Report

Explaining Outputs in Modern Data Analytics

Author(s):
Chothia, Zaheer; Liagouris, John; McSherry, Frank; Roscoe, Timothy

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-010601932

Rights / License:
In Copyright - Non-Commercial Use Permitted
Technical Report

DOI: 10.3929/ethz-a-010601932

Systems Group, Department of Computer Science, ETH Zurich

Explaining Outputs in Modern Data Analytics

by

Zaheer Chothia, John Liagouris, Frank McSherry, Timothy Roscoe

March 4, 2016

This technical report is an extended version of a paper submitted to VLDB ’16, March 2016
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 naïve 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 data curation [8], system debugging [46, 3, 29], access control [17], workflow analysis [20, 4, 42, 26], probabilistic data management [6], network management [47, 43], data integration [12, 28], and query result explanation [37], among others.
At the same time, the scale and scope of modern data analytics have moved past what traditional databases were designed to handle.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXX-X/XX/XX ...$15.00.

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 [24, 41, 13, 22], and (ii) lazy techniques which determine necessary input records by “inverting” operators in the computation [15, 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 [18] and (ii) too few inputs to reproduce the output [26]. 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. 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 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 [29, 46] 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. One algorithm to do so is first to 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 from the words in each document. In the example, the resulting count for Doc1 is 3. Tracing backwards, the three records observed to contribute to this count derive only from Doc1. 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 [26] 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 [26] is a theorem about reproducing target outputs:

**Theorem 1.** Output Reproduction (from [26])

Consider a workflow \( W(I) = O \) with input \( I \) and output \( O \). Let \( o \in O \) be an output element and consider that the output of a naive backward tracing from \( o \) is the set of input elements \( \Gamma \subseteq I \).

1. If all \texttt{map} and \texttt{reduce} operators in \( W \) are one-to-one and many-to-one respectively, then \( o \in W(\Gamma) \).
2. If there is at most one non-monotonic\(^1\) data transformation in \( W \), then \( o \in W(\Gamma) \).
3. Otherwise, there is no guarantee that \( o \in W(\Gamma) \).

\(^1\)According to [26], a data transformation \( T \) is monotonic if for any \( I_1, I_2 \) with \( I_1 \subseteq I_2 \), then \( T(I_1) \subseteq T(I_2) \).

The “unique word count” example falls into the third case of Theorem 1 because the workflow includes a one-to-many \texttt{map} operator – the operator that takes each input document and splits it into individual words – followed by two non-monotonic operations: (i) group by \texttt{word} and output only singleton groups, and (ii) group by \texttt{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 \( W(I) \), but appears in \( W(\Gamma) \), i.e., when replaying the computation using as input only the result of backward tracing from \( o = (\text{Doc1}, 3) \). To ensure consistency between the reference and the replayed computation, the authors of [26] 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 [26, 27, 4, 31, 46, 29, 1] provide explanations that guarantee the reproduction of the output only when the conditions of Theorem 1 are satisfied.

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 \texttt{map}, \texttt{reduce}, \texttt{join}, and \texttt{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.

The remainder of the document is organized as follows. Section 2 clarifies several important concepts of data provenance, and provides some background on differential dataflow which is the basis of our approach. In Section 3 we present our generalized backward tracing, and we demonstrate how it accommodates existing works on data provenance. In Section 4 we evaluate each aspect of our system, and in Section 5 we provide an overview of the systems in the field. We conclude in Section 6 with directions for future work.

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. The rest of the related work on provenance-aware systems is given in Section 5.
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 [14]. 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 [15]. 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. [9] 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 in [10]. More recently, [23] formalized the term \( \text{How} \) provenance in the form of *semirings* and extended this idea to Dialog. In an attempt to provide a consistent terminology, [39] considers two general notions of provenance: (i) Provenance of Data (PoD) that provides information about the input data items that contribute to the result of a query \( q \), and (ii) Provenance of a Data Product (PoDP) that provides information about the particular data transformations that led to the result of \( q \). As we explain in the following, the terms lineage, Why and Where provenance are classified under PoD whereas the \( \text{How} \) provenance lies between PoD and PoDP.

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 of all customers with an order of type \( \text{Orders} \). The result of \( q \) on \( D \) contains the tuple \( t_8 = <\text{Bob}, 99 \text{ High St}> \), 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 \( \times \) denote the joins of pairs \( (t_t, t_s) \) and \( (t_t, t_r) \) that produce the intermediate tuples \( t_5 \) and \( t_4 \), respectively. Nodes labeled with \( \pi \) denote the projection of fields Name and Address from \( t_4 \) and \( t_3 \), both resulting in the same output tuple \( t_6 \).

The lineage of \( t_6 \) is the set of highlighted tuples \( \{t_1, t_3, t_4\} \) from the two input relations Customers and Orders. The Why provenance of \( t_6 \) 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 \( t_6 \), i.e., a subset of the input tuples that are sufficient – but not necessary – to have \( t_6 \) in \( q(D) \). Note that the lineage is the union of data elements in the Why provenance. According to [23], the \( \text{How} \) provenance of \( t_6 \) is expressed as a semiring of the form \( t_1 \cdot (t_1 + t_2) \), where \( t_1 \cdot t_2 \) means "both \( t_1 \) and \( t_2 \)" whereas \( t_1 + t_2 \) is "either \( t_1 \) or \( t_2 \)." In general, \( \text{How} \) provenance is more informative than \( \text{Why} \) provenance and the former can be used to generate the latter. On the other hand, \( \text{How} \) 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 \( \text{How} \) provenance, called transformation provenance [22]. Transformation provenance describes all possible ways of having \( t_6 \) in \( q(D) \) – just like \( \text{How} \) provenance – but also includes the actual query operators that are involved in this process. Finally, the \( \text{Where} \) provenance is defined at the attribute level; the \( \text{Where} \) provenance of \( t_6 \), 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 \( t_6 \). In this sense, the \( \text{Where} \) provenance has the copy-contribution semantics as explained in [22].

Our work focuses on tracking, maintaining, and querying lineage and \( \text{How} \) provenance in the context of arbitrary iterative dataflows. The notion of explanations we also consider in this paper is related to the \( \text{Why} \) provenance described in the previous. Intuitively, a witness of \( t_6 \) "explains" the existence of \( t_6 \) in \( q(D) \) although it does not provide any information about the process that led \( t_6 \) to \( q(D) \).

2.2 Differential Dataflow

Differential Dataflow [33] (DD) is a data-parallel programming and execution model, layered on top of the timely dataflow model implemented in the Naiad system [34]. Differential dataflow was originally designed to support incremental updates to the inputs of iterative data-parallel computations. As we explain in the following, differential dataflow is a good fit for backward tracing, especially if we want to update backward tracing computations efficiently. 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 (\( \text{reach} \)) with the collection of edges (\( \text{edges} \)), and retains distinct elements (map is used to project b):

```plaintext
initial.iterate([\text{reach}]) {
  \text{reach}.join(\text{edges})
  .map(|(a,b)| b)
  .distinct()
}
```

Differential dataflow transparently provides incremental execution. Although the program above appears to repeatedly re-execute the \( \text{join} \) on the full collection \( \text{reach} \), the implementation is actually delta-based; work is only performed as the collections \( \text{reach} \) 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:

\[
\text{collection}(\text{record}, c, t)
\]

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 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 traces maintain the property that, at any logical time \( t \), a collection is determined by accumulating all update records with a logical time \( r \leq t \).

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:

\[
\begin{align*}
\text{collection}(\text{"a"}, +1, (0,0)) \\
\text{collection}(\text{"b"}, +1, (0,0)) \\
\text{collection}(\text{"c"}, +1, (0,0))
\end{align*}
\]

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:

\[
\begin{align*}
\text{collection}(\text{"d"}, +1, (0,1)) \\
\text{collection}(\text{"c"}, +1, (0,1))
\end{align*}
\]

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 [30, 29]), 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 – 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

\[
\text{collection("c", -1, (1,0))} \\
\text{collection("d", -1, (1,1))} \\
\text{collection("d", +1, (1,2))}
\]

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 [30, 29]), differential dataflow’s explicit use of timestamps extend it to arbitrary iterative computations.
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 \( \text{col} \) in the computation, we introduce another collection \( \text{col}_q \) of the same type, containing records from \( \text{col} \) that require explanation. To define the contents of this new collection, for each data-parallel operator with input collection \( \text{col} \) and output collection \( \text{col}_q \), we introduce a new join operator whose inputs are \( \text{col} \) and \( \text{col}_q \), and whose output is \( \text{col}_q \). Intuitively, the role of this join operator is to select elements of \( \text{col} \) matching requests in \( \text{col}_q \). 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 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.

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 [34]. 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 [19, 23, 29, 18]. 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 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 \( \text{In1}: (k, v_1) \) and \( \text{In2}: (k, v_2) \), and produces an output of the form \( \text{Out}: (k, (v_1, v_2)) \). 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.

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

---

\(^3\)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.
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 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 \texttt{enter}, which adds a timestamp coordinate that is easily removed, and to \texttt{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 \texttt{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.

**Join.** The binary \texttt{join} operator matches key-value pairs with common keys, producing the triple (key, first value, second value). In differential dataflow, the timestamp of the output is the least upper bound of the timestamps of the input records. Although we do not know the timestamps of the input records from the output alone, we can just produce requests for the appropriate inputs using the timestamp of the output record. All identified input records must have been present at the logical time the output was produced.

### 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).

![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.](image-url)
wards 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-processing. 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 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 \( O' \subseteq O' \).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Input facts</th>
<th>Derived facts</th>
<th>Derivations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNOMED CT</td>
<td>1,030,336</td>
<td>12,970,258</td>
<td>238,078,602</td>
</tr>
<tr>
<td>GALEN8</td>
<td>976,552</td>
<td>24,483,561</td>
<td>441,464,399</td>
</tr>
</tbody>
</table>

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.

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.

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-Threadining) and 512GB RAM, running Debian v7.9. We noticed substantial variation in performance when we used 32 threads, finding the best measurements for all systems at 16 threads.

4.1 Provenance in Datalog

For our Datalog experiments we use two real-world ontologies, SNOMED CT [21] and GALEN8 [36], which are the standard ontologies for representing and exchanging knowledge in the biomedial domain. Each of these ontologies describes a schema for medical terms, which have complex dependencies 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 generate a large number of results (facts), whose sizes are summarized 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 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.

![Figure 9: Query result sizes plotted against query latency on one core for How-provenance. The correlation between response time and result size indicates that the variation in query latencies is explained by the volume of data returned.](image)

Provenance Overhead. Our provenance tracking requires the materialization and indexing of the intermediate results of the computation (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 [30], and multi-threaded LogicBlox (LBX) [5] that was allowed to use all resources of the machine. These systems do not have native provenance support, but we implemented the techniques in [29], 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. Note that DLV did not terminate 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 1, 10, and 100 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 a small fraction of the facts $U$, chosen randomly. With the smallest batch size, 500 update rounds were addition of single facts from $U$ and the remaining 500 rounds were removal of existing facts. For larger batch sizes of 10 and 100 updates, the addition and deletion of facts were interleaved within each batch. Elapsed times are measured from the introduction of the batch of updates until the derivation reaches a fixed-point. For small batches, the single core is fastest because there is not much parallel work, whereas for large batches multiple cores bring down the maximum latency. The largest latency is 62.18s, for a single core on batch size 100 for GALEN8, 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 distributions of latencies without provenance, and the distribution of point-wise latency ratios with and without provenance appear in the Appendix.

![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 by intersecting the latter with the set of input facts, thus, it has a small additional latency. As more workers are introduced, work is shared, and the minimum and maximum time close in.](image)

![Figure 10: Derivation times for the Datalog program of Section 4.1 with and without provenance tracking. DLV is a single-threaded Datalog engine, and LBX (LogicBlox) is a multi-threaded Datalog engine. The DD-x measurements correspond to our approach with x numbers of workers (cores).](image)

### 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 naively, 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 neighbors, updating the label each time it receives an improved value. In the pseudo-code below, we start from initial labels $\texttt{init}$, 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.
The implementation of label propagation in Differential Dataflow is straightforward:

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

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>4,847,571</td>
<td>66,993,773</td>
</tr>
<tr>
<td>Twitter</td>
<td>41,652,230</td>
<td>1,468,365,182</td>
</tr>
</tbody>
</table>

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 has the effect of giving smaller labels a “head start” and reduces the amount of churn among nodes with large identifiers. It accomplishes the same goal as asynchronous systems, like Myria [40] and SociaLite [38], that propagate more successful labels.

**Explanation Queries.** The result of the label propagation algorithm includes pairs \((node, label)\), where \(label\) denotes the smallest node id contained in the connected component of \(node\). An explanation of an output pair using naïve backwards tracing includes the edges of all paths from \(label\) to \(node\), which in a symmetric graph would be all edges in the component. The graphs of Table 2 are largely connected, and the explanations are roughly the full set of edges in the graph, so we do not evaluate the performance of this technique.

Figure 12 presents both the query latencies (top) and the correlation between query latency and explanation size (bottom) for the label 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 identifier as the explanation request passes through the join, and repeating with the new node identifier 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 small diameter. Figure 13 presents query latencies for the explanations of 1,000 randomly chosen outputs of the connected components computation on the LiveJournal and Twitter graphs. 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 inspect are among the first issued, suggesting some warm-up issues.

**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 6 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 [40] and Socialite [38], on the label propagation algorithm. Note that 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=16 threads.

Incremental Updates. Figure 15 presents the observed latencies to perform 1,000 updates of size 1, 10, 100 (top to bottom) to the input graphs of the label propagation algorithm. Although 1% of the updates increase by more than 20x, the update times remain interactive, typically tens of milliseconds. Excluding the 32-core measurements, the time to process all updates increases by at most a factor of 8.6x. This relative increase is largely due to the very low update latencies for the reference computation, which each take only tens to hundreds of microseconds. The latencies without provenance, and the point-wise ratios with and without provenance appear in the Appendix.

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 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. According to Theorem 1, naive backward tracing does not necessarily identify a sufficient explanation here, and we need multiple iterations of tracing to do so.
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 preferred \(b\), 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 fix-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 off proposals in the former but not the latter. Iterated to a fix-point, this emulates the above algorithm, as differential dataflow only communicates updates when proposals change. The pseudo-code is given below.

```rust
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 accept =
        props.map(|x| (x.b, (x.pb, x.a, x.pa)))
            .topk(1, |x| x);

    // discard unaccepted proposals.
    active.except(props.except(accept))
});
```

**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 roughly a factor of three but, more importantly, reduces the maximum explanation size to something manageable.

![Figure 16: Observed query latencies for 1,000 output explanations for the stable matching algorithm. The latencies are generally bounded by one second, but some variation does exist, and is explained in Figure 17.](image)

**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.

![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). The correlations indicate that the variation in query latencies seen in Figure 16 is explained by the complexity of the explanations. We also see that some outputs require as many as one hundred rounds of backward tracing to explain.](image)

**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 fix-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.

![Figure 18: Execution times for stable matching with and without provenance tracking.](image)
**Incremental Updates.** Figure 19 presents the observed update times for the stable matching problem with batch sizes $1$, $10$, and $100$. 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 a small fraction of the matchings $U$, chosen randomly. With the smallest batch size, 500 update rounds were addition of single matchings from $U$ and the remaining 500 rounds were removal of existing matchings. For larger batch sizes of 10 and 100 updates, the addition and deletion of matchings were interleaved within each batch. The largest update latency is 52 seconds, for two threads with batches of size 10 on Twitter*. For 99% of the 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 latencies without provenance, and the point-wise ratios with and without provenance appear in the Appendix.

![Observations](image1)

![Observations](image2)

![Observations](image3)

**Figure 19: Observed latencies to perform 1,000 updates of size 1, 10, 100 (top to bottom) to the input graphs for the stable matching algorithm. The update times remain largely interactive, and 99% of the time increase by less than 2.85x. The time to process all updates increases by at most 1.8x for LiveJournal, and at most 4.5x for Twitter*. The latencies without provenance, and the point-wise ratios with and without provenance appear in the Appendix.**

5. RELATED WORK

Provenance has a long history in databases and scientific workflows, and there are several comprehensive surveys in both fields [11, 16]. 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 in NoSQL Systems.** The first work on providing provenance support in MapReduce jobs is RAMP [26]. 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], other implementations leverage native provenance support that results in even smaller runtime overheads. Recently, Newt [31] 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 [27] made some nice progress in extending Spark [45] 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, showing the efficiency and scalability of their system. Another interesting work is [4], 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 $G$ graph $G$ in advance; then, provenance queries are expressed as graph-matching queries on $G$. A nice feature of [4] 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, [35] provides limited lineage support in Pig by propagating user-defined tags from the input to the output records in an eager fashion.

**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 the SPJU queries of relational algebra, making the application of traditional provenance techniques even more challenging. After the seminal work in [23], [19] was the first to identify that annotation-based approaches explode in size when applied to recursive computations. To provide more concise provenance information, [18] proposed an algorithm for retrieving sub-graphs of the how provenance graph based on user-defined patterns. [29] presents an early effort to add how provenance support in a general-purpose operational Datalog engine (LogicBlox). This work augments Datalog rules automatically with additional predicates that are used to track the dependencies between inputs and outputs. An experimental comparison with [29] is given in Section 4.1.

**Network Provenance.** Data provenance is a versatile concept and has been used for network management, most notably in ExSPAN [47] and DTAP [46]. Both these systems are based on NiLog, a variation of Datalog with aggregation (e.g., $\min$), which was first introduced in [32] for the declarative definition and analysis of routing protocols. These works extend NiLog with support for how provenance, in a way similar to [29], 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 returning 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 guarantees 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 [7] has huge potentials for improvement.

### 7. REFERENCES


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

<table>
<thead>
<tr>
<th>Computational Model</th>
<th>Provenance Storage &amp; Query Engine</th>
<th>Provenance Tracking Granularity</th>
<th>Intermediate Data Tracking</th>
<th>Incremental Provenance Maintenance</th>
<th>Output Reproduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>this paper</td>
<td>Arbitrary Dataflows (even nested iterations)</td>
<td>Native (Differential Dataflow)</td>
<td>Operator Level (Section 3.2)</td>
<td>✓ (Section 4)</td>
<td>✓ (Section 4)</td>
</tr>
<tr>
<td>RAMP [26]</td>
<td>DAG Dataflows (no iteration support)</td>
<td>Native Storage (HDFS) External Query Engine (Java implementation)</td>
<td>Map &amp; Reduce Level (Hadoop)</td>
<td>- (see also [27])</td>
<td>-</td>
</tr>
<tr>
<td>Titan [27]</td>
<td>DAG Dataflows(^1) (no iteration support)</td>
<td>Native (Spark)</td>
<td>Spark Stage Level</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Newt [31]</td>
<td>DAG Dataflows(^2) (no iteration support)</td>
<td>External (MySQL)</td>
<td>Multiple Levels (instrumentation-based)</td>
<td>- (see also [27])</td>
<td>-</td>
</tr>
<tr>
<td>Lipstick [4]</td>
<td>DAG Dataflows (no iteration support)</td>
<td>Native Storage (Pig) External Query Engine (Java implementation)</td>
<td>Operator Level (Pig Latin)</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>ExSPAN* [47]</td>
<td>Datalog-based</td>
<td>Native (NDlog Engine on ns-3)</td>
<td>Operator Level (NDlog)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DTaP* [46]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) The current version of Titan does not support iteration through GraphX [44].
\(^2\) Newt has been applied to Hyracks and Hadoop [31], and to Spark [27], all of which support DAG dataflows.

* These systems are not general-purpose data processing systems but they offer interesting features regarding provenance management.
8. APPENDIX

Here we present the augmented form of the Datalog program used in Section 4.1 to track provenance in [30] and [5]. We also present additional measurements describing the distribution of ratios of update latencies for Datalog (Figure 21), label propagation (Figure 22), and stable matching (Figure 23). Figure 24 depicts the latencies for label propagation without tracking explanations.

\[
p(X, Y) := \text{p\_base}(X, Y).
\]

\[
q(X, P, W) := \text{q\_base}(X, P, W).
\]

Rule_1(X, Y, Z) := p(X, Y), p(Y, Z).

\[
p(X, Z) := \text{Rule}_1(X, Y, Z).
\]


\[
\]


\[
q(X, Y, Z) := \text{Rule}_3(X, W, Y, Z).
\]


\[
\]


\[
\]


\[
q(X, Y, Z) := \text{Rule}_6(X, R, W, P, Z, Y).
\]

Figure 20: The Datalog program used to track provenance in DLV and LBX (LogicBlox). This program is a simple augmentation of the Datalog program in Section 4.1.
Figure 21: Ratios of observed latencies with and without provenance tracking for 1,000 updates of sizes 1, 10, and 100 (top to bottom) to the input of the Datalog program of Section 4.1. The ratios on the x-axis are given in ascending order from left to right. The vast majority of ratios are close to 1.

(a) SNOMED CT  
(b) GALEN8

Figure 22: Ratios of observed latencies for 1,000 updates of sizes 1, 10, and 100 (top to bottom) for label propagation on the LiveJournal and Twitter graphs (Section 4.2). The ratios on the x-axis are given in ascending order from left to right. The large ratios are due to the optimized nature of the reference computation: the update latencies without provenance tracking are given in Figure 24. We also observed substantial variation in performance when using 32 threads.

(a) LiveJournal  
(b) Twitter
Figure 23: Ratios of observed latencies for 1,000 updates of sizes 1, 10, and 100 (top to bottom) for stable matching on the LiveJournal and Twitter* graphs (Section 4.3). The ratios on the x-axis are given in ascending order from left to right. The vast majority of ratios are close to 1.

(a) LiveJournal  
(b) Twitter

Figure 24: Times to perform 1,000 updates of size 1, 10, 100 (top to bottom) to the LiveJournal and Twitter graphs for the label propagation algorithm, without provenance tracking. The updates on the x-axis are given in ascending order of their latencies from left to right.

(a) LiveJournal  
(b) Twitter