Programming Abstractions and Security for Cloud Computing Systems

By

Sapna Bedi

A MASTERS'S ESSAY SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2010

Abstract

In cloud computing, resources and infrastructure are delivered to a client's computer over the internet. Cloud computing offers a number of advantages to its users such as scalability, economy and location independent access to data. A popular application of cloud computing is cloud storage, where an organization outsources its data for storage on a remote server. This gives the organization a flexible, economical and easily accessible store for its enormous amount of data. However, since the data is stored on an external server, cloud storage involves risk of information leakage and data corruption. Hence security measures must be deployed to ensure the privacy and integrity of this outsourced data. In addition to outsourcing data, information processing tasks involving large datasets can also be outsourced to the cloud. But in such cases, the developer faces the challenging task of scheduling the large scale distributed computation on a set of computers. Programming abstractions such as MapReduce and Dryad allow the developer to write programs for these large scale computations, as if they are writing for a single computer, while these abstractions manage the scheduling of the distributed computation on the underlying cluster of machines. This paper is a survey of such programming abstractions and security models for cloud computing systems. It discusses the MapReduce and Dryad models in detail, along with their applications. The paper also discusses some models which ensure the security of outsourced data and distributed computations, such as cryptographic file systems, Airavat for MapReduce etc.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

D…….Daemon

DBMS ……Database Management System

DPDP……..Dynamic Provable Data Position

EPG…….Execution Plan Graph

HDFS……..Hadoop Distributed File System

IP-ECC ……Integrity-Protected - Error Correcting Code

JM…….Job Manager

LINQ……Language Integrated Query

NS……Name Server

PDP…….Provable Data Possession

Acknowledgement

I would like to thank my supervisor, Dr. Charles Krasic for his continuous support and guidance throughout the completion of this essay. He provided invaluable advice at each stage of the essay, which helped me progress in the right direction. He directed me to a wide range of resources and provided invaluable feedback and inputs, which helped me stay on track. Without his constant help and direction, the completion of this essay would not have been possible. I would also like to thank Dr. Norman C. Hutchinson for agreeing to be the second reader for my essay.

1. Introduction

The term "cloud computing" is defined as "Internet-based computing, whereby shared resources, software and information are provided to computers and other devices on-demand, like a public utility" [12]. This delivery of infrastructure and resources over the network enables a client to purchase these resources only when needed while saving the cost of purchasing software and hardware [1]. It helps them accommodate peaks in demand, without having to pay for unused capacity when demand is low [1]. The scalability offered by cloud storage is another advantage, especially for web services which store a large amount of data and computation, which will increase even more in the future. The primary requirements of such web applications are scalability, good response time and high availability [5].

Cloud computing systems allow data intensive applications to be run with ease and with high scalability and throughput [24]. But in such applications, the developer faces the challenging task of scheduling the large scale distributed computation on a set of computers. Programming abstractions such as MapReduce and Dryad allow the developer to write programs for these large scale computations, while these systems manage the execution of the distributed computation. Chapter 2 discusses such programming abstractions by discussing in detail two popular models – MapReduce [15] and Dryad [19].

Since the daily operations of a large number of corporations and web services today depend on the use of data which is outsourced to an external server, i.e., cloud storage, it is extremely critical that this data be secured. Many of these servers are untrusted and hence the confidentiality, integrity and privacy of the client's (data owner) data must be protected [28]. Besides secure cloud storage, another important aspect of cloud computing security is the integrity of distributed information processing tasks and distributed web applications. Chapter 3 of this survey discusses models which provide security guarantees for outsourced data and computations.

2. Programming Abstractions for Cloud Computing

An important application of cloud computing systems is the outsourcing of information processing tasks to the cloud. MapReduce [15] and Dryad [19] are highly popular programming abstractions for such large scale distributed data processing. This chapter discusses the implementation, performance evaluation and variants of these systems which are in use today.

2.1 MapReduce

MapReduce is a programming model for performing large scale distributed computations on large data sets [15]. It is used across a wide range of applications in Google, for instance solving large scale machine learning problems, extracting properties of web pages, extracting data to produce reports of popular queries, etc. The sections below describe the working, features and performance evaluation of MapReduce [15].

2.1.1 Implementation

MapReduce [15] was implemented on machines present on a large cluster of thousands of computers. Figure 2.1 [15] below shows the sequence of operations occurring in a MapReduce operation (the steps described correspond to numbers in the figure):

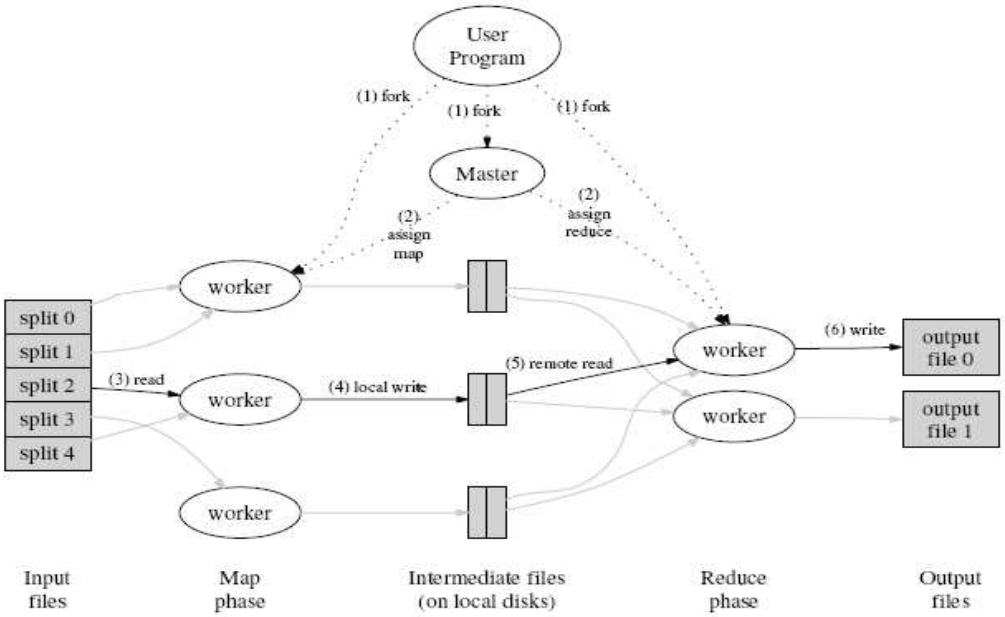

Figure 2.1 [15]: Execution Overview for MapReduce

1. The MapReduce library partitions the input data into M pieces. Each piece has a size of 16-64 MB, which can be specified by the user. Many copies of the MapReduce program are started at some machines in the cluster.
2. One machine in the cluster has the Master program running, which allocates map or reduce tasks to idle workers (machines with non-master copies of the program).
3. Each worker assigned a map task (called Mapper) parses the key-value pairs out of the input data and passes each pair to the 'Map' function defined by the user. The intermediate key value pairs produced by this Map function are stored in the memory of the machine.
4. These pairs are written to local disk and split into R regions using a partitioning function such as 'hash (key) mod R'**.** The locations of these pairs on the local disk is communicated to the Master, which then forwards them to the reduce workers (called Reducers).
5. The Reduce worker then reads data from the Mappers through remote procedure calls, sorts it by the intermediate key and groups together all occurrences of a key.
6. The Reducer passes each key and corresponding set of values to the user's Reduce function. The output of this function is appended to the final output file for that partition. The output of the operation is stored in R output files, one for each reduce task where file names are specified by the user.
7. Execution returns to the user program after all map and reduce tasks are complete.

2.1.2 Features

Some features of this MapReduce implementation are described below [15]:

a) **Data Structures:** The Master stores the identity and the state (idle, progress or completed) of the worker machines [15].
b) **Fault Tolerance:** The Master pings the worker periodically and if no reply is received the node is marked `failed`. The map or reduce tasks in progress at that node are rescheduled at another worker node. When a map task is re-executed on a different node, workers executing reduce tasks are notified of this, so that they

can read directly from the new worker with the map task. Completed reduced tasks are not rescheduled because their output is stored in files. However, completed map tasks are rescheduled because after node failure, the local disk of the machine containing the output of these tasks becomes inaccessible. If the master task fails, either a new copy can be started from the last check-pointed state or the computation is aborted [15].

Each ongoing task writes its output to temporary files, e.g., a reduce task produces one such file, and a map task produces R such files, one for each reduce task [15]. When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure. When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file [15].

c) **Locality:** The master tries to schedule map tasks on or near (e.g., on the same network switch) a machine which contains a replica of the input data. In this way, data is read locally and no network bandwidth is consumed**.**

d) **Backup Tasks:** The MapReduce operation is sometimes slowed down by the presence of machines which run especially slow (e.g., due to a bad disk). Such machines are called stragglers. To solve this problem, when the MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is declared complete whenever the primary or any of the backup executions complete.

2.1.3 Refinements

This section describes some extensions which were made to the original MapReduce implementation [15]:

(i) **Ordering:** Within a given partition, the intermediate key-value pairs are processed in increasing order of the key. This results in a sorted output file for each partition which in turn simplifies random access look-ups by key.

(ii) **Combiner Functions:** Combiner functions are used to do partial merging of the output data of a map task, before transferring it over the network. They can be used if there is repetition of the intermediate keys and the reduce function is commutative and associative. The output of this function is written to an intermediate file which is transferred over the network to the reduce worker node.

(iii) **Input-Output Types:** Data can be read in several different formats, e.g., a sequence of key value pairs sorted by key, text mode input where the key is the offset of each line in the file and the value is the content of that line, etc.

(iv) **Side-Effects:** Some MapReduce operations produce additional outputs which are written to auxiliary files.

(v) **Skipping Bad Records:** There is an optional mode of execution where the MapReduce library detects which records could cause crashes and skips these records to make progress so that the MapReduce operation would complete. This is accomplished with the help of a signal handler which is installed in each worker. Whenever a segmentation violation or bus error occurs, it signals the Master. When the master sees more than one failure in a particular record, it skips that record in the next re-execution of that task.

(vi) **Local Execution** of the MapReduce operation can be done on the local machine in a sequential manner, to facilitate debugging and small-scale testing.

(vii) **Status Information:** The master exports status pages containing information such as progress of computation, how many tasks are in-progress, how many have been completed, number of bytes of input, intermediate and output data of each task, processing rates, which workers have failed and which map or reduce task they were executing, etc. This information can be used to predict computation time and resources and to detect bugs in the user code.

(viii) **Counters:** The MapReduce library provides the user with counters – which can be used to count occurrences of various events, e.g., the number of words, the number of German documents, etc. The counter values from individual workers are periodically sent to the master, which aggregates them and returns the final value to the user code at the end of the operation. The MapReduce library also maintains some counters such as the number of input key-value pairs processed and the number of output key-value pairs produced. Such counters are used by the user to check the behaviour of the operation, for instance to check if the number of input pairs processed is equal to the number of output pairs produced by the operation.

2.1.4 Performance Evaluation

The performance of MapReduce was evaluated on sorting and searching tasks on a large cluster of about 1800 machines [15]. Each machine had two 2Ghz Intel Xeon Processors, 4 GB Memory, two 160 GB IDE disks, and a gigabit Ethernet link. Machines were arranged in a two level tree shaped network. The round trip time between any pair of machines was less than one millisecond. The search task was a 'grep' program which scanned 10^{10} 100-byte records for a three character pattern, occurring once in 92337 records. The number of input files was 15000 (64 MB each) and there was one output file. The performance of this task is shown in figure 2.2 [15].

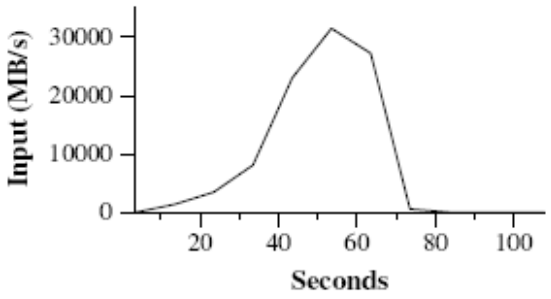

Figure 2.2 [15]: Data Transfer rate for searching task

The rate at which input data is scanned increases as more machines are assigned to the task [15]. The entire operation takes about 150 seconds. There is a start up overhead of one minute due to delays in propagation of the program to all workers and interaction with the underlying file system (GFS) to open the set of input files. The sort program sorts 10^{10} 100-byte records. There were 15000 64 MB input files and 4000 output files. Figure 2.3 [15] shows the execution of the sort program.

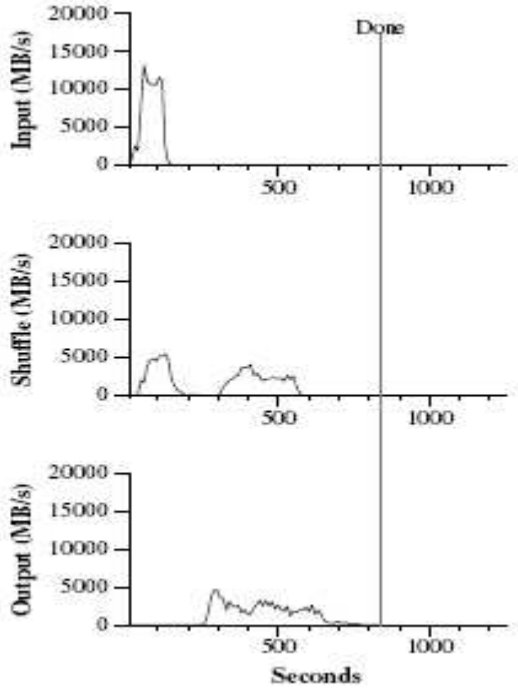

Figure 2.3 [15]: Data Transfer Rate for the sorting task

The input rate is less than that for the grep task because the sort map tasks spend half their time and bandwidth in writing their intermediate output to local disks. This intermediate output for the grep task is of negligible size [15].

Hence MapReduce is highly useful for performing large scale distributed computations. Mechanisms such as skipping of bad records, preserving network bandwidth, locality, etc., help improve the performance of MapReduce. The scalability and fault tolerance of MapReduce has made it extremely popular for a wide variety of distributed processing tasks [15].

2.2 MapReduce and Parallel Database Management Systems

Parallel Database Management systems have been in use for almost two decades now and are useful partitioned execution of SQL queries on segments of large relational tables [20]. MapReduce [15], as discussed above is used for performing computation on partitioned data. The sections below compare and contrast parallel database management systems and MapReduce based on arguments made by Ghemawat and Dean, [14] and Stonebraker et al., [20]. Stonebraker et al., [21] compared the performance of the Hadoop open source implementation of MapReduce with Parallel DBMSs on a benchmark consisting of a set of tasks such as the grep task, analytical tasks related to HTML document processing, etc. [21]. Results and observations from this paper are discussed in an article [20], which tries to bring out that MapReduce and parallel DBMSs complement each other, and don't compete with each other.

2.2.1 Parallel DBMSs and MapReduce

MapReduce is best suited for performing ETL, i.e., Extract-transform-load tasks [20]. Typical MapReduce tasks such as filtering, transformations, etc., of data items can be performed in parallel DBMSs using SQL or user defined functions, as suggested by Stonebraker et. al, [20]. The reshuffle occurring between map and reduce tasks is similar to the GROUPBY operation in SQL. Also, parallel DBMSs are highly scalable, as the number of nodes increase the database size increases proportionally, without degrading the response time. Production databases operate on multi-petabyte data sizes, with cluster

sizes of order 100 [20]. Hence scalability comparable to MapReduce can be obtained by Parallel DBMSs as well [20].

However, in spite of the fact that parallel DBMSs can perform almost all tasks typical of MapReduce, there are certain applications for which MapReduce does better [20], such as:

(i) **ETL and Read Once Data sets:** ETL systems are usually placed upstream from a database and the load phase of these systems feeds directly into the DBMS. Parallel DBMSs don't perform ETL tasks and use external products for performing these operations. On the other hand, MapReduce performs transformations on raw data and makes it ready for being loaded into another system. E.g., it reads logs of information from different sources, parses and cleans the log data, performs transformations, decides which attributes are to be stored and then loads this information in to a storage system. Hence MapReduce system can be considered as a general purpose ETL system [20].

(ii) **Complex Analytics:** Some data mining operations require multiple passes over the data. Such operations cannot be expressed as single SQL aggregate queries [20]. MapReduce is useful for such tasks since they require the output of one part of the application to be the input for the next part [20].

(iii) **Semi-Structured Data:** MapReduce does not require users to declare a schema for their data and can work with semi-structured data as well. Hence MapReduce would work well with loading tasks for semi-structured data, while parallel DBMSs will work better if a large number of analytical queries are to be performed on this data [20].

(iv) **Easy installation**: The installation and configuration of parallel DBMSs is difficult and involves tuning a large number of parameters. MapReduce on the other hand is easier and quicker to install. DBMSs also require the user to specify a schema for their data, unlike MapReduce which only requires users to copy their data in to the underlying distributed file system. Users who just want to perform a limited amount of data analysis on the input would prefer

because of the better start up time, where as professional DBMS programmers may choose longer start up times for the faster query performance provided by the latter [20].

(v) **Budget***:* Most MapReduce systems are open source projects and are available for free, while parallel DBMSs are expensive. Again, enterprise users with huge demand and budgets would be ready to pay for parallel DBMSs while users with a smaller budget and requirements would prefer open source systems.

2.2.2 Performance Analysis

Stonebraker et al., [21] defined a benchmark consisting of a collection of tasks and compared the performance of the Hadoop open source implementation of MapReduce, with Vertica (a column based parallel DBMS) and DBMS-X (a row based parallel DBMS) on this benchmark, on a 100 node cluster. The first task was the 'grep' task from the original MapReduce paper [15], where the system scans a set of 100 byte records looking for a three character string. A 1TeraByte data set was spread over the 100 nodes. It was expected that MapReduce would perform better at this task, because this task did not require the use of indices and did a full scan of the data [20]. However the two databases were found to be about two times faster than Hadoop MapReduce, Vertica being the faster of the two as can be seen in the figures 2.4 [21] and 2.5 [21].

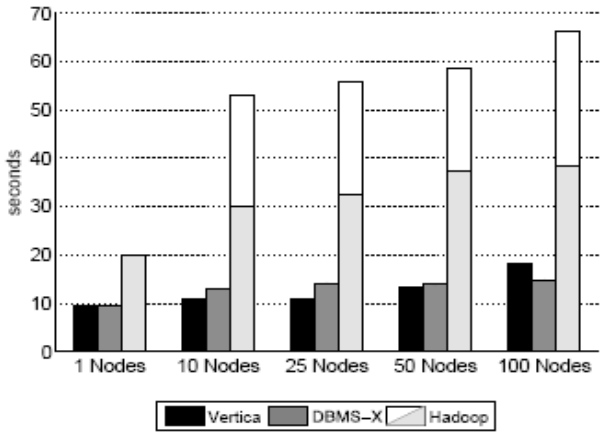

Figure 2.4 [21]: Grep Task Results (535 MB/node Data Set)

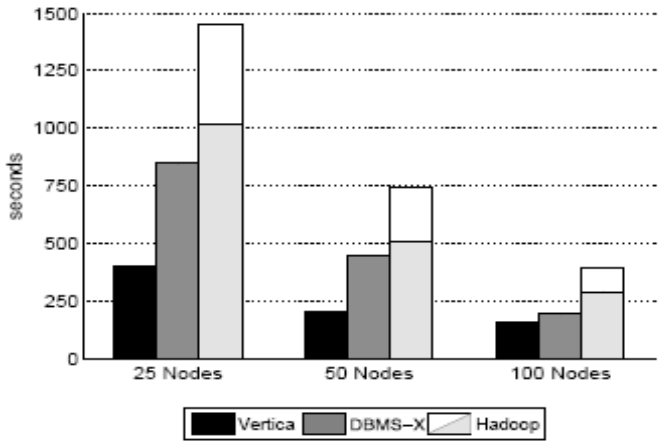

Figure 2.5 [21]: Grep Task Results (1TB /cluster data set)

The second task was a "Web log task" [20], performing SQL aggregation, using a GROUPBY clause on a table of user visits in a Web Server log. A 2TB data set consisting of 155 million records spread over 100 nodes (20 GB/node) was used. Each system had to calculate the total ad revenue generated by each visited IP address from the logs [20]. The performance of parallel DBMSs was better than Hadoop, by a greater margin than in the grep task, as shown in figure 2.6 [21].

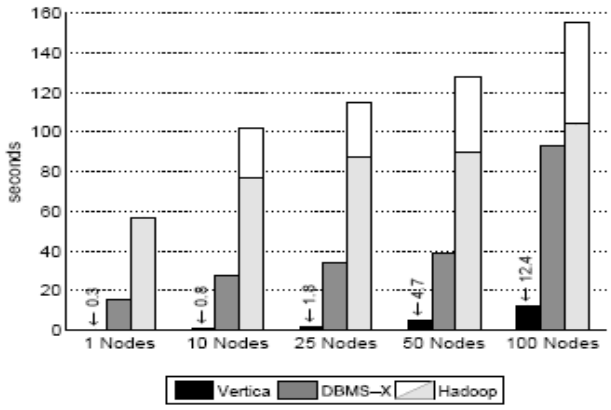

Figure 2.6 [21]: Selection task Results

The third task was a complex join over two tables requiring an additional aggregation and filtering operation [20]. The user visit data set from the previous task was joined with a 100GB table of PageRank values for 18 million URLs (1 GB/node). In this task, the system must first find the IP address that generated the most revenue within a specific date range in the user visits. It should then calculate the average PageRank of all pages

visited during this interval. The parallel DBMSs were a factor of 36 (Vertica) and 21 (DBMS-X) times faster than Hadoop in this task [20].

The authors [20] explained the poorer performance of MapReduce based on the architectural differences of the two systems (Hadoop MapReduce and parallel DBMSs) as follows:

(i) **Repetitive Record Parsing:** Hadoop MapReduce stores data in the Hadoop Distributed File System (HDFS) in the same textual format in which it was generated. This requires each Map and Reduce task to repeatedly parse and convert string fields into their appropriate type. One solution to this is the Sequence files storing data as key/value pairs. However, if the "value" portion of the record contains multiple attributes, it still needs to be parsed by the user code. The slower performance of Hadoop was attributed to this parsing overhead, and the use of sequence files without compression [20]. In contrast to MapReduce, parallel DBMSs parse records when the data is initially loaded and hence no record interpretation needs to be done when the queries are executed.

(ii) **Compression:** Data compression improved performance in the parallel DBMSs DBMS-X and Vertica, by a factor of two to four. On the other hand, compression improved performance only by about 15 % for Hadoop, and in some cases even slowed it down. One reason for this could be the use of finely tuned compression algorithms in commercial DBMSs [20], which ensure that the cost of decompressing records does not offset the performance gains from the reduced I/O cost of reading compressed data.

(iii) **Pipelining:** The lack of pipelining could be another reason for the slower performance of Hadoop [20]. In DBMSs, query execution follows a query plan which is delivered to nodes at the time of execution. When one operator in this plan needs to send data to another operation, the data is pushed to the second operator, without being written to disk [20]. However, in MapReduce the intermediate data is written to large data structures (and ultimately to

disk), introducing a bottleneck. Inspite of the advantage of fault tolerance provided by this technique, it introduces performance overhead [20].

(iv) **Scheduling:** In parallel DBMSs, workload is allocated to nodes in advance according to a distributed query plan. This helps the system optimize the execution plan to minimize data transmission between nodes. On the other hand, each task in MapReduce is scheduled on worker nodes at runtime, one storage block at a time. Although this runtime scheduling helps the MapReduce scheduler adapt to workload skew and performance differences between nodes, it is more expensive than the compile time scheduling in DBMSs [20].

(v) **Column oriented Storage**: Column store based databases (e.g., Vertica), only read the attributes which are necessary for answering the user query. This is more efficient than reading all attributes off the disk, which is done by row – stored databases such as DBMS-X and Hadoop [20].

The paper [20] concludes by saying that given the varied advantages of the two, it is best to use a system that is a hybrid of the two. MapReduce implements fine granularity fault tolerance and is good at ETL tasks and performing complex analytics. It is however not well suited for query intensive workloads and does not take advantage of high level languages [20]. Parallel DBMSs, on the other hand give high performance on query intensive workloads, but implement transaction level fault tolerance, which might be insufficient as databases get larger and the number of nodes increases. Also, they should move towards easier and quicker installation procedures [20]. This motivates the need for interfacing MapReduce with Parallel DBMSs, so that MapReduce can perform ETL functions while queries can be executed by the database. Hive [3] is an example of such a hybrid system.

2.2.3 In Defence of MapReduce

The above section was a summary of arguments made by Stonebraker et. al., [20], [21] to compare MapReduce with parallel databases [20]. Ghemawat and Dean responded with

the article “MapReduce: A Flexible Data Processing Tool” [14], to clarify some misconceptions about MapReduce introduced in the article [20]:

(a) MapReduce inputs and outputs don’t always need to be simple files in a file system [14]. Production environments today use a mixture of storage systems, e.g., customer data is stored in a relational database, while user requests are logged in a file system [14], and as these systems evolve, data may move into new storage systems. MapReduce provides a framework for analyzing data in such heterogeneous systems. It can be extended to support a new type of storage system by defining simple reader and writer implementations for that storage system. Hence a MapReduce operation can process and combine data from different types of storage systems. Some supported storage systems are BigTable [8], B-Trees, etc. If a system using only parallel DBMS is used, a loading phase is required to copy all input data into the DBMS. Loading of a large amount of data to perform a single scan or analysis only after complete loading will reduce the efficiency.

(b) Contradictory to what was suggested in the comparison paper [21], MapReduce makes use of indices [14]. For instance a large dataset can be partitioned into a set of databases using a hash function and an index can be added to each database. The result of running a query using this index, on each of these databases can act as inputs to MapReduce. Another example is when MapReduce reads data from BigTable [8] - if the data maps to a specific range of data in BigTable, only this range needs to be read. Similarly, if only specific attributes need to be read, then MapReduce can read from only specified columns of BigTable. A last example is the Join task used as a benchmark for testing in the previous article [20, 21], where only log records in a specific date range need to be processed. Every logging system rolls over to a new log file periodically and embeds the rollover time in the name of the log file. The MapReduce operation can be run over the log files that belong to the specified date range, instead of reading all log files [14].

(c) Not all Map and Reduce operations are simple enough to be expressed as SQL queries [14], e.g., extracting outgoing links from a set of HTML pages and aggregating by target document, joining overlapping satellite images to remove seams and to select high imagery for Google Earth, generating a collection of inverted index files using a compression scheme tuned for efficient support of Google search queries, fault tolerant parallel execution of programs written in higher level languages [14], etc. MapReduce is a better choice for such operations than SQL's selection and aggregation or user defined functions, which may not be present in all DBMSs [14].

(d) The benchmarks in paper [21] used an inefficient textual format, where attributes were separated by vertical bars, whereas MapReduce operations at Google use the Protocol buffer format [11] to read and write data. A high level language is used to describe the input and output types and compiler generated code hides the encoding-decoding details from application code. The protocol buffer format allows types to be upgraded without having to modify the application [14]. They use an optimized binary representation which is compact and faster to encode and decode than the benchmarks used in the paper [21].

(e) The pull model (where data moves from mappers to reducers through the file system) is preferred over the push model (mappers directly sending their output to reducers) in MapReduce, because it implements fault tolerance [14]. In the absence of this, if a reducer fails, all map tasks would have to be re-executed. As the size of data grows in the future, having a mechanism for fault tolerance like the one in MapReduce will become more important.

(f) The assumption in the comparison that MapReduce requires a full scan of the input file to start an operation, effects the results of the selection, aggregation and join tasks and hence shows poorer performance of MapReduce compared to parallel databases [14]. MapReduce does not require a full scan of the input file; it requires only an implementation of its input interface to produce a set of records which match some input specification [14], e.g., all records in a set of files, all records in a specified date range, all records with a specified value of a particular attribute from BigTable, etc. It may need a full scan over a set of files, but

alternate implementations can be used, – e.g., input can be a database with an index which provides efficient filtering or indexed file structure [14].

(g) The measurements for Hadoop in the comparison benchmark take into account the cost of a final phase which merges the results of MapReduce into a file. In practice, this merging step is not necessary since the MapReduce output is usually fed into another MapReduce task which can operate directly on the input of the previous task, or the output can be fed into BigTable or a parallel database table [14].

Therefore some advantages of MapReduce over parallel databases are fine-grain fault tolerance, handling data loading and processing with heterogeneous data storage systems and ability to handle complex functions not supported by SQL queries [14].

MapReduce is a flexible data processing tool. It parallelizes and executes the program on a large cluster of machines. The MapReduce library manages scheduling of program execution, partitioning the input data; handling machine failures and managing inter-machine failure. This makes it easy to use even for programmers with little or no experience with distributed systems [14]. However, it could do better with use of natural indices such as timestamps in logs, making the data shuffling phase between Map and Reduce stages more efficient and decreasing start up time [14].

2.3 Pipelined MapReduce

The paper [26] proposes some of modifications to the open source Hadoop MapReduce implementation, to support pipelining, while preserving the fault tolerance mechanisms of the original implementation. Pipelining improved utilization and parallelism, reduced response time and supported online aggregation and continuous queries. This modified version of was called the Hadoop Online Prototype (HOP) [26]. Some of these modifications are discussed below.

2.3.1 Pipelining within Jobs

The master sends the location of each map task to each reduce task [26]. The reduce task communicates with each map task at the beginning of the job and opens a socket, which is used to pipeline the output of the map function. Once this output is generated, the map worker sends it to the appropriate reduce task. If a reduce task hasn't been scheduled, this output is written to disk and is fetched by the reduce task after it is scheduled. The reduce task stores the received data in a memory buffer, and stores sorted runs of this data to disk. When all map tasks complete, the reduce task merges these runs and applies the reduce function to them. The map task and writing of data to pipeline sockets are carried out in separate threads so that a busy reducer may not block the functioning of the mapper [26].

The process defined above involves eager pipelining of records to the reducer. This prevents the use of combiners at the map workers and also moves the sorting to the reducer. To overcome this, the output is first stored in an in-memory buffer until it reaches a certain threshold, after which it is sorted (by partition and reduce key) and written to disk in spill file format [26]. A separate thread sends these spill files to the reducer as soon as they are produced. If the reducer cannot keep up with this rate, the mapper periodically applies the combiner function to the unsent spill files, merging them into one large file. Hence the load can be adaptively moved from the mapper to the reducer (or vice versa). The difference between the traditional and pipelined versions of MapReduce can be seen on figure 2.7 [26] below.

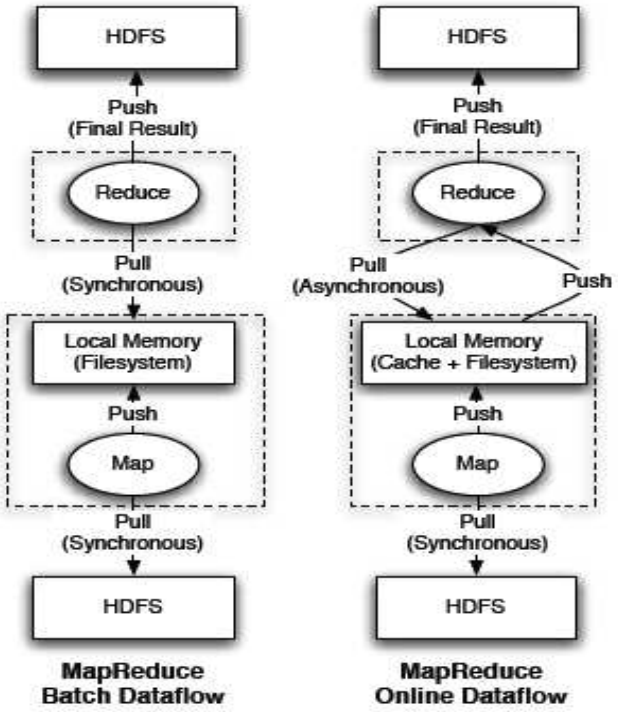

Figure 2.7 [26]: Dataflow for Batch (left) and pipelined (right) MapReduce operations

Pipelining between jobs can also be done, when the output of reduce tasks acts as input for map tasks. This is useful in practical large scale computations which cannot be expressed as a single MapReduce job. The MapReduce interface in this case accepts a list of jobs, such that each job in the list depends on the job before it. Each job is labelled with the identifier of the job it depends on. The master refers to these labels and co-schedules jobs with their dependencies [26].

2.3.2 Fault Tolerance

To simplify recovery from map task failures, the reducer treats the output of the map task as tentative and ignores this data only if the task fails [26]. It combines the spill files of a map task with outputs of other tasks only when the master confirms that the map task has completed successfully. An alternative for this is that the map task should periodically notify the master that an offset x in its input split has been reached. The master communicates this to all reducers, which can then proceed with merging map task output until offset x with outputs of other map tasks. When a reduce task fails, it needs to be sent

all previous data again. This is made possible by having the map task retain all its output data by writing the output file to disk.

2.3.3 Performance Evaluation

The performance evaluation was done on a 60-node cluster on Amazon EC2, each running "high-CPU medium" with 2 virtual cores and 1.7 GB memory [26]. Each virtual core was equivalent to a 2.5 GHz Intel Xeon processor. The task was to sort 5.5 GB of article text extracted from Wikipedia, where each word was treated as a record. 59 Reducers were used. For the blocking execution, the reducers are idle for the first 192 seconds where as this idle time is 20 seconds for pipelined execution, indicating that pipelining can help improve utilization and response times. The graphs below, figure 2.8 [26], show this performance for blocking and pipelined executions.

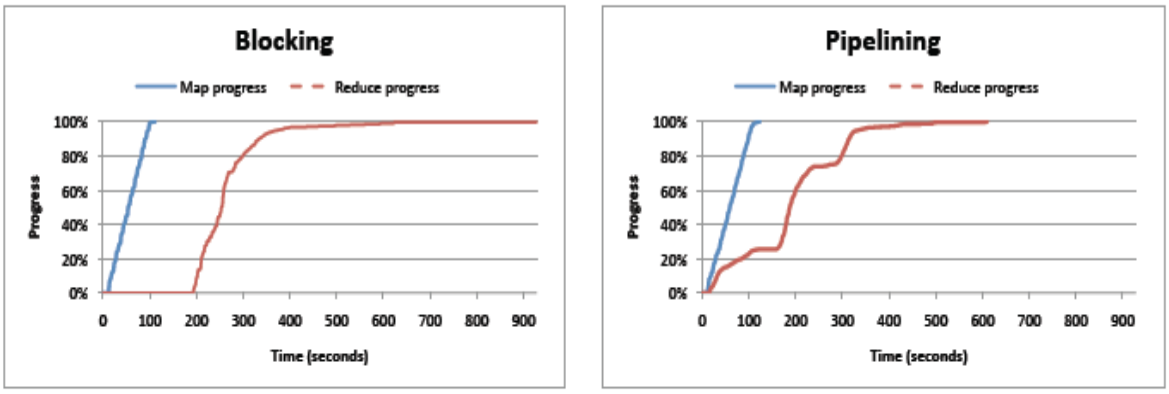

Figure 2.8 [26]: Task completion times for Blocking and Pipelined executions

2.3.4 Online Aggregation

This modified version of MapReduce can be used for interactive data analysis [26], i.e., a user submits a job and extracts information from the data set, and then waits for the results before proceeding with the data analysis. The user prefers using an approximation for this analysis rather than waiting for a long time for the final result, as would be the case with the original MapReduce. Online aggregation is used to solve this problem in the modified MapReduce [26].

(a) **Single Job Online Aggregation** can be achieved by producing "snapshots". A snapshot is the output obtained by applying the reduce function to the data received by the reduce task so far. Snapshots are computed periodically as new data arrives at each reducer. The user specifies how often these snapshots have to be computed, e.g., when 25%, 50%, 75%, etc., of the input has been seen. The snapshots are stored in a directory in the underlying file system, the Hadoop Distributed File System (HDFS) [26]. The name of this directory contains the progress value associated with the snapshot. Progress value is a score in the range [0, 1], which is assigned to a map task based on how much of its input it has consumed. It helps determine how much progress is represented by the current input to a reduce task, and when a new snapshot should be taken. To achieve this, the spill file includes the map task's current progress score. When a partition in this spill file is sent to the reducer, it includes the spill file's progress score. The progress of a snapshot is computed by taking the average of the progress scores associated with each spill file used to generate the snapshot [26].

(b) **Multi-Job Online Aggregation** is useful in cases where one MapReduce job consumes the output of the previous MapReduce job. In this case the snapshot computed by the first job's reducers is also sent to the map tasks of the second job [26]. Let the two jobs be j_1 and j_2 such that j_2 consumes the output of j_1. If a task in j_2 fails, the failed task is restarted. The next snapshot received by the restarted task in j_2 will have a higher progress value. Tasks in j_2 cache are the most recent snapshot received by j_1. They replace the snapshot when they receive a new snapshot with a higher progress, which occurs if j_1 fails. If tasks from both the jobs fail, a new task in j_2 recovers the most recent snapshot from j_1 while waiting for snapshots with a higher progress. In online aggregation, snapshots can be pipelined to the jobs which request them. This can be done through an asynchronous request call export interface supported by each reduce task. Through this map, tasks from other jobs can request snapshots and the final output. The final output is concurrently written to the underlying file system for fault tolerance [26].

2.3.5 Continuous Queries

MapReduce is used for analysis of streams of continuously arriving data. E.g., access logs, system console logs, etc. The original MapReduce implementation can only do this periodically and introduces delay in the data analysis. In the modified version of Hadoop, discussed in [26], near-real-time analysis of data streams is possible, because new data is analyzed as it becomes available. Hence this can be used in monitoring of the environment, real-time fraud detection, etc.

The pipelined version can give better response times, support online aggregation and real-time data analysis, while preserving the simple programming model and fault tolerance of the original MapReduce implementation [26].

2.3.6 Summary

MapReduce is a highly useful distributed data processing tool. It is easy to use since it hides intricacies related to parallelization, scheduling, fault tolerance, etc., from the developer [15]. Moreover a large number of computations can be expressed in the form of MapReduce operations, such as graph processing, text processing, machine learning etc [15]. As mentioned earlier, it can be used to complement existing storage systems, e.g., it can be used as an ETL system for parallel databases [20] or in combination with other data storage systems such as BigTable [8]. Its flexibility has also led to its use outside of Google, in open source implementations such as Hadoop [14] and the pipelined Hadoop Online Prototype [26], which implements pipelining in the original MapReduce implementation. Hence, the simplicity and flexibility of MapReduce have made it a popular and widely used cloud computing paradigm.

2.4 Dryad

Dryad [19] is a general purpose high performance distributed execution engine. It manages creation of large distributed applications, scheduling resources, optimizing the level of concurrency within a computer, recovering from communication or computer failures, and data delivery. It supports data transport mechanisms between computation

vertices and explicit dataflow graph construction. This section describes the construction and working of Dryad and its performance evaluation [19].

2.4.1 Dryad Architecture and Operation

Dryad enables developers to write efficient parallel and distributed applications with ease [19]. A Dryad job is a directed acyclic graph in which each vertex is a program and edges represent data channels. The graph represents logical computation and is mapped into physical resources at runtime. At runtime each channel is used to transport a finite sequence of structured items. The channel could be TCP pipes or shared memory or temporary files. The vertex program reads and writes data in the same way irrespective of the channel implementation. The vertices provided by the developer are written as sequential programs. Concurrency is introduced by Dryad. It schedules vertices to run simultaneously in multiple computers. The application determines the size and placement of data at runtime and modifies the graph as the computation progresses if needed [19].

Architecture: The Dryad job is coordinated by a process called the job manager (JM) [19]. The JM runs within each cluster and contains application specific code to schedule work across available resources. Data is sent directly between vertices. Hence the job manager only makes control decisions and is free from bottlenecks occurring due to data transfer. Each cluster also has a Name Server (NS) which enumerates all computers in the cluster. It also helps make scheduling decisions by exposing the positions of all computers in the network topology. A daemon (D) also runs on each computer in the cluster which creates processes on behalf of the job manager. The first time a vertex is executed on a machine, its binary is sent from the job manager to the daemon, after which it is executed from the cache. This daemon helps the job manager communicate with remote vertices. It monitors the state of the computation and how much data has been read and written on the channels [19]. Figure 2.9 [19] below shows this architecture.

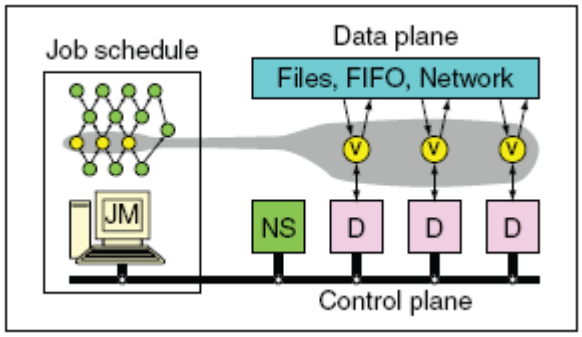

Figure 2.9 [19]: The Dryad System Organization

The job manager (JM) consults the name server (NS) to obtain a lost of available machines. It manages the job graph and schedules running vertices as computers become available using the daemon (D) as proxy. The shaded bar in the figure represents the currently running vertices in the job [19].

Graphs in Dryad are constructed by combining simple sub-graphs using a small set of operations such as 'cloning', addition of new graph edges, merging of two graphs, etc., all of which are described in detail in [19]. All these operations preserve the property that the resulting graph should be acyclic. A graph is denoted as :

$$G= (V_G ,E_G, I_G ,O_G)$$

where V_G is a sequence of vertices, E_G is a set of directed edges, I_G and O_G represent input and output vertices respectively [19]. Dryad libraries define a C++ base class from which all vertex programs inherit. A graph vertex is created by calling the appropriate program. Vertex-specific parameters are set by calling methods for the object. New edges are created by applying a composition operation to two existing graphs. The composition of two graphs A and B contains the union of all the vertices and edges in A and B, with A's inputs and B's outputs. New edges are introduced between vertices in O_A and I_B. A point wise composition is formed when A>=B, i.e., a single outgoing edge is formed from each of A's outputs or a single incoming edge to each of B's inputs (depending on which of O_A and I_B is larger). If A >> B, a complete bipartite graph is formed between O_A and I_B. Merging of two graphs creates a new graph which has the concatenation of the vertices of A and B with duplicates removed. Table 2.1 [19] below indicates the channel types the user can specify for the graph.

Channel protocol	Discussion
File (the default)	Preserved after vertex execution until the job completes.
TCP pipe	Requires no disk accesses, but both end-point vertices must be scheduled to run at the same time.
Shared-memory FIFO	Extremely low communication cost, but end-point vertices must run within the same process.

Table 2.1 [19]: Channel Types for Dryad

Large input files are partitioned and distributed across the cluster. After the job finishes execution, output partitions are logically concatenated to form a single named distributed file. When the graph is constructed every vertex is placed in a 'stage' which helps simplify job management. A topology of such stages is called the skeleton or summary of the job [19].

Vertex Program: Dryad has a runtime library which manages execution of vertices [19]. The runtime receives a notification from the job manager describing the vertex to be run, and URIs describing the input and output channels to connect to it. The vertex is invoked through by a 'Main' method, which has channel readers and writers as its argument list. The vertex reports status and errors to the job manager. The channel implementation schedules read, write, serialization and deserialization tasks on a thread pool which is shared between all channels in a process. This enables a vertex to concurrently read or write on hundreds of channels. The runtime tries to enforce pipelined execution and at the same time presents the developer with the simple abstraction of reading and writing a single record at a time [19].

Job Execution: If the job manager's computer fails the job is terminated which may lead to multiple executions of the vertex over the length of the job [19]. Also, more than one instance of a given vertex may be executing at any given time. Each execution of the vertex is associated with a version number and an execution record. This execution record contains the state of execution and the versions of the predecessor vertices from

which its inputs are derived. When a vertex's input channels are ready, a new execution record is created and placed in a scheduling queue. A vertex and any of its channels may specify a preference to run on a particular set of computers. Such constraints are attached to the execution record when it is added to the scheduling queue. A graph visualizer and a web-based interface showing summary statistics of a running job are used to monitor computations. The statistics show parameters such as the number of vertices that have completed or been re-executed, the amount of data transferred across channels, and the error codes reported by failed vertices. A developer can download logs or crash dumps for debugging [19].

Fault Tolerance: When a vertex execution fails the job manager is informed [19]. If the vertex reported an error, the process forwards it to the job manager via the daemon before exiting. However if the process crashes, the daemon notifies the job manager and if the daemon fails the job manager receives a timeout. If the failure was due to a read error on an input channel, the execution record that generated that version of the channel is marked as failed and terminates its process if it is running. This causes the vertex that created the failed input channel to be re-executed, and hence leads to the failed channel being recreated. A vertex whose execution record is set to 'failed' is immediately considered for re-execution [19].

2.4.2 Performance Evaluation

Dryad was evaluated by running two sets of experiments. The first experiment implements a complex SQL query as a Dryad application, and was run on a cluster with 10 computers. The query used was the most time consuming run on the Sloan Digital Sky Survey database [25], taken from a published study [16] on this database. The corresponding Dryad communication graph is shown in Figure 2.10 [19].

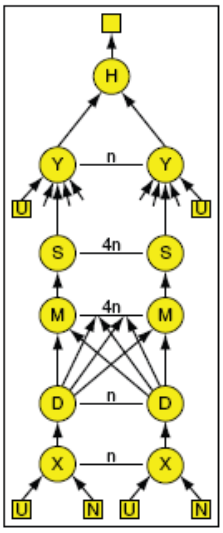

Figure 2.10 [19]: Communication Graph for the SQL Query

Dryad performance is compared with that of a commercial SQL server. The two-pass version of the Dryad job works on all cluster sizes, with almost linear speed-up. The in-memory variant works for n = 6 cluster size and above, again with almost linear speed-up, and approximately twice as fast as the two-pass variant. The Dryad program runs faster than SQLServer's general-purpose query engine. The SQLServer implementation cannot be distributed across multiple computers. Table 2.2 [19], shows the elapsed times in seconds for each experiment.

Computers	1	2	3	4	5	6	7	8	9
SQLServer	3780								
Two-pass	2370	1260	836	662	523	463	423	346	321
In-memory						217	203	183	168

Table 2.2 [19]: Time in seconds to process an SQL query using different number of computers (n)

The second experiment was a map-reduce style data-mining operation done on 10.2 Terabytes of data using a cluster of 1800 computers [19]. The purpose of this experiment was to verify that Dryad can work for such operations at large scales. The operation reads query logs gathered by the MSN Search service, extracts the query strings and builds a histogram of query frequency. Starting with a basic communication graph, many refinements were done to obtain a final graph which scaled well [19].

Hence Dryad is a general purpose data- parallel execution engine [19]. It scales well for small cluster sizes, as seen from the results. Dryad gives the developer fine control over the communication graph making it simple for programmers to express the distributed computation [19]. The developer can specify a directed acyclic graph to describe the applications communication patterns and express the data transport mechanism between vertices. The developer can pick a suitable mechanism for data transport so as to maximize performance gains [19]. Other programming interfaces can be built on top of Dryad, such as DryadLINQ [29] discussed in the next section.

2.5 DryadLINQ

DryadLINQ [29] is a system and a set of language extensions which make it easy for the developer to write imperative or declarative operations on large datasets using a high level language. It makes use of Language Integrated Query (LINQ) which is a set of .NET constructs, to provide a hybrid of declarative and imperative programming [29]. Since the objects in DryadLINQ are of .NET type, it is convenient to deal with data in the form of image patches, vectors and matrices. This section discusses the architecture, operation and performance evaluation of the DryadLINQ [29] system.

2.5.1 Architecture and Operation

A DryadLINQ program is a sequential program composed of LINQ expressions performing transformations on datasets [29]. The system gives the programmer the illusion of writing for a single computer, while the system deals with scheduling and distribution complexities. The DryadLINQ system then translates the data parallel portions of the program in to a distributed execution plan, which is then passed to Dryad for execution [29]. Figure 2.11 [29], below, shows the flow of execution of a DryadLINQ program.

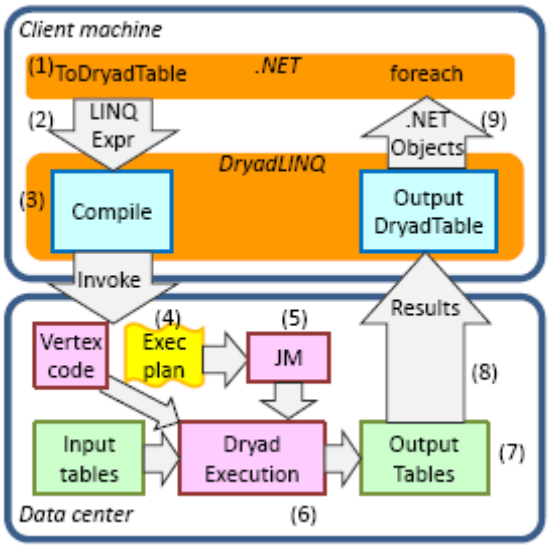

Figure 2.11 [29]: LINQ expression execution in DryadLINQ

Execution Steps are described below [29]:

1. A .NET user application creates a DryadLINQ expression object.
2. The application calls ToDryadTable and triggers a data-parallel execution. The expression object is passed on to DryadLINQ.
3. DryadLINQ compiles the LINQ instructions, resulting in a distributed Dryad execution plan. It performs the decomposition of the LINQ expression into sub-expressions for each Dryad vertex and generates code and static data for the vertices.
4. DryadLINQ invokes a DryadLINQ-specific Dryad job manager.
5. This job manager creates the job graph using the plan created in Step 3. It schedules and creates vertices as resources become available.
6. Each Dryad vertex executes the program generated for it in Step 3.
7. When the Dryad job completes, it writes the output data to output tables.
8. The job manager process terminates and returns control to DryadLINQ. DryadLINQ then creates local DryadTable objects to encapsulate the outputs of the execution from Step 7.
9. Control returns to the user application. The DryadTable contents can be read by using an iterator interface defined over these objects. This interface allows its contents to be read as.NET objects.

2.5.2 Programming with DryadLINQ

LINQ allows the programmer to express complex transformations over datasets. The exact implementation of these computations is decided at runtime [29]. The base type for a LINQ collection is IEnumerable<T>. This is an abstract data set of objects of type T accessed using the iterator interface. Another interface is the IQueryable<T> interface which is a subtype of IEnumerable<T>. It represents an unevaluated expression constructed by combining LINQ datasets using LINQ operators [29].

DryadLINQ extends the LINQ programming model to data-parallel programming by defining a set of new operators and data types. Each DryadLINQ dataset is distributed across the computers in a cluster, as shown in Figure 2.12 [29].

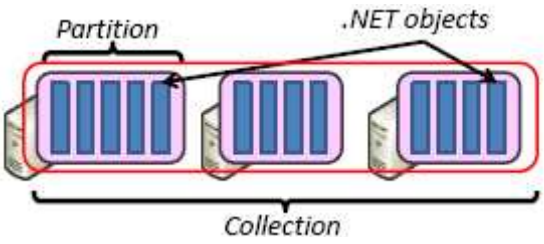

Figure 2.12 [29]: The DryadLINQ data model: strongly-typed collections of .NET objects partitioned on a set of computers

Data set partitioning can be done using hash-partitioning, range partitioning or round robin [29]. The inputs and outputs of a DryadLINQ computation are represented by objects of type DryadTable<T>, which is a subtype of IQueryable<T>. Subtypes of DryadTable<T> support underlying storage systems such as distributed file-systems, NTFS files, SQL tables, etc. DryadTable objects may also contain metadata read from the file system describing table properties such as schemas for the data items in the table. All functions called in DryadLINQ expressions are required to be 'side effect free', i.e., shared objects can be referenced and read freely and will be automatically serialized and distributed where necessary [29].

DryadLINQ contains two data re-partitioning operators - HashPartition<T, K> and RangePartition<T, K> to enforce a partitioning on the output dataset [29]. "Apply" is an operator which passes an iterator over the entire input dataset, as an argument to any

function f, hence allowing arbitrary streaming computations to be performed on the dataset. The "Fork" operator is similar to Apply, except that it takes a single input and generates multiple output datasets. It can be used as an optimization to eliminate common sub-computations. The system allows programmers to specify annotations to guide optimizations which the system otherwise might be unable to perform, e.g. Apply uses annotations in the form of simple .NET attributes to indicate possible parallelization [29]. DryadLINQ converts the raw LINQ expression into an execution plan graph (EPG), where each node is an operator and edges represent its inputs and outputs. This EPG forms the skeleton of the Dryad data-flow graph that will be executed. [29]

2.5.3 Performance Evaluation

The paper compares the performance of DryadLINQ with Dryad by evaluating their performance on the most time consuming query (Q18) from the Sloan Digital Sky Survey database [16]. This is the same query which was used to evaluate the Dryad [19] model. The query retrieves records for a 'gravitational lens' effect by comparing the locations and colors of stars in a large astronomical table. The query performs a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. The two-pass variant of the Dryad program described in [19] is compared with DryadLINQ. The Dryad program is about 1000 lines of C++ code whereas the corresponding DryadLINQ program is only around 100 lines of C#. The input tables were manually range-partitioned into 40 partitions using the same keys. Table 2.3 [29] shows the elapsed times in seconds for the native Dryad and DryadLINQ programs as the number of computers (n) was varied between 1 and 40.

Computers	1	5	10	20	40
Dryad	2167	451	242	135	92
DryadLINQ	2666	580	328	176	113

Table 2.3 [29]: Time in seconds to process the query Q18 using different number of computers

The DryadLINQ implementation is 1.3 times slower than the native Dryad model [29], attributed to a fast hand-tuned sort strategy used by Dryad, and the slow automatic

parallel sort strategy in DryadLINQ [29]. However, the DryadLINQ program is written at a higher level, abstracts the distributed nature of the computation from the programmer, and is only 10% of the length of the original code.

Another task was to sort 10 billion 100-byte records using the Terasort bench mark [27]. The DryadLINQ program defines the record type, creates a DryadTable for partitioned inputs and calls OrderBy. After this the system generates an execution plan using dynamic range partitioning [29]. The execution times on n machines are given in Table 2.4 [29] below.

Computers	1	2	10	20	40	80	240
Time	119	241	242	245	271	294	319

Table 2.4 [29]: Time in seconds to sort different amounts of data

This table shows execution times as the number of machines increases from 1 to 240 and the amount of data sorted increases from 3.87 GBytes to 10^{12} Bytes. After repeated runs of the experiment, these times had a stable value which was within 5% of the average time [29].

Hence the DryadLINQ system compiles transformations on large data sets expressed as LINQ programs into distributed computations running on the Dryad [19] cluster computing architecture [29]. The above section discusses the operation and performance evaluation of this system and also describes a set of operators designed to use LINQ support for parallel computations [29].

2.6 Summary

Each of the three architectures discussed above, Dryad [19], MapReduce [15] and DryadLINQ [29], have some key advantageous features. Graphs in Dryad can have an arbitrary number of inputs and outputs whereas, in MapReduce, the operation can have only a single input set and a single output set [19]. However, the architecture of Dryad is more complex than MapReduce [19]. With Dryad, developers can easily create large-

scale distributed applications by only drawing a graph with data dependencies. Similar to MapReduce, developers need not get into details about concurrency in Dryad [19]. In DryadLINQ [29], the computation can be expressed in a popular high level language, again without delving into details about concurrency. This system takes advantage of .NET objects to represent a wide variety of data, e.g., image patches, vectors and matrices. It also allows debugging using popular tools such as Visual Studio [29]. It is also useful for implementing complex transformations such as joins efficiently [29].

Hence this chapter discusses two popular cloud computing paradigms which support large scale distributed computations -MapReduce [15] and Dryad [19]. MapReduce allows users to express computations as Map and Reduce tasks on the input data set (in the form of key-value pairs). The distribution of these tasks to a set of 'mapper' and 'reducer' machines is managed by the MapReduce library, hence saving the programmer the overhead of dealing with parallelization details [15]. In Dryad, the computation is expressed in the form of a data flow graph with the vertices representing sequential programs running on machines and edges representing data transport channels [19]. This chapter summarizes the implementation and evaluation details of both these architectures.

3. Security in Cloud Computing Systems

As more and more users are moving their data and computations to the cloud, the security of cloud computing systems has become more important than ever before. When an organization outsources its data to an external server for storage, it runs the risk of having the data being leaked to an adversary or the modification/corruption of the file content by the adversary. Some techniques discussed below, namely HAIL [17] and a cryptographic distributed file system [4], address these issues by developing systems to verify the integrity of files stored at a remote server. Another system discussed in this section is the dynamic provable data possession scheme [6], which also supports the verification of the intactness of a file after updates have been made to it, unlike HAIL, which only supports verification for static files.

Airavat [13], a security paradigm designed for MapReduce, discusses issues related to privacy of user data in distributed computations**.** In cloud computing, each user contributes his or her individual data and obtains useful services from the cloud, e.g., a user's click-stream may be used to provide targeted advertisements [13]. Distributed processing on such sensitive data requires high security guarantees. For example [13], a medical patient who is deciding whether to participate in a large health-care study may be concerned that a careless application operating on her data may expose it. Another concern would be that even if all computations are done correctly and securely, the result itself, i.e., the aggregate health-care statistics computed in the study, may leak sensitive information about her personal medical record [13]. The major security concerns that can be seen here are [13]:

(a) Keeping data safe from being accidently exposed due to a faulty application operating on the data.

(b) Ensuring that published results of the study should not reveal information about the record of an individual who participated in the study.

Ripley [18] secures distributed web applications by replicated execution of client side code. Unlike the previous chapter of this survey, this chapter adopts a breadth approach

and discusses a number of different systems used to ensure security in cloud computing systems (instead of going in depth into any particular approach).

3.1 HAIL

HAIL (High Availability and Integrity Layer for Cloud Storage) [17] is a cryptographic system that allows a set of servers to verify to the client that the file stored on them is intact. HAIL is based on a test-and-redistribute strategy, where the client uses PORs (proofs of retrievability) [2] to detect file intactness and retrievability. If the challenge response protocol detects a fault in a file stored at the untrusted server, the client recovers the corrupted portion from cross-server redundancy built into the encoded file, and replaces the faulty server with the corrected version [17]. This section discusses the implementation of HAIL [17].

3.1.1 Encoding in HAIL

HAIL makes use of the following error-correcting codes [17]:

(i) Dispersal Code: It is used for distributing file blocks across servers. This paper introduced a new primitive called integrity-protected error correcting code (IP-ECC). It acts as an error correcting code and a message authentication code (MAC) at the same time, with an additional overhead of only one extra codeword symbol.

(ii) Server Code: This is an error correcting code used to encode file blocks within each server, and protects against low level corruption.

(iii) Aggregation Code: This code is used to compress responses from multiple servers.

In HAIL, the client distributes the file F with redundancy among n servers, while storing some metadata locally [17]. It ensures protection from a mobile adversary, i.e., one which can control b out of n servers within any given time step (called an epoch). In each epoch, the client performs an integrity check for F. If corruptions are detected on some servers, it reconstructs the file from redundant copies on other servers and replaces the faulty server [17].

The simplest way to do the integrity check is when the client chooses a random file block F_j of F from each server [17]. If all these blocks are identical, the client concludes that the file is intact. However, this has can lead to 'creeping corruption attacks' - when the adversary picks a random position i, and corrupts the file block F_i in all servers in a given epoch. After a certain number of epochs this file block is corrupted in all servers. So when the client compares the copies of file block retrieved from all servers, they pass this consistency check, although the block is corrupted. The adversary can make this test fail even by corrupting a little over half the copies of the file block F_i [17].

To avoid the creeping corruption attack, the file F is encoded under an error correcting code, called 'server code' [17]. This code renders each copy of the file F robust against a fraction ϵ_c of the corrupted file blocks, protecting against single-block corruptions, where ϵ_c is the error rate of the server code. Also, HAIL tries to reduce the storage overhead by using a dispersal code [17]. Instead of replicating the file F across all servers, it is distributed using the 'dispersal code', which is an error correcting code, containing two parameters n and l – n is the total number of servers, l is the number of 'primary servers', which will store the fragments of the original file F and the remaining n-l are the 'secondary servers' which maintain additional redundancy or parity blocks and help recover from failure. The figure 3.1 [17] shows a graphical representation of dispersal-encoding.

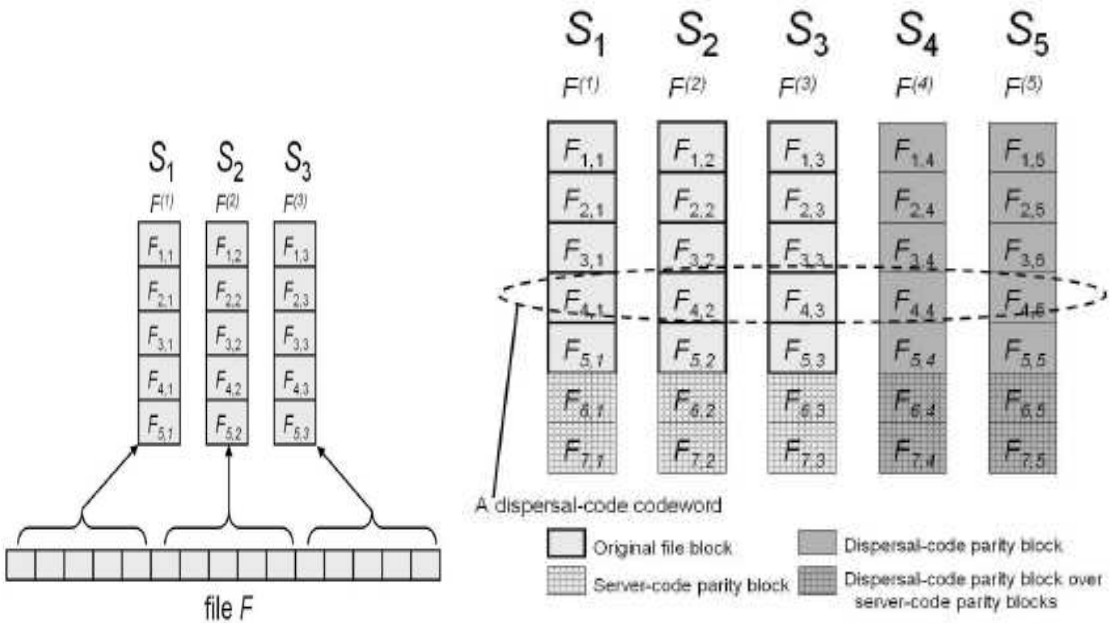

Figure 3.1 [17]: On the left, Original file represented in matrix form, on the right, Original file with parity blocks for both server and dispersal codes

Before encoding, the file F is split into l segments (S_1, S_2 … S_l) which are stored on the primary servers [17]. This clear text representation remains untouched by the subsequent encoding. The goal is to obtain a final encoding F_d of the file F. Then each of the l segments is encoded under the server code rate ϵ_c, to protect against small corruption at each server. This code extends the columns of the encoded message F_d by adding parity blocks. Followed by this, the dispersal code is applied to create parity blocks to reside on the secondary servers, S_{l+1} … S_n. It extends the rows of F_d across all of the n servers. These two encoding steps can be swapped, because the server code and the dispersal code commute. This scheme enables the client to verify that the blocks in a given position constitute a valid codeword in the dispersal code [17].

To embed the dispersal code in the IP-ECC, Pseudo Random Function values are added on the parity blocks for each row, i.e., on the blocks contained in secondary servers S_{l+1} … S_n [17]. To overcome the problem of creeping corruption attacks, matrix rows are authenticated with a message authentication code (MAC), which is computed with a secret key known by the client. MACs are embedded in the parity blocks of the dispersal code. Thus, IP-ECC code is constructed by making one block act as both the MAC and

parity block. This code helps verify the response received and also checks that the rows are not different from their original values in F [17].

A linear code called the 'aggregation code' is used for combining servers' responses in the challenge-response protocol. The aggregate response is a linear combination of rows of the encoded file matrix, and is a codeword in the dispersal code. The aggregation code in HAIL is computed on the fly and has zero storage overhead [17].

Hence HAIL [17] uses a challenge-response protocol to prove to a client that a stored file is intact. It improves on existing PORs (proofs of retrievability) techniques, while providing comparable per-server computation and bandwidth. It offers a three layer encoding and protects against a mobile adversary, i.e., which can corrupt servers and alter file blocks, and can corrupt every server in the system over time [17].

3.2 Cryptographic file system for Outsourced Data

A cryptographic file system was designed to protect the integrity (unauthorized alteration of data) and confidentiality (information leakage) of outsourced data [4]. It uses a block cipher mode of encryption for confidentiality and a Merkle hash tree construction for integrity. The following sections discuss the encryption scheme introduced in this paper [4], along with its implementation and evaluation.

3.2.1 Model

A file Φ is stored and maintained in chunks. Let $D = D_1 \ldots\ldots D_n$ be the content of the file Φ [4]. D_i is the i^{th} file block of the file Φ and n is the number of chunks. A block cipher of size b is used. The encrypted file content is stored in an untrusted storage, denoted by S. There is a separate, fixed-size, public trusted storage T, for each file. Hence the state of the file is described by $(t,s) \in T \times S$. Each file operation interacts with this state information of the storage spaces, and reads and updates the state (t; s). An algorithm halts if failure is declared. Hence if the state at the beginning of the algorithm was $(t,s) \in T \times S$, then after failure the state is $(t,s') \in T \times S$, where the t is unchanged but the state s' is modified [4].

Integrity of a file is defined as the "infeasibility of alteration of the file content, under a message attack"[4], i.e., an attacker A is allowed to make any number of file operation requests. It succeeds in violating the integrity of file Φ if on issuing a read request for block D_i, the block $D_i' \neq D_i$ is returned by the file. If $Adv_{\Phi}^{int}(A)$ is the probability that A may violate the integrity of Φ, then secure integrity is provided by the file if $Adv_{\Phi}^{int}(A)$ is negligible for all efficient attackers A [4].

Confidentiality of a file is defined as the "infeasibility of an attacker to learn any information about any file block, other than by explicitly reading the file block" [4]. If the attacker can eavesdrop or coerce a party with legitimate access of the file to read some parts of the file, still the other, unread portions are safe. The file Φ provides secure confidentiality if $Adv_{\Phi}^{conf}(A)$ is negligible for all efficient attackers A [4].

3.2.2 MAC Tree and Encryption Scheme Construction

Figure 3.2 [4], below shows a 3-ary MAC tree, where M is the MAC algorithm used and the dashed line denotes the 'nonce' for authentication [4]. Here a is the "arity" of the tree, $N^{(j)}_i$ represent counters, and a of the j^{th} level counters are concatenated and authenticated using the $(j+1)^{th}$ level parent counter $N^{(j+1)}_i$, to obtain $T^{(j+1)}_i$, the authentication tag [4]. The root counter is N^{L}_1, where L is the depth of the tree. This root counter, the depth L and the total number m of leaf counters are stored in trusted storage. The rest of the tree (counters $N^{(j)}_i$ and authentication tags $T^{(j)}_i$) is stored in an untrusted storage.

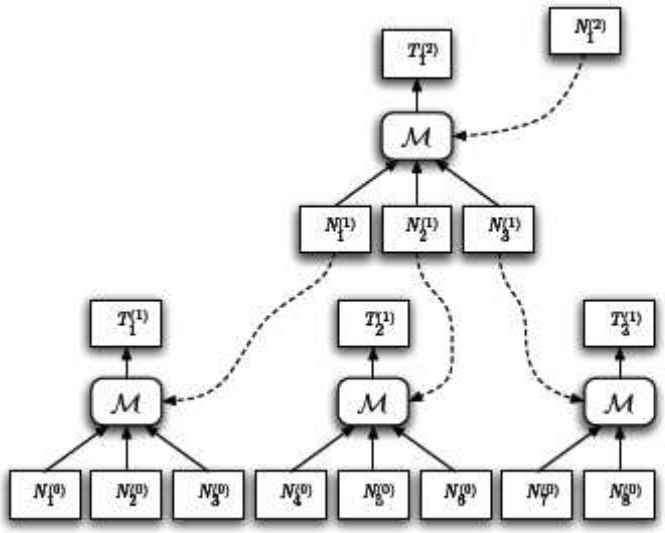

Figure 3.2 [4]: MAC tree construction

The MAC tree ensures that whenever all the verifications are passed, the leaf counters can be considered to be stored in trusted storage [4]. This transfer of trust from the root counter to the lower level counters can be done - since the root counter is stored in a trusted storage, and when the tag verification for the root level is successful, the children of the root counter can be trusted. This trust is passed down to the leaf counters [4].

A simple authenticated encryption scheme is constructed by the composition of the CTR encryption mode of operation and the MAC scheme. Figure 3.3 [4] illustrates the composite scheme [4].

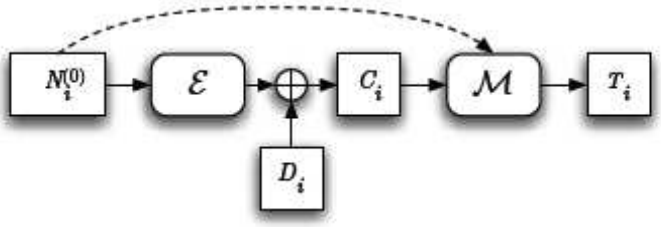

Figure 3.3[4] : Block wise authenticated encryption using leaf counters

Using the counter $N^{(0)}_i$, a pseudorandom sequence is generated, the file block D_i is XORed to this sequence to produce the cipher text block C_i. The counter $N^{(0)}_i$ is again used to authenticate C_i, and produce T_i the authentication tag [4].

3.2.3 Implementation

The prototype cryptographic file system [4] is based on CoreFS [7]. CoreFS is a FUSE [9] based network file system which provides core functionalities of a distributed file system. Figure 3.4 [4] represents the basic architecture of this cryptographic file system.

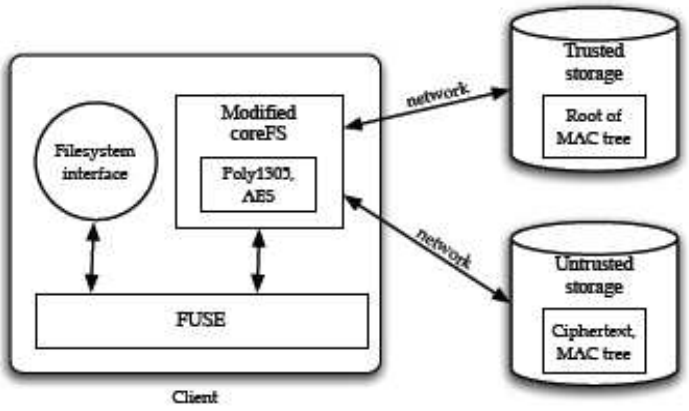

Figure 3.4 [4]: Basic Architecture

Core FS sends and receives encrypted and authenticated data to and from storage [4]. A file block of 4 KB is used. The AES block cipher and the Poly1305-AES MAC algorithm are used to build a 64-ary MAC tree. The client file system interacts with the server through the network [4].

3.2.4 Performance Evaluation

Two machines were used for the evaluation, one each for the client and server [4]. The Bonnie++ [23] benchmark was used to measure the performance of sequential read/write operations. Tests were done for 2 GB files. Elapsed CPU cycles were measured at various points during the benchmark. The performance was compared with Merkle hash tree based constructions. The network overhead was about 65- 85%, as compared to other costs. For Merkle hash tree based constructions, the cost for authentication is 60-70% of the cost of encryption - decryption. In this scheme, the cost for authentication is half of the cost for the Merkle hash tree. Figure 3.5 [4], shows the computational overhead of authentication and encryption/decryption for different tests in 10^6 cycles [4].

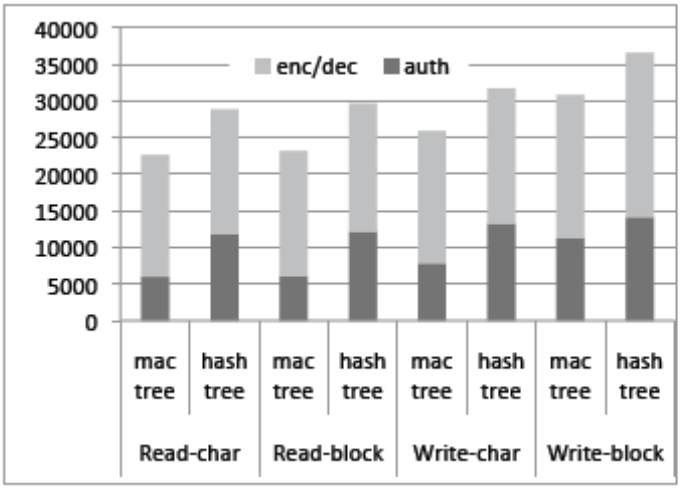

Figure 3.5 [4]: Micro benchmark result, in 10^6 cycles

Hence this paper introduced a new file encryption scheme for a distributed cryptographic file system, which provides confidentiality and security guarantees for outsourced data. A performance evaluation was done to compare this scheme, based on MAC tree construction, with Merkle hash tree based schemes. It was found to provide integrity protection at a smaller cost than Merkle hash trees [4]. The paper [4] also describes proofs to show that new scheme satisfies confidentiality and integrity guarantees.

3.3 Dynamic Provable Data Possession

Like the techniques above [4, 17], this paper [6] also discusses a framework where the server storing client data proves the integrity of this data to the client. However, where the former approaches discussed ways to verify the integrity of static data, this paper [6] introduces a framework for supporting provable updates on stored data.

In the PDP model, introduced in [10], by Ateniese et al., data, represented as a file F, is pre-processed by the client, and some metadata needed for verification purposes is stored at the client. The file is sent to an untrusted server for storage. The client keeps some secret information to verify the server's responses at a later stage. The server responds to challenges posed by the client to prove that the data has not been tampered with. However PDP schemes [10] apply only to static files, which never change and do not

consider updates such as insertions, deletions, etc., to the outsourced data [6]. The Dynamic Provable Data Possession (DPDP) Model introduced in this paper extends the PDP model to support verification of updates on data [6]. Given a file F, consisting of n blocks, an update is defined as an insertion, deletion, modification that can be done on the file. This DPDP framework is based on authenticated dictionaries, where rank information is used to organize dictionary entries [6]. This section discusses the performance and evaluation of the DPDP scheme.

3.3.1 The DPDP Model

The following procedures are a part of the DPDP scheme definition [6]:

(i) **KeyGen** is run by the client and outputs a secret key s_k and a public key p_k. The client stores the secret and public keys, and sends the public key to the server;

(ii) **PrepareUpdate** is run by the client. It prepares the file for untrusted storage. As input, it takes secret and public keys, a part of the file F, the update to be performed, e.g., full re-write, modify block, delete block etc and the previous metadata M_c. The output is a version of that part of the file (e (F)), encoded information e (info) about the update and the new metadata e (M). The client sends this output to the server. Encoding can be done by adding noise, encryption, etc.

(iii) **PerformUpdate** is a response to an update request from the client. It is run at the server and takes as input the public key, the previous version of the file F_{i-1}, the metadata M_{i-1} and the outputs of PrepareUpdate. Output is the new version of the file F_i, metadata M_i, metadata to be sent to the client, M'_c and its proof.

(iv) **VerifyUpdate** is run by the client to verify the server's behaviour during the update. It takes all inputs of the PrepareUpdate algorithm, plus the metadata M'_c sent by the server to the client. The output is an acceptance or rejection signal.

(v) **Challenge** is run by the client to create a challenge for the server. It takes as input the secret and public keys and the latest client metadata M_c. The output is a challenge c which is sent to the server by the client.

(vi) **Prove** is run at the server in response to a challenge from the client. The input is the public key, the latest version of the file and metadata, and the challenge c. It outputs a proof P that is sent to the client.

(vii) **Verify** is run by the client in response to the proof P from the server. It takes as input the secret and public keys, the client metadata M_c, the challenge c, and the proof P sent by the server. The output is "accept" if the server still has the file intact.

The DPDP scheme implementation is based on a 'rank-based authenticated skip list' [6]. In a skip list each node v stores two pointers, denoted by rgt (v) and dwn (v) used for searching. In an authenticated skip list, the node v also stores a label f (v) computed by applying a hash function to f (rgt(v)) and f(dwn(v)). Let F be a file with n blocks m_1, m_2 …… m_n. At the i^{th} bottom-level node of the skip list, a representation T (i) of block m_i is stored. The block is actually stored elsewhere by the server. Node v in the skip list stores the number of nodes at the bottom level that can be reached from v, called the rank of v, r(v). Figure 3.6 [6] below, shows nodes in a skip list. An update, such as insertion, deletion, or modification of a file block affects only the nodes of the skip list along a search path.

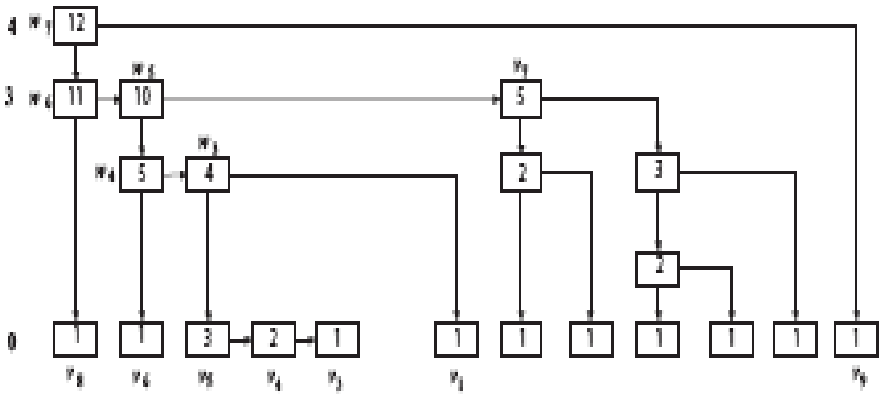

Figure 3.6 [6]: Rank based skip list

The top leftmost node of the skip list is the start node (w_7) [6]. For a node v, low (v) and high (v) are the indices of the leftmost and rightmost nodes at the bottom level reachable from it. These ranks help in accessing the i^{th} node at a lower level, by traversing a path that begins at the start node [6]. Efficiency can be improved by representing a block with

a tag. Tags are smaller in size than data blocks. Hence the skip list can be kept in memory and the client can download tags, instead of downloading the skip list. Tags protect the integrity of blocks, while the skip lists ensure security of tags [6].

3.3.2 Performance Evaluation

The performance of the DPDP scheme [6] was evaluated by comparing the communication and computational overhead with the original static PDP scheme [10]. Overheads for both schemes were compared using a test where the server wants to prove possession of a 1 GB file. The distance between the two lines in Figure 3.7 [6] represents the communication overhead of the DPDP scheme over the PDP scheme, which is attributed to the cost of proving updates to the stored data [6].

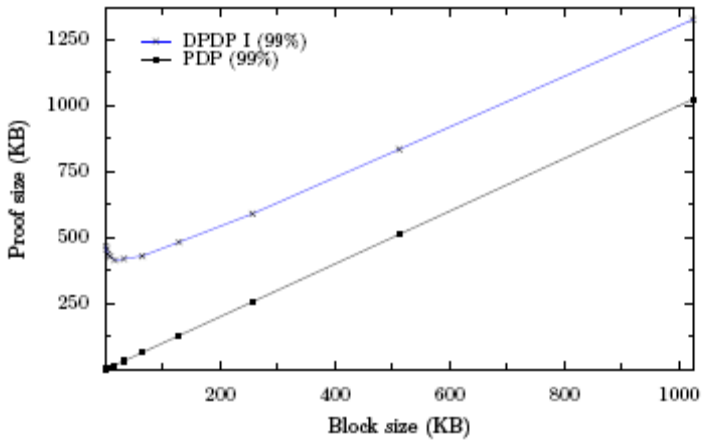

Figure 3.7 [6] : Communication overhead in DPDP due to updates

The paper [6] introduces a framework for dynamic provable data possession using a rank based authenticated skip list, with a small communication overhead. This goes a step ahead of other file verification techniques, which work for static files and do not support updates. This scheme can be used for verification at different levels in the file system with different users, e.g., each user can verify his own home directory [6].

3.4 Airavat

Airavat is a system designed to provide confidentiality, integrity, and privacy guarantees to MapReduce computations [13]. It uses a combination of DIFC, i.e., Decentralized information flow control and differential privacy. It protects against information leakage

through unauthorized storage channels (e.g., Mappers that leak data through unsecured network connections or unsecured local files) and protects the privacy of user data involved in MapReduce operations [13]. This section briefly discusses DIFC and Differential Privacy, followed by the implementation of Airavat.

3.4.1 DIFC and Differential Privacy

Decentralized information flow control (DIFC) is an access control paradigm that allows users to specify how data can propagate through a system, using labels and capabilities [13].

- Labels: A label (L) consists of a set of tags. Each data item or principal x, has an associated secrecy label, S_x, and an integrity label, I_x. The secrecy label prevents data leakage and the integrity label prevents data corruption.
- Capabilities: A capability C, allows a principal to add or drop labels. Each principal has a capability set C_p which determines what tags can be added or removed by the principal from its labels.
- Label change: A principal p can change its label from L_1 to L_2 if it has the capability to add the additional tags in L_2 and drop the tags that are not present in L_2.
- Label reduction is a change that causes an entity's label to become less restrictive, e.g., removing a security tag from a file's label.
- Flow restriction**:** Information can flow from an entity x to y if :

$$S_x > S_y \text{ and } I_x < I_y$$

Differential privacy: "A computation on a set of inputs is said to be differentially private if, for any of the inputs, the same result is produced whether this element is included in the input dataset or not" [13]. Differential privacy is well suited for cloud computing systems because the result of two differentially private computations is also differentially private. Another reason is that when the differential privacy is satisfied, it holds regardless of the background knowledge of the adversary. It guarantees the owner of any piece of data that the same privacy violations will occur, whether their data is included in the aggregate computation or not [13]. Hence no additional risk to privacy

occurs by participating in the computation. A differentially private computation reveals only limited information about its inputs [13].

3.4.2 Programming Model and Design

This section explains the components of the Airavat System [13].

(i) **Data providers** store data on a distributed file system [13]. They provide labels for pieces of data at the file level or at finer granularity, e.g., labels for individual data records. Data providers must specify the privacy bound, the privacy budget, and the reduction label for their data.

The privacy bound limits the dependence of the result of a computation on any single input [13]. Privacy budget is the limit that a data provider sets on the amount of permitted information leakage. Each differentially private computation with a privacy bound of ϵ_i, subtracts this amount from the privacy budget. If some privacy budget is leftover, the reduction label can still be removed from the labels associated with the result of the differentially private computation. After the privacy budget is exhausted, automatic declassification of more results cannot be done [13].

(ii) **Computation providers** (users) write the mappers and reducers executed by Airavat. Access control within these mappers and reducers is enforced using DIFC. If the data provider has approved the automatic reduction of labels on its data, Airavat restricts which mappers and reducers can be used in the computation [13]. Airavat adds appropriate random noise to each reducer's output to achieve differential privacy. The 'sensitivity limit' of each mapper must be specified by the computation provider. This limit is needed to calculate the amount of random noise to be added to the reducer to achieve differential privacy. If the output of the user's mapper has greater sensitivity than the pre-specified limit, Airavat modifies the output so that it remains within the limit. This change to the output

may compromise the correctness of the mapper to a certain extent, but it does not compromise the privacy guarantee given to the data owner [13].

(iii) **Workflow:** A MapReduce computation in Airavat starts with an Airavat-supplied mapper [13]. This is because the initial mapper reads the data schema and since data records are labelled, the mapper must be trusted. As long as a computation starts with trusted mappers and reducers, its result can have its label reduced using differential privacy (provided the overall privacy budget for the dataset is not exhausted).

The figure 3.8 [13] below shows the architecture of Airavat. The trusted components in this architecture are starred.

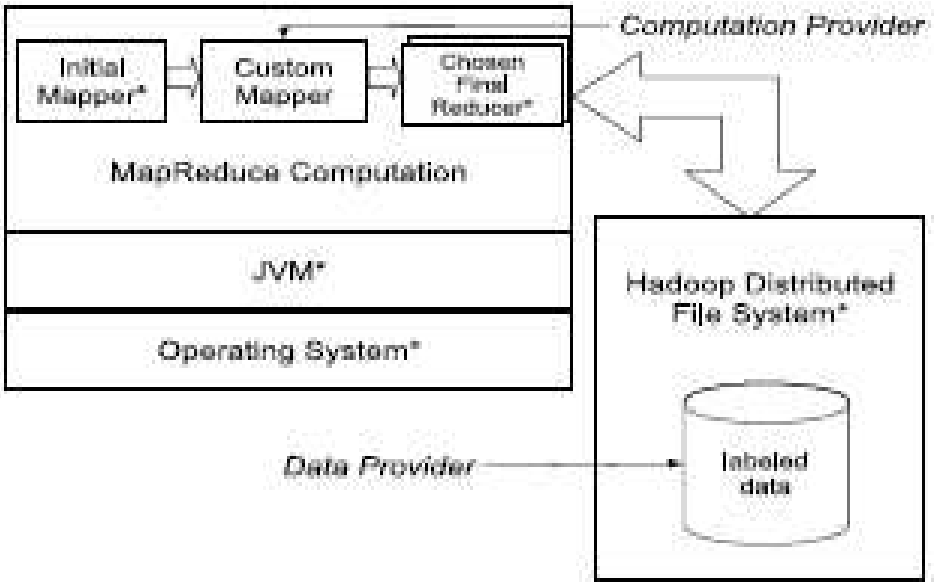

Figure 3.8 [13]: High level architecture of Airavat

Some computations cannot be differentially private [13], especially functions which are highly sensitive to specific inputs, e.g., locating a particular subsequence within a DNA sequence or determining the presence of a specific word in a Web page. In such cases, Airavat uses DIFC labels to secure both the data and the final output [13].

3.4.3 Evaluation

This section discusses the results of performance evaluation of MapReduce computations run using Airavat [13]. The experiments were run on Amazon EC2 on a cluster of 20 machines. Figure 3.9 [13] shows the percentage of the total time spent in the map, sort, and reduce phases of each benchmark. The execution times are normalized with respect to the time of the running on the original Hadoop MapReduce. The benchmarks show that Airavat slows down the computation by less than 25% [13].

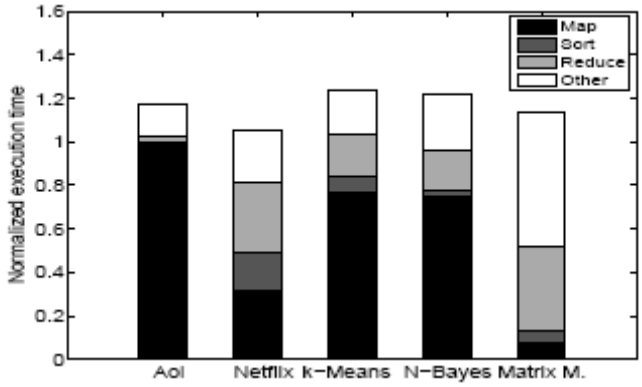

Figure 3.9 [13]: Normalized execution time of benchmarks when running on Airavat, compared to execution on Hadoop.

Airavat is the first system intended to provide security guarantees to MapReduce computations [13]. It provides trusted initial mappers and reducers, at the same time allowing users to insert their own mappers. It combines differential privacy with DIFC to secure MapReduce computations, with only a small amount of overhead [13]. The above section discusses the implementation and performance of Airavat.

3.5 Ripley

Ripley is a security paradigm in which replicated execution of client side code is done to secure distributed web applications [18]. A multi tier application consists of J2EE/.Net code executing on the server side and JavaScript executing at the client in the browser. This makes the application more responsive since the computation has moved closer to the client. However this placement of code at the client makes the overall computation less trustworthy. The other alternative (executing the entire code at the server) though more secure, leads to high latency round trips and poorer performance.

Ripley replicates a copy of the client side computation on the trusted server, including user events [18]. The results of this code are computed both at the client and by the replica of its code being executed at the server. Ripley observes the result of both computations and a discrepancy indicates a security violation by the code. Hence, Ripley helps maintain the responsiveness of client side execution, while keeping the computation as secure as it would be if the application had been run entirely on the server. The Ripley implementation discussed in this paper [18] is based on Volta [22], which is a distributing compiler for splitting .NET applications and translates them into JavaScript when needed.

3.5.1 Security threats and assurances

Some common threats to distributed applications are [18]:

(a) Data Manipulation: Manipulation of data before being sent to the server can happen within the browser or during transfer or due to a malicious client manufacturing new messages. Ripley resolves this issue by re-execution of the client code replica C, which takes into account the user - events received, but ignores the data manipulations occurred at the client.

(b) Code manipulation: The code sent to the client can be manipulated in the browser to produce undesirable effects, such as removal of consistency or input validation checks. E.g., in a game application the user might try to change the code to circumvent the rules of the game. While Ripley does not prevent code tampering, it does try to prevent code modifications which would lead to different RPCs being issued by the client.

(c) Script Injection and JavaScript worms: Ripley can help detect situations when benign users are affected by a malicious environment. This happens in case of cross site scripting attacks and JavaScript worms, both of which allow malicious actions to be executed on behalf of the user as long as it results in RPCs to the server. Ripley protects against script injection by executing the client replica C in a .NET CLR instead of JavaScript, thus rendering the injected JavaScript code

non executable when executed at the client replica. Thus if the client side component issues an RPC not issued by the replica, a RPC is detected.

Hence, Ripley does not eliminate the need for input sanitization however it restores the security that has been lost due to the application being distributed [18].

3.5.2 Architecture

As mentioned before, Ripley extends a distributed compiler Volta [22]. Volta originally takes a .NET application as input and splits it into a client side and a server side component. The client side component is translated into JavaScript for execution in a browser. This is achieved through class level annotations placed by the developer indicating which parts of the code are to be run on the client and the server. Ripley enhances the Volta tier splitter to perform additional rewrite steps [18].

Ripley makes the following modifications to Volta , as discussed in [18]:

(a) Capturing user events within the browser
(b) Transfer of events to the clients replica at the server for replay
(c) The server component S contains a Ripley checker which compares the arriving RPCs m and m' received from the client C' and server based client replicas C, for discrepancies.

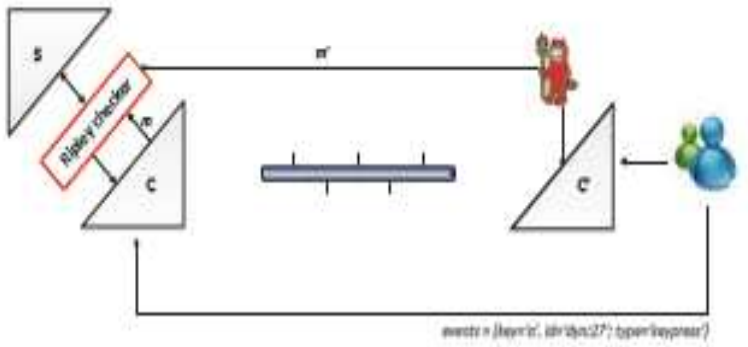

Figure 3.10 [18]: Architecture of RIPLEY:

Figure 3.10 [18] above indicates the Ripley architecture. The server receives and handles events arriving from the client C', and relays them to the client replica C [18]. Ripley

receives RPCs from both the client C' and the replica C, saves them in audit logs. and waits for RPCs m and m', which are the results of the computation at the client replica and the client respectively. It compares the two results and relays the RPC call to the application server only when the m and m' are equivalent. The response of the server is delivered to both the replica C and the client C'.

There could be multiple clients connected to the same server, hence the client replica C is executed in a light weight process like abstraction, called APPDOMAIN. A separate APPDOMAIN for each client replica has the advantage of memory isolation and eliminates inter process communication overhead, as inter-APPDOMAIN communication is comparatively cheaper [18]. This solution scales because the replicas are run in .NET instead of JavaScript making it about two times faster. Also, a lightweight browser emulator is used instead of a full-fledged browser to reduce the memory and CPU utilization. This browser emulator hosts the client replica C instead of an actual browser. It keeps track of relevant UI state including the structure of the DOM and contents of editable elements. Since it performs no rendering or layout related computations, it avoids a lot of computation [18].

3.5.3 Performance Evaluation

Five benchmark applications were built to test the performance of Ripley [18]. The applications were a Shopping Cart application, online Sudoku Game, Blog Application, Speed typing quiz and online Quiz, all built on top of Volta. The Shopping Cart application faces client side security threats, e.g., a malicious client can manually reset the total before it is sent to the server. With Ripley, these computations can be done at the client side – since the user events are replicated at the server side, the server also maintains an abstract of the cart and can verify the total amount as soon as it is received from the client. Similarly in the Sudoku game, the local and global validation checks of the game state can be bypassed by a malicious user, leading them to declare the puzzle finished without actually playing it. Here Ripley checks the validity of the final solution based on the event stream which it receives as input. Similarly Ripley protects against

JavaScript worms in the blog application, tampering with word spelling checks in the speed typing test application and bypassing solution checks in the Online Quiz [18].

However there is some network overhead, and excess CPU and memory utilization associated with Ripley. E.g., the network overhead is shown in Figure 3.11 [18].

	Application RPCs		Network overhead for event transfer					
					Uncompressed		Compressed	
Benchmark	RPCs	Bytes	Events	RPCs	Total	Norm.	Total	Norm.
Shopping Cart	1	157	13	1	1,548	119	300	23
Game of Sudoku	1	160	146	8	16,953	116	812	5.6
Blog Application	9	1,595	252	11	31,090	123	863	3.4
Speed Typing Test	4	1,598	556	28	63,945	115	1422	2.6
Online Quiz	2	275	66	4	7,801	118	445	6.7

Figure 3.11 [18]: Network Overhead measurements with Ripley

Columns 2-3 in the table show the network usage of the application itself. Columns 4-7 show the network overhead introduced by Ripley. The "total" column shows the total number of bytes and the "Norm" column shows number of bytes per event. Extra network activity occurs due to transmission of event data to the server. However since these messages are sent asynchronously, they don't slow down client side execution. This overhead can be reduced by compressing the event stream, shown in columns 8-9. CPU overhead is introduced because of the replica running on the server [18].

The above section discusses Ripley, which is a fully automated approach to ensuring the security of distributed web applications. The efficiency of Ripley was proved by demonstrating the use of Ripley in five realistic web applications, with a small amount of overhead in terms of CPU, memory and network utilization [18].

3.6 Conclusion

The above sections discussed some popular approaches for implementing security in cloud computing systems. The most significant class of techniques involves verification of the integrity of data stored at the server (external storage), by the client. Some systems in this category which I discussed above were HAIL [17] and Cryptographic distributed file system [4]. Both provide ways for ensuring the availability and integrity of the file

stored at a server. While HAIL [17] makes use of a three layer encoding scheme, the cryptographic file system [4] makes use of a combination of MAC tree and block encryption scheme. However none of these approaches, explicitly address the problems of verifying the intactness of a file which is not static, i.e., files to which some updates and deletions have been made. This is addressed in the dynamic provable data possession model [6], which extends a previous static provable data possession scheme to support provable updates to stored files, using rank based skip lists. Another system, Airavat [13] discussed above is of a different category and is designed for large scale distributed computations, specifically MapReduce. This approach combines the use of decentralized information flow control (DIFC) and differential privacy to ensure secure computations. It also ensures that the results of these computations don't reveal sensitive user data which was used in the computation, with minimal overhead. Such security models will gain even more importance in the future, considering that more and more users are moving data and computations to the cloud, hence giving rise to the need for stronger security measures.

4. Summary

This essay focuses on two important applications of cloud computing – distributed information processing and outsourced data storage. It discusses in depth two distributed processing programming models: MapReduce [15] and Dryad [19]. MapReduce [15] allows the distributed information processing task to be expressed as Map and Reduce functions on data sets in the form of key/value pairs. MapReduce has been used in a wide number of applications such as sorting, searching, data mining tasks, etc. Although the original implementation of MapReduce does not involve pipelining, a pipelined version has been proposed and evaluated in [26], called the Hadoop online prototype, since it is built on the Hadoop open source implementation of MapReduce. Dryad [19] on the other hand allows the computation to be expressed in the form of an execution graph, comprising of vertices as sequential programs and edges as data transport channels. Many programming models can be built on top of Dryad such as DryadLINQ [29]. DryadLINQ [29] allows the developer to write programs for large scale computations in a popular high level language, leaving the concurrency and scheduling intricacies to the system.

The second half of the survey discusses security models for outsourced data as well as distributed computations. Security models such as HAIL [17] and cryptographic file system [4] help in verifying the integrity of static files stored at a remote server. HAIL [17] makes use of three layer encoding where as cryptographic file system [4] uses MAC trees and encryption to provide privacy and integrity guarantees. Airavat [13] guarantees the confidentiality and integrity of data involved in large scale distributed computations in MapReduce, whereas Ripley [18] , uses replicated execution to secure distributed web applications. To conclude, this essay discusses popular programming abstractions for cloud computing in detail and gives an overview of a number of security models for outsourced data and distributed computation. Hence, this survey will act as useful compilation of important areas related to cloud computing.

Bibliography

1. 7. Things you should know about Cloud Computing. http://net.educause.edu/ir/library/pdf/EST0902.pdf
2. A. Juels and B. S. Kaliski. PORs: Proofs Of Retrievability For Large Files. *Proceedings of the 14th ACM conference on Computer and communications security*, 584–597, 2007.
3. A. Thusoo et. al. Hive: A warehousing solution over a Map-Reduce framework. In *Proceedings of the Conference on Very Large Databases,* 1626-1629, 2009.
4. A. Yun, C. Shi and Y. Kim. On Protecting Integrity and Confidentiality of Cryptographic File System for Outsourced Storage. *Proceedings of the 2009 ACM workshop on Cloud computing security,* 67-76. 2009.
5. B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment,* 1277-1288. 2008
6. C. C. Erway, A. Kupcu , C. Papamanthou and R. Tamassia. Dynamic Provable Data Possession. *Proceedings of the 16th ACM conference on Computer and communications security.* 213-222.2009.
7. CoreFS, available at http://sourceforge.net/projects/corefs.
8. F. Chang, J. Dean , S. Ghemawat, W. C. Hsieh, Deborah, A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS),* 26(2), 2008.
9. FUSE, available at http://fuse.sourceforge.net/
10. G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song . Provable Data Possession at Untrusted Stores. *Proceedings of the 14th ACM conference on Computer and communications security.*598-609, 2007.
11. Google. Protocol Buffers: Google's Data Interchange Format. Documentation and open source release; http://code.google.com/p/protobuf
12. http://en.wikipedia.org/wiki/Cloud_computing
13. I. Roy, S.T.V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce, *NSDI 2010*. San Jose, USA. 2010.

14. J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM, 53(1)*, 72-77, 2010.
15. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on large data clusters. *USENIX Association OSDI '04: 6th Symposium on Operating Systems Design and Implementation,* 137- 149, 2004.
16. J. Gray, A.S. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. Vandenberg. Data-mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, 189–210, 2002. Carleton Scientific. Also as MSR-TR-2002-01.
17. K. D. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. *Proceedings of the 16th ACM conference on Computer and communications security,* 187-198. 2009.
18. K. Vikram, A. Prateek and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution**.** *Proceedings of the 16th ACM conference on Computer and communications security,* 173-186. 2009.
19. M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 59-72, 2007.
20. M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo and A. Rasin. MapReduce and parallel DBMSs: Friends or Foes? *Communications of the ACM 53 (1)*, 64-71, 2010.
21. M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo and A. Rasin. A comparison of approaches to large scale data analysis. *Proceedings of the 35th SIGMOD international conference on Management of data,* 165-178, 2009.
22. Microsoft Corporation. Microsoft Live Labs Volta. http://research.microsoft.com/~emeijer/CloudProgrammability.html, 2007.
23. R. Coker, Bonnie++ File System Benchmark, http://www.coker.com.au/bonnie++/
24. S. Jha, A. Merzky and G.Fox. Programming Abstractions for Clouds. *Conference Proceedings, Cloud Computing and its Applications, 2008*.
25. Sloan Digital Sky Survey. http://skyserver.sdss.org

26. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy and R. Sears. MapReduce Online. *Technical Report No. UCB/EECS-2009-136,* 1-16, 2009.
27. Tera Sort benchmark http://research.microsoft.com/barc/SortBenchmark/.
28. W. Wang, Z. Li, R. Owens and B. Bhargava. Secure and Efficient Access to Outsourced Data. *Proceedings of the 2009 ACM workshop on Cloud computing security.*55-66. 2009.
29. Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language *Symposium on Operating System Design and Implementation (OSDI),* San Diego, USA. 2008.