ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/303685763

MrLazy: Lazy Runtime Label Propagation for MapReduce

Conference Paper · June 2014

CITATIONS
7

READS
12

4 authors, including:

Sherif Akoush
University of Cambridge
9 PUBLICATIONS **297** CITATIONS

SEE PROFILE

Lucian Carata
University of Cambridge
14 PUBLICATIONS **52** CITATIONS

SEE PROFILE

All content following this page was uploaded by Sherif Akoush on 31 May 2016.

The user has requested enhancement of the downloaded file.

MrLazy: Lazy Runtime Label Propagation for MapReduce

Sherif Akoush, Lucian Carata, Ripduman Sohan and Andy Hopper
Computer Laboratory, University of Cambridge
`firstname.lastname`@cl.cam.ac.uk

Abstract

Organisations are starting to publish datasets containing potentially sensitive information in the Cloud; hence it is important there is a clear audit trail to show that involved parties are respecting data sharing laws and policies.

Information Flow Control (IFC) has been proposed as a solution. However, fine-grained IFC has various deployment challenges and runtime overhead issues that have limited wide adoptation so far.

In this paper we present MrLazy, a system that practically addresses some of these issues for MapReduce. Within one trust domain, we relax the need of *continuously* checking policies. We instead rely on lineage (information about the origin of a piece of data) as a mechanism to retrospectively apply policies on-demand. We show that MrLazy imposes manageable temporal and spatial overheads while enabling fine-grained data regulation.

1 Introduction

Currently, governments and regulatory agencies are introducing data protection and compliance laws that cloud computing providers have to comply with [1, 2, 3]. This usually entails (*i*) *tracking* and (*ii*) *enforcing* data propagation according to certain policies; for example, ensuring that data is not shared with unauthorised third parties.

Information Flow Control (IFC) has been advocated as a good mechanism for regulating data usage in the Cloud [4]. IFC works by attaching labels (i.e., privacy or security metadata) to input data and *continuously* propagating these labels through computations to control where data flows.

Although IFC can operate at different granularities, field-level granularity is required to effectively track data transformations. However, practitioners generally believe that fine-grained IFC is not feasible as it incurs deployment challenges and prohibitive runtime overheads [5].

We observe, nonetheless, that tracking data flows (i.e., audit) is orthogonal to the use of IFC to limit data propagation (i.e., control). In particular, we advocate that by delaying the enforcement of data dissemination policies to the point where data crosses a trust boundary, we can significantly reduce the overheads of maintaining fine-grained IFC in modern Big Data frameworks.

Within a given trust domain a company can do computations on sensitive datasets without the need to actively check complex policies at runtime. Regulations have to be enforced only when data is about to leave a trust domain (e.g., publishing results). It is, therefore, enough to have an audit mechanism to ensure we can provide enforcements when needed.

Based on this observation, we propose the use of lineage [6] to check, at a later stage, if a given output record complies with rules and regulations. Lineage associates an output record with all input records contributing to its creation. For example if we are computing the average of 5 inputs, lineage represents a link between the output value and these specific 5 inputs.

We introduce the *Lazy* label propagation mechanism: we do not propagate any labels associated with input data (e.g., security metadata) during the actual computations. We only track lineage during computation, which we use to construct output labels and enforce policies retrospectively when data is about to leave the trust domain.

In this paper we present MrLazy, a prototype system that implements lazy label propagation for Hadoop MapReduce. We choose MapReduce because it is a popular computation framework in the Cloud. Our preliminary evaluations show that MrLazy incurs 19% additional runtime overhead on the actual computation while typical policy enforcement queries run in 15–20% of the actual computation time. Additionally, MrLazy has storage requirements of up to 50% of the output dataset size.

We start by describing a motivating use case that we use to illustrate the benefits of MrLazy (Section 2). Based on this use case we discuss the design of MrLazy (Section 3) and show overhead measurements from our prototype implementation (Section 4). We then present related work (Section 5) and conclude (Section 6).

2 Motivating Use Case

In our simple example we assume that a large retailer (say Walmart) performs data analytics on customer order logs from its stores. Walmart's data scientists that work on this project have full-access to this dataset and they use MapReduce to execute their jobs. This analysis requires a number of computations. The resulting data are not expected to leak any sensitive data as all outputs are solely produced for the consumption of Walmart personnel (e.g., senior management). Therefore there is no need to control data flows at this point.

Let us further assume that Walmart wants to provide some of these results to a marketing agency. As these datasets are produced from potentially sensitive customer individual orders, Walmart is concerned whether this action might breach data sharing policies. Some of Wal-

mart's customers have specifically requested that their data is not exposed to third parties. Additionally, Walmart wants to enforce some policies to protect their competitive edge; for example, recent orders are not shared. Consequently, Walmart has to sanitise results before they can be transferred to the marketing agency.

Walmart might naively decide to filter out sensitive records from the source dataset and *rerun* the needed computations. However, we argue that this is not an efficient solution. Walmart already has the results computed by its data scientists. All it needs to do is to remove any output records that are produced from sensitive input records.

2.1 Features

Based on the example above, we identify the following properties in a practical IFC deployment: (*i*) The system should incur acceptable runtime overhead because it has to be always enabled. (*ii*) We can omit checking complex policies within a trust domain. It is just enough to have a mechanism to enforce these policies on data that is crossing borders (e.g., publishing results). (*iii*) The framework should be able to enforce complex policies without requiring the data scientist to manually check rules and regulations. (*iv*) Additionally, IFC should be enforcable at field-level granularity because in a single record some fields are sensitive but others are not.

The system should also support multiple labels for the same record (e.g., metadata representing terms for different output views). Hence, it would be more practical if the system deals seamlessly with these labels without the need to change the underlying structure or data model. Moreover, labels can be missing (or even wrong) when the computation is executed. The system should be able to retrospectively deal with these situations efficiently.

3 MrLazy System

MapReduce is a paradigm [7] for large-scale data processing and analysis. This simple but effective model consists of two higher-order functions: `map` and `reduce`. The user-provided Map function reads, filters and transforms data from an input file, outputting a set of intermediate records. These intermediate records are typically split according to their key into disjoint buckets. Further on, the user-provided Reduce function processes and combines all intermediate records associated with the same key into new records which are written to an output file. Programs developed according to this model are considered "embarrassingly parallel" as there are no inter-key data dependencies. Moreover MapReduce is tolerant to failures as erroneous and incomplete tasks can be restarted independently of each others. Usually computations are expressed as a series of MapReduce jobs constituting a workflow.

MrLazy enables lazy label propagation in Hadoop MapReduce by maintaining lineage that is captured during job execution. In other words, labels are not propagated while the job is running; hence, we do not incur any overheads associated with processing and shuffling this additional data. As MrLazy knows the origin of any piece of data, it can effectively generate output labels post-processing (Figure 1).

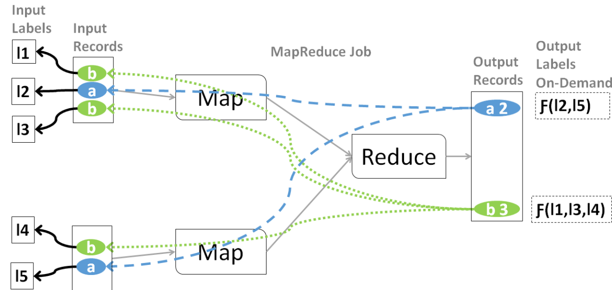

Figure 1: Lazy Output Label Generation in MrLazy: MrLazy uses lineage (dashed-lines) to link output values to relevant input records and their labels (metadata). Output labels are defined as a function of the corresponding input labels. Input labels are not propagated through the MapReduce job.

MrLazy design is composed of four subsystems: Lineage Tracking, Garbage Collection (Lineage Reconstruction), Static IFC, and Label Generation and Policy Enforcement.

3.1 Lineage Tracking

MrLazy tracks lineage of records as they are being processed in MapReduce. Lineage is defined as the causal relationship between input and output records [6]. In MapReduce terms, MrLazy links an output key/value record back to the input key/value record(s) affecting its creation [8, 9].

As a MapReduce job has a Map phase emitting intermediate records that are then shuffled to a Reduce phase, we can leverage this behavior to treat Map and Reduce tasks independently. For example, at the end of a Map task, we materialise links between intermediate (hash) keys and input positions. Likewise at the end of a Reduce task, we record links between intermediate (hash) keys and output positions.

A major design decision is *not* to augment key/value records with lineage as this mechanism will add temporal and spatial overheads to the MapReduce framework (especially during the Shuffle phase). Propagating the full lineage during the actual MapReduce job has previously been shown to incur prohibitive overheads up to 75% on job runtime [10].

3.2 Garbage Collection (Lineage Reconstruction)

As we discussed previously, MrLazy captures lineage for Map and Reduce tasks independently to lower the over-

head on the actual job. Although the resulting raw lineage files can be used to answer any query, a job is required to (*i*) construct *direct links* between input and output records and (*ii*) *discard* intermediate keys.

This Garbage Collection (Lineage Reconstruction) job takes raw lineage files as inputs, joins them by intermediate keys, and then outputs the final lineage graph. It also carries out optimisations necessary to speed up typical queries in MrLazy. The final lineage file has a much smaller size compared to the raw input lineage files as we discard intermediate keys and remove any redundancies in the data. This process works iteratively for multi-stage jobs.

To optimise the join procedure of these two sets it is important to record, at the Reduce stage, from which Map tasks a given record has been emitted. This limits the search to only relevant Map side lineage files. Moreover, we leverage the fact that MapReduce already sorts intermediate keys, which further optimises the join procedure.

Garbage Collection (Lineage Reconstruction) is not strictly required in the design but we highlight it is a one-time cost that provides a good optimisation for reducing storage overhead and query time. Moreover it can be scheduled when there is slack in the cluster and not necessarily upon main job completion.

3.3 Static IFC

Lineage Tracking as described in the previous section will only provide record-level causality. However, in many cases we require field-level visibility.

MrLazy design has, therefore, a Static IFC (Taint Analysis) [4] component to address this requirement. It works by analysing the job binaries (jar files). As Hadoop MapReduce is Java-based, analysis at the byte code is practical [11].

The output of Static IFC is a conservative list of each input field (to the Map and Reduce tasks) that *might* result in an output field being emitted. Having this information will enable MrLazy to verify whether a given job has used a sensitive field to produce an output.

We note here a subtle behavior that we can model correctly with Static IFC. A sensitive field might be used as the intermediate key in the MapReduce job; but if its value is not ultimately emitted in the final output, no sensitive data has been inadvertently leaked.

3.4 Label Generation and Policy Enforcement

Queries in MrLazy involve generating labels for outputs records based on labels associated with corresponding input records (Figure 1). The generated labels can then be used to retrospectively control what the system should do with a specific output according to policies. For example if the label of an input record is evaluated as sensitive, the output record can also be labeled as sensitive.

Once an output record is deemed sensitive we can easily omit it from the output dataset. This process is cheap compared to the naive approach of rerunning the computation with sensitive data filtering out from the input dataset. Intuitively MrLazy can deal with arbitrary labels and policies.

Output label generation is efficient using lineage. We just need to read labels of input records that contributed to the creation of a specific output record and apply a function. For multi-stage jobs, once we build lineage (iteratively) to the source input we can effectively apply the same process to produce output labels.

By combining static IFC with lazy runtime label propagation we can have a system that enforces dynamic complex policies. For example if "only recent dates (during the past week) are sensitive", MrLazy can practically filter out records that are linked to these sensitive dates.

3.5 Other Technical Challenges

To design an end-to-end IFC system, there are some additional challenges that we need to address. So far we assumed that there are labels associated with the input data. However, translating high-level rules and policies to fine-grained labels is not simple. Practically, the process of labeling should be applied automatically; for example, a tool that scans files and tags labels on matched patterns. Labels in the general form can be defined as a tuple (filename, offset, length, metadata) [12]. We can also have labels that are formulated as a function of the input data (e.g., dates before 2010 are sensitive) marked per file.

MrLazy tracks lineage for pure Map and Reduce tasks. This is not always the case; for example, a Map task can save some state during different `map` invocations and then emits records during task `cleanup`. We are exploring the possibility of intercepting these side-effects and augmenting the job binary to disclose hints to the framework so that lineage is precise.

We assume that the framework is trusted. However, a malicious developer might leak data in the output. Although MrLazy knows which individual input values have been used to compute a specific output value, it is not always straight forward to generate the output label when part of the input is sensitive (e.g., computing averages or emitting a special value for a specific input). In these cases, MrLazy will conservatively mark the resulting output as sensitive. Alternatively, we can employ differential privacy techniques to prevent against any indirect data leakage [13].

3.6 Implementation

In our prototype implementation of MrLazy we focus on evaluating Lineage Tracking, Garbage Collection (Lineage Reconstruction), and Label Generation as these are the distinguishing factors in the design. We have augmented the core of Hadoop MapReduce v2 to record lineage with low

overheads. As MrLazy required changes to the framework, jobs run seamlessly without modifications. For Garbage Collection (Lineage Reconstruction), we developed a separate MapReduce job that runs on the cluster to construct the final lineage from task-level lineage files. We implemented Label Generation as a set of jobs that read the lineage output and evaluate simple labels based on input records.

4 Evaluation

4.1 Workload

In this section we discuss preliminary results from MrLazy. We choose a join workload from the BigDataBench benchmark suite [14], which comes with a data generation tool that we used to produce two synthetic tables `ORDER` and `ORDER_ITEM` (Figure 2) for our experiments. The total size of these two tables is 120 GB (3 billions records) and they are stored in text format on HDFS.

The cluster that we use for our evaluation consists of 7 machines having each 8x2.2 GHz CPUs, 24 GB RAM and a 4 TB local disk. The cluster is virtualised using XenServer 6.3 and connected with a 1 Gbps ToR switch.

```
ORDER: ORDER_ID int, BUYER_ID int, CREATE_DT string
ORDER_ITEM: ITEM_ID int, ORDER_ID int, GOODS_ID int,
GOODS_NUMBER int, GOODS_PRICE double, GOODS_AMOUNT
double
```

Figure 2: `ORDER` and `ORDER_ITEM` Tables.

```
SELECT BUYER_ID, GOODS_PRICE, GOODS_AMOUNT
FROM ORDER, ORDER_ITEM
JOIN ORDER.ORDER_ID = ORDER_ITEM.ORDER_ID
WHERE GOODS_PRICE > 800
```

Figure 3: Evaluation Workload.

We run a single MapReduce job that joins the two tables on `ORDER_ID` while selecting only orders more than 800 price units (Figure 3). The output of this query is approximately 10 GB and 300 millions records. The job takes on average 27 minutes running on unmodified Hadoop MapReduce.

4.2 Overheads

Runtime Overhead In this section, we discuss the overheads of MrLazy. Numbers present averages of 5 runs. We first measure the overhead of tracking lineage in MrLazy. By running the same job on our modified version of Hadoop, the job takes on average 32 minutes which constitutes approximately 19% increase in runtime.

Garbage Collection (Lineage Reconstruction) Overhead As we described before, MrLazy has a Garbage Collection (Lineage Reconstruction) job that joins Map and Reduce side raw lineage files on the intermediate keys to produce the final input to output lineage. This jobs takes on average 5 minutes (18% of the actual job runtime). We highlight that this job can be scheduled asynchronously and is not required right after the main job.

Space Overhead While tasks are running MrLazy captures their lineage files. These files are saved on HDFS. For our example job these raw lineage files have a combined size of 25 GB.

After Garbage Collection (Lineage Reconstruction), size is significantly reduced to 5 GB as we discard intermediate keys after the join procedure. This overhead is 50% of the job output size (10 GB) and 5% of the job input size (120 GB).

Label Generation and Policy Enforcement (Query) Overhead We now present two example queries that might be asked by MrLazy users. The first query (Query 1) illustrates how the system deals with users that decide to opt-out of data sharing. It is still legal to process their information privately. We only need to exclude records of users that opted-out when sharing data with third parties.

We assume that 5% of users decided not to share their data. We then run Query 1 to identify output records linked to input records belonging to these users. Answering Query 1 takes on average 4.5 minutes which is 16% of the main job runtime.

The alternative solution to answer this query is to rerun the computation but filtering out sensitive users from the input. To rerun the job with this extra filter, the recomputation job takes approximately 26 minutes, almost equal to the actual job runtime. Clearly answering queries in MrLazy is considerably faster compared to the naive case.

The second query (Query 2) that we test is: transactions made in the last two years are considered sensitive. Notice that results of Query 2 depend on when it is executed, which means that even if we propagate labels during the actual job results will be outdated. Unless we use MrLazy, the only other option is to recompute the job on-query.

In MrLazy Query 2 incurs on average 5 minutes or 18.5% of the time required to run the actual job. This is still reasonable considering that the alternative (recomputation-based) approach is expensive.

4.3 Discussion

Our preliminary results (summarised in Figure 4) show that overheads are non-prohibitive considering that we maintain record-level lineage at runtime. MrLazy is designed as a general tool that can scale with the number of labels. It assumes that we have field-level labels for all input records. We might even have multiple labels for the same records (e.g., different country regulations and multiple output formats). Labels can also provide an indication of data quality. Moreover, labels can change dynamically or might actually be missing at computation time. MrLazy

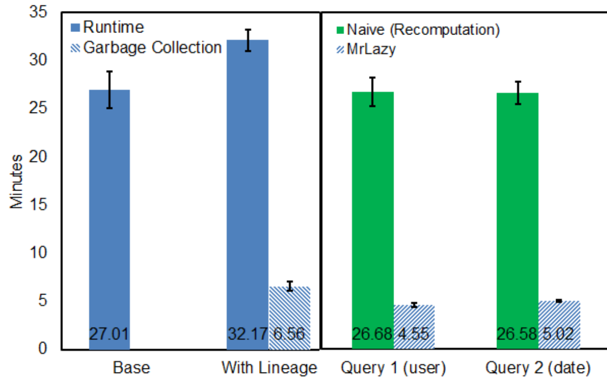

Figure 4: Runtime and Policy Enforcement Overheads in MrLazy Compared With Recomputation-Based Approach (Average and Standard Deviation of 5 Runs).

provides a mechanism to retrospectively deal with these situations.

However, we acknowledge the fact that these overheads are not negligible and there is still room for improvement. There is a direct correlation between runtime overhead and the size of data stored. We are investigating techniques to capture less data for lineage without loosing granularity.

We also note that MrLazy might not work out of the box for special types of jobs that depend on the number of records emitted. For example, top-k queries might produce less than k items when sensitive records are later omitted from the output. MrLazy can at least inform the user on how many records were removed. Alternatively, we can detect these jobs and revert back to standard IFC.

On the other hand, lineage as an end product is very useful. Fine-grained lineage can be used in audit systems to provide a detailed log of data usage and transformation. Moreover, lineage can enable interesting system optimisation (e.g., recovery from failures [15]).

The total size of lineage might be a concern for some providers. MrLazy allows users to delete lineage for files that are not expected to be used (i.e., cold data). In doing so they can bound the storage requirement of the system.

5 Related Work

There is no system, to our knowledge, that performs record-level audit and flow control specifically for MapReduce. The closest work is RAMP [10] which augments values with lineage. Alternatively, Newt records fine-grained lineage in data intensive workflows by instrumenting the framework [9]. Both approaches can be extended for audit using captured lineage but they report larger overheads. Precisely, RAMP and Newt incur up to 75% and 36% runtime overhead respectively while the prototype implementation of MrLazy achieves less than 19%.

Airavat [13] focuses more on security and privacy for MapReduce. It employs differential privacy to circumvent against indirect leakage of data in the output. Differential privacy complements IFC; However, it compromises the output quality.

On the other hand Sedic [12] takes a different approach to privacy. It automatically partitions data based on sensitivity and arranges the computation on hybrid clouds in such a way that sensitive data is only processed on private clouds. We argue that it is not always feasible to restructure data accordingly.

SilverLine enforces IFC by augmenting untrusted MapReduce job binaries with information flow tracking code [16] as they arrive at the Cloud. However, this system focuses on file-level policy enforcements for workflows.

6 Conclusion and Future Work

In this paper we presented MrLazy; a system that tracks information flow using record-level lineage in Hadoop MapReduce. Lineage provides an audit mechanism that is used to control data sharing with third parties. Enforcement of policies is not expected to be frequent; we delay this process until an output is about to be exposed.

In MrLazy labels are not propagated. Output labels are generated from the relevant input labels and enforcements can be applied retrospectively.

In our prototype implementation we showed that MrLazy has non-prohibitive runtime overhead in the general case. We are actively working on bringing this overhead to less than 10%. We are also testing with other workloads to obtain bounds on overheads.

We are planning for large scale experiments—including extensive comparisons with different IFC mechanisms—to confirm these preliminary results. We are developing an end-to-end system that includes multi-stage jobs, static IFC, labeling, and policy enforcements.

For the longer term, we will explore lazy IFC in other cloud computing frameworks. Specifically, we will be looking at in-memory processing and key/value stores.

7 Acknowledgment

We appreciate the feedback of George Coulouris, Nikilesh Balakrishnan, Thomas Bytheway, James Snee and Alastair Beresford on the paper. We also thank the anonymous reviewers for their comments.

References

[1] " European Commission: Proposal for a General Data Protection Regulation." http://ec.europa.eu/justice/data-protection.

[2] CNIL, "Summary of responses to the public consultation on Cloud computing," in *2012*.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Commun. ACM*, Apr. 2010.

[4] J. Bacon, D. Eyers, T. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch, "Information Flow Control for Secure Cloud Computing," *Network and Service Management, IEEE Transactions on*, 2014.

[5] B. Livshits, "Dynamic Taint Tracking in Managed Runtimes," Tech. Rep. MSR-TR-2012-114, Microsoft Research, 2012.

[6] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper, "A Primer on Provenance," *Commun. ACM*, May 2014.

[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI'04*.

[8] S. Akoush, R. Sohan, and A. Hopper, "HadoopProv: Towards Provenance As a First Class Citizen in MapReduce," in *TaPP'13*.

[9] D. Logothetis, S. De, and K. Yocum, "Scalable Lineage Capture for Debugging DISC Analytics," in *SOCC'13*.

[10] H. Park, R. Ikeda, and J. Widom, "RAMP: a system for capturing and tracing provenance in MapReduce workflows," *Proc. VLDB Endow.*, 2011.

[11] J. Graf, M. Hecker, and M. Mohr, "Using JOANA for Information Flow Control in Java Programs A Practical Guide," in *Karlsruhe Reports in Informatics'12*.

[12] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan, "Sedic: Privacy-aware Data Intensive Computing on Hybrid Clouds," in *CCS'11*.

[13] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and Privacy for MapReduce," in *NSDI'10*.

[14] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: a Big Data Benchmark Suite from Internet Services," in *HPCA'14*.

[15] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica, "Tachyon: Memory Throughput I/O for Cluster Computing Frameworks," in *LADIS'13*.

[16] S. M. Khan, K. W. Hamlen, and M. Kantarcioglu, "Silver Lining: Enforcing Secure Information Flow at the Cloud Edge," in *IC2E'14*.

View publication stats