A Survey on Cloud Computing

AMIT GOYAL and SARA DADIZADEH
University of British Columbia, Vancouver

Cloud computing provides customers the illusion of infinite computing resources which are available from anywhere, anytime, on demand. Computing at such an immense scale requires a framework that can support extremely large datasets housed on clusters of commodity hardware. Two examples of such frameworks are Google's MapReduce and Microsoft's Dryad. First we discuss implementation details of these frameworks and drawbacks where future work is required. Next we discuss the challenges of computing at such a large scale. In particular, we focus on the security issues which arise in the cloud: the confidentiality of data, the retrievability and availability of data, and issues surrounding the correctness and confidentiality of computation executing on third party hardware.

1. INTRODUCTION

Today, the most popular applications are Internet services with millions of users. Websites like Google, Yahoo! and Facebook receive millions of clicks daily. This generates terabytes of invaluable data which can be used to improve online advertising strategies and user satisfaction. Real time capturing, storage, and analysis of this data are common needs of all high-end online applications.

To address these problems, a number of cloud computing technologies have emerged in last few years. Cloud computing is a style of computing where dynamically scalable and virtualized resources are provided as a service over the Internet. The cloud refers to the datacenter hardware and software that supports a clients needs, often in the form of datastores and remotely hosted applications. These infrastructures enable companies to cut costs by eliminating the need for physical hardware, allowing companies to outsource data and computations on demand. Developers with innovative ideas for Internet services no longer need large capital outlays in hardware to deploy their services; this paradigm shift is transforming the IT industry. The operation of large scale, commodity computer datacenters was the key enabler of cloud computing, as these datacenters take advantage of economies of scale, allowing for decreases in the cost of electricity, bandwidth, operations, and hardware [Armbrust et al. 2009].

It is well known that writing efficient parallel and distributed applications is complex. Google proposed the MapReduce [Dean and Ghemawat 2004] programming framework in 2004 to address this complexity. It allows programmers to specify a *map* function that processes a key/value pair to generate an intermediate key/value pairs, and a *reduce* function that merges all the intermediate key/value pairs to produce the required output. Many real world tasks, especially in the domain of search can be expressed in this model.

Hadoop[1] is the most popular open source implementation of MapReduce. It has been widely adopted both in academic and industrial users, including at organiza-

[1]The Apache Hadoop Project. `http://hadoop.apache.org`

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

Fig. 1. Map function for counting.

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Fig. 2. Reduce function for counting.

tions such as Apache, Cornell University, Yahoo!, Facebook, IBM, Microsoft and many others[2]. Recently, [Zaharia et al. 2008] identifies performance problems with Hadoop's scheduler in heterogeneous environment. They proposed a new scheduling algorithm: Longest Approximate Time to End (LATE) to address the issue and showed the improvement in response times by a factor of 2. In parallel, [Sandholm and Lai 2009] also proposed MapReduce optimizations using Regulated Dynamic Prioritization. Moreover, [Isard et al. 2007] and [Yu et al. 2008] proposed Dryad and DryadLINQ, a general purpose execution environment for distributed, data parallel applications. The applications are modeled as directed acyclic graphs in this framework. It enables Dryad to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers. In addition, it allows automatic management of scheduling, distribution and fault tolerance.

Although the advantages of using clouds are unarguable, there are risks involved with releasing data onto third party servers. A client places her computation and data on machines she cannot control, and the provider agrees to run a service whose details she does not know. It is natural for the client to have concerns about the data confidentiality, security and integrity. There is a clear need for technical solutions so clients can be confident about the security and integrity of their data in the face of an untrusted cloud.

In this paper we first explore the various frameworks which provide the resources to make computing at an immense scale possible, then discuss the confidentiality, security and integrity challenges presented by this infrastructure.

2. MAP-REDUCE

In this section, we describe the MapReduce framework, its implementation and refinements originally proposed in [Dean and Ghemawat 2004; 2008].

2.1 Programming Model

The processing is divided into two steps: *Map* and *Reduce*. *Map* take key/value pair as input and generate intermediate key/value pairs. *Reduce* merge all the pairs associated with the same (intermediate) key and then generates output. Many real world tasks, especially in the domain of search can be expressed in this model. As an example, consider the problem of counting the number of occurences of each word in the collection of documents. The two functions are given in Figure 1 and 2.

[2]Technologies Powered by Hadoop. `http://wiki.apache.org/hadoop/PoweredBy`

In the *map* function, one document is processed at a time and the corresponding count (just '1' in simple example) is emitted. Then, in the second phase, the *reduce* function sums together the count of each word and emit it. Abstractly, the two phases can be represented as follows

$$\begin{aligned} &\text{map:} && (k1, v1) \rightarrow \mathtt{list}(k2, v2) \\ &\text{reduce:} && (k2, \mathtt{list}(v2)) \rightarrow \mathtt{list}(v2) \end{aligned}$$

2.2 Implementation

2.2.1 *Execution Overview.* Figure 3[3] gives an overview of the execution. The data is split into a set of M small (say 64M) *splits*. The driver (or master) machine is responsible for the co-ordination process. It remotely forks many mappers, then reducers. Each mapper read file splits from Google File System [Ghemawat et al. 2003], parses the key/value pairs out of it, and applies the user-defined logic (*Map* function). The output is store in local filesystem locations. The information about the locations are sent to master. When a *reducer* is notified of these locations by the master, it uses Remote Procedure Calls to read the data from the local disks of the *mappers*. Once a *reducer* has read all the intermediate data, it sorts it by the intermediate keys so that all the occurences of the same key are grouped together. An external sort may be used in case the data is too large to fit in memory. Then, the user defined *reduce* function is applied on the sorted data. Finally, the output is saved on the global disks.

2.2.2 *Fault Tolerance.* Instead of using expensive, high-performance, and reliable multiprocessing (SMP) or massively parallel processing (MPP) machines equipped with high-end network and storage subsystems, the map-reduce framework uses large clusters of commodity hardware. Thus, failure of machines is a common phenomenon. Hence, it is designed to handle the fault tolerance in a simple and easy to administer manner. The master pings every *mapper* and *reducer* periodically. If no response is received for a certain time window, the machine is marked as failed. Ongoing task(s) and any task completed by the *Mapper* is reset back to the intial state and re-assigned by the master to other machines from scratch. Completed *map* tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed *reduce* tasks do not need to re-executed since their output is stored in the global file system. The probability of failure of master is very less as it is a single machine. In case if the master fails, then the program has to be started from scratch.

2.2.3 *Locality.* The authors [Dean and Ghemawat 2004] observed that network bandwidth is precious and scarce. The input data is stored using GFS on the local disks of the machines that makes up the cluster as well. GFS divides each file into 64 MB blocks, and stores several copies (typically 3) of each block on different machines. The MapReduce master use this location information of the input data and attempts to schedule a *map* task on a machine that contains a replica of the corrsponding input data. Failing that, it attempts to schedule a *map*

[3]Figure taken from [Yang et al. 2007]

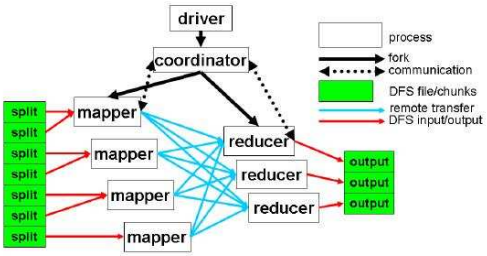

Fig. 3. Execution Overview for MapReduce.

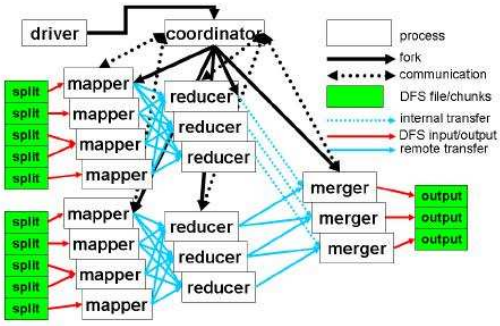

Fig. 4. Execution Overview for Map-Reduce-Merge.

task on a machine which is "closest" to the data. Usually, the machine is on the same network. Using this heuristic, the network bandwidth consumption by *map* tasks is minimal.

2.2.4 *Backup Tasks.* There are times when some nodes perfom poorly, a condition that is called "straggler". Straggler can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to the competition for CPU, memory, local disk, or network bandwidth. When a MapReduce operation is close to completion, the master schedules backup execution of the remaining *in-progress* tasks. Whenever the task is completed, either via primary or the backup execution, it is marked completed. It is found that this strategy significantly reduces the time to complete large MapReduce operations. As an example, the sort program tkaes 44% longer to complete when the backup task mechanism is disabled.

3. RECENT EXTENSIONS TO MAP-REDUCE

3.1 Map-Reduce-Merge

Most web scale search engines like Google, Yahoo!, instead of relying on generic DBMS (database management systems) for data processing and indexing, build their own customized systems. MapReduce framework as described above is one such example. Though search engines have very specific focus compared to DBMS, a lot of problems are common in both the systems. Hence, it is natural to learn and apply techniques from one field to the other. [Yang et al. 2007] tries to extend the MapReduce framework for DBMS operations.

Typically, the most expensive operations in DBMS systems are *joins*. A *join* clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A *join* is a means for combining fields from two tables by using values common to each. There are different kinds of joins in SQL (or DBMS): *inner-join, outer-join, self-join.* Three fundamental algorithms are used for performing a join operation: *nested loops, merge-join* and *hash-join* [Pratt 1995].

MapReduce is designed primarily for the data processing problems in search engines. Hence, it focuses mainly on processing homogeneous datasets. For example, all the *mappers* assumes the input data among all the files has the same schema. Although MapReduce can be made to process heterogenous datasets, there are two problems in it: It is not clean, the users may end up writing awkward map/reduce code [Yang et al. 2007]. Second, it is inefficient. In particular, [Pike et al. 2005] indicates that *joining* multiple heterogenous datasets does not quite fit into the MapReduce framework, although it can still be done with extra Map-Reduce steps.

To address the issue of heterogenity, and thus extending the MapReduce framework to efficiently and effectively process the *join* queries in DBMS, [Yang et al. 2007] proposed adding an extra step "merge" in the MapReduce framework. The signature of the new framework are as follows, where α, β, γ represent dataset lineages, k means keys, and v stands for value entities.

$$\begin{aligned}
&\text{map:} && (k1, v1)_\alpha \rightarrow \mathtt{list}(k2, v2)_\alpha \\
&\text{reduce:} && (k2, \mathtt{list}(v2))_\alpha \rightarrow (k2, \mathtt{list}(v3))_\alpha \\
&\text{merge:} && (k2, \mathtt{list}(v3))_\alpha, (k3, \mathtt{list}(v4))_\beta \rightarrow \mathtt{list}((k4, v5)_\gamma
\end{aligned}$$

The execution overview is shown in figure 4. In this extended model, the *map* function transforms an input key/value pair $(k1, v1)$ into a list of intermediate key/value pairs $\mathtt{list}(k2, v2)$. The *reduce* function aggregates the list of values $\mathtt{list}(v2)$ associated with $k2$ and produces a list of values $\mathtt{list}(v3)$, which is also associated with $k2$. Note that inputs and outputs of both functions belong to the same lineage, say α. Another pair of *map* and *reduce* functions produce the intermediate output $(k3, \mathtt{list}(v4))$ from another lineage, say β. Based on keys $k2$ and $k3$, the *merge* function combines the two reduced outputs from different lineages into a list of key/value outputs $\mathtt{list}(k4, v5)$. This final output becomes a new lineage, say γ. If $\alpha = \beta$, then this merge function does a *self-merge*, similar to *self-join* in relational algebra.

The authors have proposed algorithms all three basic algorithms for joins can be implemented using this new Map-Reduce-Merge model. In addition to *joins*, they have suggested algorithms to implement primitive and some derived relational operators using this framework. These operators include *Projection, Aggregation, Generalized Selection, Set Union, Set Intersection, Set Difference, Cartesian Product and Rename*. Thus, they claim that this Map-Reduce-Merge framework is *relationally complete*. It has a potential to change how the traditional databases systems work.

3.2 Longest Approximate Time to End (LATE)

As mentioned above, Hadoop[4] is an open source implementation of MapReduce. Nowadays. Hadoop is embraced by a variety of academic and industrial users, including Apache, Cornell University, Yahoo!, Facebook, IBM, Microsoft and many others[5]. The performance of Hadoop depends largely on its scheduler, which implicitly assumes that the cluster nodes are homogeneous and tasks make progress

[4] The Apache Hadoop Project. `http://hadoop.apache.org`
[5] Technologies powered by Hadoop. `http://wiki.apache.org/hadoop/PoweredBy`

linearly, and uses these assumptions to decide when to speculatively re-execute (backup) tasks that appears to be stragglers. In practice, the homogenity assumptions do not always hold. An especially compelling environment where Hadoop's scheduler is inadequate is a virtualized data center. As mentioned above, virtualized "utility computing" environments, such as Amazon's Elastic Compute Cloud (EC2)[6], are becoming an important tool for organizations that must process large amounts of data, because large number of virtual machines can be rented by the hour at lower costs than operating a data center year round (EC2's current cost is $0.10 per hour). [Zaharia et al. 2008] shows that hadoop's scheduler can cause severe performance degradation in heterogenous environments and proposed a new scheduling algorithm, Longest Approximate Time to End (LATE), that improves Hadoop's response time by a factor or 2. In the following, we explain the Hadoop's scheduler followed by LATE.

3.2.1 *Hadoop's Scheduler.* The goal of Hadoop's Scheduler is to minimize a job's response time. Response time is important in a pay-by-the-hour environment like EC2. Hadoop monitors task progress using a *progress score* between 0 and 1. For a *map* , the progress score is the fraction of input data read. For a *reduce* task, the execution is divided into three phases, each of which accounts for 1/3 of the score: (i) the *copy* phase, when the task fetches *map* outputs; (ii) the *sort* phase, when *map* outputs are sorted by key and; (iii) the *reduce* phase, when a user-defined function is applied to the list of *map* outputs with each key. In each phase, the score is the fraction of the data processed. For example, a task halfway through the *copy* phase has a progress score of $1/2 * 1/3 = 1/6$, while a task halfway through the *reduce* phase scores $1/3 + 1/3 + 1/2 * 1/3 = 5/6$.

Hadoop looks at the average progress score of each category of tasks (maps and reduces) to define a *threshold* to identify *stragglers*: When a task's progress score is less than the average for its category minus 0.2, and the task has run for at least one minute, it is marked as a *straggler*. This type of scheduling implicitly makes the following assumptions: (i) Nodes can perform work at roughly the same rate. (ii) Tasks progress at a constant rate throughout time. (iii) There is no cost to launching a speculative backup task on a node that would otherwise have an idle slot. (iv) A tasks progress score is representative of fraction of its total work that it has done. Specically, in a *reduce* task, the copy, sort and reduce phases each take about 1/3 of the total time. (v) Tasks tend to finish in waves, so a task with a low progress score is likely a straggler. (vi) Tasks in the same category (*map* or *reduce*) require roughly the same amount of work.

It is easy to see how these assumptions break in heterogenous environments. Hadoop assumes that a node is slow only if it faulty, but it can be slow because there may be other processes running on the same node, which is common in virtualized data centers as described above. Thus, the first two assumptions break in heterogenous environments. In a pay-by-the-hour environment, there is a cost attached to launch a backup task, hence assumption (iii) is not valid. Assumption (iv) attach equal expected time to all three phases, which may not be true for many tasks. Because of assumption (v), Hadoop may execute backup tasks for new, fast

[6] Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2`

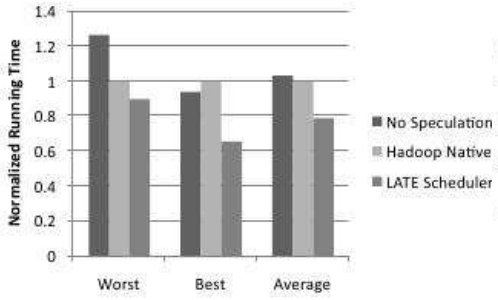

Fig. 5. EC2 Sort running times in heterogenous cluster

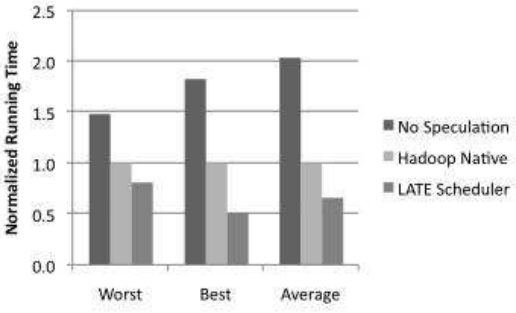

Fig. 6. EC2 Sort running times with stragglers

tasks instead of old, slow tasks that have more total progress. The paper does not deal with assumption (vi) effectively.

3.2.2 *LATE Scheduler.* Instead of focusing on *progress score*, LATE focus on expected time left. Different methods can be used to estimate expected time left. For the sake of simplicity, LATE scheduler estimate it as *(1-ProgressScore)/ProgressRate*, where *ProgressRate* for each task is defined as *ProgressScore/T*. Here T is the amount of time the task has been running. There are cases where this heuristic can fail, but it is effective in typical Hadoop jobs. The scheduler also have upper bound on the number of backup tasks that can be run at once. Moreover, it doesn't run backup copy for young tasks (determined by a threshold again). LATE execute backup copy of only those tasks that will improve the job response time. For example, if task A is 5x slower than the mean but has 90% progress, and task B is 2x slower than the mean but is only at 10% progress, then task B will be chosen for speculation first, even though it is has a higher *progress rate*, because it hurts the *response time* more.

3.2.3 *Results.* Figure 5 and 6[7] shows the effectiveness of LATE scheduler. In summary, LATE improves Hadoop's response time by a factor of 2 on an average.

3.3 MapReduce Optimization Using Regulated Dynamic Prioritization

In the paper [Sandholm and Lai 2009], the authors extend the MapReduce scheduling to take into account the user-preferences. In a virtualized data center, customer often wants answers to the questions: How much do I want to spend? How do I want to spend? Should I spend more or less? At the same time, customers want some controls over what jobs should have more priority compared to others. Using the system proposed in the paper, users can change the priority of an application over time and assign different priority to different components. This is regulated by limiting the total priority the users can assign. Due to the lack of space, we omit the details.

[7] Figures taken from [Zaharia et al. 2008]

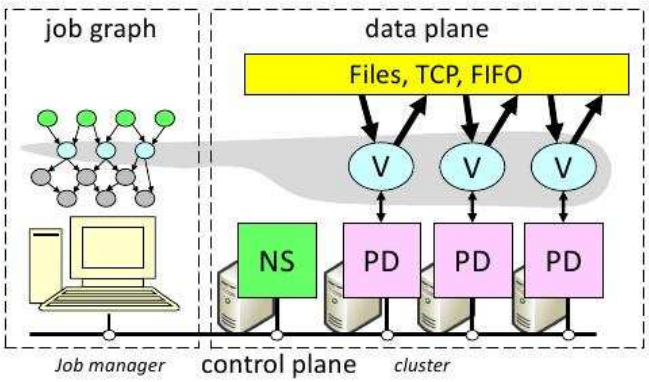

Fig. 7. Dryad system architecture. NS is the name server which maintains the cluster membership. The job manager is responsible for spawning vertices (V) on available computers with the help of a remote-execution and monitoring daemon (PD). Vertices exchange data through les, TCP pipes, or shared-memory channels. The grey shape indicates the vertices in the job that are currently running and the correspondence with the job execution graph.

4. OTHER FRAMEWORKS

4.1 Dryad

Dryad [Isard et al. 2007] is a cloud computing framework proposed by Microsoft Research. An application written for Dryad is modeled as a directed acyclic graph (DAG). The DAG defines the dataflow of the application; here each vertex is a program and the edges represents the data channel. It enables Dryad to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers. The Dryad execution engine is capable of handling all the issues in creating a large distributed cloud computing framework: scheduling the jobs to the computers taking into account various factors, fault tolerance and data transportation between vertices.

Figure 7[8] illustrates the Dryad system architecture. The execution of a Dryad job is orchestrated by a centralized "job manager". The job manager is responsible for: (i) instantiating a jobs dataow graph; (ii) determining constraints and hints to guide scheduling so that vertices execute on computers that are close to their input data in network topology; (iii) providing fault-tolerance by re-executing failed or slow processes; (iv) monitoring the job and collecting statistics; and (v) transforming the job graph dynamically according to user-supplied policies. The job manager may contain its own internal scheduler that decides what modules to execute on what machines. Data transfers takes place directly between the vertices and thus the job manager is only responsible for control decisions and is not a bottleneck for any data transfers.

The name server (NS) sends the list of all the available compute nodes with their location on the cluster to the job manager. The location information can be utilized

[8]Figure taken from [Isard and Yu 2009]

for scheduling decisions to take into account the locality. There is a simple daemon (PD) running on each cluster machine that is responsible for creating processes on behalf of the job manager. The first time a vertex (V) is executed on a machine its code is sent from the job manager to the daemon, or copied from a nearby computer that is executing the same job, and it is cached for subsequent uses. The daemon acts as a proxy so that the job manager can talk to the remote vertices and monitor the state and progress of the computation.

We omit the details of Dryad Execution DAG and programming model due to the lack of space, but they can be find in [Isard et al. 2007].

4.2 Query Based Frameworks

Both MapReduce and Dryad paradigms are low-level and rigid, and may lead to difficulties in code maintainence and reuse. All three big organizations: Yahoo!, Google and Microsoft built query based frameworks on top of these low level frameworks. Google developed Sawzall [Pike et al. 2005] on top of MapReduce; Yahoo! developed PigLatin [Olston et al. 2008] on top of Hadoop and Microsoft developed DryadLINQ [Yu et al. 2008] based on Dryad and LINQ. All these frameworks allows users to express their tasks in high-level SQL-like queries without worrying about the details of dependency analysis. All the three papers claim that these query based frameworks have significantly reduced the development time of the applications in their organizations. In this paper, we don't go into the details of query based frameworks.

5. CHALLENGES FACING THE CLOUD

Concerns surrounding the confidentiality and integrity of data and computation are a major deterrent for enterprises looking to embrace cloud computing. In recent years, there has been a lot of research exploring the various methods which can be used to provide trust in the cloud. In this section we give an overview of some of these issues, and solutions which have been proposed.

5.1 Data Confidentiality

Clients must have a mechanism to ensure that their data is secure and private in an untrusted cloud. This can be realized through cryptographic methods, where a client can verify the integrity of his remote data by storing a hash in local memory and authenticating server responses by re-calculating the hash of the received data and comparing it to the locally stored value. When dealing with large datasets, this method is implemented using hash trees, where the leaves of the tree are hashes of data blocks, and internal nodes are hashes of their children. A user verifies a given data block by storing the root hash of the tree; this results in a logarithmic number of cryptographic operations in the number of blocks [Cachin et al. 2009].

Recent research has focused on efficiency of cryptographic methods, in particular [Papamanthou et al. 2008] propose a mechanism to verify the correctness of server responses to queries, as well as integrity of stored data. They augment a hash table with a secure digest, a "fingerprint" of the entire stored dataset, which serves as a secure set description to which the answer to a query will be verified at the client, using a corresponding proof provided by the server. They meet their efficiency

goals, as their method is the first which authenticates a hash table with constant query cost and sublinear update cost.

5.2 Data Retrievability and Availability

The methods described above allow a user to verify the integrity of data returned by a server, however aonther concern is whether a clients data is still in the cloud, and still accessible. When storing large amounts of data, it is infeasible to download all the data, just to ensure that it is stored correctly. Instead, [Juels and Kaliski 2007] introduce proofs of retrievability (PORs), a scheme which allows a provider to assure a client that his data is retrievable with high probability.

In the protocol, the provider encrypts a file and injects additional information into random blocks in the file before storing it. If the provider has corrupted or deleted a significant portion of the file, then it can be shown with high probability that it has also corrupted the injected blocks. If a client samples the blocks at these well-known locations, and the file has been corrupted, then the provider is unlikely to respond correctly to the client, and thus the client knows that his data is no longer correct. In this scheme, the provider incurs a small, nearly constant overhead in compuational complexity and communication - it need only store a single cryptographic key (no matter the size or number of files it needs to verify), along with a small amount of dynamic state (tens of bits) for each file.

We note that this POR system protects only static files; it requires extension to cover the case when a client partially modifies a file, only changing a subset of the blocks. [Wang et al. 2009] devise a scheme that supports dynamic operations on data blocks (i.e. update, delete and append), and they show through extensive analysis that their method is highly efficient and resilient against attacks and failures. Further research built upon [Juels and Kaliski 2007] provides support against a fully Byzantine adversarial model [Bowers et al. 2008].

[Bowers et al. 2009] point out that PORs are of limited value in a single server environment, when a client can detect that a file is irretrievable and then has no recourse. They propose the High-Availability and Integrity Layer (HAIL), which improves upon PORs deployed on individual servers, providing protection against an active, mobile adversary that may progressively corrupt the full set of servers. When a client discovers file corruption on a particular server, she can proposition the other servers for file recovery. HAIL works by combining within-server redundancy (through PORs) and cross-server redundancy (through traditional distributed file system protocols).

5.3 Trusting Computation

As described above, a client can encrypt data stored on a cloud to ensure privacy, but this is not possible when compute services are requested, as the unencrypted data must reside in the memory of the host running the computation. Amazon's EC2, and other IaaS (Infrastructure as a Service) cloud services typically host virtual machines (VMs) where a clients computation can be executed. In these systems, anyone with privileged access to the host can read and manipulate client data.

Given this security hole, cloud service providers are going to great lengths to secure their systems against insider attacks. They restrict access to hardware facil-

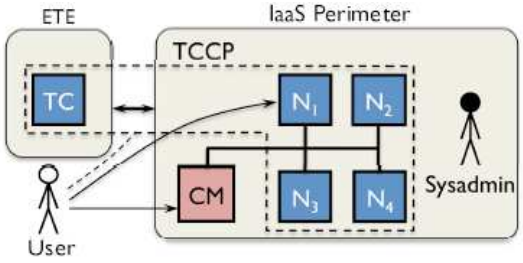

Fig. 8. The trusted cloud computing platform includes a set of trusted nodes (N) and a trusted coordinator (TC). Users talk to the cloud manager (CM) to request services. The TC is maintained by an external trusted entity (ETE) in order to provide isolation from the IaaS perimeter.

ities, adopt stringent accountability and auditing procedures, and minimize access to critical components of the infrastructure. Despite these efforts, customers VMs are still susceptible to insiders, and clients need a solution that guarantees the confidentiality and integrity of their computations in a way that is verifiable [Peltier et al. 2003]. [Santos et al. 2009] propose a trusted cloud computing platform (TCCP) which enables IaaS providers to serve a closed box execution environment that guarantees confidential execution of guest VMs. This system allows a customer to verify if its computation will run securely, before requesting the service to launch a VM.

TCCP is based off a trusted computing platform called Terra [Garfinkel et al. 2003], which uses a trusted virtual machine monitor (TVMM) to partition a single platform into multiple isolated VMs. The TVMM allows an application in a VM to authenticate itself to remote parties in a process called attestation. Attestation must identify each component of the software stack; this is achieved by building a certificate chain, starting from tamper-resistant hardware all the way up to a VM. Terra uses private, permanent keys embedded in a tamper-resistant chip to certify the system firmware (i.e. the BIOS). This firmware certifies the system boot loader, which then certifies the TVMM, which can finally certify the VMs that are loaded. TCCP follows a similar method, using the trusted platform module (TPM) chip which is bundled with commodity hardware. Attestation is performed using simple public key cryptography to authenticate between a remote party and a platform running on an untrusted host.

Cloud providers house datacenters with many machines, and a client VM is dynamically scheduled to run on any machine. This means that the attestation procedure described above must be implemented on each node in the cloud. A sysadmin, however, can divert a clients VM to a node not running the platform, either when the VM is launched, or during execution (using migration, which is supported by Xen). Although Terra was successful in a single host environment, we require a platform with stronger guarantees in the face of an untrusted cloud. TCCP provides these stronger guarantees by combining a TVMM with a trusted coordinator (TC).

Figure 8[9] outlines the components of this platform. The TC manages the set of "trusted nodes" - the nodes that can run a client VM securely. A node is trusted if it is both running the TVMM, and it resides inside the security perimeter (i.e. it is physically inaccessible to attackers). The TC needs to record the nodes located in the security perimeter, and attest to the node's platform to ensure that the node is running a trusted platform implementation. A client can then verify whether the IaaS secures its computation by attesting to the TC. The key to preventing insider attacks is the fact that the TC is hosted by an external trusted entity (ETE), which securely updates the information provided to the TC about the nodes in the perimeter. Admins with privileged access to the servers do not have access to the ETE, and they therefore cannot tamper with the TC.

Although TCCP provides an execution environment that guarantees confidential execution of guest VM's, we note that moving the TC onto an external server poses other risks. We must be assured that the TC is running on a trusted server, and also that communication between the TC and IaaS perimeter is secure. We also note that while there is a working version of the Terra system, TCCP does not yet have a working prototype.

5.4 Accountability

In addition to the security challenges we have described, there remains the issue of accountability in the face of incorrect behaviour. If data leaks to a competitor, or a computation returns incorrect results, it can be difficult to determine whether the client or the service provider are at fault. [Haeberlen 2009] argues that the cloud should be made accountable to both the client and the provider, and in the event of a dispute, there should be a way to prove the presence of the problem to a third party (i.e. a judge).

Traditionally, a client maintains a server farm on their premises, where the client has physical access to the machines, and can directly oversee the maintenance and management of the machines. When this job is outsourced, management of these machines is delegated to the service provider, and the client has relinquised control over her computation and data. When an issue does arise, it is difficult to agree on who is responsible - the provider will blame the clients software, while the client will place blame on the provider. The lack of reliable fault detection and attribution not only deters potential customers, but it may also prevent certain applications from being hosted on the cloud, when strict laws regulate the release of sensitive data (for example, personal health information) [ama].

[Haeberlen 2009] proposes the notion of accountability: "a distributed system is accountable if a) faults can be reliably detected, and b) each fault can be undeniably linked to at least one faulty node". They do not have a prototype of this system, but their goal is to implement a primitive called $\mathsf{AUDIT}(A, S, t_1, t_2)$ that can be invoked by a client to verify whether the cloud has fulfilled agreement A for service S in the time interval t_1, t_2. AUDIT either returns OK, or some evidence that S has failed to meet agreement A. At first it may seem as though accountability only benefits the client, and places a greater burden on the provider. However the provider has incentives to adopt this system as well: aside from the obvious

[9] Figure taken from [Santos et al. 2009]

reason, that it may attract previously reluctant customers, the provider can now proactively detect and diagnose problems.

Some of the necessary steps in reaching this goal including building a system which supports tamper-evident logs, virtualization-based replay, trusted timestamping, and sampling using checkpoints. Due to lack of space we refer the reader to [Haeberlen 2009] for more details. The accountable cloud is still being designed, and questions remain about whether such a system can be put in place with acceptable performance.

6. CONCLUSION

The long held dream of computing as a utility is finally emerging. The cloud provides the infrastructure necessary to provide services directly to customers over the Internet. In this survey, we studied various frameworks in detail, laying special emphasis on MapReduce which has emerged as the most popular framework. We also surveyed recent extensions of MapReduce and other frameworks like Dryad. In addition, we studied these systems from a security point of view.

Several challenges remain. Even though most of these frameworks come with debuggers, they are naive and not very effective. Security remains the biggest barrier preventing companies from entering into the cloud. Moreover, dynamic scheduling in heterogenous environments is still an issue in MapReduce like frameworks.

REFERENCES

Amazon web services. tc3 health case study. `http://aws.amazon.com/solutions/case-studies/tc3-health`.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. 2009. Above the clouds: A berkeley view of cloud computing. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

Bowers, K. D., Juels, A., and Oprea, A. 2008. Proofs of retrievability: Theory and implementation. Cryptology ePrint Archive, Report 2008/175. `http://eprint.iacr.org/`.

Bowers, K. D., Juels, A., and Oprea, A. 2009. Hail: a high-availability and integrity layer for cloud storage. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 187–198.

Cachin, C., Keidar, I., and Shraer, A. 2009. Trusting the cloud. *SIGACT News 40,* 2, 81–86.

Dean, J. and Ghemawat, S. 2004. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 10–10.

Dean, J. and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51,* 1, 107–113.

Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. 2003. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th SOSP*.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, New York, NY, USA, 29–43.

Haeberlen, A. 2009. A case for the accountable cloud. In *Proceedings of the 3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'09)*.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, New York, NY, USA, 59–72.

Isard, M. and Yu, Y. 2009. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 987–994.

Juels, A. and Kaliski, Jr., B. S. 2007. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 584–597.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 1099–1110.

Papamanthou, C., Tamassia, R., and Triandopoulos, N. 2008. Authenticated hash tables. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 437–448.

Peltier, T. R., Peltier, J., and Blackley, J. 2003. *Information Security Fundamentals*. Auerbach Publications, Boston, MA, USA.

Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Sci. Program. 13,* 4, 277–298.

Pratt, P. J. 1995. *A Guide to SQL*. International Thomson Publishing Company.

Sandholm, T. and Lai, K. 2009. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. ACM, New York, NY, USA, 299–310.

Santos, N., Gummadi, K. P., and Rodrigues, R. 2009. Towards trusted cloud computing. In *Proceedings of the Workshop On Hot Topics in Cloud Computing (HotCloud)*.

Wang, C., Wang, Q., Ren, K., and Lou, W. 2009. Ensuring data storage security in cloud computing. Cryptology ePrint Archive, Report 2009/081. `http://eprint.iacr.org/`.

Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. S. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 1029–1040.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., and Currey, J. 2008. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA*. 1–14.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. 2008. Improving mapreduce performance in heterogeneous environments. In *OSDI'08: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA*. 29–42.