Chapter 11
Automatic Big Data Provenance Capture at Middleware Level in Advanced Big Data Frameworks

Anu Mary Chacko, Alfredo Cuzzocrea, and S.D. Madhu Kumar

Abstract Huge amounts of data are being generated by Internet of Things (IoT) devices. Termed as Big Data, this data needs to be reliably stored, extracted, and analyzed. Capturing provenance of such data provides a mechanism to explain the result of data analytics and provides greater trustworthiness to the insights gathered from data analytics. Capturing the provenance of the data stored in NoSQL databases can help to understand how the data reached its current state. A holistic explanation of the results of data analytics can be achieved through the combination of provenance information of the data with results of analytics. This chapter explores the challenges of automatic provenance capture at the middleware level in three different contexts: in an analytics framework like MapReduce, in NoSQL data stores with MapReduce analytic framework, and in NoSQL stores with SQL front ends. The chapter also portrays how the provenance captured in the MapReduce framework is useful for improving the future executions of job reruns and anomaly detection, apart from its use in debugging.

11.1 Introduction

With the rise in usage of the Internet and social media websites, digital data is now treated as an asset and is used to derive insights or meaningful information. With the advent of the Internet of Things (IoT), the amount of data has increased exponentially. Most of the data generated are unstructured and are of different file types. As data are generated in large volumes, they are termed as "Big Data." Big Data can contain information generated by sensors, chatter in the social media like Twitter or

A.M. Chacko (✉) • S.D. Madhu Kumar
National Institute of Technology Calicut, Kozhikode, Kerala, India
e-mail: anu.chacko@nitc.ac.in

A. Cuzzocrea
University of Trieste and ICAR-CNR, Trieste, Italy

Z. Mahmood (ed.), *Connected Environments for the Internet of Things*,
Computer Communications and Networks,
https://doi.org/10.1007/978-3-319-70102-8_11

Facebook, or loads of information collected for user profiling. This data can act as powerful trend predictors if they can be reliably analyzed and mined. The reliability of the analytic results depends on how "good" the data used for analysis is, which in turn depends on the source of the data and transformations that the data underwent. Data provenance is the metadata that captures the history of data from its creation to how it reached its current state. In our day-to-day activities, different levels/types of provenance are collected by audit trails, logs, and change tracking software. All such data gives information that contributes to the history of data or provenance. Provenance metadata focuses on isolating all relevant details of history in one metadata in a systematic way, such that the advantages of verifiability and querying are obtained.

With the increase in complexity of data management, data provenance research is gaining a lot of attention. Every aspect of provenance handling, starting from capture and storage to representation, security, and querying, needs efficient schemes so that provenance can be seamlessly used. In the literature, there are schemes for applications to disclose provenance explicitly and schemes to capture provenance automatically at operating system and middleware level. Making all applications provenance aware is not a feasible solution, and so automatic capture of provenance is needed. Automatic capture can be done at operating system or at middleware layer. At the operating system level, the system is not able to understand the context in which data is used, and so if provenance is collected at this level, it is very fine grained, making it difficult to query and use the provenance collected. Automatically capturing provenance at middleware level gives the application designers the flexibility to focus on logic of application without worrying about provenance disclosure. Especially, in the context of Big Data, where a large number of Big Data applications are being deployed every day, automatic provenance capture at middleware layer is a feasible option for provenance capture.

This chapter focuses on processing of IoT data on the Big Data analytic frameworks. The next section provides a background to the work done in provenance research, and the rest of the chapter discusses approaches to capture provenance of analytics done on MapReduce framework and data stored and analyzed in NoSQL data stores.

11.2 Background

In eScience, many tools like *Chimera*, *myGrid*, and *CMCS* [1] were developed for provenance capture of scientific workflows. The primary focus for collecting provenance in workflows was to ensure reproducibility of experiments and providing provisions for debugging. Provenance was very interesting to the database community as it provided explanation for the results obtained. Tools like *DBNotes* [2], *Trio* [3], and *PERM* [4] focused on database provenance. Automatic provenance capture was explored in the construction of *PASS* [5], a modified Linux kernel that captured provenance of all operations happening in the kernel by observing the read/write

system calls. Similar approach was used in *SPADE* [6] where provenance capture scheme was instrumented into the application to capture intra-provenance at compile time. Most of the works except *PASS* and *SPADE* described in the literature followed a disclosed provenance approach where specific applications were made provenance aware for domain-specific requirements.

Provenance is of interest in the area of Big Data, as provenance provides a mechanism to explain the results and provide proofs for the validity of data. The main focus areas of Big Data provenance is in storage, analytics, and data stores. Munniswamy et al. [7] developed *PASS* to work for cloud storage. They provided different versions that store provenance along with data in SimpleDB or Amazon S3. Another work in this area is by Sletzer et.al. [8] who proposed techniques to instrument Xen hypervisor to capture provenance of operations on the virtual hypervisors. In Big Data analytics, a major work was done to develop the analytic framework MapReduce provenance aware. *RAMP* (Reduce and Map Provenance) [9] captures provenance of MapReduce workflows while the job executes. The provenance is generated at the end of job execution resulting in a performance overhead of 20–70% as reported by the authors. *HadoopProv* [10] attempts to improve the performance of job execution of MapReduce jobs while capturing provenance by deferring the generation of provenance to the time when it is needed. *Lipstick* [11] tool enables database style workflow provenance to be captured for jobs written in Pig script. *Titian* [12] is a library that has been created for provenance support for jobs running in Apache Spark, and the authors claim that observed overhead for job execution is below 30%.

The early works in data provenance were mainly domain specific and consisted of making particular applications provenance aware. Through this approach, rich provenance information is obtained, as the semantics of the applications is an integral part of the provenance capture system. But in Big Data scenario, retrofitting all applications to make them provenance aware is not practical. On the other hand, capturing provenance at the operating system level, e.g., *PASS* [5], x being captured. The main issue here is the large size of provenance and false dependencies. Hence there is a need for schemes to capture provenance automatically at middleware layer.

Typically, the applications or software that acts as glue between operating system and applications are categorized as middleware [12]. Semantically, the middleware layer is placed between the operating system and application layers. Middleware caters to multiple applications at a time. Creating middleware to make a set of applications provenance aware provides the developer with the option of capturing provenance of multiple applications/data in applications in one go. In the Big Data landscape, where the number of applications for processing data is as well big, retrofitting provenance into all applications is not a practical solution.

In the literature, there are scientific experiments that followed this approach, where the workflow middleware was adapted to capture provenance of all workflows running on top of it e.g., *MyGrid* [13]. By making the workflow queue provenance aware, all the jobs running on it become automatically provenance aware. In the big data scenario, provenance capture contexts can be broadly divided into two: in the context of analytic tools and in case of Big Data stores. The following sec-

tions explain the techniques proposed for capturing provenance using middleware approach, in analytic tool like MapReduce and NoSQL store like MongoDB.

11.3 Provenance in MapReduce Workflows

In the context of Big Data applications, the collected data is useful only if it is amenable to analytics. The result of the analytics can be confidently used if and only if it is verifiable. So capturing provenance for analytic frameworks is a must. The major challenge with provenance capture is the high performance overhead caused to the job during provenance capture. The provenance collected is usually used for debugging results. This section explores a different approach for capturing provenance of MapReduce workflows and explores the use of provenance collected for improving the execution of MapReduce jobs during incremental runs and anomaly detection.

11.3.1 Provenance Capture

In the context of MapReduce, three types of provenance can be collected – job provenance (coarsely grained), data provenance (finely grained), and transformation provenance (process provenance):

- *Job provenance* is an example of coarsely grained provenance and captures the signature of job.
- *Data provenance* captures relation between the output data and the input data of a MapReduce job.
- *Transformation provenance* goes beyond the job execution and tries to capture details of job execution.

A lazy approach of generating provenance after the completion of job execution is adopted in our approach so that results of job are available for the user for review, while provenance is being generated. In this approach, provenance is captured by writing a wrapper code to the classes like Mapper and Reducer so as to capture details important for provenance into temporary files. At the completion of the job, a background MapReduce job is executed to consolidate the temporary files and generate provenance. Provenance thus generated constitutes the fine-grained data provenance. This provenance is useful for debugging the result or to understand flow of data from input to output.

Job provenance is the coarsely grained provenance captured by modified MapReduce framework so as to create signature of a particular run of a job. The details captured as part of job provenance are details of input-output, file names, input-output key types and input-output file formats, Mapper, Reducer and Combiner class names, MD5 hash of jar files, and offsets to which data is read in the current job run.

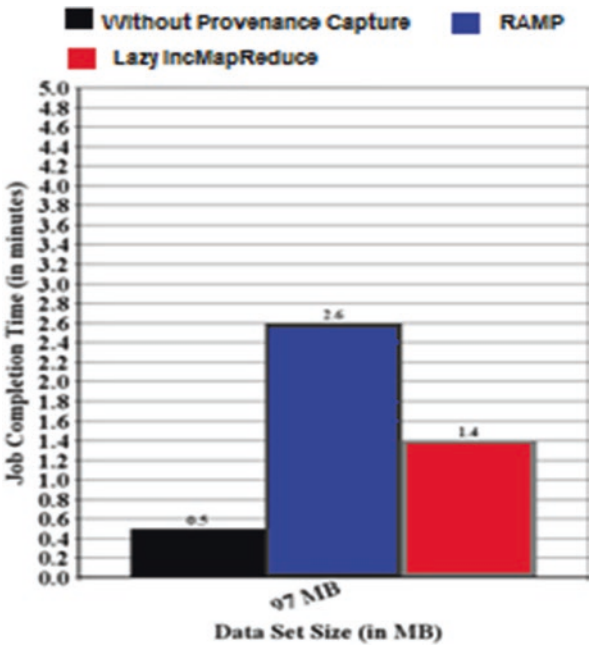

Fig. 11.1 Comparison of performance (job completion time) in word count problem

Modified MapReduce (*Lazy IncMapReduce*) was tested on a cluster of nine DataNodes and a NameNode for Hadoop. The HBase cluster consisted of nine region servers and a master server. Each system was configured with 4GB RAM and 500 GB hard drive. The results of experiment by running the above jobs are discussed next.

Provenance collection showed a performance and storage improvement for word count problem as shown in Figs. 11.1 and 11.2, respectively. For the word count problem, proposed method showed an average 50% improvement in the job completion time and an average 70% storage optimization over RAMP. This storage optimization is obtained as provenance collected is preprocessed and stored in HBase.

Another experiment was conducted to filter random *Apache WebLog* [14] data. A sample of 1 lakh weblogs was used to filter good weblogs out of ill-typed weblogs. Around 1 lakh logs were analyzed in Lazy IncMapReduce, and performance analysis is shown in Figs. 11.3 and 11.4. HBase storage required 186% more memory than RAMP as shown in Fig. 11.4.

In case of WebLog filtering, for each output record, a corresponding provenance record is written. As the number of output records increases, the number of write operations increases, and hence the storage requirement becomes larger, and job execution time degrades. These experiments indicate that significant storage and performance improvements are obtained in *Lazy IncMapReduce* for jobs where the number of output keys is less than the number of input keys.

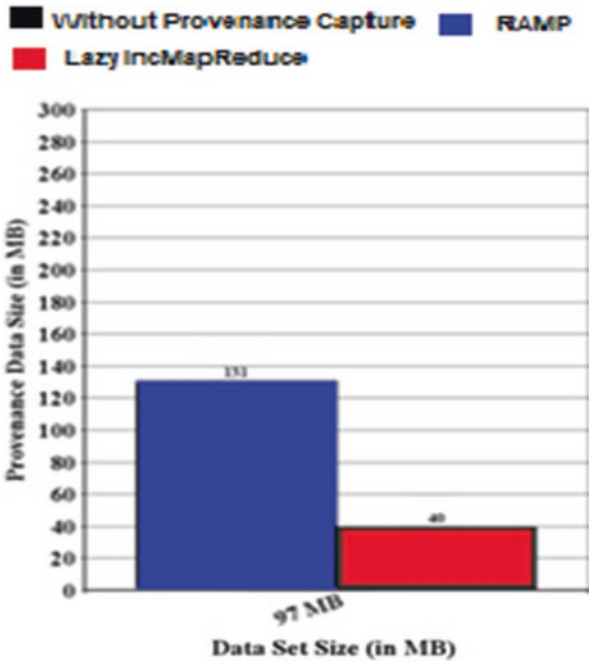

Fig. 11.2 Comparison of storage requirement – word count problem

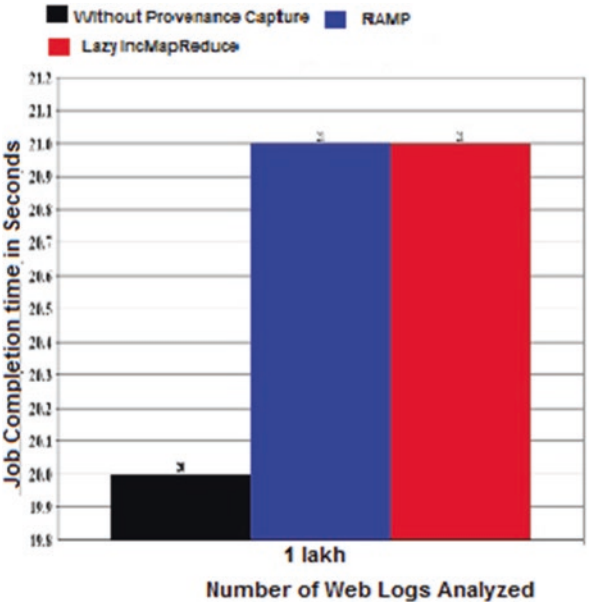

Fig. 11.3 Comparison of performance (job completion time) in WebLog filtering problem – MapReduce without provenance vs RAMP vs Lazy IncMapReduce

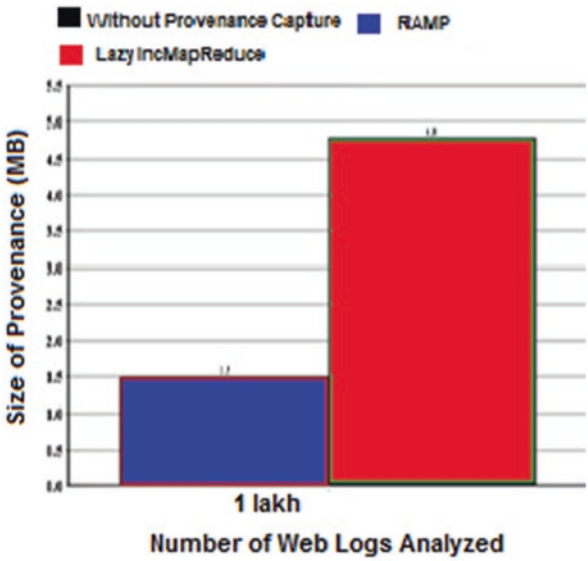

Fig. 11.4 Comparison of storage requirement: WebLog filtering problem – RAMP vs Lazy IncMapReduce

Transformation provenance consists of details of job execution. This can be extracted from the various logs created as part of the job execution. Once the job execution is over, the logs from the different nodes are consolidated, and transformation provenance can be mined from the logs using a rule-based execution framework. This is done by identifying patterns in the logs and defining rules to extract the information from log to deduce provenance. This provenance captures information on MapReduce execution, like details of task and job execution, split creation, dataset access, etc. Here, there is no change made to the MapReduce framework, but provenance is deduced from the preexisting logs.

Hadoop generates detailed log for all the services running in the cluster like *NameNode*, *DataNode*, *JobTracker*, and *TaskTracker*. The details of job extracted from the logs are used to generate a transformation provenance profile for the job. Provenance profile is captured as XML file so as to enable easy querying. The provenance profile contains complete information about the execution of the job run, cluster configuration information, as well as ERROR and WARNING messages generated.

The three provenances together provide the holistic picture about the MapReduce job execution and its results. In the literature the use of provenance collected has been demonstrated mainly for debugging of results. In the rest of this section, two novel uses of provenance collected are discussed: (1) the use of data and job provenance to improve the workflow execution of subsequent runs of MapReduce jobs and (2) the use of transformation provenance for anomaly detection.

11.3.2 Incremental MapReduce Using Provenance

In the literature, there are schemes like Incoop [15] and Itchy [16] that implement incremental MapReduce. *Incoop* [15] uses the concept of memoization and needs modified HDFS to implement incremental MapReduce. *Itchy* [15] uses the term provenance, but the provenance used is not conventional but a mapping between intermediate map result and input. Proposed approach, *Lazy IncMapReduce*, aims to reuse the provenance generated as part of workflow execution to improve the execution of job reruns.

In many MapReduce applications, the input data is of *append* only variety. For such MapReduce jobs, the old results can be reused, and computation can be restricted to the new appended input values alone. This will result in significant reduction in execution time. The following cases were evaluated as part of this work. Input file is considered to be *append* only:

- Case 1: Input file is appended with data or when input files are added.
- Case 2: Input file is processed as a sliding window of data.

The following section describes how *Lazy IncMapReduce* works for the two different cases described above.

Case 1: Jobs Rerun with Additional Data Appended to Input File or with Additional Files

When a MapReduce job is submitted by the user, its coarse-grained provenance is captured, and provenance store is queried to see if it is the first run of the job or rerun. It is considered as an incremental run if the provenance store returns a job with the following conditions satisfied:

- Jar file with same MD5 hash as current job
- Same Mapper, Reducer and Combiner classes as current job
- Same input files as current job
- Same type of output key and values as current job
- Same input format as current job

After verification, the current job submitted is classified as:

- *New run*: if no matching job is found in the provenance store, in this case, the job is run as a single MapReduce job with provenance capture.
- *Incremental run*: if a previous run of the same job is found, the input file is checked to see whether it is a case of new data appended to existing input files or new input files added. In both cases, MapReduce program runs only on the new data that was not processed in earlier run. Output of this job is combined with the output of old job by executing MapReduce job with *Identity Mapper*. This is the default Mapper class provided by Hadoop that writes all input key value pairs into output. This is diagrammatically illustrated in Fig. 11.5.

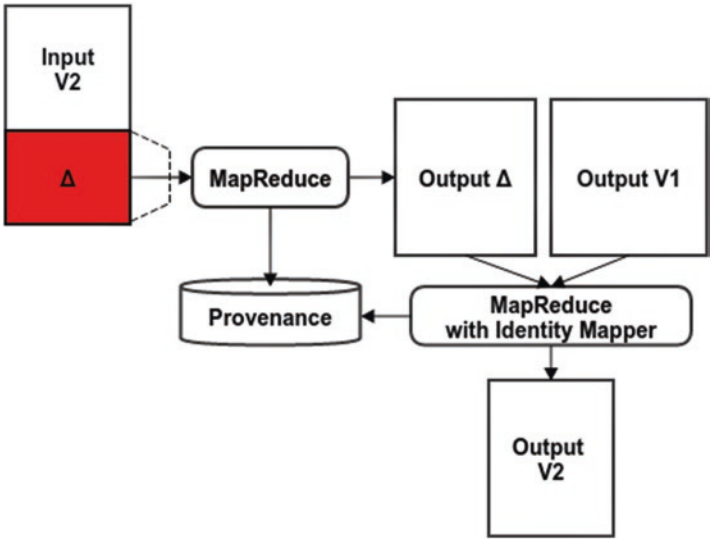

Fig. 11.5 Incremental MapReduce when new data is appended to input file

Case 2: Job Rerun on Sliding Window of Input Data
Frequently there are cases where MapReduce jobs are run for a window of data (e.g., last 30 days data). Every consecutive day, the window slides, deleting a day's information and adding a new day's information. *LazyIncMapReduce* is designed to handle incremental runs for such MapReduce jobs that process window of input data using tuple level fine-grained provenance. The first run of the job processes the window selected with provenance capture. In the next run of the job, the window has some new data appended and some old data removed. The data can be considered as having three sections as shown in Fig. 11.6:

- *Old data*: Data which is part of the old window but not included in the current job's window
- *Common data*: Data which is common to both old job and current job
- *New data*: Data which is newly added in the file and not part of old job.

The strategy for job reruns is as follows:

- Perform MapReduce on the *new data*.
- Refresh the previous job output file to reflect the removal of *old data* from input file. This is achieved by doing selective refresh of the output file of the previous run. The fine-grained provenance captured in the previous job run is used here to trace back the input for each output element. Depending on which part of the input file the input records lie, the following strategies are opted to prepare the refreshed output file:
  - Scenario 1: If inputs fall completely in common data, no refresh is done, and output file is used as such.

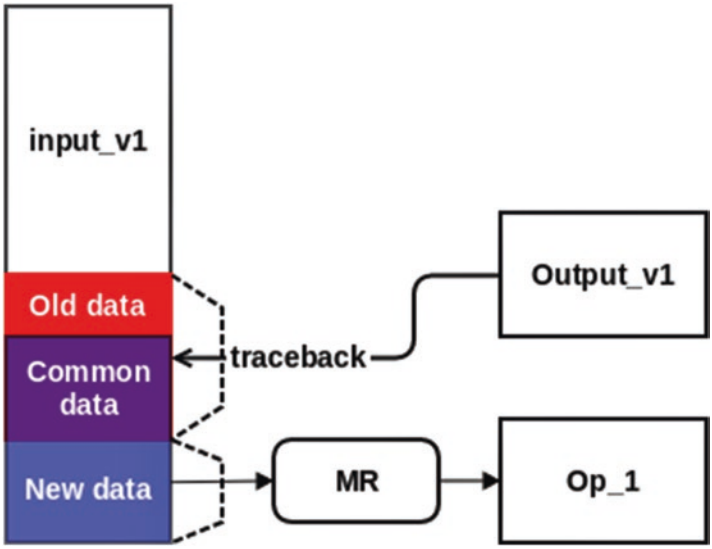

Fig. 11.6 Rerun of job in incremental MapReduce for the sliding window case

- Scenario 2: If dependent inputs fall in both common data and old data, then a selective refresh needs to be done for those input offsets using a MapReduce job.
- Scenario 3: If the dependent inputs are completely in old data, then the records in the old file can be discarded.

- Combine all the results by running a MapReduce job with Identity Mapper.

11.3.2.1 Experimental Results

For the evaluation of incremental MapReduce, two jobs whose number of output keys is less than number of input keys were considered: *word count* job and *grep* job. In these two cases, input file was appended with data, and sliding window of data approach was tested. Performance for the incremental run was analyzed.

Case 1: Input File Appended with 500 MB Data for Incremental Run
Performance analysis was done for incremental run when an input file (4.4GB) is appended with additional 500 MB data for both word count job and grep job. In the first run, a small run time overhead of 5 s was observed. But in the incremental run, our prototype outperforms the traditional MapReduce with 50% of run time improvement. Figure 11.7 shows a reduction of 50% in execution time of incremental run of word count job, and Fig. 11.8 shows a 98% reduction in execution time of incremental run of grep job.

Thus, there is a significant performance improvement for job reruns in *Lazy IncMapReduce* when jobs are rerun with additional data appended in the file, as the

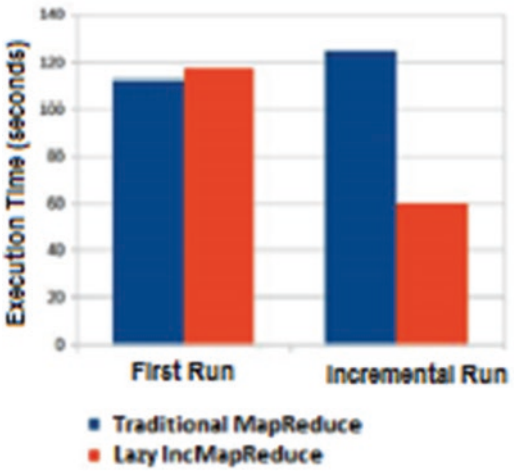

Fig. 11.7 Job execution time (word count problem) when 500 MB data is appended

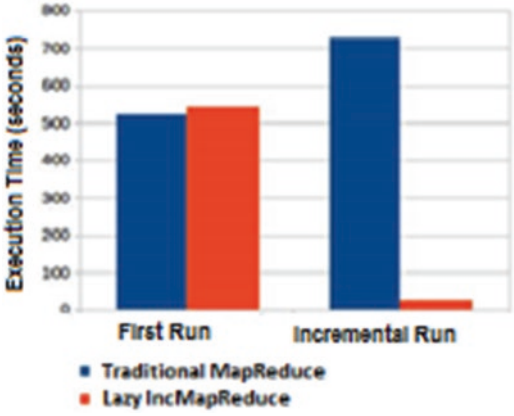

Fig. 11.8 Job execution time (grep problem) when 500 MB data is appended

data in the previous run is not processed but output of the previous run is merged with MapReduce output of new data.

Case 2: Processing Input File with a Sliding Window of 500 MB Data for Incremental Run

To evaluate the performance of *Lazy IncMapReduce* in such cases, incremental MapReduce job was executed by moving the processing window by 500 MB. Performance analysis of incremental MapReduce was done for both *word count* Job and *grep* Job. The results obtained for the *word count* problem is shown in Fig. 11.9 and for *grep* problem, in Fig. 11.10.

In the case of experimental run of sliding window *word count* problem, a performance overhead of 400% was found. On analysis, it was found that this overhead was because of the bottleneck caused by *NameNode* during selective refresh. The inherent design of MapReduce gives *NameNode* the task of preparation of splits

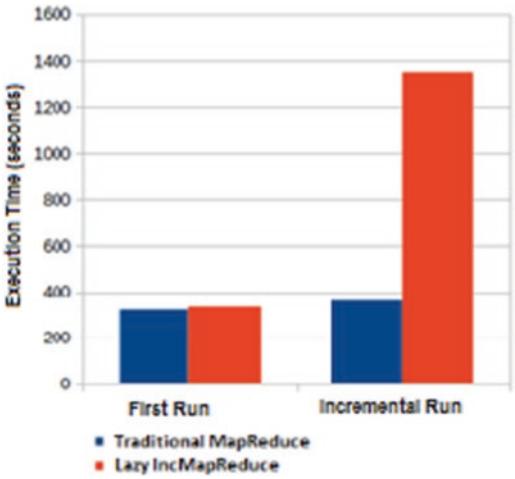

Fig. 11.9 Job execution time (word count problem) when window of processing is "slided" by 500 MB

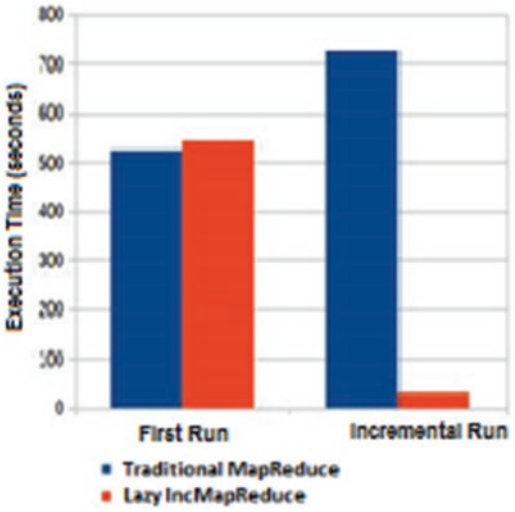

Fig. 11.10 Job execution time (Grep problem) when window of processing is "slided" by 500 MB

during selective refresh. When there are lots of output records that need refreshing, many splits have to be generated for facilitating selective refresh. Out of 1352 s of the incremental run, 1155 s were taken for selective refresh. The preparation of files for selective refresh was the main cause of the overhead. In the case of *grep* job, provenance query provided very few records for selective refresh, and so the time for preparing splits was greatly reduced. Thus, in the case of *grep*, incremental run in *Lazy IncMapReduce* gives a better performance over traditional MapReduce.

11.3.3 Anomaly Detection Using Transformation Provenance

Execution of MapReduce is handled transparently by Hadoop. Hadoop is an open source project designed to optimize handling massive amount of data through parallelism using inexpensive commodity hardware. The earlier versions of Hadoop concentrated on task distribution, and very little attention was given to security. In later versions, various techniques like mutual authentication, enforcement of HDFS file permission, using tokens for authorization, etc. were provided to enhance security. But Hadoop has a serious lack in detection of anomalous behavior. Hadoop does the data processing and scheduling in a way which is transparent to the user. There is a possibility that a compromised user or compromised node could do some malicious activity to gain additional resource usage and obstruct services to the other nodes for its purposes. An attacker could perform some attacks to slow down the data processing and create a denial of service situation in the cluster. Currently, any such anomalous activity would go unnoticed despite having security features enabled in Hadoop. Transformation provenance captured can throw light on many such malicious activities happening during MapReduce workflow.

After job execution, a provenance file is generated, and this provenance profile is used to detect anomalous behavior. The tool developed performs the set of checks as listed below:

- Check if input to all the tasks are valid.
- Check if output is stored in proper location.
- Total number of tasks performed.
- Status of nodes in cluster.
- Analyze task execution times.

Checking input and output file locations from the configuration files and actual execution log can throw light if any malicious user has made changes for leaking confidential data. The check on total number of tasks performed helps to identify any skipped computations. Logs provide information on the status of cluster. As job allocation is handled transparently by the framework, the user does not know whether the resources are properly utilized. The task execution times on different nodes can further throw light on the efficiency of nodes. This was verified by simulating a SYN flood attack on a slave machine in the cluster of three machines to make the slave system less responsive. The run times of all the map tasks were collected with and without attack. The mean and standard deviation for both the set of values were calculated. When there is an attack, the deviation is high (approx 50%) from the mean indicating that the run times of map tasks vary. Figure 11.11 describes a sample output of anomaly detection using provenance profile.

This section described the capture of data, job, and transformation provenance for jobs executing in MapReduce framework and the uses of provenance captured. The provenance collected is not only useful for error debugging but also for improving incremental runs of jobs. Transformation provenance captured is useful in

```
Check-1 : Input to all the tasks are valid inputs.


INFO: Tasks taking input from the correct directories
------------------------------------------------------------
Check-2 : Output are stored in proper locations


INFO: Task is storing output to the correct directories
------------------------------------------------------------
Check-3 : Checking Number of tasks performed


There are  10 map attempts and  10 map tasks and  1 reduce tasks
------------------------------------------------------------
Check-4 : Status of nodes in cluster


INFO : No Anamalous Behaviour from the nodes in the cluster. ['hmaster', 'sc1', 'sc2']   nodes performed [4,
 5, 4]  completed Tasks respectively
------------------------------------------------------------
Check-5 : Analyzing Task Execution times


Mean Values of Map and Reduce Exection times are : 113192.7 323908.0
Standard Deviation of Map and reduce  Execution times are 89495.8571623 0.0
There is a deviation of 79.0650432071 % of values from mean for Map task and 0.0 % of values from mean for r
educe task
```

Fig. 11.11 Example of anomaly detection by analyzing run times in provenance profile

detecting anomalous behavior in the cluster. The next section describes a novel approach to capture provenance of data stored in NoSQL data stores.

11.4 Provenance for Big Data Stores

The massive data generated from the different IoT sources are usually stored in highly scalable databases like NoSQL data stores. In order to have an end-to-end provenance captured, there needs to be provenance captured in NoSQL stores and also in analytic frameworks. This section explains the type of provenance required for NoSQL stores and approaches to capture provenance in two different contexts:

- Data stored in NoSQL store, analyzed using MapReduce Framework
- Data stored in NoSQL store, analyzed using SQL interface

11.4.1 Data Provenance Requirement in NoSQL Stores

To vouch for the credibility of data in the NoSQL stores, there is a need for three levels of provenance capture: tuple and schema provenances for data stored and data provenance for output of analytics done.

In NoSQL stores, the data on operations that cause the tuple to reach its current state can be categorized as *how provenance.* The *how provenance* answers the query on how the tuple attained its current value. Complex operations like join and aggregate are not present in NoSQL queries. So in the context of data stored in NoSQL store, *why provenance* is not relevant. However when analytics are done to produce

meaningful insights, the *why provenance* becomes critical to explain the result. When analytics are done on the data stores, the provenance of output constitutes the details of the input tuples that contributed to selection of the output and history of how each of the input tuples reached its current state.

NoSQL databases are designed with fault-resistant logs to enable replication of changes to ensure transparent scalability. The logs are fixed size tables (capped collection) that capture changes happening in the data store. The information from logs can be augmented and reused to deduce *how provenance* of data stored. *Why provenance* is captured for analytics done on the data in the NoSQL data stores. Two strategies of analytics are explored here.

1. Using inbuilt MapReduce
2. Using SQL interface

In the next section, MongoDB is used as an example to demonstrate the practical approach for capture of "how provenance" and "why provenance."

11.4.2 Capture of "How Provenance"

MongoDB supports basic CRUD (create, read, update, and delete) operations only. It provides an inbuilt MapReduce option to run complex analytic queries. The *how provenance* was tracked by setting up a tailable cursor in Python on the *operation log (oplog)* of MongoDB. *Oplog* is a special capped collection that keeps a rolling record of all operations that modify the data in the database. As provenance capture incurs storage overhead, the logger provides provision to select the tuples/documents that need to be tracked for provenance by using *resource expression.* Logger monitors the *Oplog* for any changes happening to the tuples for which provenance tracking is requested for. Whenever a log entry is made about tuple/collection that is being tracked, the cursor reads the data and deduces provenance details and records the provenance in an "append only" provenance collection. The information thus deduced from the log constitutes the *how provenance* and gives information on how a data item stored in the data store reached its current state. The following example demonstrates the use of provenance captured.

In the MongoDB database called "hospital," there exists a collection called "patients." To track the provenance for a particular patient, say "P123," resource expression is specified as <hospital/patients/P123>. The current state of the patient record is shown in Fig. 11.12. "How provenance" captured is shown in Fig. 11.13.

Both data and schema provenance are available on querying and are demonstrated by an example. Data provenance shows how the data reached its current state, i.e., the details of document creation and details of when each field value was added/updated. Schema provenance shows the addition and deletion of new fields in the document. For example, in the "hospital" database sample, a new field called "Allergy" has been added by user "Dr Jacob" on 29 April 2015 which was not there initially.

Fig. 11.12 Current state of patient record P123

```
{
"_id" : "P123",
"Name" : "John",
"Doctor" : "Dr.jacob",
"Disease" : "Asthma",
"Medication" :[ "Doxil4","Laxin" ],
"Allergy" : "Sneezing "
}
```

```
{
  "_id" : "hospital.patient.P123",
  "Provenance" : [
  {       "Op_Type" : "i"
          "Operation" : "{   'Name': 'John', 'Disease' : 'Asthma',
                             'Medication': ['Doxil4' , 'Aadrone' ],
                             'Doctor' : ' Dr. James '
                        }",
          "Time" : ISODate("2015-04-29T12:56:49Z"),
          "user"  : "Dr. James",
  },
  {       "Op_Type" : "u"
          "Operation" : "{'$set': {'Medication':['Laxin']},
                                  {'Doctor': 'Dr. Jacob'}}",
          "Time" : ISODate("2015-04-29T1:57:08Z"),
          "user" : "Dr. Jacob",
  },
  {       "Op_Type" : "u"
          "Operation" : " {'$set': {'Allergy': 'Sneezing'},
                                  {'Medication:[Doxil4, Laxin]'}}",
          "Time" : ISODate("2015-04-29T32:57:16Z"),
          "user" : "Dr. Jacob",
   }
              ]
}
```

Fig. 11.13 "How provenance" for P123

11.4.3 Capture of "Why Provenance"

"Why Provenance" is significant to explain results of analysis done on data stored in NoSQL stores. This section explores the capture of *why provenance* in two scenarios of analytics:

1. When MapReduce is used to conduct analytics on data stored in the NoSQL stores
2. When SQL interface is used to analyze the data in the NoSQL stores.

11.4.3.1 "Why Provenance" for Analytics Using MapReduce

Why provenance was captured for the MapReduce shipped with MongoDB. A wrapper-based approach similar to the approach used in the previous section was used to make MapReduce provenance aware. The provenance collected characterizes as *why provenance* as it gives reason/witness for why an output was obtained.

MongoDB MapReduce runs on one input collection at a time. The mapper reads the output of the document reader and emits them as key value pairs (k_i,v_i). Along with the input for the reducer, the mapper writes the provenance-related information (p_i,k_i) to a temporary file, *file1*, where p_i is a provenance id that uniquely identifies the document which consists of key k_i and value v_i. The reducer applies the reducer logic and processes (k_i,[v_1,v_2…v_n]) and generates the output key value pair (k_i,V). The document writer writes the key value pair (k_i,V) generated by the reducer to the output collection and temporary file,*file2*.Once the MapReduce task is complete, the provenance logger reads *file1* and *file2* and extracts the ids {p_1,p_2,…p_n} of the documents with key k_i from *file1* and appends the set {p_1,p_2,…p_n} to the pair (k_i,V) in the output collection specified with MapReduce. Thus the set {p_1,p_2,…p_n} is the provenance of the pair (k_i,V). From this, one can identify and trace back the documents inside the collection that contributed to that particular output value.

To illustrate *why provenance*, a simple example is considered. The collection of patient's medication bills at different times in hospital database is illustrated in Fig. 11.14.

The total bill for each patient can be calculated by running a MapReduce job. The output of the job is shown in Fig. 11.15.

Patient id	Bill Date	Prescribed Doctor	Items	Price(₹)
P127	2012-12-13 22:00:00	Dr.Jacob	{"Medicine":"Aidol7","qty":10,"price":2.5} {"Test":"MRI","qty":1,"price":1250}	1275
P133	2012-09-04 00:00:00	Dr.Ajeeb	{"Medicine":"Laxin","qty":5,"price":10} {"Medicine":"Mentol","qty":5,"price":2.5} {"Test":"Blood Test","qty":1,"price":50}	111.5
P123	2012-10-03 14:00:00	Dr.Ajeeb	{"Medicine":"Ameco7","qty":5,"price":20} {"Medicine":"Mentol","qty":5,"price":2.5} {"Test":"ECG","qty":1,"price":125}	234.5
P127	2012-12-13 22:00:00	Dr.Jacob	{"Medicine":"Aidol7","qty":10,"price":2.5}	25
P127	2012-12-13 22:00:00	Dr.Ajeeb	{"Medicine":"Demol Tab","qty":20,"price":2.5} {"Test":"ECG","qty":1,"price":250}	300
P123	2012-12-13 22:00:00	Dr.Jacob	{"Medicine":"Abeol","qty":10,"price":25}	250
P123	2012-10-04 00:00:00	Dr.Jacob	{"Medicine":"Laxin","qty":5,"price":2.5} {"Test":"ECG","qty":1,"price":125}	137.5
P133	2012-12-04 04:00:00	Dr.Ashly	{"Medicine":"Laxin","qty":5,"price":2.5} {"Medicine":"Aloxenol","qty":25,"price":25}	625
P333	2013-01-04 04:00:00	Dr.Hema	{"Medicine":"Laxin","qty":5,"price":2.5} {"Medicine":"Aloxenol","qty":25,"price":25} {"Test":"ECG","price":125}	150

Fig. 11.14 Snapshot of patient's medical bill collection

Key	Value
P127	1600
P333	150
P123	622
P133	736.5

Fig. 11.15 Snapshot of output MapReduce to consolidate total bill

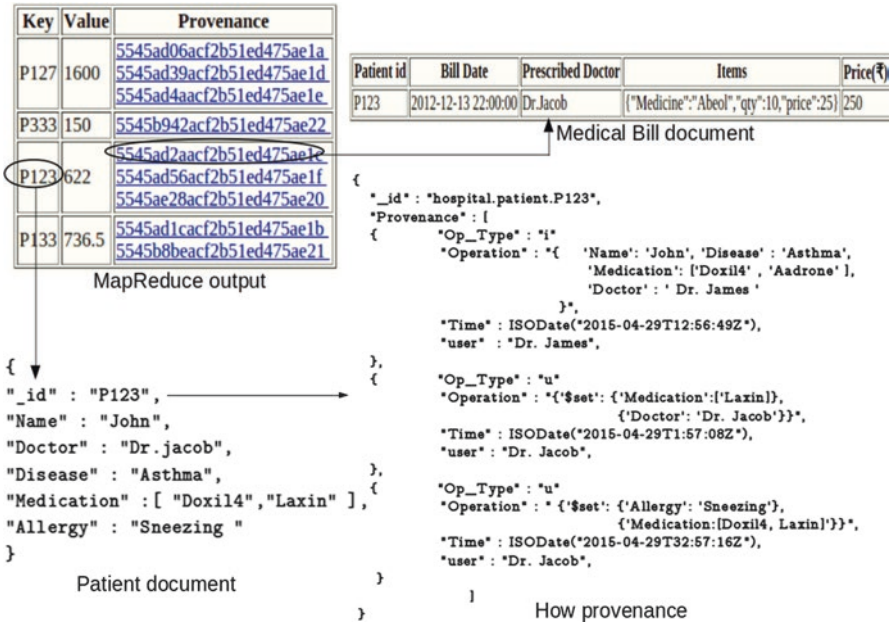

Fig. 11.16 Holistic explanation of a query by combining "why provenance" and "how provenance"

The output does not give any detail regarding the source documents that contributed to the result. Now if the same query was run with provenance collection, the *why provenance* and *how provenance* can be together viewed to have a holistic explanation of the result as shown in Fig. 11.16.

11.4.3.2 Provenance of NoSQL Stores Queried through SQL Interface

The SQL/MED, or Management of External Data, extension to the SQL standard defined by ISO/IEC 9075–9:2008 (originally defined for SQL:2003) [17] provides extensions to SQL to define foreign data wrappers (FDW) and data link types to allow SQL to manage external data. Popular commercial relational databases like

PostgreSQL and IBM DB2 adopted these standards so as to work with data stored in external data stores by providing provisions to define FDWs.

FDW defines external data views called "foreign tables" to access external data through foreign data wrappers. Thus, in this approach data always resides in the remote data store, and query manipulations are done on the "view" defined by the foreign table. Provenance of query results run through FDW is important for debugging result, in case of unexpected results.

A novel idea for provenance representation is used in provenance model called PERM (Provenance Extension of the Relational Model) [4], developed by IIT (Illinois Institute of Technology) database group. The provenance model defined by PERM [4] attaches provenance information to query results by extending the original query result with the details of tuples that contributed toward the query result. PERM displays provenance by means of query rewrite mechanism which transforms a normal query *Q* into provenance query *Q+* that computes provenance of *Q*. PERM module rewrites the query so as to include provenance specific details. This rewritten query is a relational query and hence gets the advantage of all inbuilt optimizations.

When analytics are done in NoSQL stores using SQL interface, the results are usually presented as views. So PERM model was extended to capture provenance of data accessed through foreign data wrappers. The idea is demonstrated by building a proof of concept to analyze data in MongoDB by building a MongoDB FDW and accessed through modified PERM interface. An extension of PERM model was built in stable PostgreSQL version 9.3 and tested by writing a FDW for MongoDB to capture "why provenance" of SQL query run on MongoDB through PERM. The result of simple query versus provenance query on a *SQL Select* statement is shown in Fig. 11.17.

SELECT BRAND,INSTOCK FROM SELLER_1 JOIN COUNTS ON ID=ITEMID WHERE INSTOCK<4;

brand	instock
lenovo	2
asus	1

SELECT PROVENANCE BRAND,INSTOCK FROMO SELLER_1 JOIN COUNTS ON ID=ITEMID WHERE INSTOCK<4;

Brand	Instock	prov_public_seller_1_id	prov_public_seller_1_brand	prov_public_seller_1_price	prov_public_counts_itemid	prov_public_counts_instock
Lenovo	2	113	Lenovo	25000	113	2
asus	1	117	asus	5000	117	1

Fig. 11.17 Normal select query result vs provenance query result

11.5 Summary and Conclusion

The power of Big Data generated through IoT can be leveraged only if the data captured can be analyzed and reliable results can be obtained. In this chapter, various schemes for capturing provenance of Big Data analytic tools like MapReduce and NoSQL data stores are discussed. It was demonstrated that provenance captured in MapReduce framework was not only useful for debugging but also for improving certain classes of job reruns and detecting anomalies in the framework. Improving performance of workflows using provenance collected as part of the workflows is a significant use of provenance, as it can save computational power and time for execution. Extending the work to efficiently perform selective refresh on MapReduce workflow is an interesting problem. The proposed approach of capturing transformational provenance using logs and the use of transformational provenance in identifying anomalies in job execution are promising and can be further improved by extending the collection of logs used in analysis. The "how provenance" and "why provenance" captured help in providing explanation for data stored and analytics done on the data stored in NoSQL stores, respectively. "How provenance" and "Why provenance" together provide a holistic picture to explain the results of decisions based on analytics on data stored in NoSQL stores.

This chapter restricted the focus to analysis of Big Data. In the context of IoT, the challenges in capturing provenance of data produced by sensors are very critical, and the area opens up many research problems which need serious research attention. Refer to [18–22].

References

1. Simmhan YL, Pale B, Gannon D (2005) A survey of data provenance in e-science. SIGMOD Rec 34(3):31–36. https://doi.org/10.1145/1084805.1084812
2. Tan W (2004) Research problems in data provenance. IEEE Data Eng Bull 27(4):45–52
3. Agrawal P, Benjelloun O, Sarma A D, Hayworth C, Nabar S, Sugihara T, Widom J (2006) Trio: a system for data, uncertainty, and lineage. In: Proceedings of the 32nd international conference on very large data bases (VLDB '06), VLDB Endowment, pp 1151–1154
4. Glavic B, Alonso G (2009) The PERM provenance management system in action. In: Proceedings of the 2009 ACM SIGMOD International conference on management of data (SIGMOD '09), ACM, New York, USA, pp 1055–1058. https://doi.org/10.1145/1559845.1559980
5. Muniswamy-Reddy K, Holland D, Braun U, Seltzer M (2006) Provenance-aware storage systems. In: ATEC '06 Proceedings of the annual conference on USENIX '06 annual technical conference, Boston, 2006, pp 4–4
6. Tariq D, Ali M, Gehani A (2012) Towards automated collection of application-level data provenance. In: Proceedings of the 4th USENIX conference on theory and practice of provenance (2012), USENIX Association, Berkeley, CA, USA, June 14–5, 2012, pp 16–16
7. Muniswamy-Reddy K K, Macko P, Seltzer M (2010) Provenance for the cloud, FAST, 15–14
8. Sletzer MI, Macko P, Chiarini MA (2011) Collecting provenance via the Xen hypervisor, TaPP
9. Ikeda R, Park H, Widom J (2011) Provenance for generalized map and reduce workflows, CIDR, 273–283

10. Akoush S, Sohan R, Hopper A (2013) HadoopProv: towards provenance as a first class citizen in MapReduce. In: Proceeding TaPP '13 Proceedings of the 5th USENIX workshop on the theory and practice of provenance, 2013, Article No. 11
11. Amsterdamer Y, Davidson SB, Deutch D, Milo T, Stoyanovich J, Tannen V (2011) Putting lipstick on pig: enabling database-style workflow provenance. In: Proceedings VLDB Endow. 5, 4 (December 2011), 346–357. http://dx.doi.org/10.14778/2095686.2095693
12. Middleware, Wikipedia – the free Encyclopedia. https://en.wikipedia.org/wiki/Middleware. Accessed 6 Mar 2017
13. Belhajjme K, Missier P, Goble C, Cannataro M (2009) Data provenance in scientific workflows, medical information science reference, 2009
14. Apache, Apache Weblog. https://httpd.apache.org/docs/1.3/logs.html. Accessed Nov 2016
15. Bhatotia P, Wieder A et al (2011) Incoop: MapReduce for incremental computation. In: Proceedings of the 2nd ACM symposium on cloud computing (SOCC '11). ACM, New York, NY, USA, Article 7, p 14. https://doi.org/10.1145/2038916.2038923
16. Schad J, Quianeé-Ruiz JA, Dittrich J (2013) Elephant, do not forget everything! Efficient processing of growing datasets. IEEE Sixth international conference on cloud computing, Santa Clara, CA, 2013, pp 252–259. doi:https://doi.org/10.1109/CLOUD.2013.67
17. SQL/MED, Wikipedia – the free encyclopedia. https://en.wikipedia.org/wiki/SQL/MED. Accessed 6 Mar 2017
18. Cuzzocrea A (2014) Privacy and security of big data: current challenges and future research perspectives. In: Proceedings of ACM PSBD 2014, pp 45–47
19. Cuzzocrea A, Bertino E (2011) Privacy preserving OLAP over distributed XML data: a theoretically-sound secure-multiparty-computation approach. J Comput Syst Sci 77(6):965–987
20. Cuzzocrea A, Russo V (2009) Privacy preserving OLAP and OLAP security. Encyclopedia of data warehousing and mining, pp 1575–1581
21. Cuzzocrea A (2015) Provenance research issues and challenges in the big data era. In: Proceedings of IEEE COMPSAC workshops 2015, pp 684–686
22. Cuzzocrea A, Fortino G, Rana OA (2013) Managing data and processes in cloud-enabled large-scale sensor networks: state-of-the-art and future research directions. In: Proceedings of IEEE CCGRID 2013, pp 583–588