Conditional Provenance: Enhance Dataflow Job Operators Based on Lineage Queries

Salvador Vigo
DFKI
Berlin, Germany

Joscha von Hein
TU-Berlin
Berlin, Germany

Jorge-Arnulfo Quiané-Ruiz
TU-Berlin
Berlin, Germany

Kaustubh Beedkar
TU-Berlin
Berlin, Germany

ABSTRACT

With the (re-) new era of big data analysis, Data-Intensive Scalable Computing (DISC) systems have gained popularity due to their implementation of the dataflow programming model. However,while big data processing has become dramatically easier in the last decade, the state of big data debugging is very much in the early stages. To overcome this problem, the database community has studied at length the data provenance field, where capturing the lineage information in the dataflow job of the DISC systems can provide a valuable information of the data in its life-cycle. Existing tools for capturing provenance, first, are designed to be part of a specific DISC system and second, they are following what we call a full-stack approach for capture and store lineage, increasing dramatically in most of the cases the size of the lineage output and for instance the overhead execution of the entire pipeline. In this paper, we propose a novel solution to improve the performance of capturing and tracing provenance instrumenting conditional operators based on the actual provenance query the user is trying to solve in his post-hoc analysis. We argue that considering the lineage queries (tracing) to be processed after the workflow job, it is possible to add conditions to the operators improving steeply the results of lineage tasks. For this experimental analysis we instrument from the TagSniff abstraction and compare our model to the full-stack approach of capturing all available lineage data. Our evaluation shows that we are able to reduce the execution time overhead and the storage footprint of the lineage data considerably. Before the conclusion, we also include a preliminary analysis of a probabilistic approach that could benefit the storage footprint results in some specific use cases.

PVLDB Reference Format:
Salvador Vigo, Joscha von Hein, Jorge-Arnulfo Quiané-Ruiz, and Kaustubh Beedkar. Conditional Provenance: Enhance Dataflow Job Operators Based on Lineage Queries. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

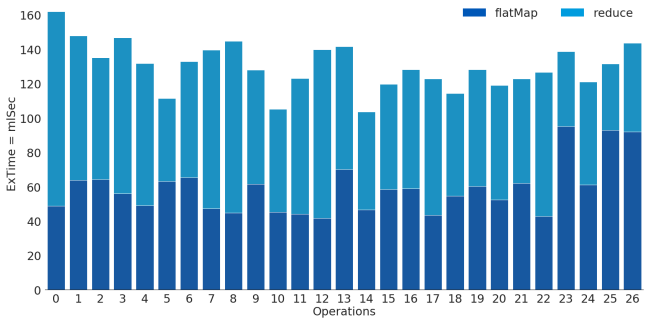


Figure 1: Runtime for a flatMap and reduce debugging job.

1 INTRODUCTION

Data Provenance information describes the origin and the story of the data flowing in its life-cycle. Capturing provenance, also called lineage, could substantially improve the way that we process the data. In the data management community, data provenance has been studied in different areas such as scientific data processing and database management systems [5]. As argued by Lynch [12], today, differing from previous centralized control systems, the data is constantly being created, copied, moved around, and combined indiscriminately from different data sources widely variable in terms of quality. It is essential to provide context information to help end users decide whether query results are trustworthy.

Provenance is also an active topic of research in DISC systems. Data-Intensive Scalable Computing (DISC) systems are extensively used for big data analysis. This is mainly because of two reasons, the systems make use of the dataflow programming model and also provide a suitable underlying implementation. Data-oriented workflow models allow programmers to express a data processing program using a direct acyclic graph where the nodes are the operators and the edges are the data flowing between operators [10]. Some examples of DISC systems are Apache Hadoop, Apache Spark and Apache Flink.

In the scope of big data analysis, DISC systems are a great asset since they can scale to (very) largest datasets by partitioning the data assigning tasks to each partition in parallel. Although this brings a great advantage, it also creates an enormous challenge for data scientists in understanding and resolving program errors [9]. Debugging data processing logic in DISC environments can be dispiriting. It is estimated that users spend 50 % of their time

debugging their applications, resulting in a global cost of 312 billion dollars per year [3]. One of the fundamental issues is that almost all of the existing tools are developed for debugging code and not (big) data. Using this example as a motivation, there is a need for capturing and supporting data lineage and suitable provenance query capabilities in DISC systems.

At this time we make the following observation. Admitting that a solid set of tools has been proposed to address this problem, however, most of the existing techniques provide solutions based on the *full-stack* approach for capture and storage lineage. We define this approach as the one in which the full lineage meta-data, that relates input with output at the operator level, is captured and stored in an external system. Although it provides *full-map* lineage information, in most of the cases it can become inefficient since for many jobs, provenance queries, e.g. forward and backward tracing, don't require the full lineage data to be processed. Added to which, when it comes to very large datasets operations, this inefficiency could lead to unsustainable system overhead and meta-data storage.

Let say that we want to implement a debugging application on top of an underlying DISC system capturing and storing provenance. This will increase the execution overhead of the application operator-wise as well as generate lineage storage information of different sizes. For example, we show in Figure 1 a slice of the run time for a debugging job in a workflow system with a flatmap and reduce operator where we can notice that both of them behave almost in the same way.

We argue that, based on specific provenance queries, we could instrument our workflow job to add conditions to the desired operators, aiming to capture (select and send out) only the lineage data which is needed to answer the provenance query the user wants to solve. Following this approach, we can significantly decrease the execution time for the selected operators as well as reducing the memory foot-print of the storage. Overall, our contributions are as follow:

- After establishing foundations, in **Section 2** we briefly review some of the relevant existing tools which help to motivate our approach. Then in **Section 3** we explain how to leverage the *TagSniff* model abstraction [6], a general purpose abstraction for capturing and debugging from data provenance.
- **Section 4** describes the methodology behind our experiment, where we instrument a Flink job for capturing and tracing provenance based on the TagSniff approach. This job allows us to compare and evaluate the execution time overhead as well as the lineage output-data size between both approaches, full-stack (standard) and conditional provenance.
- **Section 5** and **Section 6** report and analyze the results of our evaluation. We split the results in 4 sub-sections, lineage capturing, lineage tracing, memory foot-print and operator-wise microbenchmarking.
- As a preliminary analysis, in **Section 7** we include a description of a probabilistic approach based on the *Bloom Filter* [2], as well as discuss the observations from a first experimental analysis.

In **Section 8** we summarize, including the pros and the cons of our model as well as discussing future work.

2 RELATED WORK

We are probably attending to the second golden era of big data. At the beginning of the past decade, studies deriving from the *MapReduce* [7] have provided a new paradigm when it comes to process and analyze very large datasets. These are the so-called DISC systems commented before. To cite some examples we have Apache Hadoop, Apache Spark or Apache Flink [4].

One of the key features of DISC systems is that they handle the escalation to largest datasets by partitioning data and assigning tasks that execute in parallel each partition of the application logic (e.g., GFS [8]). At the same time, gains in scalability pose new challenges for programmers who find it harder to understand and deal with erroneous data in the dataflow pipeline.

Dealing with these challenges, the study of data provenance has been the focus of several research works and is still gaining more attention from the database research community. For instance, RAMP [10] is a toolkit abstraction for capturing data provenance and supporting post-hoc analysis of DISC systems. It is one of the earlier tools to be on top of workflow programming models to support lineage capture. RAMP operates externally to the target system, e.g. Hadoop, mapping all inputs to the outputs for each map reduce operator. The lineage data resulted is stored in the Hadoop Files (HDFS) component. Due to the full-stack provenance approach, once the underlying task escalates, RAMP falls off in a number of issues, including scalability when sharing amount of lineage data in tracing stage, incurred overhead in the per-job from capturing stage and memory foot-print.

TITIAN [11] is similar to RAMP, adopting the same concept of full-stack provenance. They claim that their approach leverages RAMP providing a unified environment, in which both data analysis and data provenance queries can operate in concert. They implement their model on top of Apache Spark, storing the lineage data in Sparks Blockmanager. BigDebug [9] extends TITIAN and focuses on debugging use cases. It provides support for simulated watch-points and break-points for debugging spark jobs, but it incurs in appreciable overhead requiring extensive modification to the underlying system and does not provide support for post-hoc analysis.

At last, Inspector Gadget [13] defines a general framework for monitoring and debugging Apache pig programs. It does not come with full data lineage support,(e.g. out-of-core analysis) but rather it provides a framework for tagging records of interest through the workflow.

3 CAPTURING CONDITIONAL PROVENANCE

Despite the fact that most of the tools represented before provide notable studies and stable solutions, we observe that none of them are facing the challenges from a different perspective, the provenance queries. We define *conditional provenance* as the method to improve the data provenance analysis capabilities adding conditions in the data-parallel operators compiled into a Direct Acyclic Graph (DISC systems). These conditions are defined based on the requirements of the provenance queries in post-hoc analysis.

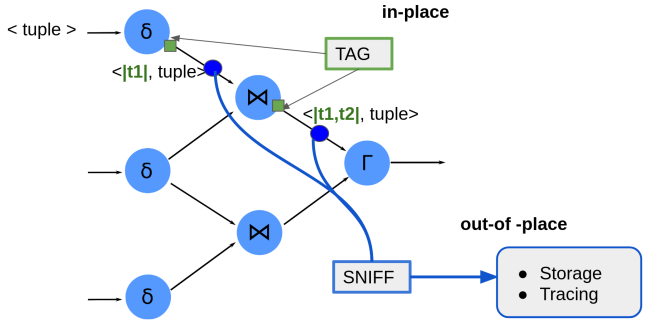


Figure 2: TagSniff model abstraction. In-place and out-of-place instrumentation.

3.1 Provenance Queries

In data-oriented workflows, tracking lineage information is indeed beneficial to understand how data elements are processing through the pipeline [14]. Provenance queries allow *post-hoc* analysis, for instance they support *backward* tracing, finding the input dataset that contributes to a specific output item, and *forward* tracing, determining the output dataset produced for an input element. In most of the cases, the amount of lineage information that we require to run these queries is only a small fraction of the whole lineage data.

Let us set an example. Suppose that we want to count in a W job how many times words from a text repeat itself. After the job, we query the lineage output to trace backward searching for a specific input set I.

```
output: (word=Word, count=N)
backward_trace(output,W,{I1,...,In}) if count==3
```

As we can appreciate in the block above, in this example we add a condition to our provenance query, where we are interested in the outputs that match the repetition number of 3. If we suppose that words repeated only 3 times represent 15% of the total, then the captured meta-data required to feed our provenance job will also be a small fraction. The operator responsible for aggregating the word counts is the *reduce* operator. If we can instrument our provenance tool with the capacity to add conditions to specific operators like reduce based on the provenance query, then the amount of lineage capture and storage will be significantly decreased. Even more, we can witness an overhead reduction in the execution time of the provenance application, since we are capturing less lineage information.

3.2 TagSniff Model Abstraction

For this purpose, we make progress leveraging the *TagSniff* model [6], a general compelling abstraction for debugging and ease provenance capture. As you can appreciate in Figure 2, TagSniff is based on two powerful primitives, *TAG* and *SNIFF* that operate in the *debug tuple*. The logic behind this model is flexible enough to allow users to instrument their dataflow applications for their analysis requirements.

The data flowing through the pipeline is considered as a tuple and the tuple that is being analysed is a debug tuple. These debug tuples are flowing between the data-flow operators with a *header* component that keeps trace information. This information is annotated with the TAG primitive in the in-place piece. Then, in the out-of-place piece, the primitive SNIFF extracts the desired tuples based on their header or value, to be analysed and transformed if necessary in an external system.

Typically these annotations describe how the user expects the system to react. To these headers we can incorporate metadata that add extra information to the debug tuples, for instance unique identifiers. The tags are inserted by the users to support refined debugging scenarios, e.g., lineage. The SNIFF primitive is used for identifying tuples requiring further analysis based on their metadata. It can capture and send metadata (and also tuple values) out-of-place for post-hoc debugging tasks.

This logic is particularly powerful, since it allows us to implement our program to add conditions at operator-wise level of the workflow job. TagSniff authors claim that wrapping the underlying system into TagSniff components gives us the flexibility to modify the operators with a comprehensible instrumentation easy to adapt to DISC systems that compiles into directed acyclic graphs. In the following, we describe how to implement a data provenance capturing task leveraging TagSniff model with a word count example. Then we will evaluate our results to observe how efficient the conditional approach behaves in comparison with a full-map lineage capture.

3.2.1 Our Implementation using Flink. While we have explained the underlying concept of Conditional Provenance and the TagSniff approach for big data debugging we will now explain our implementation of Conditional Provenance and the TagSniff method using Flink as the underlying DISC system. To find the motivation of using Apache Flink and the methods selected in the next paragraph, please refer to section 4.

To enable adding tags to tuples and capture provenance data we created a wrapper for Flink *DataSet* object and provided abstractions for filter, map, flatmap, reduce and groubBy methods. These methods call custom Operator Wrappers which contain the logic to add specific Tags and to Tuples and to (conditionally) capture the lineage data and send it out to external storage via Kafka.

The implemented DataSet wrapper works with a custom data type provided by TagSniff called debug tuple which contains a header that includes all available metadata about the object and the value containing the payload the user actually needs in his job. The operator wrappers unwrap each incoming debug tuple, extract the ID from the header, add tags if necessary, perform the actual operation on the unwrapped value, wrap the result in a new debug tuple and then send out the lineage information.

Capturing and sending out the lineage data is only performed if necessary and where the user provided condition is fulfilled. For anyone who is interested to learn more about the details feel free to contact us or look up the code directly on GitHub.

4 METHODOLOGY

In this section, we describe the main concept behind our experiment.

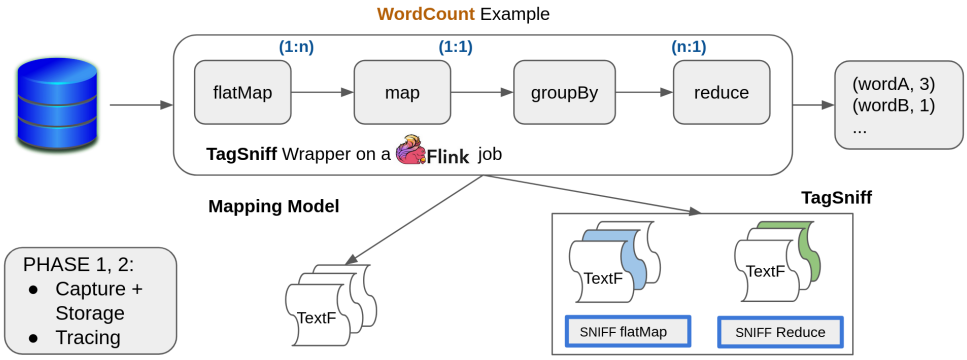


Figure 3: TagSniff wrappers of the WordCount job model

Figure 3 represents the experiment placed in the Post-hoc analysis scenario of an Apache Flink job [4] in which we want to query provenance applying forward and backward trace. We have chosen Flink as an underlying system due to its versatility at the time of implementing data-oriented workflows. Flink is designed to run stateful streaming and batch applications at any scale. Applications are parallelized into possibly thousands of tasks that are distributed and concurrently executed in a cluster. For this case, we leverage the Flink *DataSet* API.

As one can appreciate in the figure, we implement a word count example compound of *flatmap*, *map*, and *reduce* operators. From a logical perspective, the reason to choose these operators is due to the fact that we generate tuple unique-ids when wrapping the operators, mapping the I/O relationship of each of them. The relations are one to many (1:n), one to one (1:1) and many to one (n:1) respectively. With this approach we cover most of the relational database operations.

The TagSniff abstraction wraps the Flink job allowing us to enhance each operator according to our provenance query needs, generating the debug tuples, tagging the headers with unique ids and using the sniffer to act depending on the tags a debug tuple has attached.

To evaluate our results, we use a baseline reproducing the *mapping* approach similar to RAMP [10] and BigDebug [9]. In phase one of the experiment, first we run different jobs depending on the provenance query case. In each of these jobs, either flatmap or reduce operators are modified to allow conditional provenance. Second, we store the captured metadata into text files saving them for the second phase. In phase two, we implement backward and forward tracing over the stored lineage information.

We select Apache Kafka to produce the data out-of-place and to consume it in the tracing job. To measure the impact of the collected metadata size, we store the lineage information into text files.

4.1 Components and Aspects

In order to properly evaluate the concept of conditional lineage capturing and our implementation of it we focus on the compute time and memory footprint.

The first being measured by total job execution time as well as Microbenchmarking at the operator level. The former by simply comparing files which contain the generated lineage information by file size.

4.2 Benchmark Details

4.2.1 query sets. To measure the effectiveness of conditional provenance and the show that it is only beneficial for certain provenance queries we establish the notion of two different *query sets* here one of which is very specific and needs only a subset of total job lineage information while the other is more general:

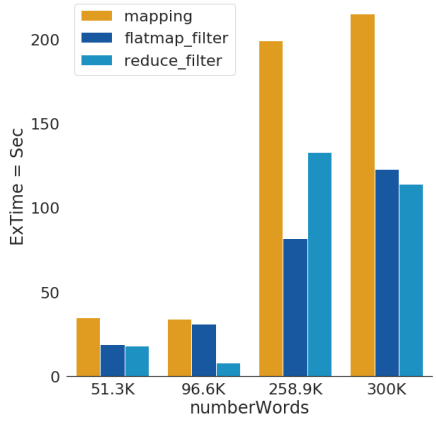
query set 1:

- **forward tracing**: "find all words which result from lines with exactly 8 words."
  Only a few lines pass this criterion which is why a lot of provenance information can be filtered out from the flatmap operator using conditional provenance capture.
- **backward tracing**: "find all lines which contain words with a wordcount larger than 100".
  As only a fraction of words occur that often in our benchmark datasets a lot of provenance information can be filtered out from the reduce operator using conditional provenance capture for this query.

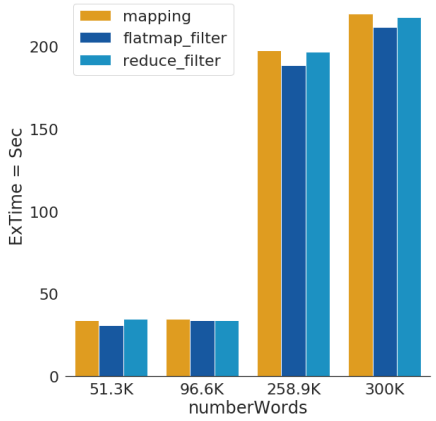
query set 2:

- **forward tracing**: "find all words which result from lines with less than 10."
  This criterion is true for a large share of lines in our datasets which limits the benefits of conditional provenance capture in the flatmap operator.
- **backward tracing**: "find all lines which contain words with a wordcount less than 100".
  This query also does not discriminate much which means there is very little potential to save space and time in the reduce operator.

We will mostly set our focus on query set 1 in the following sections as it shows the potential of conditional provenance very well. Query set 2 is there to showcase that the benefits of conditional provenance are very dependent on the underlying provenance query one intends to solve. It is therefore a judgement call by the user to decide whether it is worth employing conditional



(a) conditional provenance for query set 1



(b) conditional provenance for query set 2

Figure 4: Capturing Job execution time benchmark

provenance capturing or whether the standard approach is fine in his use case.

4.2.2 capturing job execution time. We ran our experiments using four different Datasets with increasing file sizes and wordcounts (50,000 - 300,000 total words).

All our measurements of the execution time in the capturing phase are done with parallelism 20 on an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz using 20 taskmanagers with a heapsize of 1GB each.

The execution time measurements for the tracing part as well as the microbenchmarking has been done on our local machine, a MacBook Pro 2 GHz Quad-Core Interl Core i7 with 8 GB DDR3 memory.

Flink automatically saves information like job runtime which we simply collect in a log file for each experiment. This is both easy and introduces no additional measurement overhead.

4.2.3 Microbenchmark details. In our Microbenchmark measurement we take the time our wrapper instances spend in their flatmap and reduce function respectively and send this information out to Kafka from which we consume that information later. This gives us a more detailed overview of where the job spends most of its time. One has to note that this more fine grained measurement incurs a significant overhead on its own though (as the program has to take the nanotime twice per operation and send that information out) which is why the microbenchmark measurements had to be seperate from the overall execution time measurements.

5 BENCHMARK RESULTS

5.1 Lineage Capturing

5.1.1 Execution Time. Figure 4 shows the execution time measurements using the standard approach and conditional provenance capturing for query set 1 and query set 2.

The results for query set 1 (figure 4a) show a significant improvement in overall execution time when using conditional provenance compared to the standard approach for all data-sets. The execution times generally increase for larger data-sets although not linearly which is explained by the heterogeneity of the data-sets in regards to words per line and number of distinct words and the distribution of word occurences. Especially noteworthy here is the difference in our two smallest data-sets (Hamlet - 51.3k words and the Hobbit with 96.6k words) which display very similar execution times during provenance capturing even though the Hobbit is considerably larger.

In figure 4b, query set 2 is a perfect example of provenance queries which can't benefit from conditional provenance capture at all. The execution times of the standard and conditional provenance approach are almost the same here for all data-sets.

5.1.2 Microbenchmarking. Figure 5 shows the Microbenchmarking results of the capturing phase for the Hamlet data set using query set 1. We measure the time it takes the flatmap and reduce operator wrapper to perform a single operation. For the flatmap that would be: extracting the ID of the incoming tuple, splitting the line (String) by whitespaces, creating new debug tuples for the resulting words and sending either all or only the pertinent lineage data (input ID - List<output ID>) to external storage via Kafka. A single flatmap operation takes approximately 160 ms in our experiment on our local machine using the standard approach, whereas it is 20 ms shorter when using Conditional Lineage Capture. Another observation is that a single reduce observation (around 60ms per operation) takes about half the time a single flatmap operation (around 150ms) in our experiment.

5.2 Capturing storage footprint

In this section we present the findings from our storage footprint analysis. Figure 6 shows the plots which display the size of the resulting lineage files of our provenance capturing jobs for all our data sets.

We compare the standard approach, where we capture all available lineage information with the conditional lineage capturing approach for query set 1.

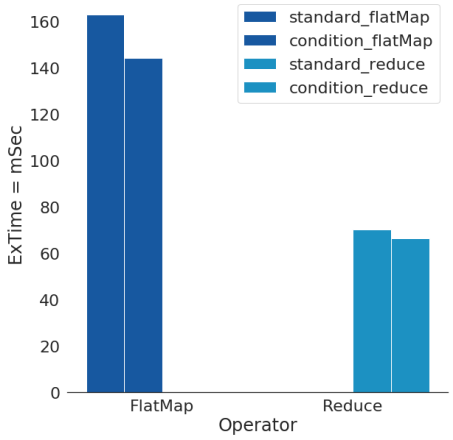


Figure 5: operator wise Microbenchmark

In figure 6a we see the results of the flatmap operator lineage. While the complete lineage information for this operator ranges from 250kb to 2400kb in file size, this is reduced considerably using conditional provenance capturing where the sizes range from 3kb to 306kb.

With this kind of provenance query, the storage needed to save the relevant lineage information of the flatmap operator can be reduced to negligible amounts.

Similar results can be observed when we compare lineage file sizes at the reduce operator. Here the complete lineage information of the operator is slightly larger, ranging from 282kb to 2700kb. If we employ conditional lineage capturing on the other hand we reduce the file sizes to 73kb to 935kb respectively.

The file size reduction here is not as profound as in the flatmap operator as our provenance query specifies to keep lineage information of words which occur more than 100 times. And while there is only a small number of words for which this is true, the provenance data for these words is especially large as the wordcount is exactly equal to the number of incoming elements of the reduce operator. Each result tuple of the reduce operator thus has at least 100 tuples containing (word, 1) which produce that exact result, which increases the size of the operator lineage.

5.3 Lineage Tracing

In this section we present the results of our execution time benchmarks in the Tracing part. The provenance queries are the one previously defined in query set 1.

We have implemented the forward and backward trace as a separate Flink program which takes the lineage files generated in the capturing phase as input. This enables us to exploit Flink inherent parallelism to work on very large lineage data.

All measurements of the tracing part have been performed on our MacBook Pro with parallelism 8.

5.3.1 Forward Tracing Query. The forward tracing query we implemented here and measured is: Do a forward trace on all lines (input elements) which contain exactly eight words. This translates to finding all output elements (words, wordcount) which occur in lines which contain exactly eight words.

In the capturing phase we generated the pertinent lineage files. We compare the forward trace using the complete lineage (standard approach) vs using the smaller lineage files containing the already filtered lineage data.

As can be seen in figure 7a the execution time of the forward trace job is about 140 ms for Hamlet data set and 250 ms for the Hobbit data set when we use the standard lineage files.

If we use lineage files where the flatmap operator lineage has already been appropriately filtered in the capturing phase the execution times are reduced to 50 ms and 150 ms respectively. Thus showing that for this kind of query conditional lineage capturing reduces execution time roughly by half.

5.3.2 Backward Tracing Query. The results for the backward tracing query are similar but not as profound. Execution times using the standard lineage files are around 130 ms for Hamlet and 260 ms for the Hobbit respectively. The execution times using the smaller conditional lineage files are 90 ms and 240 ms, still representing a noticeable reduction but much smaller than the one found in the forward tracing query.

This difference can most likely be explained by the fact that the filtered lineage information of the reduce operator is still very sizable. Even though there are only very few distinct words which occur more than 100 times in both Hamlet and the Hobbit, they naturally make out a much larger share and occur in a lot of lines.

6 ANALYZING RESULTS

The experimental results in the previous section show that conditional provenance capturing massively reduces execution time when using query set 1.

While this was expected, the scale of the reduction in execution time surprised us because it means that the overall overhead of the lineage capture, compared to the plain flink job with no provenance and debugging capabilities has to be large. If the overhead was small, then the differences in execution time of the standard approach and conditional capturing had to be small as well. We suspect the reason to be that in our implementation the process of sending out the lineage information is responsible for almost all of the capturing overhead.

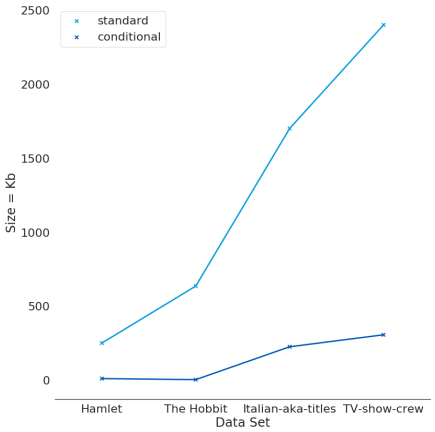
Improving the capturing efficiency would therefore be a necessary first step when working further on the concept of conditional provenance capture.

These findings do not however invalidate the general benefits of Conditional Lineage - but rather questions only the scale of the benefits in terms of the Capturing Execution time.

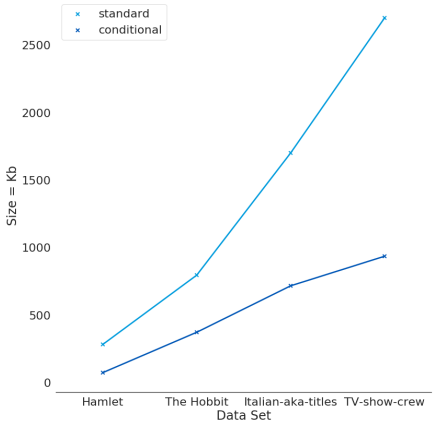
Capturing lineage information will always incur some execution time overhead which can be reduced by only storing the lineage information actually pertinent for a given provenance query.

The reduction in lineage file sizes is also remarkable and arguably the biggest selling point of using conditional lineage capture as those will persist even when a new implementation can reduce execution time overhead of lineage capture.

The serialized lineage data has usually the same order of magnitude as the source data set and can thus be extremely large. Keeping

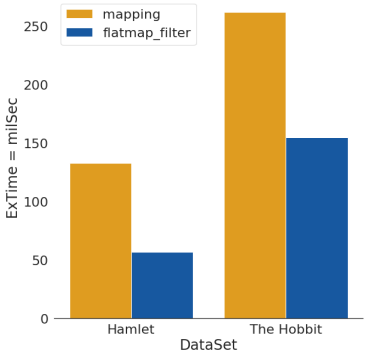


(a) flatmap operator lineage file sizes

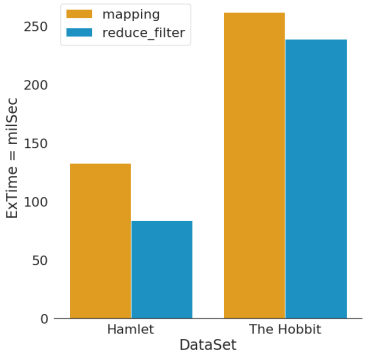


(b) reduce operator lineage file sizes

Figure 6: Lineage Data Storage Footprint



(a) forward trace



(b) backward trace

Figure 7: Tracing Job Execution Time Benchmark

the storage footprint as low as possible should be a high priority for every DISC systems programmer.

The reduction in execution time in the tracing part is also quite noticeable and another reason to use conditional lineage capture.

When someone is already using a framework which can save lineage data there is basically no reason to enhance that by adding a filter function which enables query sensitive conditional provenance capturing.

7 FROM CONDITIONAL TO APPROXIMATE PROVENANCE

7.1 Concept and Motivation

In the previous section we have seen that conditional provenance capture has significant benefits in terms of reduced execution times during capturing and tracing as well as much reduced storage needs for lineage information.

During our work on the topic of Conditional Provenance and thinking about ways to reduce the lineage data while still being able to answer the provenance query, we came about the idea of *Approximate Provenance*. Since the idea of Approximate Provenance is very related to Conditional Provenance Capture we decided to explore the topic further and managed to include a running prototype into our implementation of a provenance capture system, the results of which we are about to share. While the topic of Approximate Provenance was already discussed in theory [1] we believe we are the first to actually implement and measure it in a running system.

While all current systems which try to answer provenance queries store operator lineage as a mapping from input to output, usually in a table, *Approximate Provenance Capture* uses probabilistic data structures instead.

These probabilistic data structures can be of much smaller size than traditional lineage mappings which are stored in tables. The drawback is that the answers they give us are as their name suggests 'probabilistic'.

We argue that while certainly not appropriate for all kinds of provenance queries, approximate provenance could be valuable in those use cases where small errors are acceptable.

We managed to successfully capture provenance information of a reduce operator using bloom filters in our approximate provenance pilot project.

Bloom filters have the property that they allow false positives but not false negatives which is why they were our first choice.

We can think of a lot of use cases where the user of a data provenance system can accept that a forward or backward tracing job yields a superset of the actual trace elements. For instance if a user sees that a certain output element is false and wants to check which input tuple(s) might be responsible, the user has to perform a backward trace from the false output element and then check the resulting trace set manually.

If the provenance information of the reduce operators in this case were mappings of a Bloom Filter to the output id instead of a table with incoming tuples to output tuples the resulting elements of the backward trace would be a superset of the actual trace elements. If that superset is not much bigger than the original set the user might prefer this approximate provenance capture process if it has other noticeable benefits.

The main benefit of using a probabilistic data structure such as a bloom filter instead of the usual tables which contain the (input, output) relationships of an operator is it's size.

If chosen carefully and used in the right place bloom filters offer considerable benefits when it comes to storage which is why they are so popular in networking applications which only have limited disc space.

In a hypothetical example where a reduce operator has a million elements as input the standard approach would be to store a million (input id, output id) tuples in a table. Using a bloom filter instead can reduce that to a single (bloom filter object, output id) tuple. Obviously the bloom filter takes more space than a single input id but far less than one million combined.

With the right queries, noticeably ones where an operator contains a lot of inputs per output approximate provenance could reduce the storage overhead of lineage capture by an incredible amount as we will see in the next section which contains a preliminary analysis.

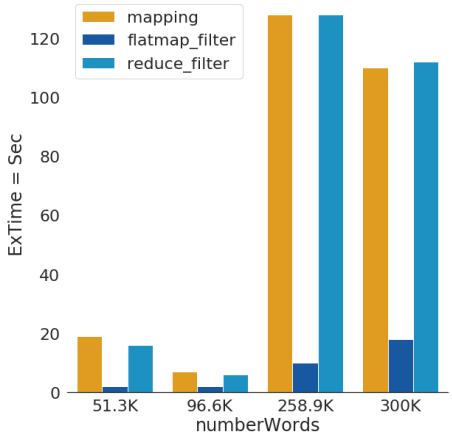


Figure 8: Approximate Provenance Capturing Execution Time

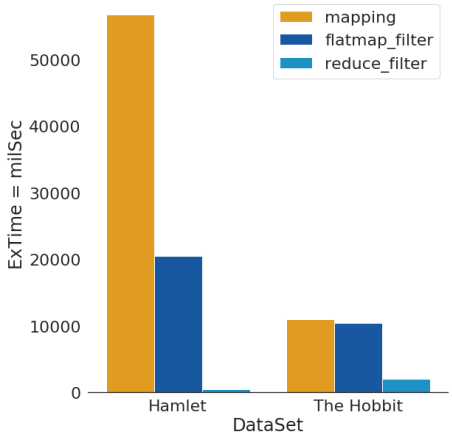


Figure 9: Tracing jobs using bloom filter lineage data. yellow and dark blue bars are forward tracing jobs while the light blue bar is the backward tracing job.

7.2 preliminary analysis

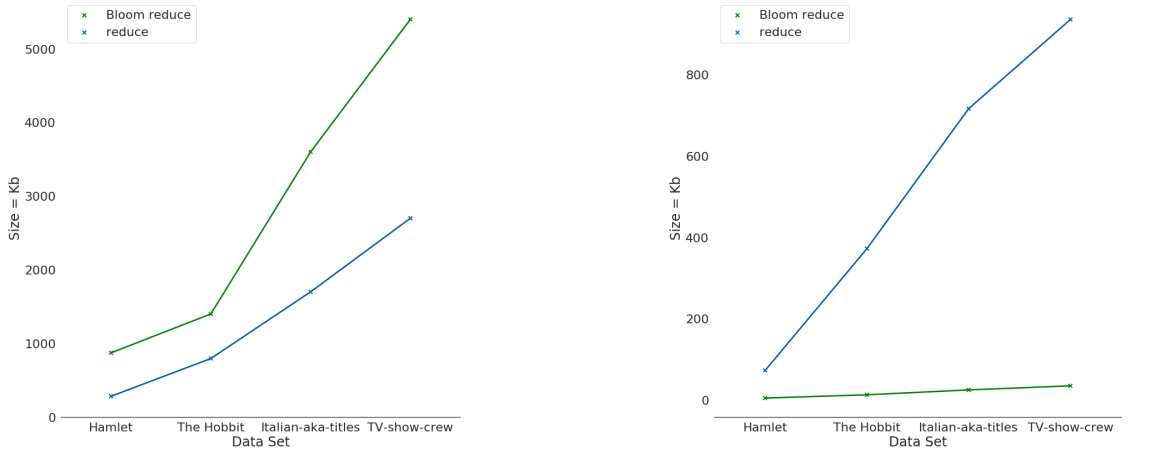
7.2.1 Capturing Execution Time. In order to get some preliminary insights into approximate provenance we have run the same provenance capture queries as before on the same data sets as before changing only the way we capture provenance in the reduce operator.

In figure 8 we see the execution time of our capturing job using a bloom filter instead of the standard mapping. Approximate and Conditional Provenance are not mutually exclusive but can rather complement each other which is why we measure the execution times when using a condition on the flatmap operator and reduce operator in addition to the standard way of storing the complete lineage information.

We can see in these graphs that the execution time using the bloom filter in the standard setting is about half for each dataset what it was when we used a simple mapping.

Even more interesting is though that the execution time when conditionally capturing provenance in the flatmap operator is extremely small while if we conditionally capture provenance in the reduce operator we barely see a difference to the standard approach. This was very unexpected but we are now able to explain this result with high certainty. We strongly believe that these results are due to the previously mentioned capturing overhead of sending out data via Kafka in our implementation. In our specific implementation we only send out the mapping data to Kafka but store the lineage information of the reduce operator using bloom filters using Flinks built in serialization and write the 'BloomToOne' Lineage information directly to a file using Flink. Apparently this approach is much faster than the approach we use with capturing mapping provenance in which we send out the lineage via Kafka.

When we run our Capturing Job using bloom filters, the flatmap operator is the only one where we send out data via Kafka. If we



(a) Standard Lineage Capture

(b) Conditional Lineage Capture

Figure 10: Approximate Provenance Storage Footprint

capture the flatmap operator lineage conditionally we reduce the total amount of data sent to Kafka considerably which results in massive performance increases.

The opposite is true for conditional capturing on the reduce side. There we don't send out lineage information to Kafka at all which results in negligible performance changes overall.

This results shows us that if someone were to continue our research in the area of conditional provenance or improve our implementation of it one of the most important things on the to do list would be to increase the performance of capturing the mapping lineage. This might be done by changing the logic so each operator instance only spawns a single Kafka producer instead of creating a new one for each operation itself.

7.2.2 Tracing Execution Time. In Figure 9 we show the results of our execution time benchmark using bloom filter lineage data for the tracing part. As before we used the queries specified in query set 1 to have a good comparison between the different approaches.

It is important to note that it is not possible to perform the backward tracing query using the standard approach and bloom filters together. This is due to the fact that the query specifies to trace only those words which occur more than 100 times. When we store lineage data as a mapping we can simply count the number of different input ids which relate to the same output id in the reduce lineage and filter the lineage based on that. This is not possible when using bloom filters as reduce operator lineage. Thankfully using conditional lineage capturing solves this problem while at the same time reducing storage and execution time overhead.

The yellow bar shows the execution time it takes the forward trace job to finish. As can be seen the execution time reaches almost 60.000 ms for the Hamlet data set.

This incredibly high amount is due to the fact that using bloom filters instead of (input, output) tuple mappings has the drawback of needing a cross product between the outputs of the flatmap lineage and the bloomToOne lineage which is considerably more compute intensive than being able to use a Hash-Join which is used in the standard way.

When using conditional lineage capturing this massive amount of execution time needed gets better but is always higher than if using mapping lineage data.

This is definitely the biggest drawback to using Approximate Provenance in its current form and would be a major contribution if it was to be solved.

7.2.3 Storage Footprint. If we compare the storage footprint of the lineage data when using bloom filters though, our expectations are met.

Figure 10 shows the storage overhead of the reduce operator using bloom filters and the standard mapping for our WordCount job with and without conditional provenance capturing.

It is important to note that we initialized our Bloom Filters to have an average false positive ratio of 3% whith 100 expected insertions per filter. As in our data sets most words occur far less than that, the bloom filters are usually much better than that.

This is apparent when we compare the lineage size where we send out every available lineage information. Here the Bloom Filter Lineage is actually larger than the corresponding mapping lineage files. This is due to the fact that the bloom filters always have a fixed size and we basically store a relatively large bloom filter for every distinct word in the data set.

When we look at the file sizes where we filtered out needless lineage data based on the query condition (only store lineage data for words which occur more than 100 times) we see that the bloom filter lineage only takes a fraction of the space needed by the mapping lineage.

In our experiments the Bloom Filter Lineage was sometimes using only 3% of the space needed by the mapping lineage.

In this regard Approximate Provenance is true to our expectations as it has the potential to massively decrease the file sizes of

lineage data. As with Conditional Provenance though, these benefits are different based on the underlying data and the provenance query itself.

Additionally the users of Approximate Provenance need to be able set the tuning parameters of the probabilistic data structures intelligently. In the case of bloom filters those are the expected numbers of insertions and the target false positive rate to determine the appropriate size of the filter.

8 CONCLUSION

This paper has provided a fair introduction to a new perspective in the study of Data Provenance. We have defined Conditional Provenance as a novel approach that improves the data provenance capabilities adding conditions in the data-flow operators in a Data-Intensive Scalable Computing (DISC) system job.

When we aim to implement a debugging system based on provenance abstraction capturing and storing lineage information, it is inevitable to commit in execution time and storage size overhead. We showed that even for simple word count examples, consuming relative small datasets, the full mapping approach of lineage capture could incur a massive amount of lineage-data. However, many debugging tasks, such as forward and backward tracing, don't need a full-map dataset when querying the provenance.

We claimed that taking advantage of the DISC system logic, it is possible to instrument the desirable operators adding conditions that reduce the amount of data captured and stored and consequently the provenance abstraction job overhead. These conditions are defined for the provenance queries to be executed.

To fully fill the experiment requirements, we leveraged the TagSniff abstraction model for capturing provenance. The paper describes how the logic behind TagSniff is flexible enough to allow users to instrument the underlying system for the analysis purpose. The model wraps the main operators in the execution job, abstracting the data-flow into *debug tuples* suitable for the provenance capture components.

A lesson learned from this paper is that, even though the results of the overhead and data size capture are very promising, our specific implementation is currently lacking in performance in the capturing process. For future work, it is necessary to engineer a better and more efficient way to capture and store the lineage data in order to be a real contender to current state of the art provenance capturing systems.

We also took the chance of presenting preliminary results for *approximate provenance*. During the process of our investigation, one of the solutions led to a different approach of mapping I/O results of each operator into probabilistic data structures instead of standard identification relationships for lineage capture. These results show that, although we increase the overhead in the tracing computation, the reduction of the size of the lineage output information when using conditional operators is astonishing. We argue that approximate provenance could be promising in specific use cases and we are willing to further investigate it, however it escapes the scope of the present work.

ACKNOWLEDGMENTS

We thank our Mentors at the BDAPRO module for their continued support and guidance. Without them this project wouldn't have been possible.

REFERENCES

[1] Eleanor Ainy, Pierre Bourhis, Susan B. Davidson, Daniel Deutch, and Tova Milo. 2015. Approximated summarization of data provenance. In *International Conference on Information and Knowledge Management, Proceedings*, Vol. 19-23-Oct-. 483–492. https://doi.org/10.1145/2806416.2806429

[2] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[3] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. [n.d.]. Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers".

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[5] James Cheney, Laura Chiticariu, and Wang-chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1 (01 2009), 379–474. https://doi.org/10.1561/1900000006

[6] Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, and Saravanan Thirumuruganathan. 2019. TagSniff: Simplified Big Data Debugging for Dataflow Jobs. *ACM Symposium of Cloud Computing conference* (2019). https://www.user.tu-berlin.de/quiane/assets/publications/socc19.pdf

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 10.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 29–43. https://doi.org/10.1145/1165389.945450

[9] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging primitives for interactive big data processing in. In *Proceedings - International Conference on Software Engineering*, Vol. 14-22-May-. ACM Press, New York, New York, USA, 784–795. https://doi.org/10.1145/2884781.2884813

[10] R Ikeda, H Park, and J Widom. 2011. generalized map and reduce workflows.pdf. http://ilpubs.stanford.edu:8090/985/

[11] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2016. Titian: Data provenance support in Spark. *Proceedings of the VLDB Endowment* 9, 3 (nov 2016), 216–227. https://doi.org/10.14778/2850583.2850595

[12] Clifford Lynch. 2001. When documents deceive: Trust and provenance as new factors for information retrieval in a tangled Web. *JASIST* 52 (01 2001), 12–17. https://doi.org/10.1002/1532-2890(2000)52:13.0.CO;2-V

[13] Christopher Olston and Benjamin Reed. 2011. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1221–1223. https://doi.org/10.1145/1989323.1989459

[14] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A Survey of Data Provenance in E-Science. *SIGMOD Rec.* 34, 3 (Sept. 2005), 31–36. https://doi.org/10.1145/1084805.1084812