World Engineering & Applied Sciences Journal 8 (2): 92-98, 2017
ISSN 2079-2204

DOI: 10.5829/idosi.weasj.2017.92.98

# Detecting High Level System Problems by Analyzing the Error Logs Using Spark

*[1]A. Sarumathi and [2]P. Ezhumalai*

[1]ME, CSE, RMD Engineering College, Kavarapettai, India
[2]HOD, CSE, RMD Engineering College, Kavarapettai, India

**Abstract:** The biggest challenge for most Ecommerce business is to collect, store and organize data from multiple data sources. Ecommerce services create huge amount of console log files which are unstructured and unfriendly. These challenges make it difficult for operators to understand log messages and extract the errors from the huge log files. This information is required to detect the performance problems. Existing disk based systems like Map Reduce cannot offer low latency services due to the high access latency to hard disks. Our project uses a novel approach for console log mining from the field of big data analytical methods and the in memory techniques to automatically monitor and detect the abnormal execution traces from the console log files The project aims to extract the useful information from the huge console logs efficiently using the in memory techniques such as Spark in a Hadoop Ecosystem. The huge unstructured log files are analyzed using the memory based techniques where the data is cached in memory rather than loading it on the disks as the traditional Hadoop systems which leads to a faster analysis and mine the useful data.

**Key words:** Big data Analytics · Hadoop · Spark · MapReduce

## INTRODUCTION

Ecommerce growth in today's world is phenomenal and the amount of users and data generated are huge. The biggest challenge for most Ecommerce businesses is to collect, store and organize data from multiple data sources. There's certainly a lot of data waiting to be analyzed and it is a daunting task for some E-commerce businesses to make sense of it all. Big Data paves the way for more organized data and enables business owners or marketing managers to track and better understand a variety of information from many different sources. To analyze such a large volume of data, big data analytics is typically performed using specialized software tools and applications for predictive analytics, data mining, text mining, forecasting and data optimization. Today's large-scale Ecommerce services run in large server clusters in datacenter and cloud computing environments. These system architectures enable highly scalable Internet services at a relatively low cost. However, detecting and diagnosing problems in such systems bring new challenges for both system developers and operators. One significant problem is that as the system scales, the amount of information operators need to process goes far beyond the level that can be handled manually and thus there is a huge demand for automatic processing of monitoring data. At these scales, "performance failures" are common and may even indicate serious impending failures [1].

Operators would therefore like to be notified of such problems quickly. Despite the existence of a variety of monitoring tools, the monitoring already available in every application is the humble console log which is usually ignored. Console logs are convenient to use and reflect the developers' original ideas about what events are valuable to report, including errors, execution tracing, or statistics about the program's internal state. But exploiting this information is difficult because console logs are both machine-unfriendly (they usually consist of unstructured text messages with no obvious schema or structure) and human-unfriendly (each developer logs information useful for his own debugging, but large systems consist of many software modules developed by different people and log messages from different

**Corresponding Author:** A. Sarumathi, ME, CSE, RMD Engineering College, Kavarapettai, India.

modules must often be correlated to identify a problem). These challenges make it difficult for operators to understand log messages and extract the errors from the huge error log files.

High-level cluster programming models like MapReduce and Dryad have been widely adopted to process the growing volumes of data in industry and the science. These systems simplify distributed programming by automatically providing locality-aware scheduling, fault tolerance and load balancing, enabling a wide range of users to analyze big datasets on commodity clusters. Most current cluster computing systems are based on the acyclic data flow model where records are read from a stable storage and written back to the stable storage. But there are many applications that are not supported by this acyclic flow of data. One such class of applications is that reuse the working set of data across multiple parallel operations. Those are applications that use the iterative algorithms commonly used in machine learning and graph applications. Our work focuses on using a new framework called SPARK that support these applications and retain the fault tolerance of Map Reduce. Spark offers a distributed memory abstraction called resilient distributed datasets (RDDs) [2].

The project aims to extract the useful information from the huge console logs efficiently using the techniques such as Spark in a Hadoop Ecosystem. The huge unstructured log files are analyzed using the in memory techniques where the data is cached in memory rather than loading it on the disks as the traditional Hadoop systems which leads to a faster analysis and mine the useful data. From the survey conducted we find that a Hadoop cluster running on over 200 machines tends to yield 24 million lines of logs during a two-day period.

Our project proposes a novel approach for console log mining from the field of big data analytical methods to automatically monitor and detect the abnormal execution traces from the console log files. Through these techniques we have found that, when analyzing the logs based on our model we can accurately extract the errors and speed up the analysis process by 10x times than a Hadoop model. The results are significant because that error information is not only used by the developers to improve their application performance but also by data scientists, operators to improve their business

**Console Log Preprocessing:** In this section we review some of the log preprocessing techniques described in [3] which we also used.

**Log Parsing:** The method presented in [4] can eliminate most of the ad-hoc guessing in parsing free text logs. The method first analyzes the source code of the program generating the console log to discover the "schemas" of all log messages. Specifically, it examines the printing statements in the source code (e.g. printf) and performs type analysis on the arguments to these statements. The technique distinguishes the parts of each message that are constant strings from the parts that refer to identifiers such as program variables or program objects with high accuracy

They first convert a log message from unstructured text to a data structure that shows the message type and a list of message variables in (name, value) pairs to get possible log message template strings from either the source code or program binaries and match these templates to each log message to recover its structure (that is, message type and variables). Their log parsing method consists of two steps: the static source code analysis and the runtime log parsing. The static source analysis step not only extracts all log printing statements from the source code, but also tries to infer types of variables contained in the log messages. Thus we can discover the message format even with the complex type hierarchies in modern object oriented languages. The runtime log parsing step uses information retrieval techniques to "search" through all possible strings extracted from template for best matching "schemas" for each log message. The process is stateless, so it is easy to parallelize and implement in a data stream processor in the online setting document [5].

**Identifying Event Traces:** The detection technique in [6] relies on analyzing *traces*, which are sets of events related to the same program object. For example, a set of messages referring to the opening, positioning, writing and closing of the same file would constitute an event trace for that file. Within the event stream, however, events of different types and referring to different sets of variables are all interleaved. One way to extract traces from the stream is to *group by* certain field of the events, a typical operation for stream data processing. The challenge of using the grouping key automatically is handled. The method described in [5], automatically finds the grouping key from historical log data by discovering which message variables correspond to *identifiers* of objects manipulated by the program. All events reporting the same identifier constitute an event trace for that identifier. This "group-by" process also occurs in a

stream processor. A group-by stream processor that converts the single interleaved event stream into many event traces, one for each identifier is implemented. They assume the event traces to be independent from each other.

**Representing the Event Traces:** The event traces are converted to a numerical representation suitable for applying PCA detector. In [5], each whole event trace is represented by a *message count vector* (MCV), which has a structure analogous to the *bag of words* model in information retrieval, where the "document" is the group of messages in an event trace. The MCV is an N-dimensional vector where N is the size of the set of all useful message types across all groups (analogous to all possible "terms") and the value of vector element yi is the number of times event i appears in a group (corresponding to "term frequency"). For example, in a system consisting of four event types, opening, reading, writing and closing, a trace of (opening, reading, closing) will be represented in MCV as (1,1,0,1) while (opening, writing, writing) can be represented as (1,0,2,0). Message count vectors are a compact representation of event traces, but two problems preclude their direct use in online scenario. First, they do not carry any time information, so they cannot be used to detect operations that are anomalous due to events being spaced too far apart in time (i.e. slowness). Second, the original MCVs are constructed based on the entire event traces which could span arbitrarily long time. This is not possible in an online setting. We use the MCV to represent a *session*, which we define as a subset events in an event trace representing a single logical operation in the system and have predictable bound in its duration.

**Technical Background:** We propose a System for console log mining using spark which is a inmemory technique[7] that executes faster than traditional disk based system. The main abstraction of spark is Resilient Distributed datasets[1] (RDD) which we will see in brief.

**Resilient Distributed Datasets (RDD):** The main goal of RDD is to provide abstraction to support applications that reuse the results in multiple parallel operations at the same time it should preserve the properties of MapReduce such as fault tolerance and locality aware scheduling and scalability. Out of our desired properties, the most difficult one to support efficiently is fault tolerance. In general, there are two options to make a distributed dataset fault-tolerant: checkpointing the data or logging the updates made to it. In large-scale data analytics checkpointing the data is expensive: it would require replicating big datasets across machines over the datacenter network, which typically has much lower bandwidth than the memory bandwidth within a machine [8] and it would also consume additional storage (replicating data in RAM would reduce the total amount that can be cached, while logging it to disk would slow down applications). Consequently, we choose to log updates. However, logging updates is also expensive if there are many of them. Consequently, RDDs only support coarse-grained transformations, where we can log a single operation to be applied to many records, series of transformations are used to build an RDD (i.e., its lineage) and use it to recover lost partitions.

RDDs are well-suited for data-parallel batch analytics applications, including data mining, machine learning and graph algorithms, because these programs naturally perform the same operation on many records. RDDs would be less suitable for applications that asynchronously update shared state. RDDs can be stored in memory between queries without requiring replication. Instead, they rebuild lost data on failure using lineage: each RDD remembers how it was built from other datasets (by transformations like map, join or groupBy) to rebuild itself. RDDs allow Spark to outperform existing models by up to 100x in multi-pass analytics [2]. RDDs can be only created through deterministic operations called transformations. Examples of transformations include map, filter, group by and join.

**Comparing RDD with Distributed Shared Memory:** The applications that uses Distributed shared memory (DSM) read and write to an arbitrary location in a global address space.DSM is a very general abstraction but difficult to implement efficiently on commodity clusters. The main difference between RDDs and DSM is that RDDs can only be created i.e. written through bulk transformations, while DSM allows reads and writes to each memory location. This restricts RDDs to applications that perform bulk writes, but allows for more efficient fault tolerance. The advantage of RDD is, not to worry about the check pointing as it can recover using the lineage. Furthermore, only the lost partitions of an RDD need to be recomputed upon failure and they can be recomputed in parallel on different nodes, without having to roll back the whole program. One interesting observation is that RDDs also let a system tolerate slow nodes (stragglers) by

running backup copies of tasks, as in Map Reduce. Backup tasks would be hard to implement with DSM as both copies of a task would read/write to the same memory addresses [9].

The other benefits are, in bulk operations on RDDs a runtime can schedule task based on data locality to improve performance. Partitions that do not fit in RAM can be stored on disk and will provide similar performance to current data flow systems

**Parallel Operations:** Several parallel operations can be performed on RDDs:

**Reduce:** Combines dataset elements using an associative function to produce a result at the driver program

**Collect:** Sends all elements of the dataset to the driver program. For example, an easy way to update an array in parallel is to parallelize, map and collect the array

**Foreach:** Passes each element through a user provided function. This is only done for the side effects of the function which might be to copy data to another system or to update a shared variable

**RDD Transformations and Actions:** RDD transformations are lazy operations that define a new RDD, while actions launch a computation to return a value to the program or write data to external storage. The various RDD transformations available in Spark is given detailed in [1]. Examples of actions include c*ount* (which returns the number of elements in the RDD), *collect* which returns the elements themselves) and *save* (which outputs the RDD to a storage system). In addition to these operators, users can ask for an RDD to be cached. Furthermore, users can get an RDD's partition order, which is represented by a Partitioner class and partition another RDD according to it. Operations such as groupByKey, reduceByKey and sort automatically result in a hash or range partitioned RDD [10].

Programmers invoke operations like map, filter and reduce by passing closures (functions) to Spark. As is typical in functional programming, these closures can refer to variables in the scope where they are created. Normally, when Spark runs a closure on a worker node, these variables are copied to the worker. However, Spark also lets programmers create two restricted types of shared variables to support two simple but common usage patterns

**Broadcast Variables:** If a large read-only piece of data (e.g., a lookup table) is used in multiple parallel operations, it is preferable to distribute it to the workers only once instead of packaging it with every closure. Spark lets the programmer create a "broadcast variable" object that wraps the value and ensures that it is only copied to each worker once.

**Accumulators:** These are variables that workers can only "add" to using an associative operation and that only the driver can read. They can be used to implement counters as in MapReduce and to provide a more imperative syntax for parallel sums. Accumulators can be defined for any type that has an "add" operation and a "zero" value. Due to their "add-only" semantics, they are easy to make fault-tolerant.

**Spark Programming:** Spark provides the RDD abstraction through a language integrated API in Scala [4]. Scala is a statically typed functional and object-oriented language for the Java VM. On top of Spark, Spark SQL, Spark Streaming, MLlib and GraphX are built for SQL-based manipulation, stream processing, machine learning and graph processing, respectively. Spark developers write a driver program that connects to cluster to run workers as shown in Fig. 1. The driver defines one or more RDDs and invokes actions on them. The workers are long-lived processes that can cache RDD partitions in RAM as Java objects. RDDs themselves are statically typed objects parameterized by an element type. For example, RDD[Int] is an RDD of integers. By transforming an existing RDD a dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called flat Map which passes each element through a user-provided function by changing the persistence of an existing RDD. By default, RDDs are lazy and ephemeral. That is, partitions of a dataset are materialized on demand when they are used in a parallel operation (e.g., by passing a block of a file through a map

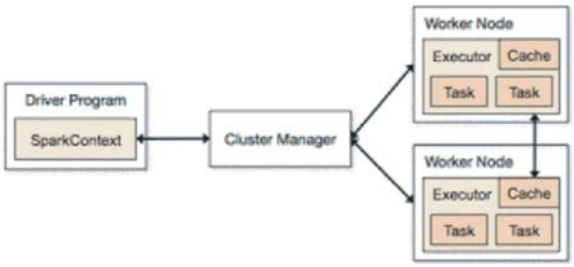



Fig. 1: Spark Execution Overview

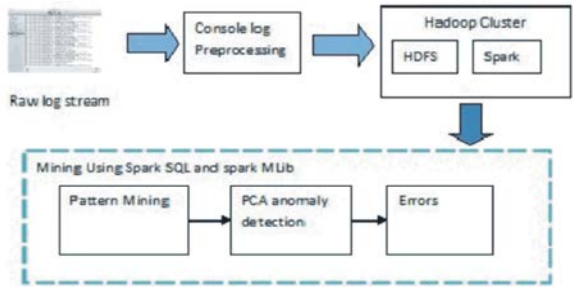


Fig. 2: Overview of console log mining

function) and are discarded from memory after use. A user can change the persistence of RDD through a cache action. It is a hint that says that it should be kept in memory so that it will be reused. However if there is no enough memory to be cached it will be recomputed [11].

**Propsed System:** Compared to the online approaches for log mining which cannot handle a huge amount of data, we propose a model that emphasizes the use of spark for detecting problems with the large scale systems. Consider a website is experiencing the errors and the operator wants to know the cause of the error. He is in need to search terabytes of console log to find the Cause of the problem. Our model is capable of handling these huge unstructured log files efficiently. Fig. 2 gives the overview of the console log mining. As mentioned in section II the console logs are preprocessed and are transferred to the Hadoop Cluster for further processing

**Mining Using Spark:** Using Spark and RDDs, the operator can load just the Error messages from the logs into RAM across a set of Nodes and query them interactively. We also use spark SQL which can effectively mine the log files in lightning speed. The following is the sample code to mine the console log using scale

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.cache()
```

Line 1 defines an RDD backed by an HDFS file where we have the preprocesed log files as a collection of lines of text), while line 2 derives a filtered RDD from it. Line 3 asks for errors to be cached.

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs. The following lines uses the spark DataFRame API [12].

```
val df = textFile.toDF("line");
val errors =df.filter(col("line").like("%error%"));
errors.collect().foreach(println);
```

Line 1 creates the DataFrame from the RDD. line 2 Filters the errors from the DataFrames.

**Frequent Pattern Mining:** We also mine the frequent patterns.. *Frequent pattern is defined* to be a session and its duration distribution such that: 1) the session is frequent in many event traces; 2) most (e.g., 99.95th percentile) of the session's duration is less than Tmax, a user-specified maximum allowable *detection latency* (the time between an event occurring and the decision of whether the event is normal or abnormal). Condition (1) guarantees that the pattern covers common cases so it is likely to be a normal behavior. Condition (2) guarantees the pattern can be detected in a short time. We mine the archived data periodically for frequent patterns. These patterns are used to filter out normal events in the online phase. We cannot apply generic frequent sequence mining techniques because 1) sessions many interleave in the event traces (e.g. two reads happen at the same time) thus "transaction" boundaries are not clear. We need to simultaneously segment an event trace into sessions and mine patterns. However, because the durations of sessions can have large variations, fixed time windows will not give satisfactory segmentation, which suggest that we shall model the distribution of durations. 2) Events can be reordered in the traces because of unsynchronized clock in a distributed system, which preludes the use of techniques requiring total ordering of events. In our algorithm described below, we use frequent patterns to tolerate the poor time-based segmentation accuracy resulting from random session interleaving. The frequent patterns, once discovered, can be used to deinterleave the events to estimate a clean duration model.

**PCA Anamoly Detection:** To uncover the true anomalies from this noisy data, we use a statistical anomaly detection method, the PCA detector, which is shown to be accurate in offline problem detection from console logs and from many other systems, [13]. As with frequent pattern mining, the goal of PCA is to discover the statistically dominant patterns and thereby identify anomalies inside data. PCA can capture patterns in high-dimensional data by automatically choosing a (smallset of) coordinates-the principal components-that reflect covariation among the original coordinates. Once we estimate these patterns from the archived and periodically updated data, we use them to transform the incoming data to make abnormal patterns easier to detect. PCA detection also has a model estimation phase followed by an online detection phase. In the modeling phase, PCA captures the dominant pattern in a transformation matrix PPT, where P is formed by the top principal components chosen by PCA algorithm. Then in the online detection phase, the abnormal component of each message count vector y is computed as ya = (I - PPT )y, i.e., ya is the projection of y onto the abnormal subspace. The squared prediction error SPE = kyak2 (squared length of vector ya) is used for detecting abnormal events: We mark vector y as abnormal if SPE = kyak2 > Q_, (5) where Q_ denotes the threshold statistic for the SPE residual function at the (1-_) confidence level [11]. Due to limitations of space, we refer readers unfamiliar with these techniques to, [13] for details. In a real deployment, the model can be updated periodically. Note that because of the noisier data in this phase and the workload dependent nature of the non-pattern data, the model update period for PCA is usually shorter than that for frequent pattern mining.

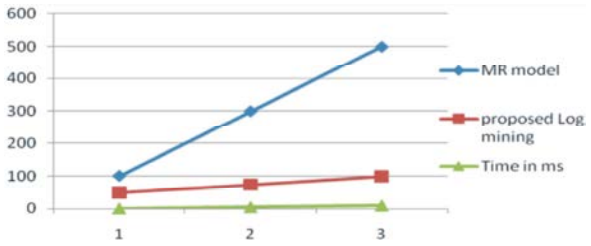
**Experimental Setup:** We evaluate our approach with real logs from a 203-node Hadoop [1] installation of a large scale Ecommerce website. Hadoop is an open source implementation of the MapReduce framework for large-scale parallel data processing. Ecommerce websites is gaining popularity tremendously. So it is important to understand its runtime behaviors, detect its execution anomalies and diagnose its performance degradation issues. To evaluate our approach we replayed the same setof logs, containing over 24 million lines of log messages with an uncompressed size of 2.4GB. The logs were generated from 203 nodes running Hadoop for 48 hours, completing many standard MapReduce jobs such as distributed sort and text scan. The average machine load varies from fully utilized to mostly idle. The log

Table 1: Frequently occurring anomaly

| | |
|---|---|
| 1 | Write exception client gives up |
| 2 | Write failed at beginning |
| 3 | Received block that does not belong to any file |
| 4 | Replication monitor timedout |
| 5 | Empty packet for block |

contains 575,319 event traces, corresponding to 575, 319 distinct file blocks in Hadoop File. The two types of false positives are 1. Normal background migration and over replication. For example, falsepositive over-replicating is due to a special application request rather than a system problem. These are indeed rare events only 368 occurrences across all traces corresponding to rare but normal operations. These cases are hard to handle with a fully unsupervised detector. In order to handle these cases, we allow operators to manually add patterns to encode domain-specific knowledge. Our approach founds all the anomaly from the logs. Some of the frequently occurring anomaly is given in the Table1

We use Spark that has the lightning speed to process the data which outperforms the traditional disk based system. The following graph illustares the timing comparison of proposed model and the other models [14].




Thus the proposed model is efficient both in terms of time and accuracy. Our current work does not make any change to the log generation code in the program. But we can also aim to improve current console log generation frameworks to allow more dynamic and fine granularity control of individual message types. With such a framework, we can do real-time control of console log generation, which will enable us to further reduce the overhead of generating unnecessary logs, while making sure the interesting and important messages are kept in the logs [15, 16].

## CONCLUSION

Our goal is to find the needles in the haystack that might indicate operational problems, without any manual input. When given a huge unstructured log files we

addressed the problem of extracting the useful error information. We proposed a new approach using Spark for finding the errors and tracking its details in a file which will help the development team to fix these errors in the future and it will improve the performance of the large scale system. The challenges of handling the unstructured and unfriendly log files are simplified. Our project's approach for console log mining from the field of big data analytical methods and the in memory techniques automatically monitor and detect the abnormal execution traces from the console log files. Through these techniques our work found that, when analyzing the logs based on our model we can accurately extract the errors and speed up the analysis process by 10x times than a Hadoop model. The results are significant because that error information is used by the developers to improve their application performance. This System would also benefit data scientists, operators to improve their business.

## REFERENCES


1. Zaharia, M., M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker and I. Stoica, 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation, pp: 2.
2. Zaharia, M., M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, 2010. Spark: Cluster computing with working sets, in Proc. 2nd USENIX Conf. Hot Topics Cloud Comput., pp: 10.
3. Shi, X., M. Chen, L. He, X. Xie, L. Lu, H. Jin, Y. Chen and S. Wu, 2014. Mammoth: Gearing hadoop towards memory-intensive mapreduce applications, IEEE Trans. Parallel Distrib. Syst., 99(1).
4. Scala. http://www.scala-lang.org.
5. Online System Problem Detection by Mining Patterns of Console Logs Wei Xu_, Ling Huang, Armando Fox, David Patterson, Michael Jordan _EECS Department, UC Berkeley, Berkeley, CA, USA.
6. Detecting Large-Scale System Problems by Mining Console Logs Wei Xu EECS Department, Ling Huang Intel Labs Berkeley Armando Fox EECS Department, David Patterson EECS Department, Michael I. Jordan EECS and Statistics Department, UC Berkeley 26 th International Conference on Machine Learning, Haifa, Israel, 2010.
7. Cheney, J., L. Chiticariu and W.C. Tan, 2009. Provenance in databases: Why, how and where, Found. Trends Databases, 1: 379-474.
8. Hindman, B., A. Konwinski, M. Zaharia, A. Ghodsi, A. DJoseph, R.H. Katz, S. Shenker and I. Stoica, 2010. Mesos: Aplatform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, EECS Department, University of California, Berkeley.
9. Scaling Spark in the Real World: Performance and Usability by Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, Matei Zaharia,Databricks Inc., MIT CSAIL.
10. Applications powered by Hadoop.http://wiki.apache.org/hadoop/Powered By. Y. Bu, B. Howe, M. Balazinska and M.D. Ernst.
11. HaLoop, 2010. efficient iterative data processing on largeclusters. Proc. VLDB Endow., 3: 285-296.
12. Hall, D., 2008. A scalable language, anda scalable framework. http://www.scala-blogs.org/2008/09/scalable-languageandscalable.html.
13. Hadoop Map/Reduce tutorial http.
14. //hadoop.apache.org/common/docs/r0.20.0/mapred tutorial.html.
15. Yu, Y., M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P.K. Gunda and J. Currey, 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In OSDI '08, San Diego, CA.
16. Ko, S.Y., I. Hoque, B. Cho and I. Gupta, 2009. On availability of intermediate data in cloud computations. In HotOS'09.