MapReduce Analysis for Cloud-archived Data

Balaji Palanisamy Aameek Singh† Nagapramod Mandagere† Gabriel Alatorre† Ling Liu‡

School of Information Sciences, University of Pittsburgh †*IBM Research - Almaden,* ‡*College of Computing, Georgia Tech*
bpalan@pitt.edu, †{aameek.singh, pramod, galatorr}@us.ibm.com, ‡lingliu@cc.gatech.edu

***Abstract*—Public storage clouds have become a popular choice for archiving certain classes of enterprise data - for example, application and infrastructure logs. These logs contain sensitive information like IP addresses or user logins due to which regulatory and security requirements often require data to be encrypted before moved to the cloud. In order to leverage such data for any business value, analytics systems (e.g. Hadoop/MapReduce) first download data from these public clouds, decrypt it and then process it at the secure enterprise site.**

We propose VNCache: an efficient solution for MapReduce analysis of such cloud-archived log data without requiring an apriori data transfer and loading into the local Hadoop cluster. VNcache dynamically integrates cloud-archived data into a virtual namespace at the enterprise Hadoop cluster. Through a seamless data streaming and prefetching model, Hadoop jobs can begin execution as soon as they are launched without requiring any apriori downloading. With VNcache's accurate pre-fetching and caching, jobs often run on a local cached copy of the data block significantly improving performance. When no longer needed, data is safely evicted from the enterprise cluster reducing the total storage footprint. Uniquely, VNcache is implemented with NO changes to the Hadoop application stack.

I. Introduction

Data Storage requirements have seen an unprecedented growth in the recent years owing to more stringent retention requirements. Cloud storage solutions can be an attractive and cost-effective choice in such cases due to its cost-effective and on demand nature. However, for certain classes of enterprise data - application and infrastructure logs, current cloud storage and analytics architectures do not support them well especially when there are stringent requirements for data privacy and security.

Logs often contain sensitive information like IP addresses, login credentials, etc. which necessitate encrypting the data before it leaves the enterprise premises. After securely archiving the data in the storage cloud, extracting any business value using analytics systems such as MapReduce[2] or Hadoop[17] is nontrivial. In these cases, using compute resources in the public cloud is often not an option due to security concerns. Most state-of-the-art cloud solutions for such cases are highly sub-optimal requiring all (encrypted) data sets to be first transferred to the enterprise cluster from remote storage clouds, decrypted, and then loaded into the Hadoop Distributed File System (HDFS)[4]. It is only after these steps complete that the job will start executing. Secondly, this results in extremely inefficient storage utilization. For example, while the job is executing, the same dataset will reside in both the public storage cloud and the enterprise cluster and is in fact replicated multiple times at both of these places for resiliency purposes, resulting in higher costs. For example, Hadoop by default will replicate the data 3 times within the enterprise Hadoop cluster. This is on top of the storage replication cost incurred at the public storage cloud.

In this paper, we propose a unique hybrid cloud platform called VNcache that alleviates the above mentioned concerns. Our solution is based on developing a virtual HDFS namespace for the encrypted data stored in the public storage cloud that becomes immediately addressable in the enterprise compute cluster. Then using a seamless streaming and decryption model, we are able to interleave compute with network transfer and decryption resulting in efficient resource utilization. Further by exploiting the data processing order of Hadoop, we are able to accurately prefetch and decrypt data blocks from the storage clouds and use the enterprise site storage only as a cache. This results in predominantly local reads for data processing without the need for replicating the whole dataset in the enterprise cluster.

Uniquely we accomplish this without modifying any component of Hadoop. By integrating VNCache into the filesystem under HDFS, we are able to create a new control point which allows greater flexibility for integrating security capabilities like encryption and storage capabilities like use of SSDs (Solid-state drives). Our experimental evaluation shows that VNCache achieves up to 55% reduction in job execution time while enabling private data to be archived and managed in public clouds.

The rest of the paper is organized as follows. Section II provides the background and the use-case scenario for supporting MapReduce analysis for cloud-archived data. In Section III, we present the design of VNCache and its optimization techniques. We discuss our experimental results in Section IV and we present a discussion of alternate solutions and design choices for VNCache in Section V. In Section VI, we discuss related work and we conclude in Section VII.

II. Background

We consider enterprise applications that perform MapReduce analysis over log data. The logs get generated at the enterprise site and archived in a public cloud infrastructure. For example, an application that monitors the status of other application software and hardware typically generates enormous amounts of log data. Such log data is often associated with a timestamp and data analysis may need to performed on them when needed in the future.

With current cloud storage solutions, the logical method to perform analysis of archived data would be as follows. Log data generated at the enterprises would be encrypted and archived at a possibly nearest public storage cloud. Upon a need to execute a Hadoop analytics job, the enterprise cluster would download all relevant input data from the public clouds (time for which depends on network latencies). It will then create a virtual Hadoop cluster by starting a number of VMs. Data is then decrypted locally (time for which depends on CPU/Memory availability on local nodes and denoted by Decryption Time) and then ingested into HDFS (Hadoop Distributed Filesystem) of the Hadoop cluster and then the job can start executing. Upon finishing the job, local copy of the data and the virtual Hadoop cluster can be destroyed.

Figure 1 shows the breakdown of execution time for running a grep hadoop job on a 5GB dataset using the conventional execution model mentioned above. The network latencies 45, 90 and 150 milliseconds represent various degrees of geographic separation such as co-located datacenters, same coast data centers, and geographically well-separated data centers. Results show that data transfer time (the time to transfer data from the remote storage cloud to the enterprise cluster) and HDFS load time can have significant impact on overall execution time, thus making this model inefficient. We also notice that the data transfer time increases with increase in network latency.

Further, depending upon the amount of data required to be loaded and connectivity between the enterprise cluster and the remote storage cloud infrastructure, this step adversely impacts performance, and while the job is running (often for long durations) the dataset is duplicated in both the public storage cloud as well as the local enterprise cluster– along with the storage cloud original, there is a copy in the enterprise cluster, leading to higher costs for the enterprise.

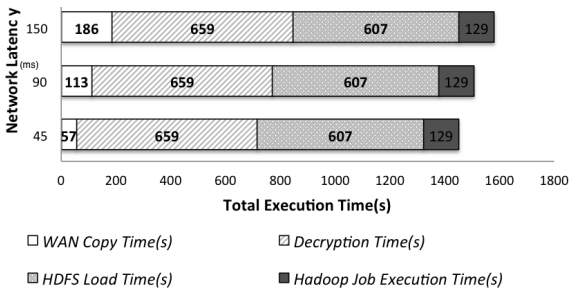


Fig. 1: Breakdown of Fullcopy Runtime: 5 GB dataset with varying network latency

In contrast to this conventional model, VNCache aims at minimizing the impact of *apriori* data ingestion and decryption steps by intelligent pipelining of compute with those steps; specifically, by creating a *virtual* HDFS namespace which lays out the HDFS data blocks across the compute cluster. Whenever the job needs to access any data block, VNCache streams it on-demand from the appropriate storage clouds, decrypting it on-the-fly, and making it available to the job. As an additional performance optimization, VNCache prefetches data ahead of processing so that the map tasks read the data from local storage.

Before presenting the design overview of VNCache and its various components, we present a brief overview of HDFS and its interaction with the underlying filesystem.

A. HDFS and underlying filesystem

Hadoop Distributed Filesystem (HDFS) is a distributed user-space filesystem used as the primary storage by Hadoop applications. A HDFS cluster consists of a Namenode that manages filesystem metadata and several Datanodes that store the actual data as HDFS blocks. HDFS is designed to be platform independent and can be placed on top of any existing underlying filesystem (like Linux ext3) on each node of the cluster. It follows a master/slave architecture. HDFS exposes a file system namespace and allows user data to be stored in files. The HDFS Namenode manages the file system namespace and regulates access to files by clients. The individual Datanodes manage storage attached to the nodes that they run on. When a client writes a file into HDFS, the file is split into several smaller sized data blocks (default size is 64 MB) and stored on the storage attached to the Datanodes.

Within the cluster, the Namenode stores the HDFS filesystem image as a file called *fsimage* in its underlying filesystem. The entire HDFS filesystem namespace, including the mapping of HDFS files to their constituent blocks, is contained in this file. Each Datanode in the cluster stores a set of HDFS blocks as separate files in their respective underlying filesystem[1]. As the Namenode maintains all the filesystem namespace information, the Datanodes have no knowledge about the files and the namespace. As a HDFS cluster starts up, each Datanode scans its underlying filesystem and sends a Block report to the Namenode. The Block report contains the list of all HDFS blocks that correspond to each of these local files.

When an application reads a file in HDFS, the HDFS client contacts the Namenode for the list of Datanodes that host replicas of the blocks of the file and then contacts the individual Datanodes directly and reads the blocks from them. We refer the interested readers to [4] for a detailed documentation on the design and architecture of the Hadoop Distributed Filesystem. In the next section, we present the design overview of VNCache and discuss its various components

III. VNCache overview

VNCache is a FUSE based filesystem [7] used as the *underlying* filesystem on the Namenode and Datanodes of the HDFS cluster. It is a virtual filesystem (similar to */proc* [6] on Linux) and simulates various files and directories to the HDFS layer placed on it. For the Namenode, VNCache exposes a virtual HDFS namespace with an artificially constructed *fsimage* file and for Datanodes, it exposes a list of data files corresponding to the HDFS blocks placed on that datanode.

[1]Location in the underlying filesystem is determined by the *dfs.data.dir* configuration setting

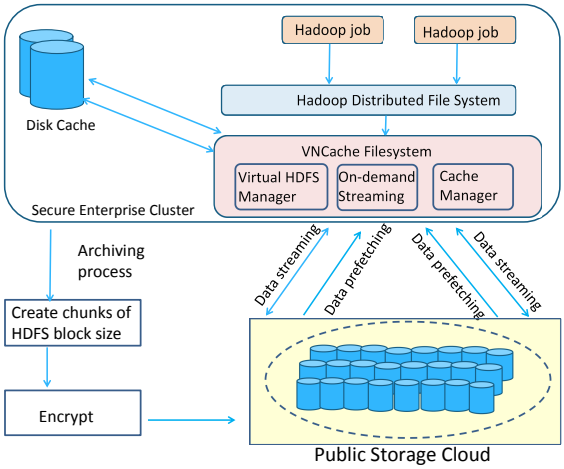


Fig. 2: System Model

TABLE I: HDFS fsimage

Image Element	Datatype
Image version	Integer
NAMESPACE_ID	Integer
NumInodes	Integer
GENERATION_STAMP	Long

TABLE II: HDFS INode

Image Element	Datatype
INODE_PATH	String
REPLICATION	Short
MODIFICATION_TIME	Long
ACCESS_TIME	Long
BLOCK_SIZE	Long
numBlocks	Integer
NS_QUOTA	Long
DS_QUOTA	Long
USER_NAME	String
GROUP_NAME	String
PERMISSION	Short

Figure-2 presents the overall framework showing various key components.

A. Data Archiving Process

In our approach, we pre-process the log data created at the enterprise cluster to encrypt and make it HDFS friendly before archiving them in a public cloud. Specifically, large amounts of log data get chunked into several small files of HDFS block size (64 MB default), get encrypted, and we label them with the timestamp information (e.g. 1-1-2013.to.2-1-2013.data) before uploading to the public storage cloud. The enterprise site uses a symmetric key encryption scheme to encrypt the dataset before archiving in the cloud. When log data belonging to a given time window needs to be analyzed later on, VNCache can identify all blocks stored in the storage cloud that contain data relevant to that analysis.

We note that archiving the data in this manner does not preclude the data being accessed in a non-HDFS filesystem when needed. In such cases when there is a need to download the data in a non-HDFS filesystem, VNCache can download it through a normal Hadoop *dfs -get* command. Next, we describe how these data blocks are presented to the HDFS layer at the enterprise cluster so that jobs can begin execution right away.

B. Virtual HDFS Creation

When a Hadoop job at the enterprise cluster needs to process an archived dataset, a virtual cluster is created by starting a number of VMs including one designated to be the primary Namenode. Before starting Hadoop in the VMs, a virtual HDFS namespace is created on the Namenode. It starts by generating a list of relevant HDFS blocks B_{job} for the job based on the input dataset. For example, for an analysis of 1 month of log data archived in the cloud, all blocks stored in the storage cloud that contains any data for the chosen time window would become part of the virtual filesystem[2].

[2]Any unaligned time boundaries are handled in a special manner, details of which are omitted due to space constraints.

A virtual file F_{job} is then created to contain $|B_{job}|$ HDFS blocks, where each block is given a unique HDFS identifier while maintaining its mapping to the filename in the remote cloud. Similar to HDFS Namenode filesystem formatting, a *fsimage* (filesystem image) file is generated and the virtual file is inserted into this *fsimage* filesystem image file using our HDFS virtualization technique described next.

The HDFS virtualization in VNCache initially creates a HDFS filesystem image and inserts an INode corresponding to the new file to be added into the virtual HDFS. The *fsimage* file is a binary file and its organization in shown in Tables I - IV. The spatial layout of the HDFS filesystem is shown in Figure 3. The *fsimage* begins with the image version, Namespace identifier and number of Inodes stored as Integers and Generation stamp stored as Long. The Generation stamp is generated by the Namenode to identify different versions of the Filesystem image. Here the INode represents the HDFS data structure used to represent the metadata of each HDFS file. For inserting a virtual file into the virtualized HDFS, VNCache creates a new INode entry corresponding to the INode organization described in Table II. The first field in the INode structure is the INode path stored as a String, followed by replication factor, modification and access times for the file. It also contains other fields such as the HDFS block size used by the file, number of HDFS blocks, namespace and disk space quotas, user name and group names and permission. The INode structure is followed by the information of each of the individual blocks of the file. As shown in Table III, each block representation consists of a block identifier, number of bytes and generation stamp. The block generation stamp is a monotonically increasing number assigned by the Namenode to keep track of the consistency of the block replicas. Since these are assigned by the Namenode, no two HDFS blocks can ever have the same Generation Timestamp. The HDFS filesystem image also has a list of INodes under construction (INodesUC) whose description is shown in Table IV.

At the enterprise cluster, the namenode is started using the virtual HDFS filesystem image which enables Hadoop to

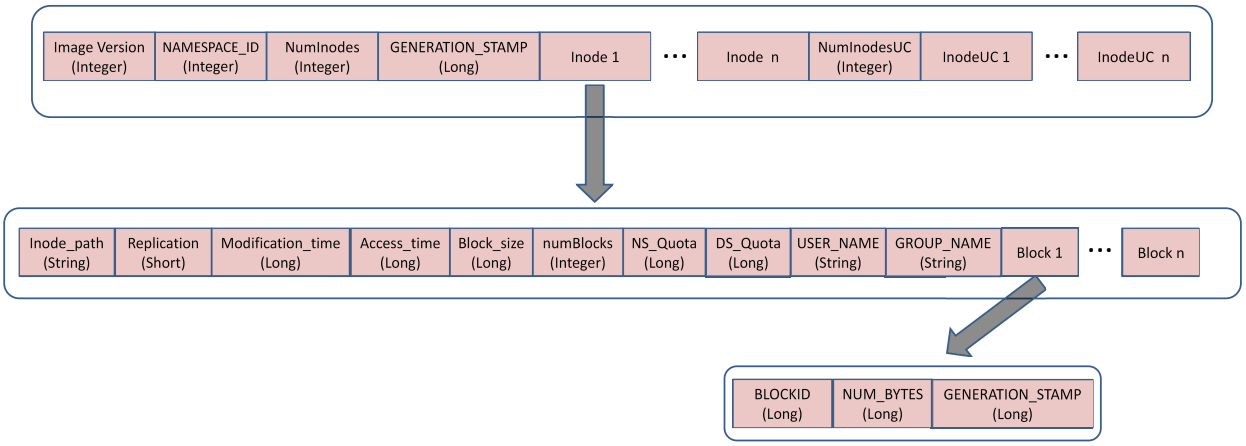


Fig. 3: HDFS: FileSystem Image

TABLE III: HDFS Block

Image Element	Datatype
BLOCKID	Long
NUM_BYTES	Long
GENERATION_STAMP	Long

TABLE IV: INodes Under Construction

Image Element	Datatype
INODE_PATH	String
REPLICATION	Short
MODIFICATION_TIME	Long
PREFERRED_BLOCK_SIZE	Long
numBlocks	Integer
USER_NAME	String
GROUP_NAME	String
PERMISSION	Short
CLIENT_NAME	String
CLIENT_MACHINE	String

understand that the required file and its individual blocks are present in the HDFS.

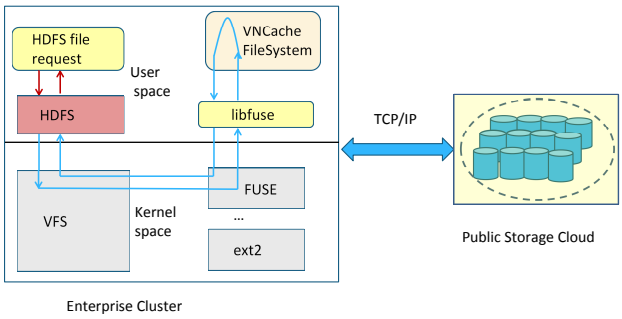


Fig. 4: VNCache: Data Flow

Next, we determine the virtual data layout of these HDFS blocks on the Datanodes. It is done similar to Hadoop's default data placement policy with its default replication factor of 3. Once Hadoop is started on the cluster, Datanodes report these blocks in the Block report to the Namenode, which assumes all HDFS blocks in the HDFS filesystem namespace are present even though initially the data still resides in the public storage cloud. Thus, from a Hadoop application stack standpoint, the job execution can begin immediately.

C. On-demand Data Streaming

VNCache enables on-demand streaming and on-the-fly decryption of HDFS data blocks. Once the read request for a HDFS block reaches the Datanode that (virtually) stores the block, VNCache on the Datanode looks up the mapping to its public cloud storage location and begins fetching the data from the public storage cloud. Once the block has been downloaded, it is decrypted before returning the data to the call. The enterprise site uses a symmetric key encryption scheme to encrypt the dataset before archiving in the cloud and therefore the downloaded blocks are decrypted using the same key prior to passing them to HDFS. Please note that the read requests received by the underlying VNCache may be for a portion of a data block (e.g. Hadoop often does multiple 128k byte reads while reading a complete 64 MB block). For our implementation, we have chosen to start downloading the block when an *open* call is received and corresponding read requests are served from that downloaded and decrypted block.

Overall, from the HDFS standpoint, the HDFS data blocks - stored as files on the VNCache filesystem - are seamlessly accessible so Hadoop works transparently without the interference of streaming and decryption happening along this process. Figure 4 shows the dataflow and the interaction between the HDFS and the FUSE-based VNCache filesystem.

D. Caching and Pre-fetching

The performance of the VNCache approach can be significantly improved if HDFS block read requests can be served from the disks of the enterprise cluster as opposed to streaming for each access. The goal of the caching algorithm is to maximize the reads from the local disks on the VMs and minimize streaming requests from the storage server in order

to minimize the read latency for the jobs. Additionally, a good caching algorithm is expected to yield high cache hit ratios even for reasonable size of the cache on the disks of the VMs and should aim at minimizing the cache space used. VNCache incorporates a distributed cache prefetching algorithm that understands the processing order of the blocks by the MapReduce workflows and prefetches the blocks prior to processing. For subsequent open and HDFS block read operations, the data from the disk cache in the enterprise cluster is used for reading. In case of a cache miss, VNCache still streams the data block from the remote storage cloud as explained above.

The cache manager follows a master/slave architecture where a dynamic workflow-aware prefetch controller monitors the job progress of the individual jobs in the workflow and determines which blocks need to be prefetched next and sends instructions to the slave prefetchers running on individual Hadoop nodes. Each slave prefetcher is multi-threaded and follows a worker model where each worker thread processes from a queue of prefetch requests. Each worker thread prefetches one HDFS data block file from the storage cloud and replicates the block within the Hadoop cluster based on the replication factor.

As mentioned earlier, the cache prefetcher logic needs to be capable of predicting the processing orders of the individual HDFS blocks by the MapReduce jobs so that the order of accesses corresponds to the prefetch order. Secondly, the caching algorithm needs to be dynamically adaptive to the progress of the jobs in terms of the map tasks that have been already launched and the ones that are to be launched next, thereby it does not attempt to prefetch data for tasks that have already completed. In addition, the prefetcher should also be aware of the rate of processing the job in terms of the average task execution time and as well as on the current network throughput available between the storage Clouds and the enterprise site.

Prefetching order: The cache prefetcher logic in VNCache is capable of predicting the processing order of the individual HDFS blocks. From the Hadoop design, we note that the default processing order of the blocks in a Hadoop job is based on the decreasing order of the size of the files in the input dataset and within each individual file, the order of data processed is based on the order of the blocks in the HDFS filesystem image file - *fsimage*. While this ordering is followed in the default FIFO scheduler in Hadoop, some other sophisticated task placement algorithms ([16], [3]) may violate this ordering to achieve other goals such as higher fairness and locality. One direction of our ongoing work is focused on developing cache-aware task placement algorithms that achieve the goals of these sophisticated scheduling algorithms in addition to being aware of the blocks that are already cached.

Dynamic rate adaptive prefetching: VNCache's prefetching algorithm is adaptive to the progress of the jobs so that it does not attempt to prefetch data for tasks that have already completed or likely to start before prefetching is complete. The algorithm constantly monitors the job progress information from log files generated in the logs/history directory of the master Hadoop node. It parses the Hadoop execution log file to obtain the Job SUBMIT_TIME and Job LAUNCH_TIME and looks for task updates related to map task launching and completion. Based on the differences in the speed of job progress (primarily dictated by the type of job) and the time being taken to prefetch a block (dependent on connectivity between the enterprise site and the public storage cloud), the algorithm aims to pick the right *offset* for fetching a block. For example, if a job is progressing quickly and is currently processing block-4, the prefetcher may choose to prefetch blocks from an offset 20; in contrast, it may start from an offset 5 for a slow job.

To further react dynamically to the prefetching requests, the prefetch controller obtains the list of all tasks that are launched since the beginning of the job and the set of tasks that have already completed. Thus, based on the task start time and completion time, the caching algorithm understands the distribution of the task execution times of the current job.

In a similar manner, the slave prefetchers periodically report the average time to prefetch and replicate an HDFS block and the bandwidth observed by them from the storage cloud to the enterprise site. Based on these reports, the cache controller understands the average time for a block prefetch operation and accordingly makes the prefetching decision. If map task is launched for an input split whose block is not prefetched, the prefetch controller understands that the prefetchers are unable to prefetch at a rate similar to the rate of processing the blocks and hence makes an intelligent decision to skip prefetching the next few blocks and start prefetching blocks that are n blocks after the currently processing block in the prefetch ordering. Concretely, if $mtime_{avg}$ represents the average map execution time of a job running on a cluster with M map slots on each task tracker and if $ptime_{avg}$ represents the average time to prefetch a block, then upon encountering a task launch for a map task t whose data block B_i is not prefetched, the cache controller skips the next few blocks and starts prefetching blocks after block B_{i+n} where

$$n = \frac{ptime_{avg}}{mtime_{avg}} \times M$$

The pseudocode of the prefetch controller is shown in Algorithm 1.

Cache eviction: Additionally, VNCache implements a cache eviction logic that closely monitors the job log and evicts the blocks corresponding to tasks that have already completed execution. It thus minimizes the total storage footprint resulting in a fraction of local storage used as compared to the conventional model in which the entire data set has to be stored in the enterprise cluster. Similar to cache prefetching, the cache manager sends direction to the slave daemons for evicting a data block upon encountering a task completion status in the job execution files. The daemons on the VMs evict the replicas of the block from the cache creating space in the cache for prefetching the next data block.

Algorithm 1 Distributed Cache Prefetching

```
1: J: a currently running MapReduce job
2: V: set of all Hadoop nodes in the system
3: Snodes: a subset of Hadoop nodes holding the replica of a data block
4: v: a variable representing a Hadoop node
5: M_i: number of Map slots in tasktracker of Hadoop node v_i
6: datablock: a variable storing the Data block object
7: tasklist: an array of map tasks to be launched by the job tracker
8: completedtasks: an array of map tasks that just completed execution
9: procedure PREFETCHCONTROLLER(J)
10:     tasklist = getOrderedTasks()
11:     // get the list of tasks in the predicted order of execution
12:     for i = 1 to |tasklist| do
13:         datablock = tasklist[i].getDatablock()
14:         Snodes = datablock.getStorageNodes()
15:         for v ∈ Snodes do
16:             sendprefetchsignal(v, tasklist[i])
17:             //signal slave prefetchers to prefetch data for task in tasklist[i]
18:         end for
19:     end for
20:     while (J.running() == true) do
21:         lasttask = getlastlaunchedtask(J)
22:         //check the last launched task by the job tracker
23:         mtime = getavgmaptime(J)
24:         // obtain the average map execution time for the Hadoop execution logs
25:         for i = 1 to |V| do
26:             prefetchtask = getPrefetchTask(v_i)
27:             if lasttask > prefetchtask then
28:                 ptime = getPrefetchTime(v_i)
29:                 //obtain the average prefetching time at the Hadoop node v_i
30:                 n = ptime/mtime × M_i
31:                 sendskipsignal(v_i, n)
32:                 //send signal to skip n blocks
33:             end if
34:         end for
35:     end while
36:     completedtasks = getcompletedtasks(J)
37:     for i = 1 to |completedtasks| do
38:         datablock = completedtasks[i].getDatablock()
39:         Snodes = datablock.getStorageNodes()
40:         for v ∈ Snodes do
41:             sendevictsignal(v, completedtasks[i])
42:             //send signal to evict the blocks for the completed task
43:         end for
44:     end for
45: end procedure
```

Workflow-awareness: When dealing with workflows (multiple back-to-back jobs processing a set of data), the cache manager understands the input and output data of the individual jobs and makes prefetch and eviction decisions based on the flow of data within the workflows. If a workflow has multiple jobs each processing the same input dataset, the cache prefetch logic recognizes it and prefetches the data blocks only once from the storage cloud and subsequent accesses to the data is served from the disk cache. Thus, the workflow-aware cache eviction policy makes its best effort to retain a data block in the cache if the workflow is processing that data block through another job in the future.

IV. Experimental Evaluation

We present the experimental evaluation of VNCache based on three key metrics: (1) *job execution time*: this metric captures the response time of the jobs. It includes data transfer time, data loading and decryption time, and job processing time. (2) *cache hit-ratio*: this metric captures the effectiveness of the VNCache's caching algorithm. It measures the amount of data read from the local disks of the enterprise site as compared to streaming from the public storage cloud. (3) *Cache size*: this metric captures the total storage footprint required at the enterprise site for processing a remotely archived data. It thus indirectly captures the storage equipment cost at the enterprise cluster.

We compare three techniques primarily:

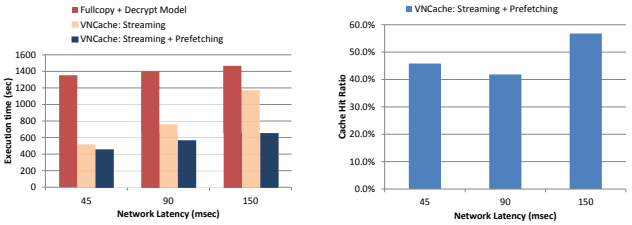
- *Full copy + Decrypt Model*: This technique downloads the entire dataset prior to processing and decrypts it and loads it onto the HDFS of the enterprise cluster. Therefore it incurs higher delay in starting the job.
- *VNCache: Streaming*: This technique incorporates the HDFS virtualization feature of VNCache and enables Hadoop jobs to begin execution immediately. It streams all data from the public storage cloud as blocks need to be accessed.
- *VNCache: Streaming + Prefetching*: It incorporates both the HDFS virtualization and streaming feature of VNCache and in addition, incorporates the VNCache prefetching and workflow-aware persistent caching mechanisms to improve job performance.

We begin our discussion with our experimental setup.

A. Experimental setup

Our cluster setup consists of 20 CentOS 5.5 physical machines (KVM as the hypervisor) with 16 core 2.53GHz Intel processors and 16 GB RAM. Out of these 20 servers, we considered 10 of them as the secure enterprise cluster nodes and used 5 other servers for functioning as public storage cloud servers. Our enterprise cluster had VMs having 2 2 GHz VCPUs and 4 GB RAM and by default we artificially injected a network latency of 90 msec (using the *tc* Linux command) between the public storage cloud and the enterprise cluster nodes to mimic the geographically separated scenario. Based on our cross-datacenter measurement experiments on Amazon EC2 and S3 (details explained in Section IV-B1), this latency setting mimics the scenario where the public storage cloud and the enterprise cluster are present within the same coast (Oregon and Northern California datacenters) but physically separated.

The FUSE-based VNCache filesystem is implemented in C using FUSE 2.7.3. Our Virtual HDFS and VNCache cache manager are implemented in Java. We use DES symmetric key encryption scheme for encrypting the blocks. We use four kinds of workloads in our study including the grep and sort workloads and the Facebook workload generated using the Swim MapReduce workload generator [9] that richly represent the characteristics of the production MapReduce traces in the Facebook cluster. The workload generator uses a real MapReduce trace from the Facebook production cluster and generates jobs with similar characteristics as observed in the Facebook cluster. The trace consists of thousands of jobs depending upon the trace duration. Out of these, we randomly pick up 5 jobs and use that as a representative sample. Each job processes 5 GB of data by default and uses 5 VMS, each having 2 2 GHz VCPUs and 4 GB RAM. In addition we consider two workflow-based workloads namely (i) *tf-idf* workflow and (ii) a workflow created as a combination of the



(a) Execution time (b) Cache hit ratio

Fig. 5: Performance of Grep Workload

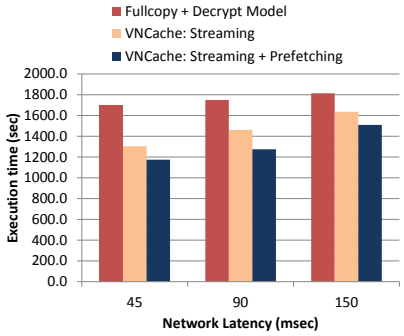
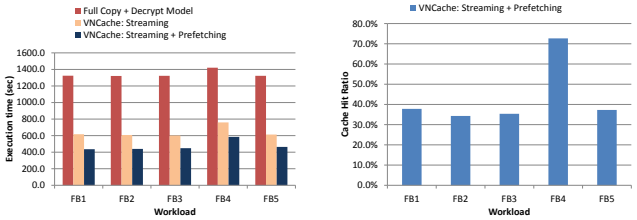


Fig. 6: Performance of Sort workload - Execution time



(a) Execution time (b) Cache hit ratio

Fig. 7: Performance of Facebook workload

jobs in the facebook workload trace. While the *tf-idf* workflow is reasonably compute-intensive, the facebook workflow is more data-intensive.

B. Experimental Results

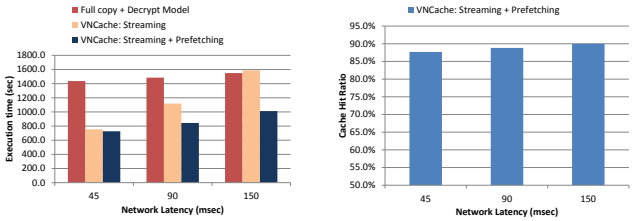
Our experimental results are organized in the following way. We first present the comparison of VNCache Streaming + Prefetching model with the basic full copy + decrypt model and the VNCache streaming model for the single job workloads. We analyze the job performance enhancements of VNCache under a number of experimental setting by varying the network latency between the public storage cloud and enterprise site, the size of the archived dataset, the size of the disk cache present in the enterprise site. We show the impact of both the HDFS virtualization and streaming techniques in VNCache as well as its caching and prefetching mechanisms on the overall job performance. We then present a performance study of our techniques by considering workflow-based workloads and show that VNCache performs better than the full copy + decrypt model even in such cases.

1) Impact of Network Latency: We study the performance of the VNCache approach for several network latencies representing various geographical distance of separation between the public storage cloud and the enterprise site. In order to simulate the scenarios of various degrees of geographic separation, we did cross-datacenter measurement experiments on Amazon EC2 and S3. As Amazon blocks ICMP packets and does not allow Ping based network measurements, we measured the average transfer time for transferring a file of HDFS block size (64 MB) between the datacenters and used that measurement to set our network latencies to obtain similar block transfer times. For example, with S3 server in Oregon and EC2 in Northern California, a 64 MB HDFS block file takes 11 seconds to get transferred. Here, the 90 msec network latency scenario represents the public storage cloud and enterprise site located within the same coast (Northern California and Oregon datacenters corresponding to 11 second transfer time in our measurements) and a 250 msec scenario would represent another extreme scenario where the public storage cloud at the west coast (Oregon site) and the compute site at the east coast (Virginia datacenter). Therefore, we use the 150 msec setting to represent a geographic separation that is in between these two extremes. In Figure 5(a), we present the execution time of the Grep workload at various latencies. We find that with increase in network latency, the execution time of the jobs increase for both the Fullcopy + decrypt model and the VNCache approaches. Here, VNCache: Streaming consistently performs better than the Fullcopy + decrypt model at various latencies showing an average reduction of 42% in execution time. Further, the execution time of the streaming approach is reduced by more than 30% by the prefetch optimization. As evident from the figure, this improvement comes from both the virtual HDFS based streaming model as well as through VNCache's intelligent prefetching. The cache hit ratios shown in Figure 5(b) illustrate that a significant amount of the input data (more than 45 %) were prefetched and read locally from the enterprise cluster.

We next consider the Sort workload. Figure 6 shows the execution time of the sort workload for the three approaches. Here we notice that VNCache achieves a reasonable improvement of 25% for even a compute-intense workload such as Sort.

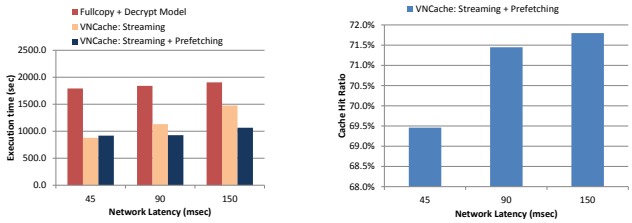
2) Performance of Facebook workload: Our next set of experiments analyze the performance of VNCache for the Facebook workload. Figure 7(a) shows the comparison of the execution time of the 5 Facebook jobs for streaming and streaming + prefetching techniques in VNCache and the basic Full copy + decrypt model. Here each job processes 5 GB of data and the network latency between the public storage cloud and enterprise cluster is 90 msec. We note that since the basic Full copy + decrypt model copies the entire (encrypted) dataset from the public storage cloud, decrypts and loads it into HDFS of the enterprise cluster, the jobs take longer time to execute.

VNCache streaming technique on the other hand uses its Virtual HDFS to start the job immediately while streaming the



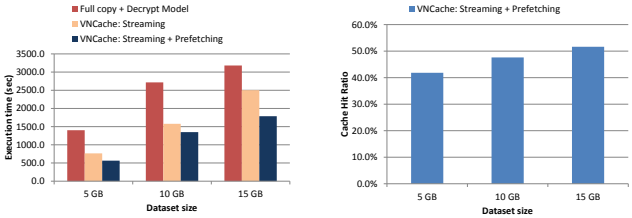
(a) Execution time (b) Cache hit ratio

Fig. 8: Performance of workflow (facebook jobs)



(a) Execution time (b) Cache hit ratio

Fig. 9: Performance of Tfidf workflow



(a) Execution time (b) Cache hit ratio

Fig. 10: Performance of Grep with different data size

required data on demand. We find that the streaming approach consistently achieves higher performance than the Fullcopy + decrypt model showing an average reduction of 52.3 % in job execution time for the jobs. Additionally, the VNCache prefetching techniques give further performance benefits to the jobs achieving an average reduction of 25.8% in execution time compared the VNCache streaming approach.

We present the obtained cache hit ratios for the VNCache: streaming + prefetching technique in Figure 7(b). We find that the prefetching optimization achieves an average cache hit ratio of 43.5% and thus serves 43.5% of the data from the local disks at the enterprise site as compared to streaming from the public storage clouds. These local reads contribute to the reduction in job execution times shown in Figure 7(a). We also notice that FB4 job has a higher cache hit ratio compared to the other jobs as its running time (excluding the data loading and loading time) is longer which gives more opportunity to interleave its compute and data prefetching resulting in higher local reads from prefetched data.

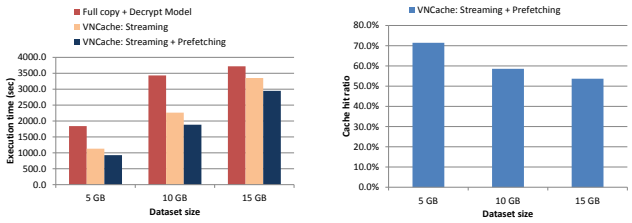
3) Performance of Job Workflows: Next, we study the performance of VNCache for job workflows that constitutes several individual MapReduce jobs. We first study the performance for a I/O intensive workflow composed of three randomly picked facebook jobs that process the same input dataset. As the three jobs in this workflow process the same dataset as input, we notice in Figure 8(a) that the VNCache:Streaming model is not too significantly better than the full copy model especially at some higher latency such as 150 msec. Here, since three individual jobs of the workflow use the same input dataset, streaming the data blocks for each of the three jobs becomes less efficient. Instead, the workflow-aware persistent caching approach in VNCache: Streaming + Prefetching caches the prefetched data at the enterprise site for the future jobs in the workflow and thereby achieves more than 42.2% reduction in execution time compared to the Full copy model. The cache hit ratios shown in Figure 8(b) shows that VNCache enables more than 88.8% of data to be read locally from the enterprise cluster for this workflow. Thus, the workflow-aware persistent caching avoids multiple streaming of the same block and helps the individual jobs read data within the enterprise cluster.

For a compute-intensive workflow, we use the *tfidf* workflow which computes the term frequency - inverse document frequency (*tf-idf*) for the various words in the given dataset. It consists of three jobs, the first two of which read the input dataset while the third job reads the output of the first two jobs. In Figure 9(a), we find that the job execution time for this workflow is again significantly reduced (by more than 47%) by VNCache. Also, the cache hit ratio in this case (Figure 9(b)) suggests that VNCache is able to prefetch a significant fraction (more than 70%) of the data.

4) Impact of Data size: Our next set of experiments vary the input dataset size for the jobs and study the performance of the individual jobs as well as the workflows. We present the execution time of Grep workload in Figure 10(a) for different input dataset size. We find that the techniques perform effectively for various datasets achieving an average reduction of 50.6% in execution time. We also find a good average cache hit ratio of 47% in Figure 10(b).

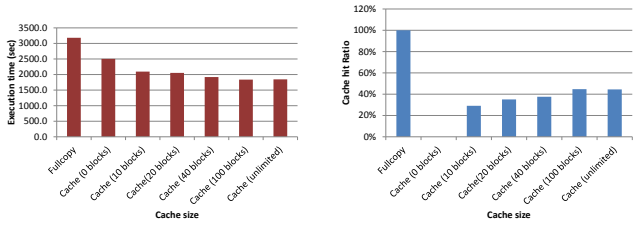
Similarly, for a compute-intensive workflow, we present the *tfidf* workflow performance for different dataset size. We find in Figure 11(a) that the VNCache techniques continue to perform well at even bigger dataset sizes with an average reduction of 35.9% in execution time. The performance improvement is further explained by the high average cache hit ratio (61.2 %) in Figure 11(b).

5) Impact of Cache Size: Next we study the impact of cache size at the enterprise cluster on the running time of the jobs. We vary the disk cache size on each VM in terms of the number of HDFS blocks that they can hold. Each HDFS block in our setting is 64 MB and we vary the cache size on the VMs from 10 to 100 blocks representing a per-VM cache of 640 MB to 6400 MB. We first study the performance of the Grep workload with cache sizes 10, 40, 100 blocks in Figure 12(a) and we find that the execution time of the VNCache:Streaming + Prefetching approach decreases with increase in cache



(a) Execution time (b) Cache hit ratio

Fig. 11: Performance of Tf-idf workflow with different data size



(a) Execution time (b) Cache hit ratio

Fig. 12: Impact of Cache size - Grep workload

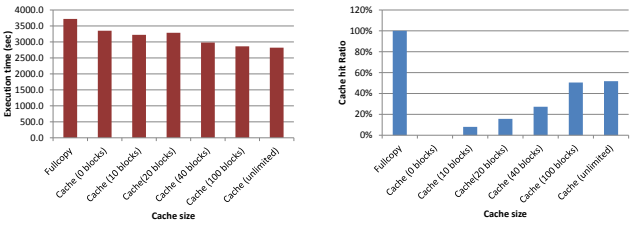
size as a larger cache gives enough opportunity to hold the prefetched data blocks. Here the cache size of 0 blocks refers to the VNCache's pure streaming approach. We find that even with a cache size of 10 blocks, VNCache achieves significantly lower execution time (Figure 12(a)) compared to the Fullcopy + decrypt model with a reasonable cache hit ratio (more than 35%) as shown in Figure 12(b).

The performance tradeoffs with cache size for the *tfidf* workflow shown in Figure 13(a) also shows that with a reasonable cache, the privacy-conscious enterprise can tradeoff job performance to save storage cost at the local cluster.

6) Effect of number of VMs: Our next set of experiments studies the performance of VNCache under different number of VMs in the Hadoop cluster. In Figure 14 we vary the Hadoop cluster size from 5 VMs to 10 VMs and compare the performance of VNCache (streaming + prefetching model) with the full copy + decrypt model. We find that VNCache continues to perform well at different cluster sizes achieving an average reduction of 51%.

V. Discussions

VNCache is developed with the goals of providing on-demand streaming and prefetching of encrypted data stored in public clouds. Here we discuss some of the merits of the design choice of implementing VNCache streaming and prefetching techniques at the Filesystem layer. We note that as an alternate solution, the Hadoop Distributed Filesystem (HDFS) can be modified to add the caching and prefetching techniques of VNCache. In a similar manner, HDFS can be also modified to implement additional functionalities such as encryption support for handling privacy-sensitive data. However, we argue that such an implementation suffers from two drawbacks. First, it can not seamlessly operate with Hadoop



(a) Execution time (b) Cache hit ratio

Fig. 13: Impact of Cache size - Tfidf workflow

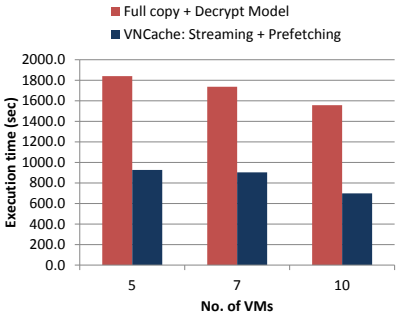


Fig. 14: Effect of number of VMs

as it requires changes to the Hadoop stack. Also, it makes it difficult to implement any changes to caching and prefetching policies as it requires modifying the Hadoop source each time. Additionally, implementing the HDFS virtualization and caching techniques at the Filesystem layer provides a seamless control point to introduce further optimizations such as dealing with storage hierarchies. For instance, VNCache can be easily extended to deal with in-memory processing of blocks by caching the blocks in a memory location and using a memory cache in addition to the disk cache. In a similar way, VNCache can also provide support to optimize for introducing SSDs into the solution where the data blocks can be moved between memory, SSDs and disks based on a prefetch/evict plan. One direction of our future research is focused on extending VNCache to optimize job latency through in-memory computations.

VI. Related Work

Hybrid Cloud solutions for MapReduce: There is some recent work on hybrid cloud architectures for security-conscious MapReduce applications [13], [14] that use public clouds for storing and processing non-private data while using a secure enterprise site for storing and processing private data. VNCache on the other hand addresses the challenge of processing archived (encrypted) data stored in public clouds in a privacy-conscious manner by providing a seamless interface for Hadoop to process the data within the enterprise site. Heintz et. al. [15] propose a solution to process geographically distributed data by scheduling map tasks close to their data. We note that such solutions are not suitable for security-conscious applications that prohibit the use of public clouds for data processing. Another direction of research is represented by

stream processing systems based on Hadoop such as Storm [19] and MapReduce Online [8] that stream data online instead of batch processing. However such solutions are not directly applicable in the context of privacy-conscious data processing and lack the caching angle of VNCache and hence these techniques stream all data during job execution without prefetch optimization.

Caching Solutions: Recently, caching techniques have been shown to improve the performance of MapReduce jobs for various workloads [11], [12]. The PACMan framework [11] provides support for in-memory caching and the MixApart system [12] provides support for disk based caching when the data is stored in an enterprise storage server within the same site. VNCache differentiates from these systems through its ability to seamlessly integrate data archived in a public cloud into the enterprise cluster in a security-conscious manner and through its seamless integration with Hadoop requiring no modifications to the Hadoop stack. Furthermore, VNCache provides a flexible control point to seamlessly introduce additional security-related functionality and other performance optimizations for storage hierarchies.

Locality Optimizations: In the past, there have been several efforts that investigate locality optimizations for MapReduce. Zaharia et al. [16] developed delay scheduler that attempts to improve job performance through increased task locality. Mantri [3] identifies that cross-rack traffic during the reduce phase of MapReduce jobs is a crucial factor for MapReduce performance and optimizes task placement. Quincy [18] is a resource allocation system for scheduling concurrent jobs on clusters considering input data locality. Purlieus [10] solves the problem of optimizing data placement so as to obtain a highly local execution of the jobs during both map and reduce phases. These above mentioned systems assume that the data is collocated with compute within the same Hadoop cluster and thus do not provide solutions for decoupled storage and compute clouds.

Resource Allocation and Scheduling: There have been several efforts that investigate efficient resource sharing while considering fairness constraints [23]. For example, Yahoo's capacity scheduler uses different job queues, so each job queue gets a fair share of the cluster resources. Facebook's fairness scheduler aims at improving the response times of small jobs in a shared Hadoop cluster. Sandholm et al [24] presented a resource allocation system using regulated and user-assigned priorities to offer different service levels to jobs over time. Zaharia et al. [21] developed a scheduling algorithm called LATE that attempts to improve the response time of short jobs by executing duplicates of some tasks in a heterogeneous system. Herodotou et al. propose *Starfish* that improves MapReduce performance by automatically tuning Hadoop configuration parameters [22]. The techniques in VNCache are complementary to these optimizations.

VII. Conclusions

This paper presents an efficient solution for privacy-conscious enterprises that deal with cloud-archived log data. We showed that current solutions are highly inefficient as they require large encrypted datasets to be first transferred to the secure enterprise site, decrypted, and loaded into a local Hadoop cluster before they can be processed. We present our filesystem layer called VNcache that dynamically integrates data stored at public storage clouds into a virtual namespace at the enterprise site. VNCache provides a seamless data streaming and decryption model and optimizes Hadoop jobs to start without requiring apriori data transfer, decryption, and loading. Our experimental results shows that VNCache achieves up to 55% reduction in job execution time while enabling private data to be archived and managed in a public cloud infrastructure.

In future, we plan to extend the principles and techniques developed in VNCache to deal with performance optimization for collocated compute and storage infrastructures where we plan to leverage VNCache to seamlessly introduce storage hierarchies including SSD based storage into the solution.

VIII. Acknowledgements

This work is partially supported by an IBM PhD fellowship for the first author. The fifth author acknowledges the partial support by grants under NSF CISE NetSE program, SaTC program and I/UCRC FRP program, as well as Intel ISTC on cloud computing.

References

[1] B. Igou "User Survey Analysis: Cloud-Computing Budgets Are Growing and Shifting; Traditional IT Services Providers Must Prepare or Perish".
[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha and E. Harris. Reining in the Outliers inMap-Reduce Clusters using Mantri. In *OSDI*, 2010.
[4] Hadoop DFS User Guide. *http://hadoop.apache.org/*.
[5] Hadoop Offline Image Viewer Guide. *http://hadoop.apache.org/docs/hdfs/current/hdfs_imageviewer.html*
[6] Proc Filesystem. *http://en.wikipedia.org/wiki/Procfs*.
[7] FUSE: Filesystem in User Space *http://fuse.sourceforge.net/*.
[8] T. Condie, N. Conway, P. Alvaro and J. M. Hellerstein MapReduce Online *NSDI*, 2010.
[9] Y. Chen, A. Ganapathi, R. Griffith, R. Katz The Case for Evaluating MapReduce Performance Using Workload Suites In *MASCOTS*, 2011.
[10] B. Palanisamy, A. Singh, L. Liu and B. Jain Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *SC*, 2011.
[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI*, 2012.
[12] M. Mihailescu, G. Soundararajan, C. Amza MixApart: Decoupled Analytics for Shared Storage Systems In *FAST*, 2013.
[13] K. Zhang, X. Zhou, Y. Chen, X. Wang, Y. Ruan Sedic: Privacy-Aware Data Intensive Computing on Hybrid Clouds In *CCS*, 2011.
[14] S. Ko, K. Jeon, R. Morales The HybrEx model for confidentiality and privacy in cloud computing In *HotCloud*, 2011.
[15] B. Heintz, A. Chandra, R. Sitaraman Optimizing MapReduce for Highly Distributed Environments *University of Minnesota, Technical Report TR12-003*, 2012.
[16] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling *EuroSys*, 2010.
[17] Hadoop. http://hadoop.apache.org.
[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.
[19] Storm *http://storm.incubator.apache.org/*
[20] Amazon Elastic MapReduce. *http://aws.amazon.com/elasticmapreduce/*
[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.
[22] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, S. Babu Starfish: A Selftuning System for Big Data Analytics. In *CIDR*, 2011.
[23] Scheduling in hadoop. *http://www.cloudera.com/blog/tag/scheduling/*.
[24] T. Sandholm and K. Lai. Mapreduce optimization using dynamic regulated prioritization. In *ACM SIGMETRICS/Performance*, 2009.