactive social networking operations. This work-in-progress paper abstracts BG's validation mechanism to realize a modular and configurable framework that evaluates the correctness of diverse applications. The input to the framework is the characteristics of either an application or an application specific benchmark. The output of the framework is a module that plugs-into either the application or the benchmark to quantify the amount of unpredictable data due to incorrect execution of operations. Inconsistent replicas of a data item may be one cause of an incorrect execution.

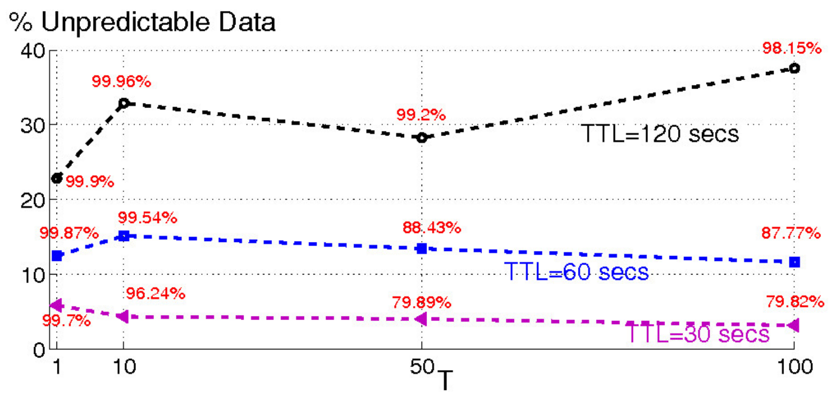

Figure 1: Amount of unpredictable data observed with a relational database management system (RDBMS) augmented with memcached.

An application's operations may update an attribute value of one or more data items. If the values produced by these operations are not stored in the database correctly then all subsequent reads may be unpredictable. Figure 1 shows the amount of unpredictable data observed with a relational database management system (RDBMS) augmented with memcached as a function of the system load emulated by a fixed number of threads, T, issuing requests. The cached entries have a fixed time to live (TTL) that is varied from 30 to 60 and 120 seconds. The numbers in the red denote the percentage of requests that observe a response time faster than 100 milliseconds. This percentage increases for different system loads with higher TTL values because servicing queries by looking up their results is faster than processing them using the RDBMS [2, 3, 4]. At the same time, a higher TTL value results in a higher percentage of read operations observing unpredictable data. This is because, while all write operations are directed to the RDBMS that provides ACID semantics, a higher TTL increases the probability of a write operation updating the relational data resulting in a stale version in the cache.

The proposed transformative framework is a component of a benchmark generator. It is appropriate for those applications with diverse use cases such as data sciences [11], empowering an experimentalist to quickly author and deploy application specific benchmarks. The rest of this paper is organized as follows. Section B presents the concept of unpredictable data and one way of quantifying it. An implementation is detailed in Section C. Finally we conclude in Section D.

B Unpredictable Reads

Consider the concurrent execution of a read operation with multiple concurrent write operations from the perspective of a client issuing them, see Figure 2. Assuming these operations reference the same data item D_i and all of them complete, the value retrieved by the read operation R_1 depends on how a data store

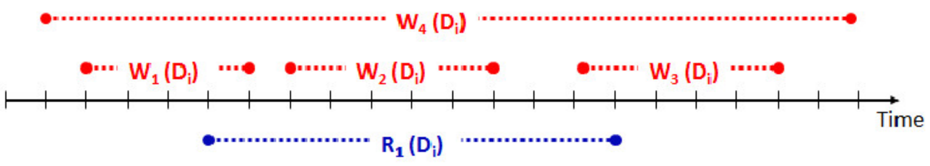

Figure 2: There are four ways for a write of D_i to overlap a read of D_i.

serializes their execution. One possibility is a serial execution of the concurrent operations in isolation [18]. Thus, the value retrieved by R_1 is a set of possible values depending on how the data store serializes the read operation relative to the write operations. In Figure 2, the read operation might be serialized to have occurred either prior to all the write operations, subsequent to each write operation incrementally (considering all possibilities of W_4 with other write operations), or after all write operations. As long as R_1 retrieves a value that is in the set, the data store has produced a correct value. Otherwise, R_1 has observed unpredictable data.

This section describes *validation* as the process of quantifying the amount of unpredictable data produced by a data store. It presents several possible design choices for validation, motivating the need for a modular and configurable component. We use a social networking application to illustrate how each module is used. We use the term application and benchmark interchangeably because our target benchmark is intended to be application specific. Moreover, a modular and configurable validation process should plug-in to either an application or a benchmark.

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 technique may implement either extremes, or a hybrid design that reports the amount of unpredictable data after some delay based on its allocated resources, see Section C.

In order to plug-in to diverse application-specific benchmarks, validation must support both primitive and custom data types. Primitive data types include int, Char, String, Boolean, etc. Custom data types include user-defined objects such as calendar events, and arbitrary array types such as lists (allow duplicate values), sets (no duplicate), and their sorted variants [15]. For example, with a social networking benchmark, the number of friends of a member is an int. However, the news feeds produced for a member is a set sorted either chronologically, by topic, or some other custom ranking metric.

To compute the amount of unpredictable data, the validation technique must be configurable with two key inputs. First, a mechanism to uniquely identify a data item, read and write operations that manipulate data items, and how the write operations change the state of a data item from one valid state to another. With a social networking benchmark, a data item might be a member profile identified by a member's unique identifier. Example read and write operations might be View Profile (VP) and Accept Friend Request (AFR), respectively. When the AFR operation is issued by Member A to confirm friendship with Member B, the write operation increments the number of friends of each member by one. This is an example of a write operation changing the state of a member profile from one valid state to another. Assuming the VP operation retrieves the number of friends of a member and executes concurrently with AFR, its order in a serial schedule dictates its observed number of friends. (One friend fewer when serialized before AFR.)

Specification of data items and dependence of operations apply to custom data types. For example, with the feed displayed to member A (an array type), when a write operation is issued by Member A to unfriend B then B's news must not appear in A's feed [15]. A challenge is how to specify these in an intuitive manner. Effective user interfaces are an integral part of our undertaking.

Second, there must be sufficient input information for the validation process to compute the value that should have been observed by a read operation. A key input is the initial value of a data item. This input might be in the form of analytical models. To illustrate, consider a social networking benchmark and a social graph consisting of 1000 members and 100 friends per member with members numbered 0 to 999. An analytical model may describe the initial benchmark database as Member i being friends with members (i+j)%1000 where the value of j varies from 1 to 50. (The torus characteristics of the mod function guarantees 100 friends per member.)

Once the validator is provided with a stream of read and write operations with well defined start and end times, it computes the amount of unpredictable data. One may configure the validation process to be applicable to new data items generated by the application, rendering the initial state of existing data items irrelevant. In this mode, the validator ignores all those operations that reference data items with original values unknown to it. With new data items, the validation process maintains the initial value of the data item to compute whether a read observes unpredictable data. This renders the resulting validation module appropriate for use within existing applications such as Google+ and Facebook.

C An Implementation

The abstraction of Section B is based on our design and implementation of BG's validation technique. BG is a scalable benchmarking framework that implements validation as an offline technique, quantifying the amount of unpredictable data after the benchmark completes. This decision minimizes the amount of resources required by BG to generate a load that fully utilizes the resources of a data store. An online variant would enable an experimentalist to view the amount of unpredictable data with other reported online metrics (such as throughput and response time). However, it may require additional CPU and memory resources.

Today's BG generates a stream of log records [6, 12] to provide the validation module with the details of the operations performed during the benchmarking phase. These are categorized into read and write log records and identify operations issued by the BG benchmark along with their start and end timestamps. The read log records maintain the value observed from a data store. The write log records identify either a change or a new absolute value for an attribute of a manipulated data item. The offline validation process uses these log records in combination with the initial state of the database to compute the amount of unpredictable reads. The initial state of the social graph is provided by a fixed number of members, friendships per member, and resources per member. Analytical models such as the mod function (see Section B) describe the friendship relationship between members.

Generation of these log records is an example of a mechanism to make the validation process aware of the operations performed by a benchmark. It is trivial to direct BG's stream of log records to a process that would perform the validation in a semi-online[2] manner. One may envision a dataflow MapReduce[3] infrastructure that incrementally refines computation of the amount of unpredictable data.

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. For example, today's implementation assumes all the write log records fit in the memory of the PC performing validation[4]. This does not work with high throughput data stores that process millions of operations per second as the validation exhausts the available memory of the PC, causing its operating system to exhibit a thrashing behavior. This happens today with our experiments involving middlewares such as KOSAR [16, 17]. Even with a sufficient amount of memory, 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. Below, we describe a host of design choices for today's validation scheme. We organize these in 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. Its 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.

[2]Not completely online because there might be a delay from when the log records are produced to the time they are processed.

[3]MapReduce is natural as the records can be partitioned based on the identity of the data items referenced by a log record.

[4]An alternative is to use a persistent store such as a relational database [6]. This is more appropriate when the total size of the write log records exceeds the available memory but is slow as it involves issuing SQL queries that incur disk I/Os.

- 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 subsystem s 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. This may report unpredictable data that is not 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 such as 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 validation phase examines the value produced by a data store and determines its correctness. It may employ either a deterministic or a probabilistic model with a pre-specified confidence to determine correctness. Ideally, these models should be plug-and-play software modules. They may require initial state of the data items and a sequence of specific write actions in order to compute correctness. Hence, this phase may have dependencies on both the Runtime configuration and Log processing steps.

D Summary

This paper presents our on-going work on a general purpose framework to evaluate the correctness of operations in big data applications. It quantifies the amount of unpredictable data produced by novel data store designs that may compromise the correctness of operations due to delayed propagation of updates and undesirable race conditions. Such a transformative framework would enable an experimentalist to develop application specific benchmarks and measure the amount of unpredictable data produced by a data store quickly. It is particularly useful for data science applications [11] with diverse data sets and use case scenarios.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

[10] 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.

[11] A. Talukder, C. Greenberg. Overview of the NIST Data Science Evaluation and Metrology Plans. In *Data Science Symposium, NIST*, March 4-5, 2014.

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

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the 1st *ACM Symposium on Cloud Computing (SoCC '10)* , 2010.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In Proceedings of Twenty-first *ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, 2007.

[15] S. Barahmand, S. Ghandeharizadeh and D. Montauk. Extensions of BG for Testing and Benchmarking Alternative Implementations of Feed Following. In Proceedings of the *SIGMOD Workshop on Reliable Data Services and Systems (RDSS)*, 2014.

[16] S. Ghandeharizadeh. KOSAR: A Game Changer for SQL Solutions. Mitra LLC, 2014.

[17] S. Ghandeharizadeh. KOSAR: An Elastic, Scalable, Highly Available SQL Middleware. USC DBLAB Technical Report 2014-09.

[18] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., 1992.