

MalStone: Towards A Benchmark for Analytics on Large Data Clouds

Collin Bennett
Open Data Group
400 Lathrop Ave Suite 90
River Forest IL 60305

Robert L. Grossman^{*}
Open Data Group
400 Lathrop Ave Suite 90
River Forest IL 60305

David Locke
Open Data Group
400 Lathrop Ave Suite 90
River Forest IL 60305

Jonathan Seidman
Open Data Group
400 Lathrop Ave Suite 90
River Forest IL 60305

Steve Vejckik
Open Data Group
400 Lathrop Ave Suite 90
River Forest IL 60305

ABSTRACT

Developing data mining algorithms that are suitable for cloud computing platforms is currently an active area of research, as is developing cloud computing platforms appropriate for data mining. Currently, the most common benchmark for cloud computing is the Terasort (and related) benchmarks. Although the Terasort Benchmark is quite useful, it was not designed for data mining per se. In this paper, we introduce a benchmark called MalStone that is specifically designed to measure the performance of cloud computing middleware that supports the type of data intensive computing common when building data mining models. We also introduce MalGen, which is a utility for generating data on clouds that can be used with MalStone.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*data mining*; H.3.4 [Information Storage and Retrieval]: Systems and Software—*distributed systems, performance evaluation*

General Terms

Performance

Keywords

benchmarks, cloud computing benchmarks, data mining benchmarks

^{*}This author is the corresponding author. He is also a faculty member at the University of Illinois at Chicago.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD'10, July 25–28, 2010, Washington, DC, USA.

Copyright 2010 ACM 978-1-4503-0055-110/07 ...\$10.00.

1. INTRODUCTION

Clouds based on the Hadoop system and associated Apache systems, such as Hbase, Apache Pig, Hive and ZooKeeper, have proved effective for processing large scale data for data mining and related applications over racks of commodity computers [11]. This type of architecture is sometimes called a large data cloud [9], and was popularized in a series of Google technical reports that described the Google File System (GFS) [7], MapReduce [5], and BigTable [3]. In a large data cloud, the data is stored over many loosely coupled distributed disks, such as you would find in racks of commodity computers. A common architecture is for the computers in each rack to communicate using a switch located at the top of each rack and for different racks to communicate using a larger switch that connects racks within a data center.

It is an important requirement now for many industry and government applications to evaluate the applicability and scalability of different large data cloud architectures and systems. This can be difficult without standardized architectures and benchmarks. In this paper, we take a first step towards a benchmark that is designed to measure in part the ability of a large data cloud system to prepare data for data mining and to build statistical and data mining models.

As motivation, think of the role that the TPC Benchmarks have played in understanding performance differences between different databases and transaction processing systems. Currently, there are no similar benchmarks for comparing two large data clouds that support building analytic models on large datasets. In this paper, we take a first in this direction by introducing a benchmark called *MalStone*. We also describe the implementation of a data generator for MalStone called *MalGen* as well as several experimental studies using MalStone that compare three different large data cloud middleware stacks.

Although using clouds for data mining and data intensive computing is an area of active research [6], [4], there is no benchmark that we are aware of for understanding the impact of different cloud middleware on the performance of a particular algorithm. MalStone is a first step in this direction.

Another way to view MalStone is as a stylized analytic computation of a type that is common in data intensive

computing. MalStone computes a ratio in moving window over aggregated data. We call MalStone stylized since it is typical of the type of derived attributes or features that are computed as part of the modeling process. For data small enough to fit in memory, in a disk, or in a network attached storage system, it is straightforward to compute the MalStone statistic. On the other hand, if the log data is so large that it requires large numbers of disks to manage it, as is the case in a large data cloud, then computing something as simple as this ratio can be computationally challenging. For example, if the data spans 100 disks, then the computation cannot be done easily with any of the databases that are common today. On the other hand, if the data fits into a database, then this statistic can be computed easily using a few lines of SQL.

The open source MalGen code to generate data for MalStone and a technical report describing some illustrative implementations of MalStone is available from

malgen.googlecode.com.

MalStone was designed to give some insight into different large data cloud systems. It was not designed to compare a large data cloud system (in which data is typically stored over multiple distributed disks) to a traditional database, databases and systems that utilize proprietary hardware, or hybrid database/large data cloud systems. Other benchmarks must be designed for this purpose. For example, MalStone does not measure the efficiency of joins and does not take into account the cost by TB of data managed, and similar considerations, all of which are important when comparing these different types of systems.

This paper is organized as follows: in Section 2 motivates the MalStone benchmark. Section 3 describes the abstract model motivating the statistic behind the MalStone benchmark. Section 4 describes the benchmark. Section 5 describes MalGen. Section 6 describes three illustrative implementations of MalStone and shows the sometimes quite significant differences that can arise with different large data cloud middleware stacks. Section 7 contains some experimental studies. Section 8 contains some discussion. Section 9 describes related work. Section 10 is the summary and conclusion.

This emerging application and technology paper makes the following contributions:

1. There is currently no benchmark that we are aware of for measuring the performance of cloud middleware designed to support building data mining models on large datasets. MalStone is such a benchmark. MalStone is defined in Section 4 and is useful for quantifying differences in system architectures and in quantifying their scalability. See Tables 4 and 5 for some examples.
2. There are currently very few data generators that we are aware of for generating data records that can be used for testing data mining algorithms designed for cloud computing platforms. MalGen is such a data generator. MalGen is described in Section 5.
3. Through three experimental studies using MalStone, we have shown that are substantial differences ($\approx 20\times$) between different cloud computing platforms designed to support building data mining models on very large datasets. This is discussed in Section 7.

4. The abstraction described in Section 3 covers a number of interesting examples as summarized in Table 1. Viewing these types of problems from this point of view has not received a lot of attention in the data mining literature to date, but represents an interesting class of problems that occur not infrequently. As the implementations demonstrate, the type of log files that these types of problems produce can be analyzed easily using MapReduce style parallel programming frameworks.

2. MOTIVATING EXAMPLE

We introduce MalStone with a simple motivating example. Consider visitors to web sites. As described in the paper “The Ghost in the Browser” by Provos et. al. [13], approximately 10% of web pages have exploits installed that can infect certain computers when users visit the web pages. Sometimes these are called “drive-by exploits.”

The MalStone benchmark assumes that there are log files that record the date and time that users visited web pages. Assume that the log files of visits have the following fields:

Timestamp | Web Site ID | User ID

There is a further assumption that if the computers become infected, at perhaps a later time, then this is known. That is for each computer, which we assume is identified by the ID of the corresponding user, it is known whether at some later time that computer has become compromised:

User ID | Compromise Flag

Here the Compromise field is a flag, with 1 denoting a compromise. A very simple statistic that provides some insight into whether a web page is a possible source of compromises is to compute for each web site the ratio of visits in which the computer subsequently becomes compromised to those in which the computer remains uncompromised. This statistic is defined in Section 3. Also, see Figure 2.

We call MalStone stylized since we do not argue that this is a useful or effective algorithm for finding compromised sites. Rather, we point out that if the log data is so large that it requires large numbers of disks to manage it, then computing something as simple as this ratio can be computationally challenging. For example, if the data spans 100 disks, then the computation cannot be done easily with any of the databases that are common today. On the other hand, if the data fits into a database, then this statistic can be computed easily using a few lines of SQL.

We abstract this problem by abstracting web sites by *sites*, users by *entities*, and visits by *events*. When entity visits a *marked* site, it may become *marked* at some time in the future. With this generalization, we assume that we have log files containing events records describing an event associated with an entity and a site and that some of these sites mark some of the entities that are associated with them. We assume that not all entities become marked and that there may be a time delay in the marking.

3. SITES, ENTITIES & MARKS

In this section, we abstract and formalize the example described in the previous section.

3.1 The Model

Example	Site	Entity
drive-by exploits	web site	computer identified by IP with browser
compromised login service	computer providing the compromised service	user providing credentials

Table 1: Some examples of scenarios producing site-entity logs. The problem of interest in these examples is to identify the *site* or *sites* that are the source of the marks assuming that we know which entities are marked. The problem is difficult since: i) the site-entity log files may be very large; ii) there may be a background process that marks some entities independent of the marked sites; and iii) not all entities that visit a marked site become marked.

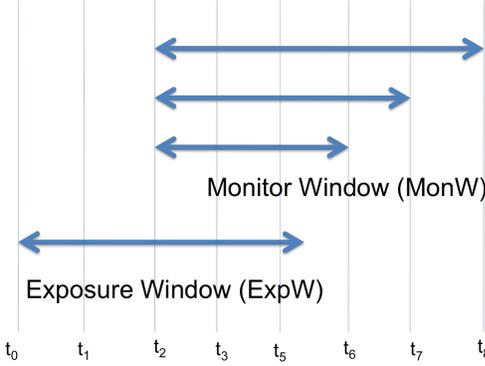


Figure 1: To define the SPM ρ_j statistic requires fixing an exposure window and a monitor window. To define the SPM $\rho_{j,t}$ statistic requires fixing an exposure window and a sequence of increasing monitor windows.

The model we use contains abstract sites and abstract entities. There are two types of activities: entities can *visit* sites and sites can *mark* entities. There is a log file that records each type of activity. The first type of log file records the times at which entities visit sites. The second type of log file records the times at which entities become marked. More precisely, *some of the entities* that visit sites became *marked at some time in the future* after the visit. The second type of log file records the times at which this happens.

We use the following notation:

- We usually let e denote an entity and let s denote a site; we also use e_i for an entity and s_j for a site.
- A, B refer to sets of entities, A_j, B_j refer to sets of entities that depend upon a site s_j .
- S refers to a set of sites and S_i refers to a set of sites associated with an entity e_i .

3.2 SPM for Fixed Windows

In this section, we define a statistic associated with sites, entities, and marks called the subsequent proportion of marks or SPM. We first define SPM-based scores for a fixed window. In the next subsection, we will consider moving windows.

We define the SPM statistic as follows:

1. Fix an exposure window ExpW and a monitor window MonW. See Figure 1.

2. Fix a site s_j .

3. Let A_j be the set of all entities e_i that: i) transact at site s_j at any time during the exposure window ExpW; and ii), in the case that the entity is marked, the transaction occurs before the entity is marked.

4. Let B_j be the set of all entities $e_i \in A_j$ that become marked at any time in the monitor window MonW.

DEFINITION 1. Define the subsequent proportion of marks ρ_j by:

$$\rho_j = \frac{|B_j|}{|A_j|}.$$

We close this section with some remarks:

- Note that $B_j \subseteq A_j$.
- It is important to note that A_j depends upon the exposure window, and B_j depends upon the monitor window, and through the relation $B_j \subseteq A_j$ upon the exposure window.
- Note that the MonW may: 1) start after the ExpW, 2) before the ExpW, or 3) include the entire data available.
- As an example, in Figure 2, for time d_k , there are no events for site s_j , but for the window starting at time d_k , extending backward to time d_{k-2} (time zero in this example), the statistic is $(1 + 0 + 0)/(1 + 1 + 0) = \frac{1}{2}$.

3.3 SPM for Moving Windows

In general, we have a sequence of monitor windows

$$\text{MonW}_{t_1}, \text{MonW}_{t_2}, \text{MonW}_{t_3}, \dots$$

depending upon time t . For example, the sequence may all have a common start time, but the end time increases by a week for each monitor window in the sequence. See Figure 1.

Using this sequence of monitor windows, we can define a sequence

$$\rho_{j,t} = \frac{|B_{j,t}|}{|A_j|},$$

where $B_{j,t}$ is the set of entities $e_j \in A_j$ that become marked at any time during the monitor window MonW_t .

Benchmark	Statistic	# records	Data Size
MalStone A-10	ρ_j	10 billion	1 TB
MalStone A-100	ρ_j	100 billion	10 TB
MalStone A-1000	ρ_j	1 trillion	100 TB
MalStone B-10	$\rho_{j,t}$	10 billion	1 TB
MalStone B-100	$\rho_{j,t}$	100 billion	10 TB
MalStone B-1000	$\rho_{j,t}$	1 trillion	100 TB

Table 2: The MalStone benchmarks use 100 byte records, with a fixed field width.

4. MalStone A & B

In this section, we define the MalStone A and B Benchmarks.

Assume that we have a collection of log files. For simplicity, we assume that the log files that describe visits of entities to sites has been joined to the log file that describes which entities are marked (and when). With this assumption, log files contain the following fields:

```
Event ID | Timestamp | Site ID |
Entity ID | Mark Flag |
```

We interpret these as recording the fact that at the time indicted by the timestamp, the entity with the entity ID visited the site with the Site ID. The Mark Flag indicates whether at the time of visit the entity was marked.

Remark. It is important to note that if the Mark Flag is 1 indicating the entity is marked, we do not necessarily know that the site identified by the Site ID marked the entity. Instead, all that we know is that *either the site, or any site that the entity has visited in the past during the exposure window* has marked the entity.

It is for this reason, that the statistic is called the subsequent proportion of marks.

MalStone A computes ρ_j for all sites j in the log files. MalStone B computes $\rho_{j,t}$ for sites j in the log files and for a sequence of moving windows that begin at time t_0 and end at time t equal to:

$$t_1 < t_2 < t_3 < \dots$$

MalStone records are 100 byte records, with a fixed width fields. Both MalStone A and B use 1 year’s worth of data. MalStone A uses a single window for the entire year, while MalStone B uses a window that begins at the beginning of the year and ends at week 1, week 2, ..., week 52.

5. MalGen

As mentioned above, we have developed an open source program called MalGen for generating site-entity log files for all the nodes in a cluster. MalGen uses a power law distribution to model the number of entities associated with a site. Most sites have few entities associated with them, while a few sites have a large number of entities. This is the case, for example, with web sites: most sites have only a few visitors, a few sites have a lot of visitors, and a power law distribution is often used to model this distribution.

MalGen are 100 bytes in size with five fixed width fields:

```
Event ID | Timestamp | Site ID |
Entity ID | Mark
```

The following is a description of each field:

- Event ID — The Event ID consists of an ID for each record that is sequential and unique when restricted to a single node followed by a hash of the hostname to create a globally unique Event ID.
- Timestamp — The date and time of the event. This is a uniformly distributed random value over a user-specified number of days. The default is to generate data distributed over a period of one year.
- Site ID — This is the ID of the site associated with the event.
- Entity ID — This is the ID of the entity associated with the event.
- Mark — The field is either 0 or 1 and indicates whether the entity is marked at the time. Note that, as discussed above, the fact that the mark is 1 does not indicate that the site with Site ID is responsible for the mark, simply that at some time prior to the Timestamp, the entity visited some marked site.

Site-entity log files are generated by MalGen in several steps. Several steps are used since the MalStone SPMstatistic requires aggregating data from different distributed nodes and computing a statistic that satisfies both certain statistical properties and certain consistency requirements. The first step generates certain seed information about the marked sites and scatters this information to all the nodes in the large data cloud. The algorithm is designed to keep certain information required for the first step in memory in order to improve the overall speed of MalGen . The subsequent steps are done independently by each of the nodes. We report on the memory utilization of the first step of MalGen below since keeping the memory utilization relatively low is important so that enough seed information is available for generating the 10 billion, 100 billion and 1 trillion records that MalGen requires.

In the first step, MalGen generates events associated with marked sites. For each marked site, a random date is generated. For a particular site, the number of events is randomly generated using a power law distribution and a set of entity IDs is randomly generated from the pool of available entity IDs. The power law distribution is constructed so that most sites are associated with a relatively few number of entity IDs (a few hundred a day), but with a long tail so that there are a small number of sites with a very large number of events. The Entity IDs are sampled until the number of events for each site is complete.

For the marked sites, a visit by an entity subjects the entity to a probability (e.g. 70%) of being marked. If an entity is marked, it is tagged as being such with a timestamp that occurs after a delay period (e.g. one week) If an entity is already marked when it visits a marked site again, it is subject to mark if the date of the current event precedes that of the event that marked it. In this case, the date-time of the mark is updated accordingly.

The initial seeding and the generation of marked entities is done on a single node. This information is then pushed out to all the nodes in the large data cloud and each local node then generates records for entities that are not marked. Table 3 shows the times required to generate 2 billion, 6 billion and 10 billion events in this way on a 20 node cloud.

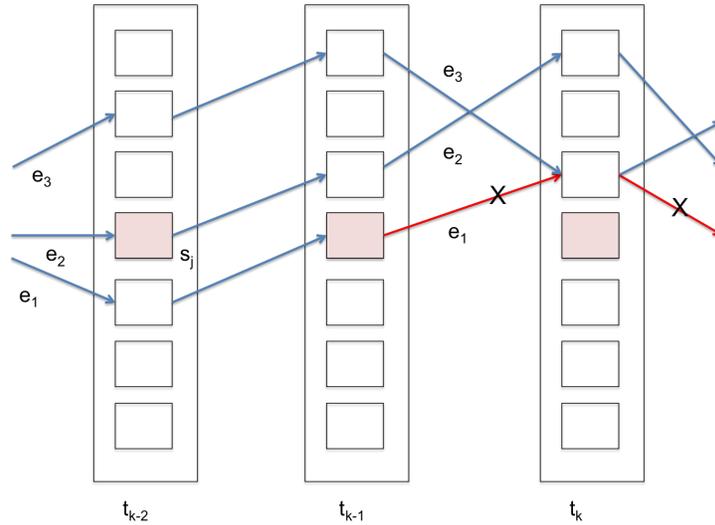


Figure 2: The diagram shows an example of how the SPM statistic is computed. Here sites s_j are represented by small rectangles and marked sites are represented by shaded rectangles. Specifically, for each site s_j at time t_k , MalStone B collects all the transactions (represented by arrows) that are associated with the site at time t_k or earlier. Notice there are no transactions associated with s_j at time t_k , but that there are two transactions associated with the site at earlier times t_{k-1} and t_{k-2} . Entity e_2 was associated with the site at t_{k-2} and entity e_1 at time t_{k-1} . Entity e_1 became marked at the site s_j at time s_{k-1} (represented by red entity arrow with an “X”). Therefore $\frac{1}{2}$ of the transactions are marked for site s_j with respect to the window (t_{k-2}, t_{k-1}, t_k) .

includegraphics[scale=0.6]figs/memory

Figure 3: The memory usage of MalGen as it generated approximately 500,000,000 log records for visits to approximately 120,000 different sites, where the number of visits to a web site follows a power law. Approximately 8.5 GB of an available 12 GB of memory were used during the 30 minute generation of data. The bottom line shows the memory used, while the top line shows the available memory.

Records/node	RAM	Time
100 M	16 GB	60 min
300 M	16 GB	142 min
500 M	16 GB	190 min

Records/node	Total Records	RAM	Time
100 M	2 B	4 GB	54 min
300 M	6 B	4 GB	157 min
500 M	10 B	4 GB	275 min

Table 3: The first table shows the time required in minutes for MalGen to seed the data generation and to generate the marked entities. This was run on a head node with 16 GB of memory. The second table shows the time required to copy the required data from the head node to each of 20 local nodes and for the local nodes to generate all the required unmarked events. For example, the time required to generate 10 billion events distributed over 20 nodes is $190 + 275 = 465$ minutes.

The time required for seeding the process is in the table below:

After all event histories for the requested number of marked sites are complete, subsequent sites are assumed to be unmarked with no possibility of being marked. This is the third step, which uses the same process that was just described to construct visits for non-marked sites.

We close this section with two remarks about MalGen.

MalGen is designed to generate large datasets that span all the nodes in a cluster. To create consistent data in parallel over all the nodes in the cluster, MalGen: i) generates the data describing all marked sites on one machine in the cluster; ii) scatters the information to all the other nodes in the cluster; iii) all the other nodes in the cluster then generate the data for all the unmarked sites.

By keeping information about sites in memory (vs on disk), MalGen can improve its performance. The default parameters for MalGen can generate approximately 500,000,000 events for approximately 120,000 sites in about 30 minutes using a Dell 1435 2.0GHz dual-core AMD Opteron 2212 processor and 16 GB of memory. Figure 3 contains a graph showing MalGen’s memory usage.

6. THREE IMPLEMENTATIONS

We have implemented the MalStone A and B benchmarks in three different ways:

1. using the Hadoop Distributed File System (HDFS) [2] and Hadoop’s implementation of MapReduce [11];
2. using HDFS, Hadoop Streams [11] and coding MalStone A and B in Python; and
3. 3) using the Sector Distributed File System and Sphere User Defined Functions [10].

6.1 Hadoop HDFS and MapReduce

We used MalGen to generate data which we stored in HDFS and then implemented MalStone B using a Mapper, Reducer and Partitioner as follows:

- Mapper — Reads the records and groups them using the Site ID as the key. The corresponding value is the timestamp and the mark flag.
- Reducer — For each key, the Reducer tracks the total number of events seen with the mark flag equal to one and the total number of events and stores them by the date. All saved values are then processed in order by the date.
- Partitioner — The Site Id is taken modulo the number of reducers.

Each record is parsed and then transformed into a key-value pair. The key is the Site Id. The value is the Flag and the bucket the time stamp is put in. The time stamp in each record can be bucketed arbitrarily. MalStone B requires that the statistic be computed for each week; we used the ISO week number (www.iso.org) for convenience.

The operation performed on each group of data is to count the number of events and the number of events with the mark equal to one each time t . For each site s , this is stored in a Java Collections Map using t as the key.

When all records for a site id are processed, the stored values are accessed in chronological order and running totals computed.

The output is the Site ID (key) j and a list of the times t and associated SPM statistics $\rho_{j,t}$

6.2 HDFS, Hadoop Streams and Python

The second implementation used Hadoop Streams [11] and Python. The mapper method reads the records from Standard Input and sends the mapped data to Standard Output. The same key and value structure as described in Section 6.1 is used.

The reducer reads the mapped data from Standard Input and for each Site ID, stores the aggregated number of events seen and those seen with the mark equal to 1 in a Python dictionary keyed by the time t .

When all the records for a site id are processed, the stored values are accessed in chronological order and running totals computed.

The output is the Site ID (key) and a list of the times t ’s and the associated SPM statistic $\rho_{j,t}$ is sent to Standard Output (value).

6.3 Sector and Sphere UDFs

Sector provides two methods for implementing processing [10]:

- Using indexed data — when using indexed data each input data file has an accompanying index file containing the offsets of each record in the data file. This index allows Sector to segment the data during processing.
- Using non-indexed data — when using non-indexed data, the processing code must manually segment the input the input data during processing.

Using non-indexed data requires somewhat more code to implement, but seems to improve processing time. MalStone B was implemented using non-indexed data.

The MalStone B code was implemented in two stages:

- In the first stage, each record in the input data is read, assigned to a bucket based upon the site ID, and each bucket file is written to disk. After this stage is completed, all records for a particular site will be in a single file.
- In the second stage, for each site, for each site j the cardinality of the sets A_j and B_j is computed. After all the records for a site j have been processed, the resulting record is saved to a file.

7. EXPERIMENTAL STUDIES

7.1 Testbed

The experimental studies used a rack of 30 Dell 1435 computers. Each computer had 12 GB memory, 1TB disk, and a 2.0GHz dual dual-core AMD Opteron 2212. Each computer had a 1 Gb/s network interface cards and were networked together with a Cisco 3750E switch.

7.2 MalStone Benchmarks

For the experimental studies reported below, we used 20 nodes. Each node was populated with 500 million records using MalGen for a total of 10 billion records. Each record was 100 bytes for a total of 1 TB of data. The results are reported in Tables 4 and 5.

Note that as measured by these benchmarks, storing the data using HDFS and implementing the benchmark using Hadoop streams and Python was substantially faster than using HDFS and Hadoop’s MapReduce.

Note also that managing the data using Sector and implementing the benchmark using Spheres UDFs was about 2.5 times faster than the Hadoop streams implementation.

8. DISCUSSION

One of the pleasant surprises is the power of Hadoop streams, which does not require the MapReduce framework. Hadoop is now a relatively mature distributed file system that can scale to over a thousand nodes and manage petabytes of data. As Tables 4 and 5 show, Python programs can be invoked by Hadoop streams and be used to efficiently process large data sets without the MapReduce framework and this approach can be faster when computing certain statistics (such as $\rho_{j,t}$) than performing the computation using MapReduce. We stress that this is a positive outcome and

	Hadoop HDFS with Streams & Python	Hadoop HDFS with MapRe- duce	Sector with Sphere UDFs
Run 1	82m 21s	458m 7s	33m 44s
Run 2	90m 31s	450m 21s	33m 26s
Run 3	89m 35s	454m 12s	33m 51s
Average	87m 29s	454m 13s	33m 40s

Table 4: This table summarizes an experimental study running MalStone A on 20 nodes. Each node had 500 million 100-byte MalStone records. The tests used version 0.18.3 of Hadoop and version 1.20 of Sector.

	Hadoop HDFS with Streams & Python	Hadoop HDFS with MapRe- duce	Sector with Sphere UDFs
Run 1	144m 10s	799m 0s	43m 57s
Run 2	146m 23s	861m 40s	43m 52s
Run 3	137m 4s	861m 51s	43m 24s
Average	142m 32s	840m 50s	43m 44s

Table 5: This table summarizes running MalStone B on 20 nodes. Each node had 500 million 100-byte MalStone records. The tests used version 0.18.3 of Hadoop and version 1.20 of Sector.

simply shows (as is obvious in hindsight) that certain statistical qualities can be computed more efficiently directly with Python over the data managed by the HDFS than by using MapReduce and the HDFS.

Another pleasant surprise is that once we abstracted the MalStone statistic as the Site-Entity-Mark Model, we found that other applications could also be modeled in this way, as Table 1 shows.

In practice, once the MalStone SPM statistic is computed, relatively effective statistical models can be computed by looking for changes over time t in the $\rho_{j,t}$ statistic using CUSUM, GLR and related change detection models [12]. Although outside the scope of this paper, if segmented models are used for each site j , the Reducer in MapReduce can be used to organize the computation so that each node in a large data cloud contains all the data required to build a change detection model for a site j [1].

9. RELATED WORK

The CloudStone Benchmark [14] is a first step towards a benchmark for clouds designed to support Web 2.0 type applications. In this note, we describe the MalStone Benchmark, which is a first step towards a benchmark for clouds, such as Hadoop and Sector, designed to support data intensive computing.

One of the motivations for choosing 10 billion 100-byte records is that the TeraSort Benchmark [8] (sometimes called the Terabyte Sort Benchmark) also uses 10 billion 100-byte records.

We note that in 2008, Hadoop became the first open source program to hold the record for the TeraSort Benchmark. It was able to sort 1 TB of data using 910 nodes in 209 seconds, breaking the previous record of 297 seconds. Hadoop set a new record in 2009 by sorting 100 TB of data at 0.578 TB/minute using 3800 nodes.

The TeraSort Benchmark is now deprecated and has been replaced by the Minute Sort Benchmark. Currently, 1 TB of

data can be sorted in about a minute given the right software and sufficient hardware.

The paper by Provos et. al. [13] describes a system for detecting drive-by malware that uses MapReduce. Specifically, MapReduce is used to extract links from a large collection of crawled web pages. These links are then analyzed using heuristics to identify a relatively small number of suspect web sites. These suspect web sites are then tested using Internet Explorer to retrieve web pages in a virtual machine that is instrumented. This allows those web sites resulting in drive-by infections to be directly monitored. In contrast, the work described in this paper is quite different. The work here uses Hadoop and MapReduce to compute the MalStone statistic from a collection of log files generated by MalGen in one of the illustrative implementations of MalStone.

The paper [4] describes how several standard data mining algorithms can be implemented using MapReduce, but this paper does not describe a computation similar to the MalStone statistic.

10. SUMMARY

In this paper, we have introduced a benchmark called MalStone for measuring the performance of cloud middleware designed for data mining and data intensive computing. Currently, gaining access to large amounts of nonproprietary data to use for benchmarking cloud middleware can be challenging. For this reason, we have developed an application called MalGen that is designed to generate synthetic log-entity files that can be used by MalStone. We have used MalGen to generate tens of billions of events on clouds with over 100 nodes.

The MalStone benchmark computes a statistic that is a stylized analytic on log files consisting of records of visits by entities to sites. Sometimes, after these visits, entities become marked at some time in the future. Note that this analytic is related to identifying the *sites* that are the sources of the marks, not the marked *entities* themselves.

As Tables 4 and 5 show, there can be substantial differ-

ences in performance, depending upon which cloud middle-ware is used to compute the MalStone statistic.

11. AVAILABILITY

MalGen is open source and available from malgen.googlecode.com. The current version of MalGen is 0.9.

12. REFERENCES

- [1] C. Bennett, D. Locke, R. L. Grossman, and S. Vejcik. Sawmill: Building segmented models in large data clouds. to appear, 2010.
- [2] D. Borthaku. The Hadoop distributed file system: Architecture and design. retrieved from lucene.apache.org/hadoop, 2007.
- [3] 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. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.
- [4] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *NIPS*, volume 19, 2007.
- [5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.
- [6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.
- [8] J. Gray. Sort benchmark home page. <http://research.microsoft.com/barc/SortBenchmark/>, 2008.
- [9] R. L. Grossman and Y. Gu. On the varieties of clouds for data intensive computing. *Bulletin of the Technical Committee on Data Engineering*, 32(1):44–50, March 2009.
- [10] Y. Gu and R. L. Grossman. Sector and sphere: Towards simplified storage and processing of large scale distributed data. *Philosophical Transactions of the Royal Society A*, also *arxiv:0809.1181*, 2009.
- [11] Hadoop Wiki. Apache Hadoop. retrieved from <http://wiki.apache.org/hadoop/>, 2010.
- [12] H. V. Poor and O. Hadjiladis. *Quickest Detection*. Cambridge University Press, 2008.
- [13] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *HotBot '07*, 2007.
- [14] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proceedings of Cloud Computing and its Applications 2008*, 2008.