Explaining Outputs in Modern Data Analytics*

Zaheer Chothia, John Liagouris, Frank McSherry, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zürich
firstname.lastname@inf.ethz.ch

ABSTRACT

We report on the design and implementation of a general framework for interactively explaining the outputs of modern data-parallel computations, including iterative data analytics. To produce explanations, existing works adopt a naive backward tracing approach which runs into known issues; naive backward tracing may identify: (i) too much information that is difficult to process, and (ii) not enough information to reproduce the output, which hinders the logical debugging of the program. The contribution of this work is twofold. First, we provide methods to effectively reduce the size of explanations based on the first occurrence of a record in an iterative computation. Second, we provide a general method for identifying explanations that are sufficient to reproduce the target output in arbitrary computations – a problem for which no viable solution existed until now. We implement our approach on *differential dataflow*, a modern high-throughput, low-latency dataflow platform. We add a small (but extensible) set of rules to explain each of its data-parallel operators, and we implement these rules as differential dataflow operators themselves. This choice allows our implementation to inherit the performance characteristics of differential dataflow, and results in a system that efficiently computes and updates explanatory inputs even as the inputs of the reference computation change. We evaluate our system with various analytic tasks on real datasets, and we show that it produces concise explanations in tens of milliseconds, while remaining faster – up to two orders of magnitude – than even the best implementations that do not support explanations.

1. INTRODUCTION

Work on data provenance describes methods for tracking and querying the dependence of individual outputs of a computation on specific inputs to the computation. Over the last decade, the idea of provenance received considerable attention in a wide range of application areas, including system debugging [34, 22], workflow analysis [3, 19], network management [33], and data integration [21], among others. At the same time, the scale and scope of modern data analytics have moved past what traditional databases were designed to handle. Modern analytic workloads involve complex iterative computations over rich graphs with billions of edges, while receiving updates in real time. Now more than ever, it is important to understand the even less obvious connections between the inputs and outputs of these computations.

Existing works on provenance are rich and varied, but each lies along a spectrum between (i) *eager* techniques which annotate each tuple with the inputs on which it depends [17, 32, 8], and (ii) *lazy* techniques which determine necessary input records by "inverting" operators in the computation [10, 2]. As their names suggest, the techniques trade-off runtime overhead against query latency; eager approaches must perform a great deal of work to compute and maintain output annotations, whereas lazy approaches may need to perform a substantial amount of work for each query.

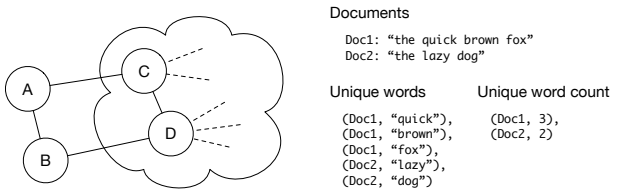
One compromise between the two is *backward tracing*, in which each operator in a dataflow graph maintains a reverse mapping from its output records to its input records, and the dataflow structure guides a trace backwards from queried output records to the input records on which they depend. Each operator eagerly maintains a reverse mapping, but the reverse mapping for the computation as a whole is computed lazily – only when requested – rather than being fully materialized. Backward tracing makes relatively few assumptions about the structure of the operators, maintains an amount of state no larger than the total data it has processed, and provides query latencies that depend on the size of the query result. However, the naïve application of backward tracing quickly runs into known issues, even for simple dataflows; backward tracing may identify (i) *too many inputs to be helpful* [14] and (ii) *too few inputs to reproduce the output* [19]. We give examples for both cases.

Too much information. Consider the problem of determining the connected components of a graph: we want to assign labels to each node so that nodes have the same label if and only if there is a path between them. This is core task in social network analysis. A common algorithm for this task assigns a distinct label to each node (often the node's id) and then repeatedly performs the following steps: (i) joins the label set with the set of edges in the graph (essentially sharing each node's label with its neighbors), and (ii) groups by edge destination to retain the minimum label for each destination node. After some number of iterations, each node will have been exposed to the smallest label in its connected component.

Figure 1a presents an example graph, in which node `B` would receive label `A` in the first iteration. Suppose that node A has the smallest id (0) among all nodes in the graph. Although the label `A` at node `B` never changes throughout the computation, it is recomputed in each round from the labels of its neighbors. Now suppose that we want to explain why node `B` has label `A`, i.e., why these two nodes are

*This work was partially supported by the "SOLAS" Project (EU Grant 612480), the "Data Center Model" Project (in collaboration with AMADEUS), and the "Online Data Center Modeling" Google Research Award (jointly with the University of Lugano).

Proceedings of the VLDB Endowment, Vol. 9, No. 12



(a) An example graph (b) An example corpus

Figure 1: Examples for the discussion of challenges

in the same connected component. In a naïve backward tracing from any output label, each neighbor contributes an input which depends in turn on inputs from its neighbors, and so on until the whole connected component is implicated. However, we know that there is a succinct explanation for why `A` and `B` are connected; specifically, the edge `(A,B)`. Existing systems [22, 34] with backward tracing support for the previous computation return the whole set of paths between `A` and `B`; this is prohibitive for large connected components and, more importantly, the explanation is very difficult for a human to digest.

The first contribution of this work is a backward tracing approach that identifies concise explanations for an output record of an arbitrary iterative computation, based on the first occurrence of the record in the computation. In the connected components example, our approach automatically reduces the explanation for "why `B` has label `A`" into a shortest path between the two nodes `A` and `B`.

Not enough information. Consider a collection of documents for which we want to compute the set of words unique to each document. This is typical step in text mining, widely used in many IR tasks. One algorithm to do so is to first produce all (*doc*, *word*) pairs, then group by *word* and retain only pairs in singleton groups, and finally group by *doc* and count the words in each group.

Figure 1b presents an example corpus where we remove all words with count greater than one, and then simply count the number of remaining words in each document. In the example, the resulting count for `Doc1` is 3. The existing backward tracing approaches report that only `Doc1` contributes to this count; however, re-running the whole computation on only `Doc1` results in a count of 4, because the record `(Doc1,"the")` is no longer too frequent. We can see that to reproduce the output we must also include `Doc2`, but for a more general text processing pipeline an analyst would need to write new queries to identify and diagnose the discrepancy.

Output reproduction is crucial for debugging purposes. The problem was first pointed out in [19] and, to the best of our knowledge, no viable solution existed until now, even for the case of DAG dataflows without iteration. Specifically, the central result of [19], a theorem about reproducing target outputs, is explicitly restricted to computations with either (i) one-to-one `map` operators and many-to-one `reduce` operators or (ii) at most one non-monotonic operation[1]. The "unique word count" example does not fall into any of these two classes because the workflow includes a one-to-many `map` operator – the operator that takes each input document and splits it into individual words – followed by two non-monotonic operations: (i) group by *word* and output only singleton groups, and (ii) group by *doc* and count the words in each group. As pointed out in the example, the discrepancy lies in the *intermediate* record that does not appear in the reference computation $C(I)$, but appears in $C(I^*)$, i.e., when *replaying* the computation using as input only the result of backward tracing from o = `(Doc1,3)`. To ensure consistency between the reference and the replayed computation, the authors of [19] proposed a quick fix in the *replaying process* (although this feature was not implemented in their system): all intermediate records that appear in the re-executed computation but not in the reference computation should be filtered out while replaying. Suppressing records ensures that each operator is correctly re-simulated, but does not solve the issue that the computation's input does not yield the desired output, which hinders the logical debugging of the program. In fact, all provenance-aware systems [19, 20, 3, 24, 34, 22, 1] provide explanations that guarantee the reproduction of the output only when the computation falls into the two classes we described.

The second contribution of this work is a generalized backward tracing approach that provides explanations sufficient to reproduce the output in arbitrary, even iterative, dataflows with any combination of monotonic and non-monotonic operators.

Our Approach. In this paper, we present a general system for explaining outputs in iterative data analytics. Our approach is based on *differential dataflow*, a data-parallel framework that supports iterative and incremental computations. In a nutshell, we express the explanation logic for each operator of differential dataflow (roughly `map`, `reduce`, `join`, and `iterate`) in the form of simple rules, and we implement these rules as differential dataflow operators themselves. This approach allows us to identify *when* (i.e., the points in the computation) and *how* a data collection changes, and provide explanations for only these few changes, rather than the rest of the unchanged data.

We evaluate our approach on several analytic tasks including: (i) complex fixed-point computations with mutual recursion, (ii) graph connectivity, and (iii) stable matching which is inherently non-monotonic. For the first set of experiments we use "hard" Datalog programs on real biomedical datasets whereas for (ii) and (iii) we use the LiveJournal (68M edges) and Twitter (1.4B edges) graphs. The results show that our system produces explanations in tens of milliseconds, for computations whose inputs are interactively updated, and remains faster – up to two orders of magnitude – than even the best provenance-free non-interactive implementations.

2. PRELIMINARIES

Our work is strongly influenced by previous work on database provenance. We highlight this connection in Section 2.1, where we also clarify several important concepts used throughout the paper. In Section 2.2 we provide an overview of *differential dataflow* which serves as a basis for our framework in Section 3.

2.1 Concepts of Data Provenance

The idea of tracking the dependencies between the individual inputs and outputs of a computation was first introduced in the database field under the term *data lineage* [11]. The focus of the first works was to provide lineage support for SPJUA queries, i.e., relational queries with Selection, Projection, Join, Union, and Aggregation [10]. Given a tuple t in the output of a query q, the lineage of t is the set of all tuples from the input relations of q that contribute in having t in the result. [6] made one step further by classifying different concepts of lineage under the term *data provenance*. This work opened a whole new area of research by introducing *Why* and *Where* provenance, which were later related to fundamental optimization problems. More recently, [18] formalized the term *How* provenance in the form of *semirings*.

Consider the example in Fig. 2 with the simple database D and the query q on the left. This query asks for the names and addresses

[1]According to [19], a data transformation T is monotonic if for any I_1, I_2 with $I_1 \subseteq I_2$, then $T(I_1) \subseteq T(I_2)$.

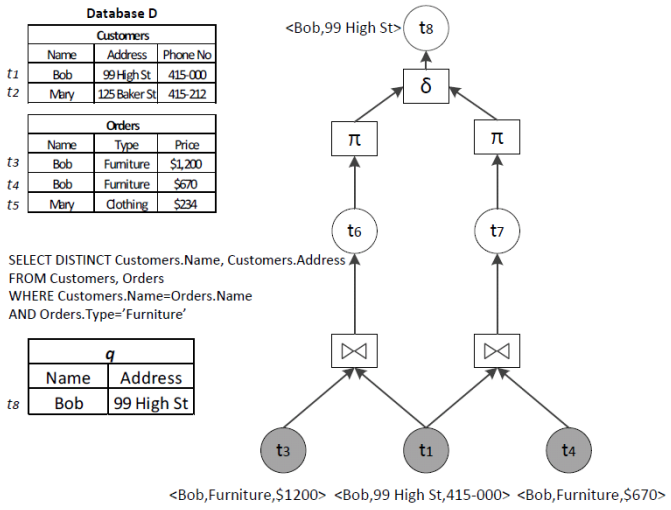


Figure 2: Example of a Provenance Graph G

of all customers with an order of type `Furniture`. The result of q on D contains the tuple t_8 = <`Bob`, `99 High St`>, i.e., $q(D) = \{t_8\}$. Now consider the graph G shown on the right of Fig. 2, which we will use to clarify the different notions of provenance in the database literature. This graph illustrates the flow of individual tuples throughout the evaluation of q, and it is also known as the *Provenance Graph*. Circular nodes in G represent input, output, and intermediate tuples whereas rectangular nodes denote operations on one or more tuples; in our example, nodes labeled with $\bowtie$ denote the joins of pairs (t_1,t_3) and (t_1,t_4) that produce the intermediate tuples t_6 and t_7 respectively. Nodes labeled with π denote the projection of fields `Name` and `Address` from t_6 and t_7, both resulting in the same output tuple t_8.

The *lineage* of t_8 is the set of highlighted tuples $\{t_1, t_3, t_4\}$ from the two input relations `Customers` and `Orders`. The *Why* provenance of t_8 is the multi-set $\{\{t_1, t_3\},\{t_1, t_4\}\}$. Each of the two subsets $\{t_1, t_3\}$ and $\{t_1, t_4\}$ corresponds to a *proof* or *witness* of t_8, i.e., a subset of the input tuples that are sufficient – but not necessary – to have t_8 in $q(D)$. Note that the *lineage* is the union of data elements in the *Why* provenance. According to [18], the *How* provenance of t_8 is expressed as a semiring of the form $t_1 \cdot (t_3 + t_4)$, where $t_i \cdot t_j$ means "both t_i and t_j" whereas $t_i + t_j$ is "either t_i or t_j". In general, *How* provenance is more informative than *Why* provenance and the former can be used to generate the latter. On the other hand, *How* provenance provides limited information about the data transformations. In our example, this information is fully captured by the provenance graph G; in fact, G is a more "verbose" form of *How* provenance, called *transformation* provenance [16]. *Transformation* provenance describes all possible ways of having t_8 in $q(D)$ – just like *How* provenance – but also includes the actual query operators that are involved in this process. Finally, the *Where* provenance is defined at the attribute level; the *Where* provenance of t_8.`Address` is t_1.`Address`, i.e., the field of the input tuple t_1 from which the value of the attribute `Address` is propagated to the output tuple t_8.

Our work focuses on tracking, maintaining, and querying *lineage* and *How* provenance in the context of arbitrary iterative dataflows. The notion of explanations we also consider in this paper is related to the *Why* provenance described in the previous. Intuitively, a witness of t_8 "explains" the existence of t_8 in $q(D)$ although it does not provide any information about the process that led t_8 in $q(D)$. Witnesses are useful in various data management tasks, including the logical debugging of complex data analytics. A witness/explanation of an erroneous output record can be used as a much smaller input to the computation in order to produce the erroneous record much faster and with less resources (e.g., nodes in a cluster) than when using the whole input. Although a witness is in the worst case as large as the input to the computation, our evaluation with different analytic tasks in Section 4 demonstrates that the size of a witness/explanation is in practice very small (less than 1% of the whole input).

2.2 Differential Dataflow

Differential Dataflow [26] (DD) is a data-parallel programming and execution model, layered on top of the timely dataflow model implemented in the Naiad system [27]. Differential dataflow was originally designed to support incremental updates to the inputs of iterative data-parallel computations. Here we summarize how differential dataflow supports efficient updates (and at what cost), both to inform our adaptation of backward tracing to differential dataflow (Sections 3.2 and 3.3), and to explain the robust performance of our general implementation in Section 4.

2.2.1 Computational Model

Differential dataflow models computation as a data-parallel dataflow over collections of records. It includes standard operators such as `join`, `group`, `map`, and `filter`, but also supports an `iterate` operator for (arbitrarily nested) iterative computations. Each operator is *data-parallel* and *functional*: its output can be determined by partitioning its input records according to some *key* and applying a function independently to each part. For example, one can implement a data-parallel graph reachability computation using the `iterate` operator whose body repeatedly joins the collection of reachable nodes (`reach`) with the collection of edges (`edges`), and retains distinct elements (`map` is used to project `b`):

```
initial.iterate(|reach| {
    reach.join(edges)
         .map(|(a,b)| b)
         .distinct()
})
```

Differential dataflow transparently provides *incremental* execution. Although the program above appears to repeatedly re-execute the `join` on the full collection `reach`, the implementation is actually delta-based; work is only performed as the collections `reach` and `edges` change. This allows differential dataflow to efficiently implement iterative computations, and also to update those computations as their inputs change.

Internally, each collection of records is represented by a *trace*, a monotonically growing set of *update* records that describe the collection's content at various points in the computation. These updates explain how records in the collection change as the computation proceeds. An update record is always defined for a specific `collection` and has the following form:

```
collection(record,c,t)
```

where: (i) `record` is the record that is inserted or removed into/from the `collection`, i.e., the actual content of the update, (ii) `c` $\in \mathbb{Z}$ is the change in the count (number of occurences) of the record in the `collection`, and (iii) `t` is the *logical time*, i.e., the point in the computation at which the update occurs. The logical time may indicate a round of input data (e.g., in a streaming scenario where data are given in batches), a number of iterations around a loop, or arbitrary combinations of these – for example, describing a change at some round of input within multiply nested loops. The collection

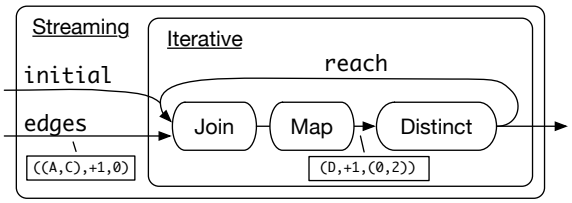


Figure 3: A sketch of the differential dataflow graph for the reachability example.

traces maintain the property that, at any logical time t, a collection is determined by accumulating all update records with a time $t' \leq t$.

Figure 3 depicts the differential dataflow graph for the previous reachability example. Consider the input collection of the `distinct` operator, which consists of nodes produced by the previous `map` operator. Suppose that, in the first round of edge data and first iteration around the loop, this `collection` consists of the nodes "a", "b", and "c", written as update records:

```
collection("a", +1, (0,0))
collection("b", +1, (0,0))
collection("c", +1, (0,0))
```

The +1 are the count values that indicate a change from the baseline (empty) collection. The `(0,0)` is the logical time that indicates the first round of input data (round 0) and the first iteration (iteration 0). The logical time contains loop counters for each nested loop in the computation, though in this case there is only one. Suppose that, in the second iteration of the first round of edge data, the computation adds node "d" as well as a second copy of "c", introducing updates whose logical time now reflects the second iteration:

```
collection("d", +1, (0,1))
collection("c", +1, (0,1))
```

In case there are no further changes, the computation quiesces, and the results of `iterate` are reported. However, suppose that in the second round of input data an edge is removed from `edges`, and the first iteration of the second round starts with a changed input for `distinct`: node "c" is now removed. This may lead to a second iteration of the second round in which "d" is also removed and added again in the third iteration. In this hypothetical scenario, the update records for the input trace of `distinct` would be:

```
collection("c", -1, (1,0))
collection("d", -1, (1,1))
collection("d", +1, (1,2))
```

In this case, the only update records we see relate to the actual changes that occur. Records corresponding to `"a"` and `"b"` are not re-considered if they do not change. Although this property is common to all systems with delta-based updating (for example, in Datalog engines that support *semi-naive* evaluation [23, 22]), differential dataflow's explicit use of timestamps extend it to arbitrary iterative computations.

Each differential dataflow operator maintains its input updates indexed by key, and reacts to new inputs by ensuring that its output updates properly describe a collection that reflects the operator's logic applied to the collection described by its input updates. The operators continue receiving and producing updates until all updates have either dissipated or reached the outputs of the dataflow (in our example, the output of `iterate`). The communication between operators and the memory footprint are both bounded by the total number of update records circulated in the dataflow over the course of the computation.

2.2.2 Implications for Explanations

The framework we present in Section 3 relies on the ability of differential dataflow to efficiently execute and update iterative data-parallel computations. Explanations are produced by iteratively updating a fixed-point implementation of backward tracing. Differential dataflow not only efficiently computes these explanations, but it also updates them automatically as the source computation changes.

Additionally, our adaptation of backward tracing to differential dataflow's updates (Section 3.2) is based on three important properties: (i) the graph-based structure of differential dataflow, which allows us to easily determine the source of an operator's input, (ii) the functional nature of each operator, which means that an output at a logical time is fully explained by the operator's input at that logical time, and (iii) the data-parallel nature of operators, which means that, for each operator, we can restrict the relevant inputs for an output to those with the same key. These three properties allow us to restrict the number of updates in the provenance information and avoid redundant computations.

3. A FRAMEWORK FOR EXPLANATION

In this section we provide the details of our approach. In Section 3.1 we present our general framework for tracking provenance in data-parallel operators. In Section 3.2 we specialize this framework for differential dataflow operators, which can be used to define arbitrary iterative computations, including those expressed in Datalog. Finally, in Section 3.3 we introduce our generalized backward tracing approach that identifies meaningful explanations in settings where the naive backward tracing fails.

3.1 Explaining Data-parallel Operators

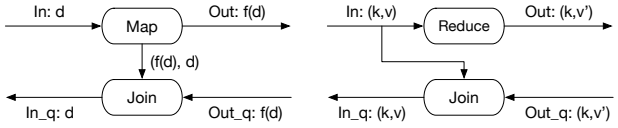
Our goal is to create a set of recursively defined rules that determine which records in a general data-parallel computation – outputs, intermediate records, and inputs[2] – require explanation. To this end, we consider a simple framework where all records have the form (k, p) representing a key k and an associated payload p. The payload may contain a value (data) along with additional metadata. In this framework we define two operators: (i) a linear `map` operator, which applies arbitrary logic on a record-by-record basis, and (ii) a `reduce` operator, which maps sets of records with the same key k to output sets with that key. The `map` operator can associate arbitrary keys with its outputs (one-to-one or one-to-many), which the `reduce` operator can then group and act on appropriately. In the following section, we will also present (optimized) versions for operators like `join`, `top-k`, `distinct`, and invertible `map`.

For each collection of records `col` in the computation, we introduce another collection `col_q` of the same type, containing records from `col` that require explanation. To define the contents of this new collection, for each data-parallel operator with input collection `col` and output collection `out`, we introduce a new `join` operator whose inputs are `col` and `out_q`, and whose output is `col_q`. Intuitively, the role of this `join` operator is to select elements of `col` matching requests in `out_q`. The result is a shadow copy of the dataflow graph, with the direction of edges reversed; the inputs in the reversed dataflow are sets of output records to explain, and the outputs are the required inputs in the original computation.

The explanation of an operator's output is the set of input records that produced it. Without any assumptions on the logic of the `map` operator, we can only record all observed pairs of input and output records and index these pairs by output. The `reduce` operator has a data-parallel structure, meaning that we only need to index the

[2]An explanation for an input record is the record itself.

input by key, and use the key when a request is made for explaining a record in the output. The two constructs are depicted in Figure 4. We emphasize that, although the general `map` and `reduce` operators do maintain a second copy of their input – which can be expensive – we have several opportunities to reduce this overhead for common data transformations, as we show in the following section.



Map operator Reduce operator

Figure 4: Generic provenance tracking constructs

Discussion. Expressing the explanation logic as a join that matches queried outputs against explanatory inputs has two major advantages. First, we do not rely on specialized data structures for maintaining provenance; hence, our approach can be easily incorporated into any system. Second, the explanation join can be implemented as a data-parallel operator itself to achieve robust and scalable provenance querying. In contrast to existing works that query provenance outside the data-parallel system [19, 3, 24], our implementation uses the built-in `join` operator of differential dataflow, which also allows for efficient incremental evaluation.

3.2 Backward Tracing in Differential Dataflow

Besides the general `map` and `reduce` operators, differential dataflow also introduces a data-parallel `join` and an `iterate` operator.

The `join` operator can be implemented as a binary `reduce` that takes a pair of inputs `In1`: (k, v_1) and `In2`:(k, v_2), and produces an output of the form `Out`: $(k, (v_1, v_2))$. Although we could implement provenance tracking with the generic construct for the `reduce` operator from Section 3.1, there is a significant optimization to apply. The join has a special structure: its output bindings reveal exactly the input records responsible for it, hence, we do not need to index the inputs for the join because they can be recovered directly from the outputs by projecting out the relevant attributes. The optimized construct is shown on the left of Figure 5.

The `iterate` operator is used to define arbitrary loops on collection of records, and breaks into three operators: `enter`, `feedback`, and `leave` [27]. These operators are applied on a record-by-record basis and manipulate only the logical timestamp of the input record[3]; respectively, extending the timestamp with a new coordinate, advancing the coordinate, and removing the coordinate. Each of these three operators can be treated using the same construct as with `map` in Section 3.1, and we will even see that the first two can be optimized further as invertible `map` operators (Section 3.2.3).

We are now able to show how our framework naturally captures existing definitions of provenance for Datalog computations. To the best of our knowledge, this is the only class of iterative computations that have been studied so far in the context of data provenance.

3.2.1 Provenance for Datalog

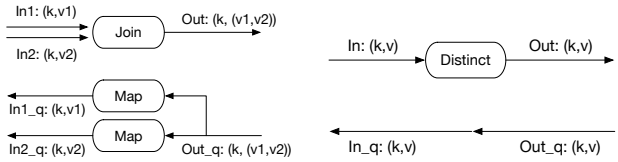
A Datalog computation is a collection of recursive rules, each of which defines how new elements of a relation may be produced from existing elements of other relations. By re-using variables in these rules, the source relations are implicitly joined. A classic example determines the transitive closure `reach` of a directed graph defined by a relation `edge` as:

```
1. reach(x,y) := edge(x,y)
2. reach(x,z) := reach(x,y), edge(y,z)
```

An intuitive way to compute provenance for output tuples is the approach followed by [22, 34], where each rule is rewritten as first determining a binding of all variables involved, and then yielding the contribution to the output relation. For example, the second rule above can be rewritten as:

```
bind(x,y,z) := reach(x,y), edge(y,z)
reach(x,z)  := bind(x,y,z)
```

It is easy to see that each Datalog rule producing a binding is a join of the participating relations and can be implemented with a `join` operator. Each rule projecting attributes from a binding can be simply treated as a `map`. In Datalog, the projections for the same relation are merged and followed by a `distinct` operator, which we can be also expressed as a `reduce`. However, although the distinct operator is a form of `reduce`, it does not need to track its inputs: each requested output record can simply be passed backwards as a required input record. The optimized construct for the `distinct` operator is shown on the right of Figure 5. With this optimized construct, the only operators that require indexed state for provenance are the projections, which must maintain maps from projected tuples back to the full bindings that produced them.



Join operator Distinct operator

Figure 5: Optimized provenance tracking constructs

3.2.2 Reducing the Provenance Size

So far we have focused on the backward tracing constructs without considering the time dimension that is natively supported in differential dataflow. In fact, the most important change we make in our explanation logic is that the inputs of an operator can be restricted to those with logical timestamp less-or-equal to that of the required output. This allows us to put a filter on `in_q` in the `reduce` construction in Figure 4, and substantially reduce the volume of records in the reported provenance. The effect is that we only see records that existed when the output was first produced; later inputs may also be relevant for the full provenance in the logical sense, but they did not participate in producing the target output in the differential dataflow computation.

3.2.3 Additional Optimizations

Several differential dataflow operators admit optimizations that can simplify their implementations, requiring less computations and memory. In each case, the optimizations take the form of slightly more interesting explanation rules, which we use instead of the generic rules of Section 3.1. We expect that other optimizations could be added similarly, but we leave this for future work.

Invertible map operators. We see many instances of invertible `map` operators used to rearrange which fields of a tuple are the key

[3]Recall from Section 2.2 that each record in differential dataflow is associated with a logical timestamp that denotes the point of the computation the record was produced.

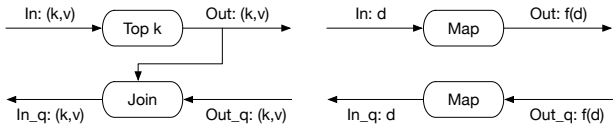


Figure 6: Additional optimized provenance tracking constructs

and which are the value. An invertible `map` operator doesn't need to maintain the mapping from output to input record explicitly since it can be computed using the inverse function. For example, this optimization applies to `enter`, which adds a timestamp coordinate that is easily removed, and to `feedback`, which increments a timestamp coordinate that can simply be decremented. The optimized construct is shown on the right of Figure 6.

Top-k. We often see the generic `reduce` operator used with logic that looks at only the first element when ordered by some metric. This happens in connected components, where each node selects the smallest value it sees, and in stable matching, where each node selects proposals by rank-order. More generally, operators that just restrict each group to the top-k elements do not need all inputs to explain outputs; the outputs (which are themselves inputs) suffice. Not only are they sufficient, but the other records are literally never exposed to user logic; the outputs cannot depend on them if they are never observed. The optimized construct is shown on the left of Figure 6. In Section 4.2, we will see how the top-k optimization can effectively reduce the size of the explanation in the computation of connected components of a graph.

Additional optimizations are given in the extended version [9].

3.2.4 Space Overhead

Provenance tracking does not come for "free", meaning that additional state has to be kept by the operators, as we explained in the previous section. We emphasize that the incurred space overhead increases linearly to the total amount of data in the computation (including the intermediate records), and this overhead is the same in all existing systems with fine-grained provenance tracking. In this sense, we do not invent a new approach for storing provenance-related information here; our contribution lies in the time-based filtering of records (Section 3.2.2), and the techniques we described for optimizing the naive materialization of dependencies in specific operators (Figures 5 and 6).

3.3 Iterative Backward Tracing

In general data-parallel computations, the input records identified by backward tracing may not be sufficient to reproduce the output. As pointed out in [19], the issue lies in *intermediate* records that appear in the re-executed computation but are not seen in the reference computation. These records may interfere in downstream computation and change the results of operators, suppressing important outputs. This problem doesn't occur in dataflows of monotonic operators, as they have the property that additional input records only lead to additional output records.

Although it appears difficult to identify the records that deviate from the reference computation early, we can discover them through their main defective property: *to affect the output they must intersect the backwards trace*. If we maintain a second instance of the computation, re-executed only on the inputs identified by the backwards trace, then any new records it produces that intersect the backwards trace can themselves be traced backwards, and ultimately corrected

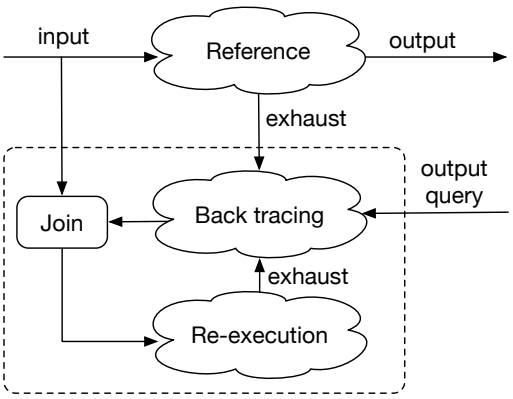


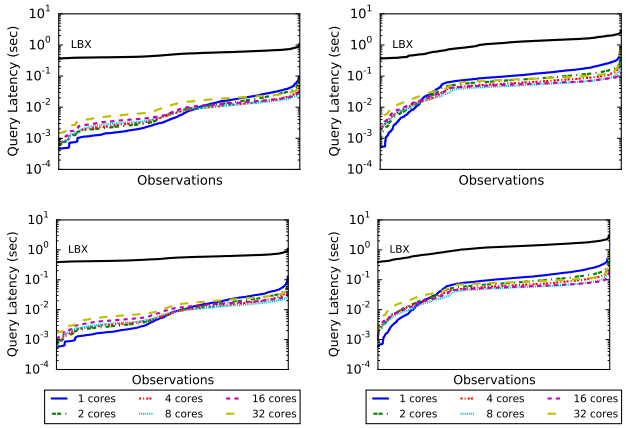
Figure 7: Iterative process to find explanations. The backward tracing and re-execution is iterated to a fixed-point for each instance of the reference computation. Any changes to the input and reference computation propagate to changes in the iterative explanation finding process.

through the introduction of more input records. Concretely, we introduce a second copy of the dataflow graph, structured as for the reference computation. For each of our explanation rules, we merge the streams from the two computations leading into the explanation logic. This has the effect of propagating requirements for explanations of inputs in either the test or the reference computation. Each new record intersecting the backwards trace is added to the backwards trace and traced backwards itself, possibly resulting in more required inputs, and this is repeated until convergence. When the fixed-point is reached, the set of explanatory input records collected by the iterative backward tracing is sufficient to reproduce the output we wanted to explain. Figure 7 sketches the overall process.

For the approach to be useful, it must also be reasonably efficient to re-execute both the computation and the backward tracing on the test inputs. Here we rely on the incremental updating infrastructure of differential dataflow, which is designed to handle this task. The re-execution of the backward tracing operates by *propagating changes*: newly introduced requirements are propagated, but old requirements are not re-processed. The same delta-based approach is followed for the re-execution of the computation.

Example. Let us go through the example of Figure 1b in the introduction to make the process more concrete. Recall that we have a document corpus and we want to compute, for each document, the number of its terms that are unique in the corpus. The output we want to explain is `(Doc1,3)`, and each of the three contributions traces back directly to `Doc1`. Including only the input `Doc1` creates the three records we are interested in, but also creates a fourth record: `(Doc1,"the")`. At this point, the backwards trace may have changed (we have new records) and so it is re-performed. We discover the new record in the backwards trace, which turns into a request to explain `(Doc1,"the")`. The record is the result of the `reduce` operator, which keeps only singleton groups, and the inputs associated with the key `"the"` in the reference computation include both `(Doc1,"the")` and `(Doc2,"the")`. The latter record leads back to `Doc2`, which is now included in the set of inputs that are sufficient to explain the output.

Convergence. The process we described increases the set of explanatory input records monotonically, with an upper bound the



(a) SNOMED CT (b) GALEN8

Figure 8: Observed latencies for 1,000 How-provenance queries (top) and 1,000 lineage queries (bottom), for the Datalog program of Section 4.1 with varying number of workers. The lineage is derived from the How provenance, thus, it has a small additional latency. LBX plots refer to the latencies of the LogicBlox system.

whole set I of input records of the reference computation, so it does converge. Given a reference computation f that converges to a fixed-point (for any subset of the input I), and an output $O' \subseteq f(I)$ of the computation that has to be explained, our iterative backward tracing performs the following steps: (i) traces O' backwards and collects a set of input records $I' \subseteq I$, (ii) re-executes the second copy of computation only for I', (iii) identifies new records that are not seen in the reference computation, (iv) traces these records backwards to a set of input records $I'' \subseteq I$, and (v) repeats the same process for the set $I'' \setminus I'$, until a fixed-point. The latter is reached due to the finite number of possible new records produced by the second copy of the computation; these records are finite since the reference computation converges on any subset of the input I.

Correctness. We still need to argue that, when the iterative backward tracing reaches a fixed-point, the output is correctly reproduced. Let $f(I) = O$ be the reference computation with input I and output O, and $O' \subseteq O$ be the output we want to explain. Let also $I' \subseteq I$ be the input records returned by the backward tracing. Assuming that O' is not included in $f(I')$, then along the backwards trace there is some first point of divergence between the reference and re-executed computations: an operator, key, and timestamp for which the inputs are not the same. The inputs from the reference computation must be present, as they are part of the reference trace and this is assumed to be the first point of divergence; the difference must lie in intermediate records of the re-executed trace. However, a backwards trace from the different intermediate records should have included additional input records from I, otherwise it would agree with the reference computation in which the intermediate records do not exist. The following holds: $f(I') = O''$ where $O' \subseteq O''$.

4. EXPERIMENTAL EVALUATION

This section provides an experimental evaluation of our provenance tracing techniques. To highlight all aspects of our approach, we divide the experiments into three classes. In the first class (Section 4.1), we evaluate our backward tracing with *lineage* and *How* provenance queries on iterative computations with complex mutual recursion expressed in Datalog. In the second class of experiments (Section 4.2), we evaluate our backward tracing on the problem of graph connectivity to show how we can efficiently reduce the size of explanations using the techniques of Section 3.2. Finally, in Section 4.3, we evaluate the iterative backward tracing of Section 3.3 on a representative non-monotonic computation, i.e., stable matching in graphs, and we show that, although explanations require many iterations of tracing in this case, they are still largely concise and interactive. In all experiments, we also measure the overhead caused by provenance tracking, and we compare the performance of our system with the state-of-the-art implementations.

Dataset	Input facts	Derived facts	Derivations
SNOMED CT	1,030,336	12,970,258	238,078,602
GALEN8	976,552	24,483,561	441,464,399

Table 1: Size of the provenance graph (cf. Section 2.1) after evaluating the Datalog program of Section 4.1 on each dataset. Facts and derivations correspond to nodes and edges resp.

Implementation and Setting. Our prototype was implemented on top of the open-source prototype of Differential Dataflow compiled with Rust v1.4. All experiments of this section were done on a machine with an AMD Opteron 6378 at 2.4GHz, 16 physical cores (32 with Hyper-Threading) and 512GB RAM, running Debian v7.9.

4.1 Provenance in Datalog

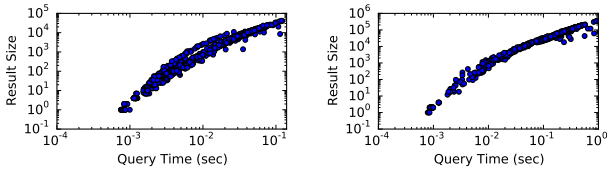
For our Datalog experiments we use two real-world ontologies, SNOMED CT [15] and GALEN8 [29], which are the standard ontologies for representing and exchanging knowledge in the biomedical domain. Each of these ontologies describes a schema for medical terms with complex dependencies, which can be partially expressed by the following mutually recursive Datalog rules:

```
p(X,Y)   :- p_initial(X,Y)
q(X,P,W) :- q_initial(X,P,W)
p(X,Z)   :- p(X,Y), p(Y,Z)
p(X,W)   :- p(X,Y), p(X,Z), c(Y,Z,W)
q(X,Y,Z) :- p(X,W), q(W,Y,Z)
p(X,W)   :- q(X,R,Y), p(Y,Z), u(R,Z,W)
q(X,P,W) :- q(X,R,W), r(R,P)
q(X,Y,Z) :- q(X,R,W), q(W,P,Z), t(R,P,Y)
```

These rules are closely related to the task of *ontology classification*, and generate a large number of results (facts) given in Table 1.

Provenance Queries. The results of the above Datalog computation are derived facts in relations `p` and `q`. The *How* provenance of a fact is the graph of facts (both initial and derived) containing all derivations of that fact. A derivation indicates not only the facts involved, but also the rules that were applied. The *lineage* of a fact is the intersection of the set of facts in its *How* provenance with the input set of facts. Figure 8 presents the latencies to produce the *How* provenance and *lineage* for 1,000 randomly chosen facts in relation `p`. The queries on the x-axis are given in ascending order of their latencies from left to right. The observed latencies for our system (skip the LBX latencies for now) range from sub-millisecond, in the best case, up to 100 milliseconds for SNOMED CT, and one second for GALEN8. The latencies smooth out as more cores are used, because the work required for the more expensive queries can be effectively shared among multiple workers. Figure 9 plots query latencies against result sizes, and demonstrates that the variation in query latencies is explained by the variation in the result sizes. The latter can be as large as 10^5 for SNOMED CT and 10^6 for GALEN8.

Provenance Overhead. Our provenance tracking requires the materialization and indexing of the intermediate results of the computa-



(a) SNOMED CT (b) GALEN8

Figure 9: Query result sizes plotted against query latencies on one core for How-provenance.

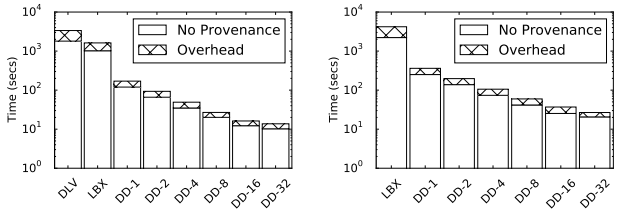
tion (variable bindings for rule firings in the case of Datalog) which has an impact on derivation time. Figure 10 presents the derivation times with and without provenance tracking. Our implementation (DD-x) exploits the parallelism of differential dataflow, and scales well with the number of threads (from x=1 to x=32), imposing an approximately fixed relative overhead. For SNOMED CT, provenance tracking increases derivation time from 42% (1 thread) to 32% (32 threads). For GALEN8, the results are similar: from 44% (1 thread) to 29% (32 threads).

Comparison with other systems. Figure 10 also presents derivation times for two state-of-the-art Datalog engines: single-threaded DLV [23], and multi-threaded LogicBlox (LBX) [4] that was allowed to use all resources of the machine. These systems do not have native provenance support, but we implemented the techniques in [22], which essentially keep the same intermediate results as we do. Overall, our approach is up to two orders of magnitude faster than even the provenance-free versions of these engines. Finally, Figure 8 also presents the LBX latencies for the same 1000 *How* provenance (top) and *lineage* queries (bottom) as in our system. In these experiments, our system is on average an order of magnitude faster than LBX for both datasets. Note that DLV does not support queries after the input Datalog program reaches a fixed point, and did not even reach the fixed point on GALEN8 after 12 hours.

Incremental Updates. Provenance tracking affects the derivation time also in the case of updates. Figure 11 present the observed latencies when applying 1,000 batches of 10 updates to the set of base facts for each dataset. The updates on the x-axis are given in ascending order of their latencies from left to right. For these experiments, we introduced all but U=5,000 facts, chosen randomly; in each batch of updates we added 5 random facts from U and removed 5 random facts from the existing ones. The largest latency we observed is 62.18s, for a single core on batch size 100 for GALEN8 (see [9]), but generally the update latencies are both small and not much larger than without provenance tracking; 99% of the latency measurements increase by less than 40%, and the time to process all updates increases by at most 37%. The extended version of this paper [9] provides the latencies for updates of various sizes, the update latencies without provenance, and the point-wise latency ratios with and without provenance.

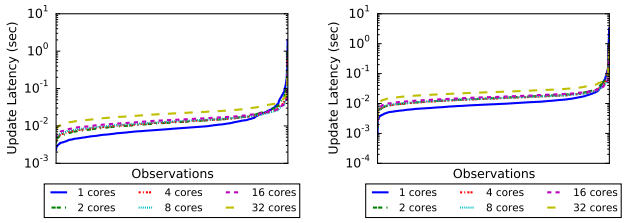
4.2 Explanations for Connected Components

We now study the problem of providing explanations for the computation of connected components in a graph. This task highlights the first challenge from the introduction: applied naïvely, backwards tracing produces explanations of enormous sizes. We will show that our approach reduces the data dramatically, to the point that the connectivity between two nodes is explained by a shortest path between them. We use a variation on the well-known "label propagation" algorithm: each graph node maintains a label (initially its own identifier), and repeatedly exchanges its label with neigh-



(a) SNOMED CT (b) GALEN8

Figure 10: Derivation times for the Datalog program of Section 4.1 with and without provenance tracking, for several systems and configurations.



(a) SNOMED CT (b) GALEN8

Figure 11: Observed latencies to perform 1,000 updates of size 10 to the input of the Datalog program of Section 4.1.

bors, updating the label each time it receives a smaller value, i.e., a smaller node id. In the pseudo-code below, we start from initial labels `init`, and repeatedly join the `values` with the set of `edges`, change the tuples to be keyed by destination `dst`, add these tuples in the original `values`, and then keep only the first (smallest) record for each destination node.

```
init.iterate(|values| {
    values.join(edges)
          .map(|(src,(lbl,dst))| (dst,(lbl,src)))
          .concat(values)
          .topk(1, |(lbl,src)| (lbl,lbl))
})
```

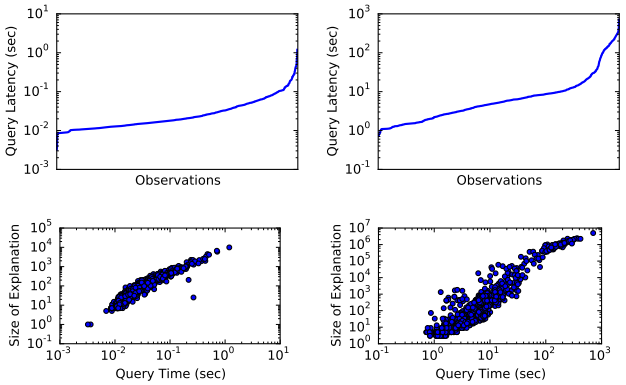
The properties of the datasets we used in the experiments of this section are shown in Table 2.

Dataset	Nodes	Edges
LiveJournal	4,847,571	68,993,773
Twitter	41,652,230	1,468,365,182

Table 2: Graphs used in the experiments of Section 4.2 and 4.3.

Remark. In the above algorithm, labels are introduced in iterations that depend on their magnitude, rather than all in the first iteration. This technique accomplishes the same goal as asynchronous systems, like Myria [31] and SociaLite [30], that propagate successful labels.

Explanation Queries. The result of the label propagation algorithm includes pairs (*node*, *label*), where *label* denotes the smallest node identifier contained in the connected component of *node*. An explanation of an output pair using naïve backward tracing includes the edges of all paths from *label* to *node*, which in a highly connected graph would be roughly the full set of edges. This approach does not terminate on the graphs of Table 2 in reasonable time (we used a 5-minutes timeout per query), as it does not apply the filtering of Section 3.2.2 and, hence, considers too many redundant edges. Figure 12 presents the query latencies (top) and the correlation between query latency and explanation size (bottom) for the label

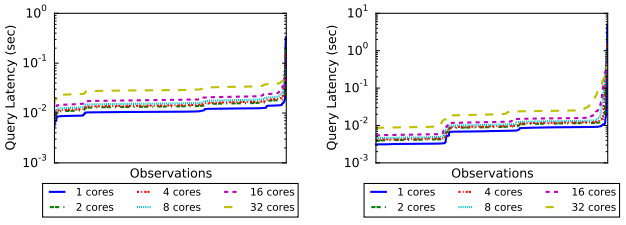


(a) LiveJournal (b) Twitter

Figure 12: Explanations for label propagation using the generic `reduce` explanation logic. The top figure plots the observed latencies for 1,000 output explanations using one core; the bottom figure plots the query latencies against the explanation sizes.

propagation algorithm, using the generic `reduce` explanation logic of Section 3.1. The queries on the x-axis of the two plots on top are given in ascending order of their latencies from left to right. The observed latencies are manageable for LiveJournal, but still too large for Twitter, even though the input of the `reduce` operator is filtered on the logical time as we explained in Section 3.2.2.

The `topk` optimization of Section 3.2.3 resolves this problem by retaining only the path which followed backwards has the least node identifiers. The result is a simple shortest path from *label* to *node*. The derivation of this path is roughly equivalent to repeated pointer chasing: looking up a key in the `topk` explanations, projecting out the current node id as the explanation request passes through the `join`, and repeating with the new node id as a key. All explanations are paths with size bounded by the diameter of the graph. In practice, we see explanation sizes from three to five edges, and this is consistent with our understanding of large social networks as having a small diameter. Figure 13 presents query latencies for the explanations of 1,000 randomly chosen outputs of the connected components computation, using the `topk` optimization. Again, the explanation queries on the x-axis are given in ascending order of their latencies from left to right. We see that the latencies are consistently small, with the exception of a few queries that upon inspection are among the first issued, suggesting some warm-up issues.



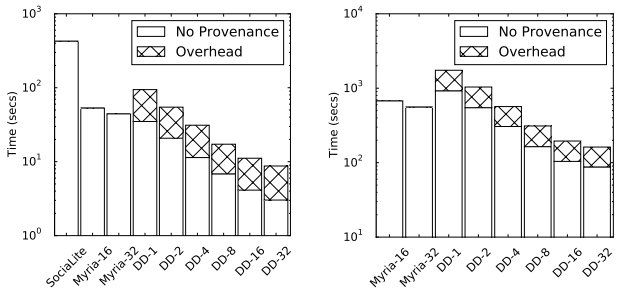
(a) LiveJournal (b) Twitter

Figure 13: Observed query latencies for 1,000 output explanations for the label propagation algorithm. Response times are largely within 10 milliseconds, other than a small number of initial samples. The single-core latencies are consistently smallest, because there is almost no parallel work to perform.

Explanation Overhead. Figure 14 shows the execution times with and without provenance tracking for connected components on the LiveJournal and Twitter graphs for our approach (DD) with 1 to 32 threads. The relative overheads for LiveJournal range from 169% (1 thread) to 188% (32 threads). The relative overheads for Twitter range from 88% (1 thread) to 84% (32 threads). These overheads are large because the reference connected components computation is well optimized: its data-parallel operators use dense integer keys and avoid hash maps, whereas in provenance-tracking mode we spend the bulk of the time populating hash maps with input records. The overheads of provenance tracking in Figure 10 are much smaller because the reference Datalog computation uses the same hash join.

Comparison with other systems. Figure 14 also depicts the performance of the state-of-the-art systems for graph analytics, Myria [31] and SociaLite [30], on the label propagation algorithm. Note that SociaLite does not accept the number of workers as a parameter. In addition, we were unable to get SociaLite to report more than 1.7M labels for the Twitter graph, thus, we do not report its elapsed time for this dataset. As a general comment, these systems do not support provenance tracking or incremental computations. Still, our system (DD-x) is 50x faster than Socialite on LiveJournal, and 4x faster than Myria on both datasets using x=32 workers.

Incremental Updates. Figure 15 presents the observed latencies to perform 1,000 updates of size 10 to the LiveJournal and Twitter graphs for the label propagation algorithm. The updates on the x-axis are given in ascending order of their latencies from left to right. For these experiments, we introduced all but U=5,000 edges, chosen randomly; in each batch of updates we added 5 random edges from U and removed 5 random edges from the existing ones. The largest update time we observed is 2.9s, for 32 threads with batch size 100 on Twitter (see [9]). Although 1% of the updates increase by more than 20x, the update times remain interactive, typically tens of milliseconds. Excluding the highly-variant 32-thread measurements, the time for all updates increases by at most 8.6x. These large relative numbers are due to the optimized nature of the reference computation; the update latencies are within 10 milliseconds for each batch. The extended version of this paper [9] provides the latencies for updates of various sizes, the update latencies without provenance, and the point-wise latency ratios with and without provenance.

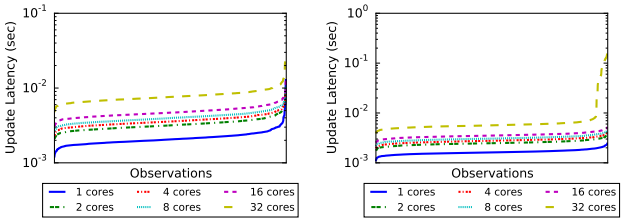


(a) LiveJournal (b) Twitter

Figure 14: Execution times for label propagation with and without provenance, for several systems and configurations.

4.3 Explanations for Stable Matching

We now evaluate our generalized backward tracing of Section 3.3, which provides explanations sufficient to reproduce the output in arbitrary non-monotonic computations. We use a representative non-monotonic computation, stable matching in graphs, that applies to a bipartite graph and a list of candidate matchings (edges), which are rank-ordered by each of the incident nodes. The goal is to find a matching (a subset of the edges where each node is incident



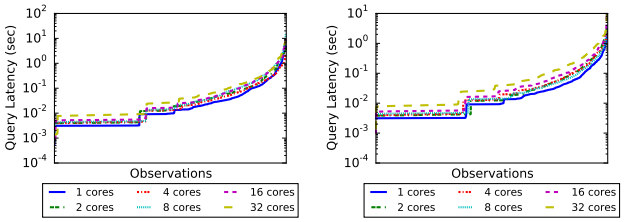
(a) LiveJournal (b) Twitter

Figure 15: Observed latencies to perform 1,000 updates of size 10 to the input graphs of the label propagation algorithm.

on at most one edge) with the property that no excluded edge is more appealing to both of its endpoints than the matchings they actually received. Existing approaches do not guarantee sufficient explanations here, as the stable matching computation does not fall into the two classes we described in the introduction.

We model the input to the computation as a collection of quadruples: (a, b, p_a, p_b) where a and b are node identifiers, and p_a and p_b are their respective rank orderings (0 being best) for this particular matching. The standard algorithm for this task has each node a "propose" to its most prefered b, at which point the b either "rejects" or "tentatively accepts" the proposal, determined by whether it has received a better proposal. Each declined proposal is crossed off the list, and the process continues. We can write this algorithm as a fixed-point computation in differential dataflow, where we repeatedly determine the best proposal for each a in parallel, from those determine the best proposal for each b in parallel, and cross of proposals in the former but not the latter. Iterated to a fixed-point, this emulates the above algorithm, as our system only communicates updates when proposals change. The pseudo-code is given below.

```
initial.iterate(|active| {
    // key records by a, order by pa.
    let props =
      active.map(|x| (x.a,(x.pa,x.b,x.pb)))
            .topk(1, |x| x);
    // key records by b, order by pb.
    let accpt =
      props.map(|x| (x.b,(x.pb,x.a,x.pa)))
           .topk(1, |x| x);
    // discard unaccepted proposals.
    active.except(props.except(accpt))
});
```



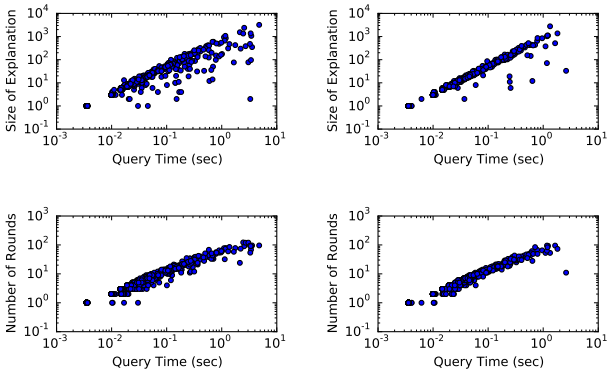
(a) LiveJournal (b) Twitter*

Figure 16: Observed query latencies for 1,000 output explanations for the stable matching algorithm.

Remark. For the experiments we used tuples of the form (a, b, p_a, p_b), where a, b are ids of adjacent nodes in the graphs of Table 2, and p_a, p_b are randomly generated unsigned integers. We run the computation on the LiveJournal graph, but we observed that explanations can be enormous for the Twitter graph, since its maximum degrees are substantially larger than preference lists we might expect. To report measurements, we restricted the Twitter graph to those nodes with degree at most 1,000. We denote this dataset as Twitter*. This reduces the number of edges by a factor of three but, more importantly, reduces the maximum explanation size to something manageable.

Explanation Queries. Figure 16 presents the observed latencies for 1,000 explanation queries for stable matching on the LiveJournal and Twitter* graphs. The explanation queries on the x-axis are given in ascending order of their latencies from left to right. The latencies are largely interactive, and only a small fraction of explanation queries takes more than a second. We also see that additional threads bring little benefit. This is because iterated backward tracing introduces considerable sequential work.

Figure 17 plots explanation latencies against both result size and rounds of backward tracing required, demonstrating that the latency is explained by the complexity of the explanation we must derive. We are not aware of prior work on explaining the results of stable matching, and have less clear intuition for whether the explanations *should* be small than we have in the case of connected components. It may be that there are simpler explanations, and it is an open question whether we could hope to find them automatically. We leave this for future work.

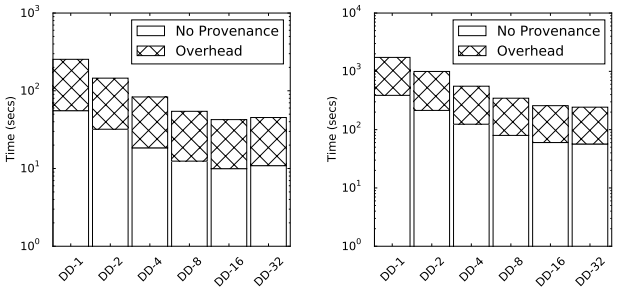


(a) LiveJournal (b) Twitter*

Figure 17: Stable matching explanation query latencies on one core plotted against result sizes (top) and the required number of rounds in backward tracing (bottom). Some outputs require as many as one hundred rounds of backward tracing to explain.

Explanation Overhead. Stable matching is not commonly implemented by graph processing systems, in part because its non-monotonic nature makes it more challenging to express. Figure 18 presents the elapsed times to compute the stable matching, with and without our explanation tracking, for varying numbers of cores. The derivations are non-trivial; for example, on the LiveJournal graph, our system takes more than 2,000 iterations to reach a fixed-point. The relative overheads for LiveJournal range from 358% (1 thread) to 316% (32 threads). The relative overheads for Twitter* range from 348% (1 thread) to 328% (32 threads). The overheads are non-trivial, largely due to the two `except` operations.

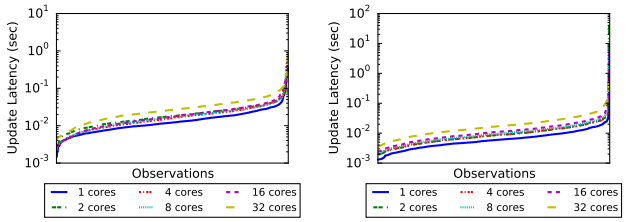
Incremental Updates. Figure 19 presents the observed update times for the stable matching problem with batch size 10. The updates on the x-axis are given in ascending order of their latencies from left to right. For these experiments, we introduced all but U=5,000 matchings, chosen randomly; in each batch of updates we added 5 random matchings from U and removed 5 random matchings from the existing ones. The largest update latency we observed is 52 seconds, for two threads on Twitter*. For 99% of the



(a) LiveJournal (b) Twitter*

Figure 18: Execution times for stable matching with and without provenance tracking.

updates, the latencies increase by at most 2.85x, and the absolute latencies remain largely interactive. The time to process all updates increases by at most 1.8x on LiveJournal, and by at most 4.5x on Twitter*. The extended version [9] provides the latencies for updates of various sizes, the update latencies without provenance, and the point-wise latency ratios with and without provenance.



(a) LiveJournal (b) Twitter*

Figure 19: Observed latencies to perform 1,000 updates of size 10 to the input graphs of the stable matching algorithm.

5. RELATED WORK

Provenance has a long history in databases and scientific workflows, and there are several comprehensive surveys in both fields [7, 12]. Here we provide an overview of the related works on provenance in (i) NoSQL systems, and (ii) Datalog engines. Table 3 summarizes the features of the most prominent systems.

Provenance for MapReduce. The first work on providing provenance support in MapReduce jobs is RAMP [19]. This work describes a methodology on building wrappers around `Map` and `Reduce` functions in order to track lineage dependencies between input and output records. The authors implement their ideas on Hadoop and provide experiments with word count and sorting jobs, showing that provenance tracking has an acceptable overhead in both space and time. Following the ideas of RAMP, [1] extends Hadoop operators with native provenance support that results in even smaller runtime overheads. Recently, Newt [24] has been introduced as a general framework for tracking lineage in big data platforms. Newt requires the manual instrumentation of the system for which lineage has to be captured, and provides a separate MySQL cluster where users can query the collected lineage data with SQL. A major drawback of RAMP and Newt is that they do not provide access to the intermediate data of the computation (in contrast to [1] that offers this functionality); consequently, these two systems cannot provide the *How* provenance of an output record. Based on this limitation, Titian [20] made some nice progress in extending Spark with step-by-step provenance tracking. Titian materializes the dependencies between individual records in a Spark job (including the intermediate ones), and offers an API for interactive forward and backward tracing of dependencies. Similarly to RAMP and Newt, the authors provide experiments with word count and grep jobs. Another interesting work is [3], which extends Pig Latin operators with built-in provenance tracking. This approach tracks the complete state of Pig Latin operators, i.e., both data and execution parameters, and represents provenance dependencies as a graph G, similar to the one shown in Fig. 2. The evaluation of provenance queries requires the construction of the whole *How* provenance graph G in advance; then, provenance queries are expressed as graph-matching queries on G. A nice feature of [3] is that it allows users to query provenance information at multiple levels of granularity through a form of zoom-in/out operations on G. Finally, [28] provides limited lineage support in Pig by propagating user-defined tags from the input to the output records in an eager fashion. The limitations of all previous works are summarized in the following: (i) they are unable to handle iteration, (ii) they do not support real-time provenance for computations over continuously updated data, and (iii) they do not provide explanations that guarantee the reproduction of the output in the presence of non-monotonic operators.

Provenance for Datalog. Database-style provenance has also received considerable attention in the context of fixed-point Datalog computations. The recursive nature of Datalog goes beyond relational algebra, making the application of traditional provenance techniques even more challenging. After the seminal work in [18], [13] was the first to identify that annotation-based approaches explode in size when applied to recursive computations. To provide more concise provenance information, [14] proposed an algorithm for retrieving sub-graphs of the *How* provenance graph based on user-defined patterns. [22] presents an early effort to add *How* provenance support in general-purpose operational Datalog engines. This work augments Datalog rules with additional predicates that are used to track the dependencies between inputs and outputs.

Network Provenance. Data provenance is a versatile concept and has also been used for network management, most notably in ExSPAN [33] and DTaP [34]. Both these systems are based on NDlog, a variation of Datalog with aggregation (e.g., `MIN`), which was first introduced in [25] for the declarative definition and analysis of routing protocols. These works extend NDlog with support for *How* provenance, in a way similar to [22], and they also provide incremental tracking of provenance through semi-naive evaluation – which is equivalent to differential dataflow for Datalog. Driven by debugging needs, DTap adopts a time-aware provenance model where each node in the provenance graph is associated with a logical timestamp that denotes the point a tuple was generated during the execution. Logical timestamps can then be used to query provenance within time windows (snapshots) specified by the user. The notion of time in DTaP is similar to the one natively used in differential dataflow (cf. Section 2.2), however, time metadata are not exploited to provide concise explanations as we do in Section 4.2. In the connected components example, DTaP explains a label r at node n by returing all paths between nodes n and r in the graph. As a final comment, NDlog is not well-suited for data analytics, e.g., it is unclear how it can express stable matching (Section 4.3). In addition, the NDlog engine is built as an extension of ns-3 (Network Simulator) and cannot be used as a standalone distributed engine.

6. CONCLUSIONS

In this paper we presented a framework for explaining outputs in modern data analytics, with a particular focus on iterative dataflows. We introduced a generalized form of backward tracing which guar-

	Computational Model	Provenance Storage & Query Engine	Provenance Tracking Granularity	Intermediate Data Tracking	Incremental Provenance Maintainance	Output Reproduction
this paper	Arbitrary Dataflows (even nested iterations)	Native (Differential Dataflow)	Operator Level (Section 3.2)	✓ (Section 4)	✓ (Section 4)	✓ (Section 3.3)
RAMP [19]	DAG Dataflows (no iteration support)	Native Storage (HDFS) External Query Engine (Java implementation)	Map & Reduce Level (Hadoop)	- (see also [20])	-	According to Theorem in [19]
Titian [20]	DAG Dataflows[1] (no iteration support)	Native (Spark)	Spark Stage Level	✓	-	According to Theorem in [19]
Newt [24]	DAG Dataflows[2] (no iteration support)	External (MySQL)	Multiple Levels (instrumentation-based)	- (see also [20])	-	According to Theorem in [19]
Lipstick [3]	DAG Dataflows (no iteration support)	Native Storage (Pig) External Query Engine (Java implementation)	Operator Level (Pig Latin)	✓	-	According to Theorem in [19]
ExSPAN* [33] DTaP* [34]	Datalog-based	Native (NDlog Engine on ns-3)	Operator Level (NDlog)	✓	✓	According to Theorem in [19]

[1] The current version of Titian does not support iteration through GraphX.
[2] Newt has been applied to Hyracks and Hadoop [24], and to Spark [20], all of which support DAG dataflows.
* These systems are not general-purpose data processing systems but they offer interesting features regarding provenance management.

Table 3: Summary of features in NoSQL systems with provenance support

antees concise explanations of outputs in computations where naive backward tracing approaches fail. The experiments indicate that our framework is suitable for a variety of data analysis tasks, on realistic data sizes, and with very low overhead in performance, even in the case of incremental updates.

We believe that our approach is here to stay and there are several directions for future work. Besides the further optimization opportunities, we are especially interested in understanding the structure of explanations for general data-parallel computations; while our intuition serves us well for tasks like graph connectivity, our understanding of explanations for problems like those in [5] has huge potentials for improvement.

ACKNOWLEDGEMENTS. We would like to thank the anonymous reviewers for their comments, Molham Aref, Martin Bravenboer, and Wael Sinno for their help with the LogicBlox system, and Matteo Interlandi for his comments on Titian, Newt, and RAMP.

7. REFERENCES

[1] S. Akoush et al. HadoopProv: Towards provenance as a first class citizen in MapReduce. In *TAPP*, 2013.
[2] B. Alexe et al. Spider: A schema mapping debugger. In *VLDB*, 2006.
[3] Y. Amsterdamer et al. Putting lipstick on Pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2012.
[4] M. Aref et al. Design and implementation of the LogicBlox system. In *SIGMOD*, 2015.
[5] Y. Bu et al. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.
[6] P. Buneman et al. Why and where: A characterization of data provenance. In *ICDT*, 2001.
[7] J. Cheney et al. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2007.
[8] L. Chiticariu et al. DBNotes: A post-it system for relational databases based on provenance. In *SIGMOD*, 2005.
[9] Z. Chothia et al. Explaining outputs in modern data analytics. TR, *ETH Zurich*, http://people.inf.ethz.ch/ioannisl/prov.pdf, 2016.
[10] Y. Cui et al. Tracing the lineage of view data in a warehousing environment. *TODS*, 25(2):179–227, 2000.
[11] Y. Cui et al. Lineage tracing for general data warehouse transformations. *VLDBJ*, 12(1):41–58, 2003.
[12] S. Davidson et al. Provenance in scientific workflow systems. 2007.
[13] D. Deutch et al. Circuits for Datalog provenance. In *ICDT*, 2014.
[14] D. Deutch et al. Selective provenance for datalog programs using top-k queries. *PVLDB*, 8(12):1394–1405, 2015.
[15] K. Donnelly. SNOMED-CT: The advanced terminology and coding system for eHealth. *HTI*, 121:279–290, 2006.
[16] B. Glavic et al. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, 2009.
[17] T. J. Green et al. Orchestra: Facilitating collaborative data sharing. In *SIGMOD*, 2007.
[18] T. J. Green et al. Provenance semirings. In *PODS*, 2007.
[19] R. Ikeda et al. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.
[20] M. Interlandi et al. Titian: Data provenance support in Spark. *PVLDB*, 9(3), 2015 (to appear).
[21] G. Karvounarakis et al. Querying data provenance. In *SIGMOD*, 2010.
[22] S. Köhler et al. Declarative Datalog debugging for mere mortals. In *Datalog 2.0*, 2012.
[23] N. Leone et al. The DLV system for knowledge representation and reasoning. *TOCL*, 7(3):499–562, 2006.
[24] D. Logothetis et al. Scalable lineage capture for debugging DISC analytics. In *SOCC*, 2013.
[25] B. T. Loo et al. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.
[26] F. McSherry et al. Differential dataflow. In *CIDR*, 2013.
[27] D. G. Murray et al. Naiad: A timely dataflow system. In *SOSP*, 2013.
[28] C. Olston et al. Inspector Gadget: A framework for custom monitoring and debugging of distributed dataflows. *PVLDB*, 4(12):1237–1248, 2011.
[29] J. Rogers et al. GALEN ten years on: Tasks and supporting tools. In *MEDINFO*, 2001.
[30] J. Seo et al. Distributed SociaLite: A Datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.
[31] J. Wang et al. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.
[32] J. Widom. Trio: A system for integrated management of data, accuracy and lineage. In *CIDR*, 2005.
[33] W. Zhou et al. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*, 2010.
[34] W. Zhou et al. Distributed time-aware provenance. *PVLDB*, 6(2):49–60, 2012.