HadoopProv: Towards Provenance As A First Class Citizen In MapReduce

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

Abstract

We introduce HadoopProv, a modified version of Hadoop that implements provenance capture and analysis in MapReduce jobs. It is designed to minimise provenance capture overheads by (*i*) treating provenance tracking in Map and Reduce phases separately, and (*ii*) deferring construction of the provenance graph to the query stage. Provenance graphs are later joined on matching intermediate keys of the Map and Reduce provenance files. In our prototype implementation, HadoopProv has an overhead below 10% on typical job runtime (<7% and <30% average temporal increase on Map and Reduce tasks respectively). Additionally, we demonstrate that provenance queries are serviceable in $\mathrm{O}(k \log n)$, where n is the number of records per Map task and k is the set of Map tasks in which the key appears.

1 Introduction

There is currently widespread interest in the MapReduce paradigm 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 a hash function into disjoint buckets. Further on, the user-provided Reduce function processes and combines all intermediate records associated with the same hash value 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.

Hadoop [1] is a popular open-source implementation of MapReduce. There already exist a few projects that extend it to support provenance capture and tracing [2, 3]. However these systems are of limited use in production environments due to their large runtime overhead (between 75%–150% on typical workloads [2, 3]).

These previous systems have been explicitly designed to avoid modification of the MapReduce framework. They rely on a wrapper-based approach augmenting key-value pairs with provenance information. Furthermore, their design precomputes the whole provenance graph for all output records while the job executes. This results in inefficient implementations with substantial runtime overhead.

We show that these challenges are mitigated by considering provenance as an intrinsic feature of the MapReduce framework. Towards this goal this paper presents the preliminary design, implementation and evaluation of HadoopProv: a modified version of Hadoop that captures provenance information without incurring the large runtime overhead associated with previous systems.

2 System Design

HadoopProv is designed to capture *data provenance* at the record level. In effect, we add a feature to Hadoop which allows users to track lineage between input key-value pairs involved in the creation of output key-value pairs. Provision of this fine grained tracking enables a host of extended functionalities such as incremental process and log analysis (described in Section 4) difficult to realise on current systems.

2.1 Architecture

HadoopProv aims to minimise the temporal overhead of capturing provenance information in MapReduce jobs. It achieves this aim by (*i*) recording provenance information in Map and Reduce phases separately and (*ii*) deferring the construction of entire provenance graphs to a later stage. While this design choice means that there might be a delay in provenance queries, we believe that keeping the temporal overhead low will encourage the use of HadoopProv as a core component in the Hadoop ecosystem. Moreover, provenance queries are infrequent and focus on a small subset of output data at a time. Therefore, incurring the large overhead of precomputing the complete provenance graph for all records at runtime is not justifiable.

We formally define data provenance of MapReduce jobs as the causal relationship between input and output records. In other words, HadoopProv provides a mechanism that links an output record back to the input record(s) affecting its creation [2]. As a MapReduce job has a Map phase emitting intermediate records that are then shuffled to a Reduce phase, we can treat each phase as an independent sub-job from the provenance capture point of view. For example, at the end of a Map task, there is a causal relationship connecting each intermediate record back to the input records involved in its creation (intermediate→input). Likewise at the end of a Reduce task, there is a causal relationship connecting each output record to the intermediate records involved in its creation (output→intermediate).

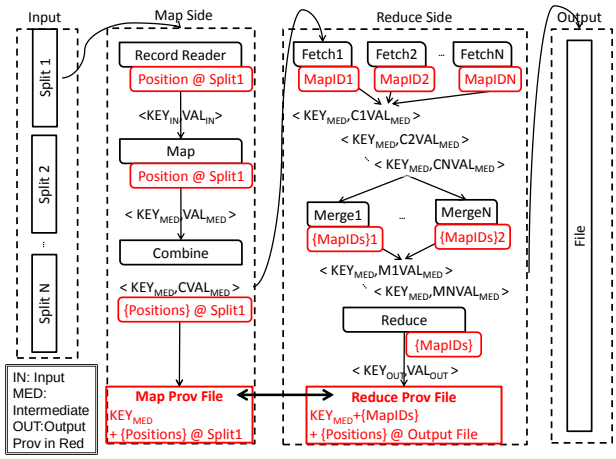

Figure 1: Provenance Capture in HadoopProv

Upon query, provenance graphs may be simply constructed by joining on intermediate key of the Map and Reduce provenance files (output→intermediate→input).

However, reliance on this basic relationship between input and output records is inefficient. Arbitrary intermediate records can be generated by *any* Map task which would lead to significant provenance construction overhead.

Alternatively, HadoopProv tracks the Map task associated with every intermediate record as it propagates through the Reducers. In this way, HadoopProv knows the Map tasks that emitted the intermediate records affecting the creation of a given output. The provenance graph of this output record is thus constructed by only considering the provenance files of these involved Map tasks.

By design, HadoopProv does not shuffle provenance information. This property, while increasing provenance query time, has the important advantage of imposing minimal I/O overhead during the Shuffle phase.

HadoopProv easily extends the same technique to link provenance information across an entire workflow. For example if Job A produces outputs that are then fed to Job B. HadoopProv joins the two provenance sets based on matching keys: outputB→(inputB=outputA)→inputA.

2.2 Implementation

Figure 1 illustrates provenance capture in HadoopProv. In summary, we annotate each emitted record with a list of objects pointing to its origin records. Provenance information is stored in a separate *provenance file* (composed of key-based *provenance objects*) that is colocated with data created by each task. In the case of a Map task, each provenance object stores the *locations* of input records associated with the emitted intermediate records. For a Reduce task, each provenance object represents a set of Map task IDs corresponding to intermediate keys coupled with locations of affected output records. Provenance queries are thus serviced by joining the provenance files on matching intermediate keys to construct the association between input and output records.

2.2.1 Map

Every Map task reads records from an input file split and applies a user-defined Map function to emit intermediate key-value pairs. HadoopProv records a mapping between each intermediate key to a list of positions at the corresponding input split that contain the input key-value pairs affecting its creation. Additionally, an index (sorted on intermediate keys and pointing to the actual provenance objects) is created to speed up provenance graph construction.

2.2.2 Combine

A Combine phase usually follows each Map task to aggregate intermediate key-value pairs and minimise the amount of data shuffled. Similarly HadoopProv combines positions at the input split for records having the same key.

2.2.3 Fetch

No record level provenance information is transmitted during the Fetch phase. However, we store in the header file of each Fetcher additional metadata describing the Map task that produced this data segment. In this manner Reducers can associate records to their corresponding Map tasks.

2.2.4 Merge

As each Reduce task fetches data from different Mappers, the MapReduce framework has to merge sort these files. This process is multi-stage depending on the amount of fetched data and physical memory available on the compute node. Accordingly HadoopProv tracks the Map task information of intermediate records as they are processed.

2.2.5 Reduce

When the user-defined Reduce function is applied to the intermediate key-value pairs, HadoopProv knows two facts about each record: (*i*) the Map tasks of the intermediate records and (*ii*) the locations (at the output file) of the final output records. This information is saved to the provenance file on disk.

2.2.6 Provenance Reconstruction

As outlined previously, the entire provenance graph of an output set of key-value pairs can be constructed by joining the Map and Reduce provenance files on all matching intermediate keys. HadoopProv leverages that the MapReduce framework already sorts intermediate records on the

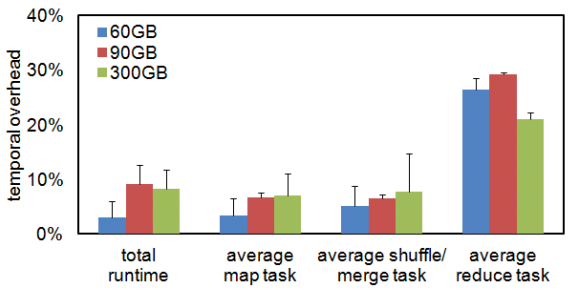

Figure 2: Temporal Overhead (Total per Job and Average per Task. Line Bars Represent Standard Deviation)

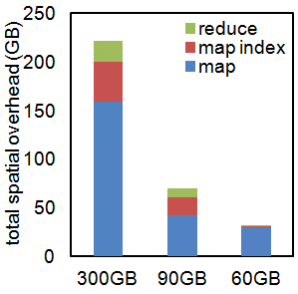

Figure 3: Spacial Overhead (Total Size per Job)

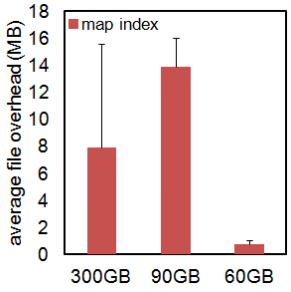

Figure 4: Lookup Overhead ($\propto$ Average Size per Task)

key. Provenance information is thus sorted which enables $O(\log n)$ single-key lookup.

The *backward* provenance of a given output record is constructed by searching the corresponding intermediate keys in the provenance files of the Map tasks. This lookup returns the locations of all records in the input files processed by the Map tasks that produced any of the intermediate records affecting this output. Recall that HadoopProv tracks lineage linking Reduce side intermediate records and generating Map tasks. Only provenance files of these tasks are searched. This technique can be adjusted to construct the *forward* provenance of an input record.

3 Evaluation

This section presents preliminary results from our prototype implementation of HadoopProv. Our instrumentation is based on MapReduce 2.0 (MRv2). Our testbed consists of four machines each having two Intel Xeon E5607 CPUs, 24 GB RAM, 1 TB SATA Disk, and connected over a 1 Gbps switch.

Our evaluation focuses on quantifying the temporal and spatial overheads of HadoopProv on a typical MapReduce workload. We carry out canonical wordcounts on public Wikipedia dumps [4] of size 60, 90 and 300 GB. For each dataset we compare job execution times with and without provenance and the size of the created provenance files. The presented results are averages of three runs.

Wordcount is an example of a many-to-one transformation where an output record is dependent on one or more input records. We expect the Wikipedia dataset to be skewed on popular words which should stress the system at the Combine and Merge phases. For one-to-one transformations (e.g. sort), we expect the overheads to be much lower as shown in related work [2, 3].

3.1 Temporal Overhead

As outlined previously, HadoopProv is designed to minimise the temporal overhead of capturing provenance in a bid to encourage its integration as a core component of the MapReduce infrastructure.

Figure 2 illustrates the overhead of provenance capture for 60, 90 and 300 GB dataset sizes. We observe up to 10% additional overhead on the runtime of jobs. With respect to individual tasks, the average temporal overhead of the Map and Shuffle/Merge tasks is approximately 7% while the overhead in the Reduce tasks increases to 20%–30%. The relatively larger overhead in the case of Reducers is attributable to the fact that the amount of I/O is doubled (HadoopProv reads per record provenance data from the Merge phase and attaches it to the output). As our cluster is small, the Map phase takes longer to complete and contributes more to the overall runtime. Consequently, the overhead of the Reduce phase is relatively masked.

We have also compared wordcount runtime in the case of HadoopProv and RAMP [2] for the 60 GB dataset. Preliminary results show that RAMP has an order of magnitude increase relative to HadoopProv in our four-machine cluster. This illustrates the benefit of implementing provenance capture as an intrinsic function of the framework.

Our preliminary results indicate that provenance capture with HadoopProv imposes a reasonably small temporal overhead and scales well with different dataset sizes.

3.2 Spatial Overhead

HadoopProv stores per-record provenance information and hence the spatial overhead of provenance capture is proportional to the number of key-value pairs in the input and output. Results presented in Figure 3 confirm this intuition.

Focusing on the 300 GB dataset, HadoopProv requires approximately 220 GB additional disk space to store the associated provenance data (Map and Reduce provenance files as well as the index used to speed up provenance query). We use hashing to encode the intermediate keys. This ratio is maintained for the other dataset sizes.

3.3 Lookup Overhead

Tracking an output record through intermediate records to source inputs requires searching each index file corresponding to the Map tasks that emit intermediate records affecting the output. Figure 4 shows that the average index file size is in the order of a few megabytes. The "small"

difference in provenance file sizes is attributed to the difference in words distribution of the datasets.

As this index file is already sorted, its lookup complexity is $\mathrm{O}(\log n)$. Effectively, the provenance query for one record is satisfiable in $\mathrm{O}(k \log n)$ where n is the average number of records per Map task and k is the set of map index files that needs to be searched. It is also possible to parallelise these k searches.

4 Use Cases

Many datasets typically grow incrementally over time. It is extremely inefficient to process the entire dataset on every addition, especially if the dataset is large [5]. Using HadoopProv, we can exploit provenance information to limit additional processing to only the *changes* between the old and new inputs. Given this delta, we compute the set of intermediate keys for it. Then for each key, HadoopProv is able to fetch (if any) the corresponding input and output records with the same intermediate keys. The subset of input records and the delta are then fed to the MapReduce job, which updates the old output records with the new values.

Similarly, HadoopProv makes it simple to filter a related subset of records without requiring a rescan of the entire dataset. For example if a job computes the minimum of an input dataset grouped by key, subsequent jobs can make use of the provenance information to compute the maximum of *specific* keys without the requirement to rescan the whole input set. This functionality can be generalised to log processing [6, 7] and indexing [8, 9]. In fact, a by-product of HadoopProv is the creation of on-the-fly index that can be exploited by such applications.

Provenance information may also be leveraged to compute statistics that assist in optimising future job execution on identical or similar datasets [10, 11]. For instance, statistics about the data skew is useful to decide on the optimal hash function in MapReduce jobs.

Moreover HadoopProv can be easily extended to record block level lineage and therefore be used to decide where to store related data blocks that might be processed jointly. This is especially beneficial for joins [12, 13] in which the network communication cost (Shuffle phase) is reduced.

5 Related Work

A few existing projects provide similar functionality to HadoopProv. RAMP [2] is a wrapper based approach of Hadoop record readers and writers enabling provenance capture and trace. It requires little user intervention in most cases and it does not modify the core of Hadoop. As a rough comparison, the reported runtime overhead of wordcount on the platform is approximately 76%. We believe that this large overhead can be attributed to (*i*) the requirement that provenance information is shuffled to the Reducers, (*ii*) the need to wrap and unwrap records in the default data path, and (*iii*) crucially the fact that the complete provenance graph for all records is constructed during the actual MapReduce job. On the other hand, we expect RAMP to be faster in provenance queries.

Similarly, Crawl et al. present a modification to Kepler and Hadoop to track provenance in scientific workflows. In their implementation, wordcount takes 2.5x longer to execute when provenance capture is enabled [3]. The integration with Kepler has an effect on this large overhead.

6 Conclusion and Future Work

In this paper we presented HadoopProv, a modified version of Hadoop that captures and analyses provenance in MapReduce jobs. We have designed it to have minimal temporal overhead. Preliminary results are quite encouraging indicating a runtime increase of less than 10% on a typical MapReduce workload. Users not requiring provenance can turn this feature off. In future work, we intend to rigourously evaluate HadoopProv on larger testbeds running commodity production MapReduce workflows taking into consideration failure modes (currently we discard provenance files of failed tasks). Moreover, we are optimising the spatial overhead using graph compression [14].

7 Acknowledgment

We thank Lucian Carata for his views on the design of HadoopProv. We also appreciate the feedback of George Coulouris, Nikilesh Balakrishnan and Thomas Bytheway.

References

[1] "Hadoop MapReduce." http://hadoop.apache.org/.

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

[3] D. Crawl, J. Wang, and I. Altintas, "Provenance for MapReduce-based data-intensive workflows," in *WORKS'11*.

[4] "Wikipedia Dumps." http://dumps.wikimedia.org/.

[5] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: MapReduce for incremental computations," in *SOCC'11*.

[6] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, "The unified logging infrastructure for data analytics at Twitter," *VLDB Endow.*, 2012.

[7] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, 2012.

[8] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *Proc. VLDB Endow.*, 2012.

[9] J. Lin, D. Ryaboy, and K. Weil, "Full-text indexing for optimizing selection operations in large-scale data analytics," in *MapReduce'11*.

[10] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *NSDI'12*.

[11] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing data shuffling in data-parallel computation by understanding user-defined functions," in *NSDI'12*.

[12] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endow.*, 2010.

[13] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: flexible data placement and its exploitation in Hadoop," *Proc. VLDB Endow.*, 2011.

[14] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long, "A hybrid approach for efficient provenance storage," in *CIKM'12*.