Differential dataflow

Frank McSherry Derek G. Murray Rebecca Isaacs Michael Isard

Microsoft Research, Silicon Valley Lab
{mcsherry, derekmur, risaacs, misard}@microsoft.com

ABSTRACT

Existing computational models for processing continuously changing input data are unable to efficiently support iterative queries except in limited special cases. This makes it difficult to perform complex tasks, such as social-graph analysis on changing data at interactive timescales, which would greatly benefit those analyzing the behavior of services like Twitter. In this paper we introduce a new model called *differential computation*, which extends traditional incremental computation to allow arbitrarily nested iteration, and explain—with reference to a publicly available prototype system called Naiad—how differential computation can be efficiently implemented in the context of a declarative data-parallel dataflow language. The resulting system makes it easy to program previously intractable algorithms such as incrementally updated strongly connected components, and integrate them with data transformation operations to obtain practically relevant insights from real data streams.

1. INTRODUCTION

Advances in low-cost storage and the proliferation of networked devices have increased the availability of very large datasets, many of which are constantly being updated. The ability to perform complex analyses on these mutating datasets is very valuable; for example, each tweet published on the Twitter social network may supply new information about the community structure of the service's users, which could be immediately exploited for real-time recommendation services or the targeting of display advertisements. Despite substantial recent research augmenting "big data" systems with improved capabilities for incremental computation [4, 8, 14, 26], adding looping constructs [7, 12, 17, 20, 25], or even efficiently performing iterative computation using incremental approaches [13, 19], no system that efficiently supports general incremental updates to complex iterative computations has so far been demonstrated. For example, no previously published system can maintain in real time the strongly connected component structure in the graph induced by Twitter mentions, which is a potential input to the application sketched above.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2012, Asilomar, California, USA.

This paper introduces *differential computation*, a new approach that generalizes traditional models of incremental computation and is particularly useful when applied to iterative algorithms. The novelty of differential computation is twofold: first, the state of the computation varies according to a *partially ordered* set of versions rather than a totally ordered sequence of versions as is standard for incremental computation; and second, the set of updates required to reconstruct the state at any given version is retained in an indexed data-structure, whereas incremental systems typically consolidate each update in sequence into the "current" version of the state and then discard the update. Concretely, the state and updates to that state are associated with a multi-dimensional logical timestamp (hereafter *version*). This allows more effective re-use: for example, if version (i, j) corresponds to the j^{th} iteration of a loop on the i^{th} round of input, its derivation can re-use work done at both predecessors $(i-1, j)$ and $(i, j-1)$, rather than just at whichever version was most recently processed by the system.

Incremental systems must solve two related problems: efficiently updating a computation when its inputs change, and tracking dependencies so that local updates to one part of the state are correctly reflected in the global state. Differential computation addresses the first problem, but as we shall see it results in substantially more complex update rules than are typical for incremental systems. We therefore also describe how differential computation may be realized when data-parallelism and dataflow are used to track dependencies, resulting in a complete system model that we call *differential dataflow*. A similar problem is addressed by incremental view maintenance (IVM) algorithms [6, 15, 23], where the aim is to re-use the work done on the previous input when updating the view to reflect a slightly different input. However, existing IVM algorithms are not ideal for interactive large-scale computation, because they either perform too much work, maintain too much state, or limit expressiveness.

We have implemented differential dataflow in a system called Naiad, and applied it to complex graph processing queries on several real-world datasets. To highlight Naiad's characteristics, we use it to compute the strongly connected component structure of a 24-hour window of Twitter's messaging graph (an algorithm requiring doubly nested loops, not previously known in a data-parallel setting), *and* maintain this structure with sub-second latency, in the face of Twitter's full volume of continually arriving tweets. Fur-

thermore, the results of this algorithm can be passed to subsequent dataflow operators within the same differential computation, for example to maintain most common hashtags for each component, as described in the Appendix.

The contributions of this paper can be summarized as follows:

- The definition of a new computation model, *differential computation*, that extends incremental computation by allowing state to vary according to a partial ordering of versions, and maintains an index of individual updates, allowing them to be combined in different ways for different versions (Section 3).
- The definition of *differential dataflow* which shows how differential computation can be practically applied in a data-parallel dataflow context (Section 4).
- A sketch of the implementation of the prototype Naiad system that implements differential dataflow, along with sample results showing that the resulting system is efficient enough to compute updates of complex computations at interactive timescales (Section 5).

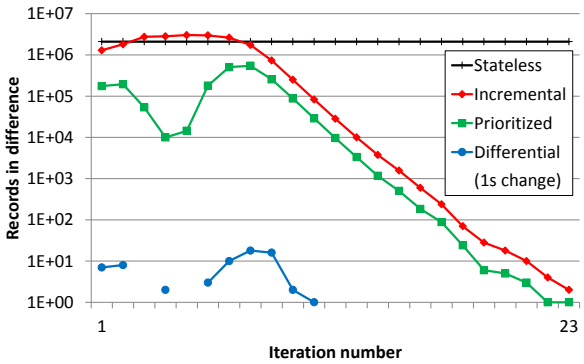


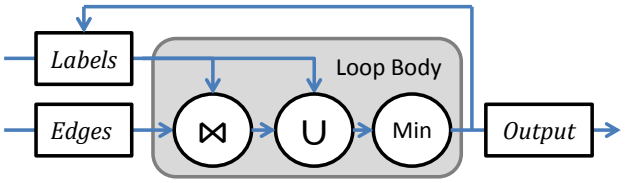
Figure 1: Number of connected components labels in difference plotted by iteration, for a 24-hour window of tweets, using three different techniques. Also plotted are the differences required to update the third as an additional second of tweets arrive.

2. MOTIVATION

To motivate our new computational framework, consider the problem of determining the connected component structure of a graph. In this algorithm, each node is assigned an integer label (initially its own ID), which is then iteratively updated to the minimum among its neighborhood. In a relational setting, a single iteration can be computed by joining the edge relation with the current node labeling, taking the union with the current labeling, and computing the minimum label that is associated with each node ID:

```
SELECT node, MIN(label)
FROM ((SELECT edges.dest AS node, label
    FROM labels JOIN edges ON labels.node = edges.src)
  UNION (SELECT * FROM labels))
GROUP BY node
```

After i steps each node will have the smallest label in its i-hop neighborhood and, when run to fixed point, will have the smallest label in its connected component. The following dataflow graph illustrates the iterative computation:



To make the problem concrete, consider the example of the graph formed by `@username` mentions in a 24-hour period on the Twitter online social network, and contrast four approaches to executing each iteration of the connected components algorithm. Figure 1 plots the number of label changes in each iteration, for the various techniques, as a proxy for the amount of work each requires. We confirm in Section 5 that running times exhibit similar behavior.

The simplest and worst-performing approach repeatedly applies the above query to the result of the previous iteration until the labeling stops changing. In this case, all previously computed results are overwritten with new labels in each round, leading to a constant amount of work each iteration and the flat line labeled "Stateless" in Figure 1. Data-parallel frameworks including MapReduce [10] and Dryad [16] maintain no state between iterations, and are restricted to executing the algorithm in this manner.

A more advanced approach ("Incremental") retains state from one iteration to the next, and uses an incremental evaluation strategy to update the set of labels based on changes in the previous iteration [13, 17, 19]. As labels converge to their correct values, the amount of computation required in each iteration diminishes. In Figure 1, the number of differences per iteration decays exponentially after the eighth iteration, and the total work is less than half of that required for the traditional approach. The incremental approach does require maintaining state in memory for performance, though not more than the full set of labels.

The incremental approach can be improved ("Prioritized") by reordering the computation in ways that result in fewer changes between iterations. For example, in connected components, we can prioritize smaller labels, which are more likely to prevail in the min computation, and introduce these before the larger labels. This is similar in spirit to the prioritized iteration proposed by Zhang *et al.* [28]. In fact, the total amount of work is only 10% of the incremental work, and corresponds to approximately 4% of that done by a stateless dataflow system.

Allowing the inputs to change.

Differential dataflow generalizes both the incremental and prioritized approaches and can be used to implement either, resulting in the same number of records in difference. Although differential dataflow stores the differences for multiple iterations (rather than discarding or coalescing them), the total number retained for the 24-hour window is only 1.5% more than the set of labels, the required state for incremental dataflow.

The power of differential dataflow is revealed if the input graph is modified, for example by the removal of a single edge. In this case the results of the traditional, incremental and prioritized dataflow computations must be discarded

and their computations re-executed from scratch on the new graph. IVM algorithms supporting recursive queries can be used, but have significant computational or memory overheads. In contrast our approach ("Differential (1s change)") is able to re-use state corresponding to the parts of the graph that have not changed. A differential dataflow system can distinguish between changes due to an updated input and those due to iterative execution, and re-use any appropriate previous state. In Figure 1 we see, when the initial 24-hour window slides by one second, that only 67 differences are processed by the system (which is typical across the duration of the trace), and in several iterations no work needs to be done. The work done updating the sliding window is only 0.003% of the work done in a full prioritized re-evaluation.

We will show in Section 5 that the reduction in differences corresponds to a reduction in the execution time, and it is possible to achieve multiple orders of magnitude in performance improvement for these types of computation.

3. DIFFERENTIAL COMPUTATION

In this section we describe how a differential computation keeps track of changes and updates its state. Since we will use the computation in later sections to implement data-parallel dataflow, we adopt the terminology of data-parallel dataflow systems here. The functions that must adapt to their changing inputs are called *operators*, and their inputs and outputs are called *collections*. We model collections as multisets, where for a collection A and record x the integer $A(x)$ indicates the multiplicity of x in A. Wherever an example in the paper describes a generic unary or binary operator it should be assumed that the extension to operators with more than two inputs is straightforward.

Collections may take on multiple *versions* over the lifetime of a computation, where the versions are members of some partial order. The set of versions of a particular collection is called a *collection trace*, denoted by a bold font, and defined to be a function from elements of the partial order $t \in (T, \leq_T)$ to collections; we write $\mathbf{A}_t$ for the collection at version t. As we shall see, different collections within a single computation may vary according to different partial orders. The result of applying an operator to a collection trace is itself a collection trace and this is indicated using a $[\cdot]_t$ notation; for example, for a generic binary operator

$$[\mathrm{Op}(A, B)]_t = \mathrm{Op}(\mathbf{A}_t, \mathbf{B}_t) \ .$$

The computation's inputs and outputs are modeled as collection traces and thus vary with a partial order. Typically inputs and outputs vary with the natural numbers, to indicate consecutive epochs of computation.

3.1 Incremental computation

In incremental computation, we consider sequences of collections, $\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A}_t$ and compute $\mathrm{Op}(\mathbf{A}_0), \mathrm{Op}(\mathbf{A}_1), \ldots, \mathrm{Op}(\mathbf{A}_t)$ for each operator. The most naïve way to do this (corresponding to the "Stateless" approach in Figure 1) is to re-execute $\mathrm{Op}(\mathbf{A}_t)$ independently for each t, as in Figure 2.

When successive $\mathbf{A}_t$ have a large intersection, we can achieve substantial gains through incremental evaluation. We can define the *difference* between two collections at subsequent versions in terms of a *difference trace*, analogous to a collection trace and once again taking on a value for each version of the collection. Differences and difference traces are denoted using a δ symbol applied to the name of the

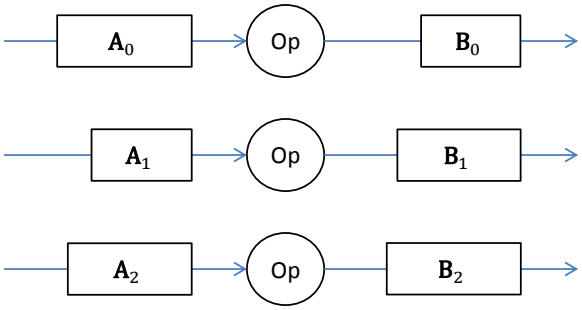


Figure 2: A sequence of input collections $\mathbf{A}_0, \mathbf{A}_1, \ldots$ and the corresponding output collections $\mathbf{B}_0, \mathbf{B}_1, \ldots$. Each is defined independently as $\mathbf{B}_t = \mathbf{Op}(\mathbf{A}_t)$.

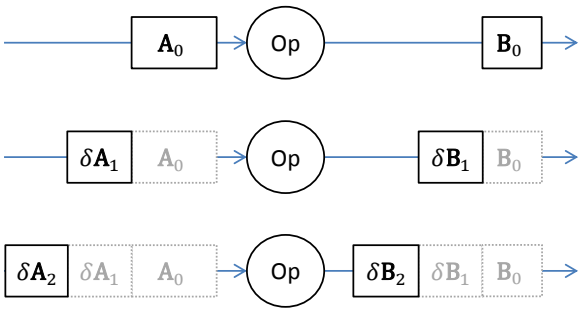


Figure 3: The same sequence of computations as in Figure 2, presented as differences from the previous collections. The outputs still satisfy $\mathbf{B}_t = \mathbf{Op}(\mathbf{A}_t)$, but are represented as differences $\delta\mathbf{B}_t = \mathbf{B}_t - \mathbf{B}_{t-1}$.

corresponding collection or trace. For each version $t > 0$, $\delta\mathbf{A}_t = \mathbf{A}_t - \mathbf{A}_{t-1}$, and $\delta\mathbf{A}_0 = \mathbf{A}_0$. It follows that

$$\mathbf{A}_t = \sum_{s \leq t} \delta\mathbf{A}_s. \tag{1}$$

Notice that $\delta\mathbf{A}_t(r)$ may be negative, corresponding to the removal of a record r from A at version t.

An operator can react to a new $\delta\mathbf{A}_t$ by producing the corresponding output $\delta\mathbf{B}_t$, as in Figure 3. Incremental systems usually compute

$$\delta\mathbf{B}_t = \mathrm{Op}(\mathbf{A}_{t-1} + \delta\mathbf{A}_t) - \mathrm{Op}(\mathbf{A}_{t-1})$$

and retain only the latest version of the collections $\mathbf{A}_t = \mathbf{A}_{t-1} + \delta\mathbf{A}_t$ and $\mathbf{B}_t = \mathbf{B}_{t-1} + \delta\mathbf{B}_t$, discarding $\delta\mathbf{A}_t$ and $\delta\mathbf{B}_t$ once they have been incorporated in their respective collections. For the generalization that follows it will be helpful to consider the equivalent formulation

$$\sum_{s \leq t} \delta\mathbf{B}_s = \mathrm{Op}\left(\sum_{s \leq t} \delta\mathbf{A}_s \right). \tag{2}$$

In practical incremental systems the operators are implemented to ensure that $\delta\mathbf{B}_t$ can usually be computed in time roughly proportional to $|\delta\mathbf{A}_t|$, as opposed to the $|\mathbf{A}_t|$ that would be required for complete re-evaluation.

Incremental evaluation and *cyclic* dependency graphs can be combined to effect iterative computations. Informally, for a loop body f mapping collections to collections, one can

reintroduce the output of f into its input. Iteration t determines $f^{t+1}(X)$ for some initial collection X, and can proceed for a fixed number of iterations, or until the collection stops changing. This approach is reminiscent of semi-naïve Datalog evaluation, and indeed incremental computation can be used to evaluate Datalog programs.

Unfortunately, the sequential nature of incremental computation implies that differences can be used either to update the computation's input collections or to perform iteration, but not both. To achieve both simultaneously, we must generalize the notion of a difference to allow multiple predecessor versions, as we discuss in the next subsection.

3.2 Generalization to partial orders

Here we introduce *differential computation*, which generalizes incremental computation. The data are still modeled as collections $\mathbf{A}_t$, but rather than requiring that they form a sequence, they may be *partially* ordered. Once computed, each individual difference is retained, as opposed to being incorporated into the current collection as is standard for incremental systems. This feature allows us to carefully combine differences according to reasons the collections may have changed, resulting in substantially smaller numbers of differences and less computation.

We must redefine *difference* to account for the possibility of $\mathbf{A}_t$ not having a single well-defined "predecessor" $\mathbf{A}_{t-1}$. Referring back to Equation 1, we use exactly the same equality as before, but s and t now range over elements of the partial order, and $\leq$ uses the partial order's less-than relation. The difference $\delta\mathbf{A}_t$ is then defined to be the difference between $\mathbf{A}_t$ and $\sum_{s<t} \delta\mathbf{A}_s$. We provide a few concrete examples in the next subsection.

As with incremental computation, each operator determines output differences from input differences using Equation 2. Rewriting, we can see that each $\delta\mathbf{B}_t$ is determined from $\delta\mathbf{A}_t$ and strictly prior $\delta\mathbf{A}_s$ and $\delta\mathbf{B}_s$:

$$\delta\mathbf{B}_t = \mathrm{Op}\Big(\sum_{s\leq t} \delta\mathbf{A}_s\Big) - \sum_{s<t} \delta\mathbf{B}_s \ . \tag{3}$$

One consequence of using a partial order is that—in contrast to incremental computation—there is not necessarily a one-to-one correspondence between input and output differences. Each new $\delta\mathbf{A}_t$ may produce $\delta\mathbf{B}_{t'}$ at multiple distinct $t' \geq t$. This complicates the logic for incrementalizing operators, and is discussed in more detail in Section 3.4.

3.3 Applications of differential computation

We now consider three examples of differential computation, to show how the use of differences deviates from prior incremental approaches. In particular, we will outline the benefits that accrue from both the composability of the abstraction, and the ability to redefine the partial order to select the most appropriate predecessors for a collection.

Ex 1: Incremental and iterative computation.

Imagine a collection $\mathbf{A}_{ij}$ that takes on different values depending on the round i of the input and the iteration j of a loop containing it. For example, $\mathbf{A}_{ij}$ could be the node labels derived from the j-hop neighborhood of the ith input epoch in the connected component example of Section 2. Consider the partial order for which

$$(i_1, j_1) \leq (i_2, j_2) \quad \text{iff} \quad i_1 \leq i_2 \text{ and } j_1 \leq j_2 \ .$$

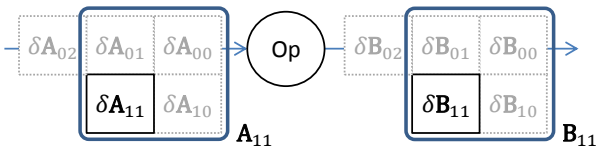


Figure 4: Differential computation in which multiple independent collections $\mathbf{B}_{ij} = \mathbf{Op}(\mathbf{A}_{ij})$ are computed. The rounded boxes indicate the differences that are accumulated to form the collections $\mathbf{A}_{11}$ and $\mathbf{B}_{11}$.

Figure 4 shows how differential computation based on this partial order would consume and produce differences.

Some of the differences $\mathbf{A}_{ij}$ are easily described:

$\delta\mathbf{A}_{00}$: The initial value of the collection (equal to $\mathbf{A}_{00}$).

$\delta\mathbf{A}_{01}$: $\mathbf{A}_{01} - \mathbf{A}_{00}$. Advances $\mathbf{A}_{00}$ to the second iteration.

$\delta\mathbf{A}_{10}$: $\mathbf{A}_{10} - \mathbf{A}_{00}$. Updates $\mathbf{A}_{00}$ to the second input.

Because neither $(0,1)$ nor $(1,0)$ is less than the other, neither $\delta\mathbf{A}_{01}$ nor $\delta\mathbf{A}_{10}$ is used in the derivation of the other. This independence would not be possible if we had to impose a total order on the versions, since one of the two would have to come first, and the second would be forced to subtract out any differences associated with the first.

It is instructive to consider difference $\delta\mathbf{A}_{11}$ and see the changes it reflects. Recall that $\mathbf{A}_{11} = \sum_{(i,j)\leq(1,1)} \delta\mathbf{A}_{ij}$, so

$$\delta\mathbf{A}_{11} = \mathbf{A}_{11} - (\delta\mathbf{A}_{00} + \delta\mathbf{A}_{01} + \delta\mathbf{A}_{10}) \ .$$

The difference $\delta\mathbf{A}_{11}$ reconciles the value of the collection $\mathbf{A}_{11}$ with the preceding differences that have already been computed: $\delta\mathbf{A}_{00}+\delta\mathbf{A}_{01}+\delta\mathbf{A}_{10}$. Note that not all previously computed differences are used: even though $\delta\mathbf{A}_{02}$ may be available, it describes the second loop iteration and is not useful for determining $\mathbf{A}_{11}$. Here the benefit of maintaining each $\delta\mathbf{A}_{ij}$ becomes apparent: the most appropriate set of differences can be used as a starting point for computing any given $\mathbf{A}_{ij}$. Consequently, the correction $\delta\mathbf{A}_{ij}$ can be quite slight, and indeed is often completely empty. In Figure 1, several iterations of the differential computation (3, 5, and after 11) are completely empty.

If a total order on differences were used, $\delta\mathbf{A}_{11}$ might be defined solely in terms of $\mathbf{A}_{11} - \mathbf{A}_{10}$. Despite already having computed $\delta\mathbf{A}_{01} = \mathbf{A}_{01} - \mathbf{A}_{00}$ (the effect of one iteration on what may be largely the same collection) the computation of $\delta\mathbf{A}_{11}$ would not have access to this information, and would waste effort in redoing some of the same work. The product partial order is a much better match for collections experiencing independent changes from two sources.

Ex 2: Prioritized and iterative computation.

Differential computation can also be used to implement the connected components optimization in which the smallest label is first propagated throughout the graph, followed by the second smallest label, and so on until all labels have been introduced [28]. This *prioritized* approach is more efficient because only the smallest label is propagated within a component: larger labels immediately encounter smaller labels, and are not propagated further.

To achieve this optimization, we use the lexicographic order, for which $(p_1, i_1) \leq (p_2, i_2)$ iff either $p_1 < p_2$ or $p_1 = p_2$ and $i_1 \leq i_2$. Each label l is propagated with priority l, and its propagation is reflected through the differences $\delta\mathbf{A}_{l0}, \delta\mathbf{A}_{l1}, \ldots$. When using the lexicographic order, $\delta\mathbf{A}_{p0}$ is taken with respect to $\mathbf{A}_{(p-1,\infty)}$, the limit of the computation at the previous priority, rather than $\mathbf{A}_{(p-1,0)}$. This results in fewer differences, as label l's progress is thwarted immediately at vertices that receive any lower label. The resulting sequential dependencies also reduce available parallelism, but this is mitigated in practice by batching the priorities, for example propagating label l with priority $\lfloor \log(l) \rfloor$.

This optimization is the basis of the distinction between the incremental and prioritized lines in Figure 1.

Ex 3: Composability and nesting.

An attractive feature of differential computation is its composability. As incremental composes with iterative, and prioritized with iterative, we can easily combine the three to create an incremental, prioritized, iterative computation using simple partial-order combiners (here, the product of the integer total order and the lexicographic order). The "Differential" line in Figure 1 was obtained with a combination of incremental, prioritized, and iterative computation. The resulting complexity can be hidden from the user, and results in real-world performance gains as we will see in Section 5.

Support for the composition of iterative computations enables nested loops: strongly connected components can be computed with an incremental, iterative, prioritized, iterative implementation (a four-dimensional partial order). We present the data-parallel algorithm for strongly connected components in the appendix.

3.4 Differential operators

We now describe the basic operator implementation that takes an arbitrary operator defined in terms of collections, and converts it to compute on differences. In the worst case this basic implementation ends up reconstructing the entire collection and passing it to the operator. Section 4.3 explains how most common operators in a differential dataflow implementation can be optimized to avoid this worst case.

As a differential computation executes, its operators are invoked repeatedly with differences to incorporate into their inputs, and must produce output difference traces that reflect the new differences. Consider a binary operator f which has already processed a sequence of updates for collection traces $\mathbf{A}$ and $\mathbf{B}$ on its two respective inputs. Suppose that new differences δa and δb must be applied to its respective inputs, where the differences in δa and δb all have version τ. Denoting the resulting updates to f's output by the difference trace $\delta\mathbf{z}$, Equation (3) indicates that

$$\delta\mathbf{z}_t = [f(\mathbf{A}+\mathbf{a}, \mathbf{B}+\mathbf{b})]_t - [f(\mathbf{A},\mathbf{B})]_t - \sum_{s<t} \delta\mathbf{z}_s$$

where

$$\mathbf{a}_t = \begin{cases} \delta a & \text{if } \tau \leq t \\ 0 & \text{otherwise} \end{cases}$$

and similarly for $\mathbf{b}$. It is clear by induction on t that $\delta\mathbf{z}_t = 0$ when $\tau \not\leq t$, which reflects the natural intuition that updating differences at version τ does not result in any modifications to versions before τ. What may be more surprising is that there can be versions $t > \tau$ for which $\delta\mathbf{z}_t \neq 0$ even if $\delta\mathbf{A}_t = 0$ and $\delta\mathbf{B}_t = 0$ for all $t > \tau$. Fortunately the set of versions that potentially require updates is not unbounded and in fact it can be shown that $\delta\mathbf{z}_t = 0$ if $t \notin T$ where T is the set of versions that are upper bounds of τ and some non-zero delta in $\delta\mathbf{A}$ or $\delta\mathbf{B}$:

$$T' = \{t' \ : \ \delta\mathbf{A}_{t'} \neq 0 \vee \delta\mathbf{B}_{t'} \neq 0\}, \text{ and}$$
$$T = \{t \ : \ t' \in T' \wedge t \text{ is the least upper bound of } t' \text{ and } \tau\}.$$

Algorithm 1 Pseudocode for operator update logic.

```
δz ← 0
for all elements t ∈ T do
  A_t ← δA_t
  B_t ← δB_t
  for all elements s ∈ lattice do
    if s < t ∧ (δA_s ≠ 0 ∨ δB_s ≠ 0 ∨ δz_s ≠ 0) then
      A_t ← A_t + δA_s
      B_t ← B_t + δB_s
      δz_t ← δz_t − δz_s
    end if
  end for
  δz_t ← δz_t + f(A_t + δa, B_t + δb) − f(A_t, B_t)
end for
δA_τ ← δA_τ + δa
δB_τ ← δB_τ + δb
return δz
```

In order to efficiently compute $\delta\mathbf{z}$ for arbitrary inputs, our basic operator must store its full input difference traces $\delta\mathbf{A}$ and $\delta\mathbf{B}$ indexed in memory. In the Naiad prototype implementation this trace is stored in a triply nested sparse array of counts, indexed first by key k, then by lattice version t, then by record r. Naiad maintains only non-zero counts, and as records are added to or subtracted from the difference trace Naiad dynamically adjusts the allocated memory.

With $\delta\mathbf{A}$ and $\delta\mathbf{B}$ indexed by version, $\mathbf{A}_t$ and $\mathbf{B}_t$ can be reconstructed for any t, and $\delta\mathbf{z}$ computed explicitly, using the pseudocode of Algorithm 1. While reconstruction may seem expensive, and counter to incremental computation, it is necessary to be able to support fully general operators for which the programmer may specify an arbitrary (non-incremental) function to process all records. We will soon see that many specific operators have more efficient implementations.

One general optimization to the algorithm in Algorithm 1 reduces the effort spent reconstructing values of $\mathbf{A}_t$ and $\mathbf{B}_t$. Rather than loop over all $s < t$ for each t the system can update the previously computed collection, say $\mathbf{A}_{t'}$ at version t'. Doing so only involves differences

$$\{\delta\mathbf{A}_s : (s \leq t') \neq (s \leq t)\} .$$

This often results in relatively few s to update, for example just one in the case of advancing loop indices. By ensuring that differences are processed in a sequence that respects the partial order, the system need only scan from the greatest lower bound of t' and t until it passes both t' and t. Additionally, if updates are available at more than one version τ they can be batched, again potentially reducing the number of collections that need to be reconstructed and the number of evaluations of f.

The above explanation assumes that difference traces $\delta\mathbf{A}$ will be kept around indefinitely, and therefore that the cost of the reconstruction looping over $s < t$ in Algorithm 1 will

grow without bound as t increases. In practice, $\delta\mathbf{A}$ can be thought of like a (partially ordered) log of updates that have occurred so far. If we know that no further updates will be received for any versions $t < t_0$ then all the updates up to version t_0 can be consolidated into the equivalent of a checkpoint, potentially saving both storage cost and computational effort in reconstruction. The Naiad prototype includes this consolidation step, but the details are beyond the scope of this paper.

4. DIFFERENTIAL DATAFLOW

We now present our realization of differential computation: differential dataflow. As discussed in Section 6, incremental computation has been introduced in a wide variety of settings. We chose a declarative dataflow framework for the first implementation of differential computation because we believe it is well suited to the data-parallel analysis tasks that are our primary motivating application.

In common with existing work on query planning and data-parallel processing, we model a dataflow computation as a directed graph in which vertices correspond to program inputs, program outputs, or operators (e.g. `Select`, `Join`, `GroupBy`), and edges indicate the use of the output of one vertex as an input to another. In general a dataflow graph may have multiple inputs and outputs. A dataflow graph may be cyclic, but in the framework of this paper we only allow the system to introduce cycles in support of fixed-point subcomputations.

4.1 Language

Our declarative query language is based on the .NET Language Integrated Query (LINQ) feature, which extends C# with declarative operators, such as `Select`, `Where`, `Join` and `GroupBy`, among others, that are applied to strongly typed collections [5]. Each operator corresponds to a dataflow vertex, with incoming edges from one or two source operators.

We extend LINQ with two new query methods to exploit differental dataflow:

```
// result corresponds to body^infty(source)
Collection<T> FixedPoint(Collection<T> source,
                Func<Collection<T>,Collection<T>> body)

// FixedPoint variant which sequentially introduces
// source records according to priorityFunc
Collection<T> PrioritizedFP(Collection<T> source,
                Func<T, int> priorityFunc,
                Func<Collection<T>,Collection<T>> body)
```

`FixedPoint` takes a source collection (of some record type `T`), and a function from collections of `T` to collections of the same type. This function represents the body of the loop, and may include nested `FixedPoint` invocations; it results in a cyclic dataflow subgraph in which the result of the body is fed back to the next loop iteration.

`PrioritizedFP` additionally takes a function, `priorityFunc`, that is applied to every record in the source collection and denotes the order in which those records should be introduced into the body. For each unique priority in turn, records having that priority are added to the current state, and the loop iterates to fixed-point convergence on the records introduced so far. We will explain the semantics more precisely in the following subsection.

The two methods take as their bodies arbitrary differential dataflow queries, which may include further looping and sequencing instructions. The system manages the complexity of the partial orders, and hides the details from the user.

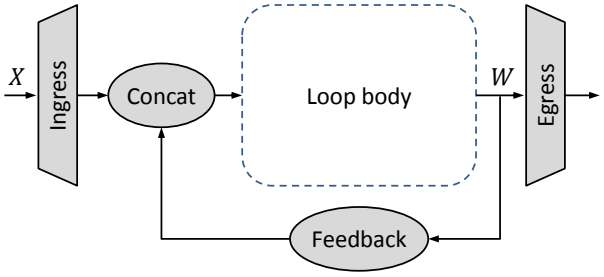


Figure 5: The dataflow template for a computation that iteratively applies the loop body to the input X, until fixed-point is reached.

4.2 Collection dataflow

In this subsection, we describe how to transform a program written using the declarative language above into a cyclic dataflow graph. We describe the graph in a standard dataflow model in which operators act on whole collections at once, because this simplifies the description of operator semantics. In Section 4.3 we will describe how to modify the dataflow operators to operate on differences, and Section 4.4 sketches how the system schedules computation.

Recall from Section 3.2 that collection traces model collections that are versioned according to a partial order. We require that all inputs to an operator vary with the same partial order, but a straightforward order embedding exists for all partial orders that we consider, implemented using the `Extend` operator:

$$[\texttt{Extend}(\mathbf{A})]_{(t,i)} = \mathbf{A}_t \ .$$

The `Extend` operator allows collections defined outside a fixed-point loop to be used within it. For example, the collection of edges in a connected components computation is constant with respect to the loop iteration i, and `Extend` is used when referring to the edges within the loop.

Standard LINQ operators such as `Select`, `Where`, `GroupBy`, `Join`, and `Concat` each correspond to single vertices in the dataflow graph and have their usual collection semantics lifted to apply to collection traces.

Fixed-point operator.

Although the fixed-point operator is informally as simple as a loop body and a back edge, we must carefully handle the introduction and removal of the new integer coordinate corresponding to the loop index. A fixed-point loop can be built from three new operators (Figure 5): an *ingress* vertex that extends the partial order to include a new integer coordinate, a *feedback* vertex that provides the output of the loop body as input to subsequent iterations, and an *egress* vertex that strips off the loop index from the partial order and returns the fixed point. (The standard `Concat` operator is used to concatenate the outputs of the ingress and feedback vertices.)

More precisely, if the input collection X already varies with a partial order T, the ingress operator produces the

trace varying with $T \times \mathbb{N}$ for which

$$[\texttt{Ingress}(\mathbf{X})]_{(t,i)} = \begin{cases} \mathbf{X}_t & \text{if } i = 0 \\ 0 & \text{if } i > 0 \,. \end{cases}$$

The feedback operator takes the output of the loop body and advances its loop index. For the output of the loop body W, we have

$$[\texttt{Feedback}(\mathbf{W})]_{(t,i)} = \begin{cases} 0 & \text{if } i = 0 \\ \mathbf{W}_{(t,i-1)} & \text{if } i > 0 \,. \end{cases}$$

Finally, the egress operator observes the output of the loop body and emits the first repeated collection

$$[\texttt{Egress}(\mathbf{W})]_t = \mathbf{W}_{(t,i^*)} \,,$$

where $i^* = \min \{i \ : \ \mathbf{W}_{(t,i)} = \mathbf{W}_{(t,i-1)}\}$.

We have said nothing specific about the implementation of these operators, but their mathematical definitions should make it clear that

$$[\texttt{Egress}(\mathbf{W})]_t = \lim_{i \to \infty} F^i(\mathbf{X}_t)$$

where this limit exists.

Prioritized fixed-point operator.

This operator assigns a priority to each record in a collection, and uses this priority to impose a total order on the introduction of records into a fixed-point loop. Starting from an empty collection, the operator sequentially introduces records at the next unintroduced priority to the collection, iterates to a fixed point (as above) and uses the result as the starting point for the next priority.

The prioritized fixed-point operator makes use of the same dataflow template as its unprioritized counterpart, comprising ingress, feedback, egress and `Concat` operators (Figure 5), but it has different semantics. The ingress operator adds *two* coordinates to each record's version, corresponding to its evaluated priority (p) and the initial iteration ($i = 0$):

$$[\texttt{PIngress}(\mathbf{X})]_{(t,p,i)}(r) = \begin{cases} \mathbf{X}_t(r) & \text{if } P(r) = p \wedge i = 0 \\ 0 & \text{otherwise} \end{cases}$$

where $P(r)$ is the evaluation of `priorityFunc` on record r. The additional coordinates (p, i) are ordered lexicographically, as described in Subsection 3.3.

The feedback operator plays a more complicated role. For the zeroth iteration of each priority, it feeds back the fixed-point of iteration on the previous priority; otherwise it acts like the unprioritized feedback.

$$[\texttt{PFeedback}(\mathbf{W})]_{(t,p,i)} = \begin{cases} \mathbf{W}_{(t,p-1,i^*_{p-1})} & \text{if } p > 0 \wedge i = 0 \\ \mathbf{W}_{(t,p,i-1)} & \text{if } i > 0 \\ 0 & \text{if } p = i = 0 \end{cases}$$

where $i^*_p = \min \{i \ : \ \mathbf{W}_{(t,p,i)} = \mathbf{W}_{(t,p,i-1)}\}$.

Finally, the egress operator is modified to emit the fixed-point after the final priority,

$$q = \max \{P(r) \ : \ \mathbf{X}_t(r) \neq 0\},$$

has been inserted:

$$[\texttt{PEgress}(\mathbf{W})]_t = \mathbf{W}_{(t,q,i^*_q)} \,.$$

4.3 Operator implementations

Section 3.4 outlined the generic implementation of a differential operator. Although the generic operator update algorithm can be used to implement any differential dataflow operator, we have specialized the implementation of the following operators to achieve better performance:

Data-parallel operation.

Exploiting data-parallel structure is one of the most effective ways to gain benefit from differential dataflow. For each operator instance f in the dataflow assume that there is a key type K, and a key function defined for each of the operator's inputs that maps records in that input to K. The key space defines a notion of independence for f, which can be written as

$$f(A, B) = \sum_{k \in K} f(A|_k, B|_k) \tag{4}$$

where a restriction $A|_k$ (or $B|_k$) is defined in terms of its associated key function key as

$$(A|_k)(r) = A(r) \ \text{ if } \ key(r) = k, \ 0 \ \text{ otherwise.} \tag{5}$$

Such independence properties are exploited in many systems to parallelize computation, since subsets of records mapping to distinct keys can be processed on different CPUs or computers without the need for synchronization. A differential dataflow system can exploit parallelism in the same way, but also crucially benefits from the fact that updates to collections can be isolated to keys that are present in incoming differences, so an operator need only perform work on the subsets of a collection that correspond to those keys. In common cases both the size of the incoming differences and computational cost to process them are roughly proportional to the size of these subsets. It is easy to modify the pseudocode in Algorithm 1 to operate only on records mapping to key k, and since $\delta\mathbf{A}$ and $\delta\mathbf{B}$ are indexed by key it is therefore easy to do work only for subsets of $\delta\mathbf{A}$ and $\delta\mathbf{B}$ for which $\delta a|_k \neq 0$ or $\delta b|_k \neq 0$.

Operators such as `Join` and `GroupBy` naturally include key functions as part of their semantics. For aggregates such as `Count`, `Sum` and `Min`, we adopt a slightly non-standard definition that effectively prepends each operator with a `GroupBy`. For example, `Count` requires a key function and returns a set of counts, corresponding to the number of records that map to each unique key in the collection. The standard behavior of these operators can be obtained by specifying a constant key function that maps every record to the same key.

Pipelined operators.

Several operators—including `Select`, `Where`, `Concat` and `Except`—are *linear*, which means they can determine $\delta\mathbf{z}$ as a function of only δa, with no dependence on $\delta\mathbf{A}$. These operators can be pipelined with preceding operators since they do not need to maintain any state and do not need to group records based on key: they apply record-by-record logic to the non-zero elements of δa—respectively transforming, filtering, repeating and negating the input records.

Join.

The `Join` operator combines two input collections by computing the Cartesian product of those collections, and yielding only those records where both input records have the

same key. Due to the distributive property of `Join`, the relationship between inputs and outputs is simply

$$\mathbf{z} = \mathbf{A} \bowtie \mathbf{b} + \mathbf{a} \bowtie \mathbf{B} + \mathbf{a} \bowtie \mathbf{b} \,.$$

While the implementation of `Join` must still keep its input difference trace resident, its implementation is much simpler than the generic case. An input δa can be directly joined with the non-zero elements of $\delta\mathbf{B}$, and analogously for δb and $\delta\mathbf{A}$, without the overhead of following the reconstruction logic in Algorithm 1.

Aggregations.

Many data-parallel aggregations have very simple update rules that do not require all records to be re-evaluated. `Count`, for example, only needs to retain the difference trace of the number of records for each key, defined by the cumulative weight, rather than the set of records mapping to that key. `Sum` has a similar optimization. `Min` and `Max` must keep their full input difference traces—because the retraction of the minimal (maximal) element leads to the second-least (greatest) record becoming the new output—but can often quickly establish that an update requires no output by comparing the update to the prior output without reconstructing $\mathbf{A}$.

Fixed-point operators.

The `Extend`, `Ingress`, `Feedback`, and `Egress` operators from Section 4.2 have simple differential implementations. The `Extend` operator reports the same output for any i, so

$$[\delta\texttt{Extend}(\delta\mathbf{X})]_{(t,i)} = \begin{cases} \delta\mathbf{X}_t & \text{if } i = 0 \\ 0 & \text{if } i > 0 \,. \end{cases}$$

The `Ingress` operator changes its output from zero to $\delta\mathbf{X}_t$ then back to zero, requiring outputs of the form

$$[\delta\texttt{Ingress}(\delta\mathbf{X})]_{(t,i)} = \begin{cases} \delta\mathbf{X}_t & \text{if } i = 0 \\ -\delta\mathbf{X}_t & \text{if } i = 1 \\ 0 & \text{if } i > 1 \,. \end{cases}$$

The `Feedback` operator is initially zero, but then changes as the previous iterate of its input changes.

$$[\delta\texttt{Feedback}(\delta\mathbf{W})]_{(t,i)} = \begin{cases} 0 & \text{if } i = 0 \\ \delta\mathbf{W}_{(t,i-1)} & \text{if } i > 0 \,. \end{cases}$$

The `Egress` operator produces the final output seen, which is the result of all accumulations seen so far

$$[\delta\texttt{Egress}(\delta\mathbf{W})]_t = \sum_{0 \le i \le i^*} \delta\mathbf{W}_{(t,i)} \,.$$

Informally stated, `Ingress` adds a new loop index and produces a positive and negative output for each input seen, `Feedback` advances the loop index of each input seen, and `Egress` removes the loop index of each input seen. The differential implementations of the prioritized fixed-point operators `PIngress`, `PFeedback` and `PEgress` follow in similar fashion.

4.4 Scheduling differential dataflow

Scheduling the execution of a differential dataflow computation is complicated by the need to reconcile cyclic data dependencies. In our Naiad prototype, the scheduler keeps track of the outstanding differences to be processed at each operator, and uses the topology of the dataflow graph to impose a partial order on these differences, enabling the system to sort them topologically and thereby obtain a valid schedule. The challenge is that the version associated with each difference orders two outstanding differences at the same operator, but says nothing in the case that there are outstanding differences for two distinct operators.

Intuitively, there is a notion of causality: a difference d_1 at operator Op_1 with version s *causally precedes* d_2 at Op_2 with version t if processing d_1 can possibly result in new data for Op_2 at a version $t' \le t$. Recall from Section 4.2 that some operators modify the version of an incoming difference: for example, the unprioritized `Feedback` operator advances the last coordinate of the version. The scheduler combines this information with the edge relation of the dataflow graph to determine the causal order and identify a set of minimal outstanding differences. Thereafter, repeatedly scheduling one of the minimal differences ensures that forward progress is made.

Whereas some iterative data-parallel systems rely on an explicit convergence test [7, 20, 25], in a differential dataflow system convergence is implied by the absence of differences. Therefore, if no outstanding differences remain, all of the input has been processed and all loops have converged to fixed points.

4.5 Prototype implementation

The Naiad prototype transforms declarative queries to a dataflow graph that may contain cycles. The user program can insert differences into input collections, and register callbacks to be informed when differences are received at output collections. The Naiad runtime distributes the execution of the dataflow graph across several computing elements (threads and computers) to exploit data-parallelism. Since operators in a differential dataflow system often compute for only a short time before sending resulting output to another computer, many of the design decisions were guided by the need to support low latency communication and coordination.

The other technical challenges that Naiad faces relate to new trade-offs that differential dataflow exposes. Many "big data" systems leverage data-parallelism heavily, as there is always a substantial amount of work available. Differential dataflow reduces this work significantly, and we must reconsider many of the common implementation patterns to ensure that its benefits are not overshadowed by overheads. For example, unlike many other distributed data-processing systems, Naiad maintains each operator's input collections (as differences) deserialized and *indexed* in memory, to allow microsecond-scale reaction to small updates. Naiad's workers operate asynchronously and independently, rather than under the instruction of a central coordinator. Most of Naiad's internal data structures are designed to amortize computation, so that they never stall for extended periods of time. Many of these properties are already in evidence in modern database systems, but their significance for big data systems is only revealed once the associated computational models are sufficiently streamlined.

5. APPLICATIONS

To support the claim that differential dataflow can lead to substantial performance improvements for incremental and iterative computations, we now describe some example

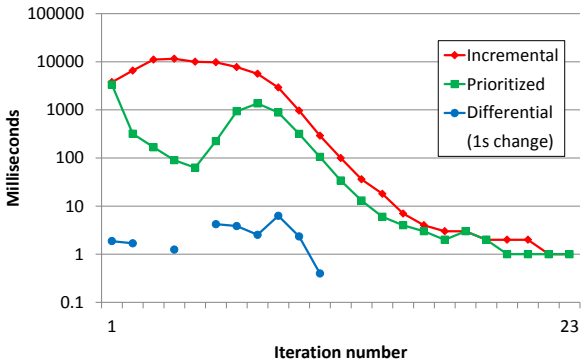


Figure 6: Execution time for each iteration of the connected components computation on the Twitter graph, as described in Section 2 (cf. Figure 1, which shows the number of label changes in each iteration). Plotted values are the medians of nine executions.

applications, and present initial performance measurements taken using the Naiad prototype.[1]

5.1 Twitter connected components

We measured the per-iteration execution times for the connected components computation described in Section 2. We performed the experiments on an AMD Opteron 'Magny-Cours' with 48 (four 12-core) 1.9GHz processors and 64GB of RAM, running Windows Server 2008 R2 Enterprise Service Pack 1. Figure 6 shows the times for the Incremental, Prioritized, and Differential (1s change) versions of the computation, when executed using eight cores. Notice that the curves exhibit the same relative ordering and roughly the same shape as the counts of differences in Figure 1. Compared to Figure 1, the one-second update is separated by fewer orders of magnitude from the 24-hour differential computation. This lower-than-expected speedup is due to per-iteration overheads that become more apparent when the amount of work is so small. Nonetheless, Naiad is able to respond to one second of updates in 24.4ms; this is substantially faster than the 7.1s and 36.4s used by either differential or incremental dataflow, and makes it possible for Naiad to maintain the component structure of the Twitter mention graph in real time.

5.2 Iterative web-graph algorithms

We have also assessed Naiad's performance on several graph algorithms applied to the Category B web-graph from ClueWeb.[2] We draw on the work of Najork *et al.* [22], which assesses the performance, scalability and ease of implementation of several algorithms on three different types of platform: the Microsoft SQL Server 2008 R2 Parallel Data Warehouse (PDW) relational database, the DryadLINQ [24] data-parallel batch processor, and the Scalable Hyperlink Store (SHS) [21] distributed in-memory graph store. To allow a direct comparison, we run the distributed version of Naiad on the same experimental cluster used by Najork *et al.*: 16 servers with eight cores (two quad-core Intel Xeon E5430 processors at 2.66GHz) and 16GB RAM, all connected to a single Gigabit Ethernet switch.

[1]Available to download from `http://research.microsoft.com/naiad`.

[2]`http://boston.lti.cs.cmu.edu/Data/clueweb09/`

Algorithm	PDW	DryadLINQ	SHS	Naiad
Pagerank	8,970	4,513	90,942	1,404
SALSA	2,034	439	163	–
SCC	475	446	1,073	234
WCC	4,207	3,844	1,976	130
ASP	30,379	17,089	246,944	3,822

Table 1: Running times in seconds of several algorithms and systems on the Category B web graph. The first three systems measurements are from [22].

Table 1 presents the results where we see Naiad's general improvement due to a combination of its ability to store data indexed in memory, distribute computation over many workers, and accelerate iterative computations as they converge.[3] Notably, each other system implements only a trimming pre-processing step for SCC, and then runs single-threaded SCC on the reduced graph; Naiad is capable of expressing the SCC computation as a declarative doubly nested fixed-point computation, and distributes the full execution across the cluster. None of these workloads are interactive, and the measurements do not exploit Naiad's ability to support incremental updates. Nevertheless, each computation is automatically incrementalized, and could respond efficiently to changes in the input graph.

6. RELATED WORK

Many approaches to incremental execution have been investigated. To the best of our knowledge, differential dataflow is the first technique to support programs that combine arbitrary nested iteration with the efficient addition and removal of input data. However, the existing research in incremental computation has uncovered techniques that may be complementary to differential computation, and in this section we attempt to draw connections between the related work in this field.

Incremental view maintenance.

As noted earlier, differential dataflow addresses a similar problem to that tackled by incremental view maintenance (IVM), where the aim is to reuse the work done on the previous input when computing a new view based on a slightly different input. Over the past three decades, the set of supported queries has grown from simple select-project-join queries [6], to fully general recursive queries [15, 23]. While the latter techniques are very general, they are not ideal for interactive large-scale computation, because they either perform too much work, maintain too much state or limit expressiveness. Gupta *et al.*'s classic DRed algorithm [15] can over-estimate the set of invalidated tuples and will, in the worst case, perform a large amount of work to "undo" the effects of a deleted tuple, only to conclude that the best approach is to start from scratch. Nigam *et al.*'s extended PSN algorithm [23] relies on storing with each tuple the full set of tuples that were used to derive it, which can require a prohibitive amount of state for a large computation. Ahmad *et al.* have improved incremental performance on queries containing higher-order joins, but do not currently support it-

[3]We are unable to present results for SALSA, as it uses a query set that is not distributed with the ClueWeb dataset.

erative workloads [3]; this approach could be adapted to the benefit of differential programs containing such joins.

Incremental dataflow.

Dataflow systems like MapReduce and Dryad have been extended with support for incremental computation. Condie *et al.* developed MapReduce Online [8], which maintains state in memory for a chain of MapReduce jobs, and reacts efficiently to additional input records. Incremental dataflow can also be useful for coarse-grained updates: Gunda *et al.* subsequently developed Nectar [14], which caches the intermediate results of DryadLINQ programs and uses the semantics of LINQ operators to generate incremental programs that exploit the cache. The Incoop project [4] provides similar benefits for arbitrary MapReduce programs, by caching the input to the reduce stage and carefully ensuring that a minimal set of reducers is re-executed upon a change to the input. None of these systems has support for iterative algorithms, rather, they are designed for high throughput on very large data.

Iterative dataflow.

To extend the generality of dataflow systems, several researchers have investigated ways of adding data-dependent control flow constructs to parallel dataflow systems.

HaLoop [7] is an extended version of MapReduce that can execute queries written in a variant of recursive SQL, by repeatedly executing a chain of MapReduce jobs until a data-dependent stopping criterion is met. Similar systems include Twister [12] and iMapReduce [27]. Spark [25] supports a programming model that is similar to DryadLINQ, with the addition of explicit in-memory caching for frequently re-used inputs. Spark also provides a "resilient distributed dataset" abstraction that allows cached inputs to be reconstructed in the event of failure. All of these systems use an execution strategy that is similar to the collection-oriented dataflow described in Section 4.2, and would perform work that is proportional to the "Stateless" line in Figure 1. D-Streams [26] extends Spark to handle streaming input by executing a series of small batch computations, but it does not support iteration. The CIEL distributed execution engine [20] offers a general execution model based on "dynamic task graphs" that can encode nested iteration; however because CIEL does not support mutable data objects, it would not be practical to encode the fine-grained modifications to operator state traces that occur during a differential dataflow computation.

More recently, several iterative dataflow systems supporting incremental fixed-point iteration have been developed, and these achieve performance proportional to the "Incremental" line in Figure 1. Ewen *et al.* extended the Nephele execution engine with support for "bulk" and "incremental" iterations [13], where monotonic iterative algorithms can be executed using a sequence of incremental updates to the current state. Mihaylov *et al.* developed REX [19], which additionally supports record deletion in incremental iteration, but the programmer is responsible for writing incremental versions of user-defined functions (UDFs). The differential operator update algorithm (Algorithm 1) would automatically incrementalize many UDFs, but the lack of a partial order on updates would limit its usefulness. Finally, Conway *et al.* recently introduced Bloom^L [9], which supports fixed-point iteration using compositions of monotone functions on a variety of lattices. The advantage of this approach is that it is possible to execute such programs in a distributed system without blocking, which may be more efficient than Naiad's current scheduling policy (Section 4.4), but it does not support retractions or non-monotonic computations.

Alternative execution models.

Automatic techniques have been developed to incrementalize programming models other than dataflow. The basic technique for purely functional programs is memoization [18] which has been applied to a variety of existing systems [14, 20]. Acar pioneered *self-adjusting computation* [1], which automatically incrementalizes programs with mutable state by recording an execution trace and replaying only those parts of the trace that are directly affected when a variable is mutated. While the general approach of self-adjusting computation can be applied to any program, it is often more efficient to use "traceable" data types [2], which are abstract data types that support high-level query and update operations with a more compact representation in the trace.

Reactive imperative programming [11] is a programming model that uses *dataflow constraints* to perform updates to program state: the runtime tracks mutations to "reactive" variables, which may trigger the evaluation of constraints that depend on those variables. The constraints in such programs may be cyclic, which enables algorithms such as connected components and single-source shortest paths to be expressed in this model. However, convergence is only guaranteed for programs where the constraints have a monotonic effect on the program state, which makes it difficult to express edge deletion in a reactive imperative program.

In principle, traceable data types or high-level dataflow constraints could be used to implement differential computation. Furthermore, differential dataflow could benefit in many cases from incrementalized user-defined functions (particularly user-defined `GroupBy` reduction functions), and the techniques of self-adjusting computation offer the potential to do this automatically.

7. CONCLUSIONS

We have presented differential computation, which generalizes existing techniques for incremental computation. Differential computation is uniquely characterized by the fact that it enables arbitrarily nested iterative computations with general incremental updates. Our initial experimentation with Naiad—a data-parallel differential dataflow system—shows that the technique can enable applications that were previously intractable and achieve state of the art performance for several real-world applications.

These promising results in the context of dataflow lead us to conclude that the techniques of differential computation deserve further study, and have the potential to similarly enhance other forms of incremental computation.

8. REFERENCES

[1] U. A. Acar. *Self-adjusting computation.* PhD thesis, Carnegie Mellon University, 2005.

[2] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Turkoglu. Traceable data types for self-adjusting computation. In *ACM PLDI*, 2010.

[3] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. In *38th VLDB*, Aug. 2012.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *2nd ACM SOCC*, Oct. 2011.

[5] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to C♯. In *22nd OOPSLA*, Oct. 2007.

[6] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *1986 ACM SigMod*, 1986.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *36th VLDB*, Sept. 2010.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *7th USENIX NSDI*, 2010.

[9] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *3rd ACM SOCC*, 2012.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th USENIX OSDI*, 2004.

[11] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *26th OOPSLA*, 2011.

[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *19th ACM HPDC*, June 2010.

[13] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. In *38th VLDB*, 2012.

[14] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *9th USENIX OSDI*, Oct. 2010.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *1993 ACM SigMod*, 1993.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, Mar. 2007.

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *2010 ACM SigMod*, June 2010.

[18] D. Michie. "Memo" functions and machine learning. *Nature*, (218):19–22, Apr. 1968.

[19] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: recursive, delta-based data-centric computation. In *38th VLDB*, 2012.

[20] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *8th USENIX NSDI*, Mar. 2011.

[21] M. Najork. The scalable hyperlink store. In *20th ACM Conference on Hypertext and Hypermedia*, 2009.

[22] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, and S. Gollapudi. Of hammers and nails: An empirical comparison of three paradigms for processing large graphs. In *5th ACM WSDM*, Feb. 2012.

[23] V. Nigam, L. Jia, B. T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. In *13th ACM PPDP*, July 2011.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX OSDI*, Dec. 2008.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX NSDI*, Apr. 2012.

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *4th USENIX HotCloud*, 2012.

[27] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. In *1st International Workshop on Data Intensive Computing in the Clouds*, May 2011.

[28] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *2nd ACM SOCC*, Oct. 2011.

Demo: sliding strongly connected components

In this demonstration, we show how Naiad can compute the strongly connected component (SCC) structure of the mention graph extracted from a time window of the Twitter stream, and then extend this to build an interactive application that uses Naiad to track the evolution of these components as the window slides back and forth in time.

Background.

The classic SCC algorithm is based on depth-first search and not easily parallelizable. However, by nesting two connected components queries (Figure 7) inside an outer `FixedPoint`, we can write a data-parallel version using Naiad (Figure 8). Strictly speaking, the `ConnectedComponents` query computes directed reachability, and the SCC algorithm repeatedly removes edges whose endpoints reach different components and must therefore be in different SCCs. Iteratively trimming the graph in alternating directions—by reversing the edges in each iteration—eventually converges to the graph containing only those edges whose endpoints are in the same SCC.

Although Naiad's declarative language makes it straightforward to nest a `FixedPoint` loop, the resulting dataflow graph is quite complicated. Figure 9 shows a simplified version with some vertices combined for clarity: in our current implementation the actual dataflow graph for this program contains 58 vertices. Nonetheless, the SCC program accepts incremental updates, and differential dataflow enables the doubly nested fixed-point computation to respond efficiently when its inputs change.

Demo.

The interactive demo shows Naiad continually executing the SCC query described above. The input is a month of tweets from the full Twitter firehose, and we compute the SCCs formed by the Twitter mention graph within a given time window. A graphical front-end lets us slide the window of interest forward and backward (in steps of at least one second), and shows how the set of SCCs changes as the Naiad system re-executes the query incrementally. In addition, we maintain a continuous top-k query on the results of each successive SCC computation, and display the most popular hashtag within each component.

As incremental outputs are produced (in real-time with respect to the Twitter stream), the GUI is automatically refreshed to show the relative size and the most popular term of the SCCs computed by Naiad. The user is able to then investigate the 'hot topics' during the window, and we can even relate specific conversations to actual events that occurred at the time (for example, we see a component favoring the hashtag `#yankees` at about the same time that an important baseball game took place).

The demo highlights the responsiveness of Naiad while executing a complicated incremental query that contains a doubly nested loop. We argue that SCC is representative of the sophisticated data analysis that is increasingly important in contexts ranging from data warehousing and scientific applications through to web applications and social networking. Our demo emphasizes the power of efficiently composing incremental update, iterative computation, and interactive data analysis in a single declarative query.

```
// produces a (src, label) pair for each node in the graph
Collection<Node> ConnectedComponents(Collection<Edge> edges)
{
  // start each node with its own label, then iterate
  return edges.Select(x => new Node(x.src, x.src))
              .FixedPoint(x => LocalMin(x, edges));
}

// improves an input labeling of nodes by considering the
// labels available on neighbors of each node as well
Collection<Node> LocalMin(Collection<Node> nodes,
                          Collection<Edge> edges)
{
  return nodes.Join(edges, n => n.src, e => e.src,
                    (n, e) => new Node(e.dst, n.label))
              .Concat(nodes)
              .Min(node => node.src, node => node.label);
}
```

Figure 7: Connected components in Naiad.

```
// returns edges between nodes within a SCC
Collection<Edge> SCC(Collection<Edge> edges)
{
  return edges.FixedPoint(y => TrimAndReverse(
                               TrimAndReverse(y)));
}

// returns edges whose endpoints reach the same node, flipped
Collection<Edge> TrimAndReverse(Collection<Edge> edges)
{
  // establish labels based on reachability
  var labels = ConnectedComponents(edges);

  // struct LabeledEdge(a,b,c,d): edge (a,b); labels c, d;
  return edges.Join(labels, x => x.src, y => y.src,
                    (x, y) => x.AddLabel1(y))
              .Join(labels, x => x.dst, y => y.src,
                    (x, y) => x.AddLabel2(y))
              .Where(x => x.label1 == x.label2)
              .Select(x => new Edge(x.dst, x.src));
}
```

Figure 8: Strongly connected components in Naiad.

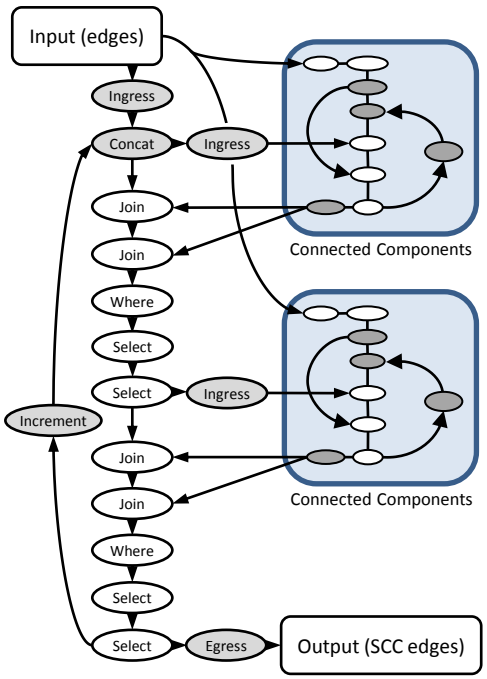


Figure 9: Simplified dataflow for strongly connected components. The outer loop contains two nested instances of the `ConnectedComponents` query.