Graphsurge: Graph Analytics on View Collections Using Differential Computation

Siddhartha Sahu  
University of Waterloo  
Waterloo, Canada  
s3sahu@uwaterloo.ca

Semih Salihoglu  
University of Waterloo  
Waterloo, Canada  
semih.salihoglu@uwaterloo.ca

ABSTRACT
This paper presents the design and implementation of a new open-source view-based graph analytics system called Graphsurge. Graphsurge is designed to support applications that analyze multiple snapshots or views of a large-scale graph. Users program Graphsurge through a declarative graph view definition language to create views over input graphs and a Differential Dataflow-based programming API to write analytics computations. A key feature of GVDL is the ability to organize views into view collections, which allows Graphsurge to share computation across views by performing computations differentially. We then introduce two optimization problems that naturally arises in our setting. First is the collection ordering problem to determine the order of views that leads to minimum differences across consecutive views. We prove this problem is NP-hard and show a constant-factor approximation algorithm drawn from literature. Second is the collection splitting problem to decide on which views to run computations differentially vs from scratch, for which we present an adaptive solution that makes decisions at runtime. Graphsurge is implemented on top of the Timely and Differential Dataflow systems. We present extensive experiments to demonstrate the benefits of running computations differentially for view collections and our collection ordering and splitting optimizations.

1. INTRODUCTION
A variety of applications, such as fraud detection, risk assessment and recommendations from telecommunications, finance, social networking, biological brain networks, and many other fields, process large-scale connected data among different entities [28]. Developers of these applications naturally model such connected data as graphs. Many of these applications require the ability to analyze different snapshots or views of a large-scale static graph, often based on selecting subsets of nodes or edges that satisfy different predicates. We refer to these views as filtered views. We first review several of these applications that motivate our current work. Figure 1 shows a call graph that we use as a running example throughout the paper. In the graph, customers are represented as nodes with profession and city properties. Phone calls are represented as edges between nodes with duration and date properties, written in curly brackets, respectively.

Example 1. Researchers and practitioners study the changes in structural properties of graphs across different views. A popular example is historical analyses of graphs where nodes or edges have some time property. A network scientist might study the history of the connectivity of the call graph from Figure 1 and compute one view of the graph for each year between 2010 to 2020. Similarly, the analyst can study the history of more complex views, where each view contains only the calls up to certain duration, say for \(\leq 1, 5, \) or 10 minutes. A classic example of such analyses from literature is reference [23] that studied the component size, vertex degrees, and diameters of different time-windows in time-stamped citation and web graphs and under different selection criteria of vertices, e.g., those belonging to a particular component or without incoming edges. In other settings, applications may study the history of social or e-commerce networks to find the trends in the centralities or importance rankings of nodes across different snapshots.

Figure 1: Example phone call graph.
Example 2. Perturbation or contingency analysis is a popular analysis done on real-world graphs to study the resilience of the graph to different failure or perturbation scenarios. For example, in network analyses in neuro-science, scientists “lesion” anatomical or functional brain networks by deleting nodes or edges randomly or in a targeted way [10], e.g., by deleting subsets of highest degree nodes, and study the effects of these lesions on the average path lengths between different nodes in these graphs. Similarly, in a recent user survey we co-authored in reference [28], we reported an application from power grids, which are modeled as graphs. The application periodically takes a static snapshot of the grid and constructs thousands views of this graph, each representing a failure scenario through the removal or updates of sets of nodes or edges. Several computations, such as power or path analysis, are performed to analyze the effects of each scenario. Similar contingency analysis applications have also been described in references from many other fields, such as communication [30], transportation [18], or other biological networks [33].

These applications, and many others, require constructing multiple, sometimes thousands of, views of a static input graph, and compute the same computations across each view. A system that is able to share computation across views would be of immense use to enable efficient development of these applications. We have developed a new open-source system called GraphSurge for this purpose. GraphSurge treats graph views as first-class citizens, and is highly optimized to share computations across views for arbitrary graph analytics computations. GraphSurge is a full-fledged analytics system that has a declarative view definition language called GVDL with which users can define: (1) individual views; or (2) collections of view graphs, which we call view collections. For running analytics computations users use a dataflow-based API. When users execute a computation on view collections, GraphSurge shares computation across the views to improve performance.

GraphSurge is developed on top of the Timely Dataflow [26, 6] system and its Differential Dataflow layer [5, 25], which adopts the differential computation model [7]. Differential computation is a computational model to incrementally maintain arbitrary, possibly iterative, dataflow computations, across evolving data sets. As such, prior literature has used differential computation primarily for maintaining streaming (i.e., continuous) computations for evolving datasets, e.g., to maintain relational queries over a changing database [16]. Our approach is based on the observation that although GraphSurge processes static (instead of evolving) graphs, one can organize view collections as edge difference sets over a base graph. This gives GraphSurge the ability to share computations for arbitrary graph analytics computations on view collections. Given a view collection, GraphSurge runs a Timely Dataflow filtering program that outputs ordered edge difference sets, which compactly represent the views. When running the same analytics computation across the views of a view collection, GraphSurge feeds the user program and the computed difference sets to Differential Dataflow, which shares computation across views internally by running the program differentially across the views.

Unlike streaming applications on Differential Dataflow or specialized graph streaming systems, such as GraphBolt [24], the static nature of the views defined in GraphSurge gives the system several interesting optimization opportunities. The study of these optimization problems constitutes the second major contribution of this paper:

Collection Ordering Problem: Intuitively, once a view collection is ordered as consecutive edge difference sets, making the neighboring views more similar allows differential computation to share more computation across views. In streaming or continuous query processing systems, a system has no choice over the order of the updates that come in, so effectively no choice as to the order of the snapshots on which a computation has to be performed. Instead, the static nature of the views processed by GraphSurge gives GraphSurge an opportunity to order the views as a preprocessing step and put similar views close to each other. We show that this problem is NP-hard through an interesting connection to the consecutive block minimization problem for binary matrices [17]. We then show a constant-factor and efficient approximation algorithm from literature for this optimization problem, which we have integrated into GraphSurge. In our evaluations, we show that our collection ordering optimization can lead to significant runtime improvements when good orderings are unclear.

Collection Splitting Problem: Even after a system has found a good ordering that minimizes differences between views and maximizes computation sharing, there are cases when differentially maintaining the computation for a view $GV_j$, given the differential computations for $GV_0,...,GV_{j-1}$, might be slower than re-running $GV_j$ from scratch. We call this the collection splitting problem, as rerunning the computation from scratch at $GV_j$ effectively splits the view collection into 2 sub-collections, each of which would be run differentially (in absence of further splittings). There are several factors that can trigger this behavior, such as the analytics computation that is executed may be very unstable, or the views may not be similar enough to benefit from differential computation sharing. Although differential dataflow maintains black-box computations, we show that a system can still monitor the runtimes of each view and the sizes of the edge differences, and make effective decision to decide at run-time whether to split a collection at each view or run it differentially. We show that our collection splitting optimization can detect cases when running all views differentially or from scratch is optimal, and lead to up to significant performance improvements over these baselines when neither baseline is optimal.

Finally, we have designed and implemented GraphSurge to be a complete and general view-based analytics system. In addition to filtered views and view collections, which is the focus of this paper, GVVL also supports defining aggregate-views, which are views that have been introduced in Graph OLAP literature [12, 35]. Aggregate views group nodes and edges into super-nodes and super-edges on a set of properties to provide higher-level summaries of an input graph. GraphSurge computes aggregate views using Timely Dataflow.

2. BACKGROUND

Property Graph Model: GraphSurge uses the property graph model, where data consists of a set of nodes and directed edges and arbitrary key-value properties on nodes and
edges. Our current implementation supports string, integer, and boolean properties.

Timely Dataflow (TD) [6, 26]: TD is a system for general, possibly cyclic, i.e., iterative, data-parallel computations that are expressed as a combination of timely operators, such as map, reduce, and iterate, that transform one or more input data streams to an output stream. TD is an inherently streaming system but supports bulk synchronous computations by giving programs the ability to synchronize operators at different timestamps, which are vectors of integers, \(<i_1, i_2, \ldots, i_k>\), where each \(i_j\) can represent different nested iterations of the computation or versions of input data streams (an important feature for differential computations). Similar to systems such as MapReduce and Spark, TD automatically scales computations to multiple workers, within or across compute nodes, where each worker processes only a partition of the data streams in the dataflow. GraphSurge uses TD directly to create individual views, view collections, and aggregate views and indirectly by using Differential Dataflow to run analytics computations.

Differential Dataflow (DD) [25, 5]: DD is a system built on top of TD for incrementally maintaining the outputs of arbitrary dataflow computations over evolving inputs. DD is based on the differential computation model [7], which we review through an example. Consider the Bellman-Ford [14] algorithm for computing shortest paths from a source \(s\) to all other vertices in a graph \(G\). Let \(c(u, v)\) be the cost of an edge in \(G\). Initially \(s\) has a distance of 0 and every other vertex has a distance of \(\infty\). Iteratively, until a fixed point, each vertex \(w\) whose distance has changed produces for each of its outgoing neighbor \(z\) a possible distance "message" \(d(z) + c(w, z)\). Vertices update their distances by taking the minimum of their latest distance and these messages. Figure 2 shows a dataflow implementation of this computation consisting of two original inputs, Edges (E) and Distances (D), and two operators: (i) a JoinMsg operator taking as input edge tuples \((u, v, c(u, v))\) and latest vertex distances and outputting the messages \(M\); (ii) a UnionMin operator taking latest distances and messages for each \(v\) and producing (possibly new) distances.

### Table 1: Differences of Bellman-Ford (BF) example.

| Time/Graph Updates | \(G_0\) | \(G_1\) | \(G_2\) |
|-------------------|---------|---------|---------|
| \(\delta E\)  | \(+ (s, w_1, 2)\), \(+ (s, w_2, 10)\), \(+ (w_1, w_2, 2), dZ_{D}\) | \(- (s, w_1, 2)\), \(+ (s, w_2, 1)\), \(+ (w_1, w_2, 1), dZ_{D}\) |
| \(\delta D\)  | \(+ (w_1, \infty), dZ_{D}\) | \(+ (w_1, \infty), dZ_{D}\) | \(+ (w_1, \infty), dZ_{D}\) |
| \(\delta M\)  | \(+ (w_1, 2), dZ_{M}\) | \(+ (w_2, 1), dZ_{M}\) | \(+ (w_2, 10), dZ_{M}\) |

Given a dataflow computation, DD stores the state of the input and output data streams of each operator as partially ordered timestamped differences and maintains these differences as the original inputs to the dataflow, e.g., stream \(E\) in our example, change. In the above computation, the timestamps are two dimensional \(<\text{graph-version}, BF\text{ iteration}>\) tuples because the streams, specifically \(D\), can change for two separate reasons: (1) changes to \(E\); and (2) changes between iterations of the Bellman-Ford computation.

For a stream \(S\), let \(S_t\) represent the state of \(S\) at timestamp \(t\) and let \(\delta S_t\) be the difference to \(S_t\) at \(t\) (defined momentarily). Consider an operator with a single input stream \(A\) and output stream \(B\). DD only keeps track of the differences \(\delta A_t\) and \(\delta B_t\) ensuring \(A_t\) and \(B_t\) can be constructed for each \(t\) by summing their differences \(\text{prior}\) to \(t\) according to the partial order of the timestamps, i.e., \(A_t = \bigcup_{s \leq t} \delta A_s\) and \(B_t = \bigcup_{s \leq t} \delta B_s\). These equations imply that \(\delta A_t = A_t - \bigcup_{s \leq t} \delta A_s\) and \(\delta B_t = B_t - \bigcup_{s \leq t} \delta B_s\), which is how DD computes and stores \(\delta A_t\) and \(\delta B_t\). Streams in DD are multisets of tuples and the tuples in \(\delta S_t\) can have negative multiplicities, implying deletions of tuples. Figure 3 shows the example of differences to the \(E\), \(D\), and \(M\) streams in the Bellman-Ford dataflow as the graph in Figure 3 is updated first by changing \((s, w_1)\)'s cost from 2 to 1 and then \((s, w_2)\)'s cost from 10 to 1. We assume, for purpose of demonstration that the graph contains billions of edges among the \(z_{ik}\) vertices and we denote the difference sets relevant to them in Figure 1 by \(\delta E, \delta D, \text{ and } \delta M\). Readers can verify in that Figure 1, \(S_t = \bigcup_{s \leq t} \delta S_s\), for every stream and \(t\) for the \(x_i\) component of the graph.

We end this section with two important notes. First, DD is not only designed for maintaining continuous computations, such as maintaining the shortest paths from a source \(s\) in an evolving graph. It is also highly efficient in performing (arbitrarily-nested) iterative computations on static datasets that run till some fixed point, as in many graph computations. For example, even if the graph was not changing, implementing Bellman-Ford algorithm in dataflow
Graphsurge is a system for performing analytics on views over static input graphs. The system is implemented in Rust. Figure 4 shows the architecture of Graphsurge. Users program Graphsurge through two interfaces: (1) A declarative graph view definition language (GVDL) to define individual filtered views and view collections over base graphs (or other views in the system) and aggregate views. (2) A DD-based API to write dataflow programs for graph analytics computations that consume the edge stream of a graph view. Graphsurge uses TD and DD as its execution layer for both creating and manipulating views as well as for running user-specified analytics programs on views. As such, it can be parallelized both in a single multi-core machine as well as in a distributed cluster.

Users import base input graphs to Graphsurge through csv files that contain the nodes and edges of the graphs and their properties. Upon loading, each node and edge is given a unique 64-bit ID. Edges of base graphs are stored as an edge stream by the Storage Manager and persisted in files. Each edge in an edge stream is a tuple of (srcID, sPtr, dID, dPtr, key1, val1, ...), where srcID and dID are pointers to source and destination nodes' properties, which are stored as a stream and accessed through Node Property Store. When running in a distributed cluster, edge streams are partitioned according to srcID or dID, but for simplicity, Node Property Store is replicated across machines.

We next describe how users program Graphsurge and the different components of the system that execute TD and DD dataflows in response to user’s programs.

### 3. Individual Filtered Views

#### 3.1 Individual Filtered View Definition

GVDL is a simple SQL-like language to define views over base graphs. A GVDL query to create filtered views has a single WHERE clause that specifies a predicate on an input graph (or another materialized view) that specifies the edges of the output view. Predicates can be arbitrary conjunctions or disjunctions and access the properties of both source and destination nodes as well as the edges. That is why we do not need a separate WHERE clause for the nodes for filtered views. This will be needed for aggregate views in Section 6.

**Example 3.** Listing 1 shows a filtered view an analyst can construct on our running example Calls graph. The view constructs the graph of calls made in California in 2019 with longer than 10 minute durations.

GVDL queries that define individual views are compiled into TD dataflow programs in a straightforward fashion. The dataflow consists of an operator implementing a standard binary join to join node IDs with the vertex property stream from Node Property Store and a filter operator to apply the user-specified predicates. The output of the program is materialized as a stream in the View Store.

#### 3.1.2 Analytics Computations on Individual Views

Users write arbitrary DD dataflow programs for performing analytics on their views with the constraint that one of the inputs to the dataflow is the Graphsurge-specific edge stream for the view. Graphsurge simply exposes a Rust interface to users with a graph_analytics function, inside of which users can write arbitrary DD programs that are expected to return per-vertex user-defined outputs, such as the connected component ID of each vertex in a connected components analytics. Listing 2 shows the interface of the graph_analytics function. Users invoke their programs through a separate command line and specify their graph_analytics function and the view on which to run this function. Graphsurge’s Computation Executor calls user’s graph_analytics function to obtain user’s dataflow, and feeds the edge stream corresponding to the view into it. When the computation is executed on a single view, then the entire edge stream is fed into the dataflow at once. How the computation is executed on a view collection is more involved and described in Section 3.2.

DD is a library built over TD, so ultimately DD dataflow programs compile to TD operators, so we could allow users to write programs directly as TD dataflows. We expose only...
create view collection call-analysis on Calls
[DV-Y2010: duration<1 and year<2010],
[D2-Y2010: duration<2 and year<2010],
[D3-Y2010: duration<3 and year<2010],
...[D34-Y2010: duration<34 and year<2010]

Listing 3: Example GVDL view collection query.

| GV1 | GV2 | GV3 | GV1 | GV2 | GV3 |
|-----|-----|-----|-----|-----|-----|
| e0  | 1   | 0   | 0   | e0  | +1  | -1  | 0   |
| e1  | 1   | 0   | 1   | e1  | +1  | -1  | +1  |
| e2  | 0   | 0   | 1   | e2  | 0   | 0   | +1  |
| e3  | 0   | 1   | 1   | e3  | 0   | +1  | 0   |
| e4  | 1   | 1   | 1   | e4  | +1  | 0   | 0   |

(a) Edge boolean matrix.  (b) Difference stream.

Figure 5: An example edge boolean matrix and its corresponding edge difference stream.

The goal of this step is to put views that overlap more edges next to each other, so that the differences between neighboring views is smaller, so that when running an analytics computation on the collection, more computation is shared. To achieve this, GraphSurge re-orders the views in EBM so that views whose predicates satisfy highly overlapping sets of edges are adjacent to each other. As we demonstrate, this optimization step can lead to significant performance benefits. The output of this step is the same EBM but possibly with a different column ordering. We defer a detailed explanation of how collections are ordered to Section 4.

Step 2. Collection Ordering: The goal of this step is to put views that overlap more edges next to each other, so that the differences between neighboring views is smaller, so that when running an analytics computation on the collection, more computation is shared. To achieve this, GraphSurge re-orders the views in EBM so that views whose predicates satisfy highly overlapping sets of edges are adjacent to each other. As we demonstrate, this optimization step can lead to significant performance benefits. The output of this step is the same EBM but possibly with a different column ordering. We defer a detailed explanation of how collections are ordered to Section 4.

Step 3. Edge Difference Stream Computation: Finally, GraphSurge takes the reordered EBMs and materializes the views in the view collection as an edge difference stream that is consistent with the semantics of difference sets of differential computation. Specifically, we treat the entire view collection C as an evolving input stream according to the order output in step 2. For simplicity, let \( GV_1, ..., GV_k \) be the order of the views after step 2, so \( C_t = GV_i \). Recall from Section 2 that according to differential computation semantics, the difference of a stream \( A \) at timestamp \( t \) is \( \delta A_t = A_t - \cup_{e < t} \delta A_e \). So the edge difference of a view \( t \), \( \delta C_t \), is computed to ensure that \( \delta C_t = GV_t - \cup_{s < t} \delta C_s \) equality holds. Specifically, the multiplicity of each edge \( e_i \) in \( \delta C_t \) is: (i) 0 if \( GV_t - 1 \) and \( GV_t \) both contain or both do not contain \( e_i \); (ii) 1 if \( GV_{t-1} \) does not contain \( e_i \) and \( GV_t \) does; or (iii) -1 if \( GV_{t-1} \) contains \( e_i \) and \( GV_t \) does not. Figure 5b shows the resulting difference set for the edge boolean matrix in Figure 5a. The contribution of each edge \( e_i \) to \( \delta C_t \) can be computed independently, so this is another embarrassingly parallelizable step.

3.2 View Collections

3.2.1 View Collection Definition

To share analytics computations across multiple filtered views of the same graph, GraphSurge allows users to organize filtered views in a view collection. A view collection organizes a set of filtered views as a single timestamped2 difference edge stream \( C \), where each view corresponds to a state of the stream at a particular timestamp \( t \).

Example 4. Listing 3 shows an example GVDL query defining a view collection. Each line is an edge predicate describing one filtered view on our Calls graph, including all calls made in 2010 with durations ranging from 1 to the 34 minutes, which is the maximum duration, in the graph.

GraphSurge materializes the view collection described by a GVDL query in three steps. Below, we let \( p_{ij} \) denote the predicate defining \( GV_i \) in a given view collection.

Step 1. Edge Boolean Matrix Computation: For each edge \( e_i \) in the base graph and each view \( GV_j \) in the collection, GraphSurge runs the predicate \( p_{ij} \) on \( e \) and outputs an edge boolean matrix (EBM) that specifies whether \( e_i \) satisfies \( p_{ij} \). Figure 5a shows an example EBM. This is an embarrassingly parallelizable computation and is performed by a TD dataflow.

Step 2. Collection Ordering: The goal of this step is to put views that overlap more edges next to each other, so that the differences between neighboring views is smaller, so that when running an analytics computation on the collection, more computation is shared. To achieve this, GraphSurge re-orders the views in EBM so that views whose predicates satisfy highly overlapping sets of edges are adjacent to each other. As we demonstrate, this optimization step can lead to significant performance benefits. The output of this step is the same EBM but possibly with a different column ordering. We defer a detailed explanation of how collections are ordered to Section 4.

Step 3. Edge Difference Stream Computation: Finally, GraphSurge takes the reordered EBMs and materializes the views in the view collection as an edge difference stream that is consistent with the semantics of difference sets of differential computation. Specifically, we treat the entire view collection \( C \) as an evolving input stream according to the order output in step 2. For simplicity, let \( GV_1, ..., GV_k \) be the order of the views after step 2, so \( C_t = GV_i \). Recall from Section 2 that according to differential computation semantics, the difference of a stream \( A \) at timestamp \( t \) is \( \delta A_t = A_t - \cup_{e < t} \delta A_e \). So the edge difference of a view \( t \), \( \delta C_t \), is computed to ensure that \( \delta C_t = GV_t - \cup_{s < t} \delta C_s \) equality holds. Specifically, the multiplicity of each edge \( e_i \) in \( \delta C_t \) is: (i) 0 if \( GV_t - 1 \) and \( GV_t \) both contain or both do not contain \( e_i \); (ii) 1 if \( GV_{t-1} \) does not contain \( e_i \) and \( GV_t \) does; or (iii) -1 if \( GV_{t-1} \) contains \( e_i \) and \( GV_t \) does not. Figure 5b shows the resulting difference set for the edge boolean matrix in Figure 5a. The contribution of each edge \( e_i \) to \( \delta C_t \) can be computed independently, so this is another embarrassingly parallelizable step.

3.2.2 Analytics on View Collections

Given an analytics program \( P \) that a user wants to run on all views of a view collection \( C \), in absence of any collection splitting, which is an optimization we describe in Section 5, GraphSurge’s Analytics Computation Executor runs \( P \) as follows. First, the system runs \( P \) on \( C_0 \), i.e., the “first” view in \( C \), and then when this computation finishes, in an outside loop advances (in DD terminology) \( C \) to \( C_1 \) by feeding \( \delta C_t \) to DD. Then the system feeds \( \delta C_t \) to DD, so on and so forth, until all views are evaluated. When computing \( P \) at each time \( t \), DD will internally share computation from the “prior” views on which \( P \) has been computed, in some cases leading to significant performance gains compared to running \( P \) on each view from scratch. The output of the DD program is a set of output difference sets for the output \( (VID, ResultValue) \) stream specified in the user-specified graph-analytics function. The output difference stream can then be stored or processed by the user.

4. COLLECTION ORDERING

Given a set of \( k \) views in a view collection \( C \) defined by an application, there are \( k! \) different ways GraphSurge can order the views before running analytics computations differentially on the collection. This is important because the number of edge differences that are generated in the final collection is solely determined by the order of the views.
Recall from Section 3.2.2 that when running analytics computations on a view collection $C$, Graphsurge iterates over neighboring views on and view $t$ feeds in the difference set $\delta C_t$ to DD (in absence of collection splitting). The smaller the size of the differences, the larger the structural overlap between view $C_t$ and the union of the views prior to $C_t$, which we expect to lead to larger computation sharing. As we present in our evaluations, by picking orderings that minimize the set of differences, Graphsurge can improve performance significantly in certain applications.

Recall our call-analysis view collection from Listing 3, which contained 34 views of calls made in 2010 with durations at most $i$ for $i \in [1, 34]$ with names $D_{1}$, ..., $D_{34}$. Sometimes, as in this application, whose predicates have an inclusion relationship, finding a good order may be easy for a user, and the system can use the user-given order, e.g., $D_{1}$, ..., $D_{34}$. In many applications, e.g., many contingency analysis applications, such an order is not obvious. In such cases, a system can perform a pre-processing step to find a good ordering that leads to small number of differences. We can formulate this problem as a concrete optimization problem as follows:

**Definition 1. Collection Ordering Problem (COP):** Given a view collection $C$, find the collection ordering that minimizes the sum of the sizes of difference sets $\delta C_t$.

We next show that COP is NP-hard through a reduction from the consecutive block minimization problem (CBMP) for boolean matrices. In a boolean matrix $B$, such as the edge boolean matrix (EBM) in Figure 5a, a consecutive block is a maximal consecutive run of 1-cells in a single row of $B$, which is bounded on the left by either the beginning of the row or a 0-cell. Given an arbitrary column ordering $\sigma$, we analyze the number of differences each row in $B$ induces in $EBM$.

- Row $r$ in $B_{0}$ yields 0 but $r^C$ yields 1 difference.
- Row $r$ in $B_{1}$ yields 1 difference but $r^C$ yields 0 difference.
- Row $r$ in $B_{0}$ requires analyzing two cases. Let $cb(r, \sigma)$ denote the number of consecutive blocks only in row $r$. (i) If $r$‘s last cell is a 0, then $r$ yields $2cb(r, \sigma)$ and $r^C$ yields $2cb(r, \sigma) + 1$ differences; and (ii) otherwise $r$ yields $2cb(r, \sigma) - 1$ and $r^C$ yields $2cb(r, \sigma)$ differences. Therefore in either case, $r$ yields $4cb(r, \sigma) - 1$ differences.

Therefore $ds(B_{EBM}, \sigma) = \sum_{j \in B_{0}} 4cb(r, \sigma) - 1 + m_{0} + m_{1}$, which is equal to $4cb(B, \sigma) - m_{01} + m_{0} + m_{1}$. This establishes a one-to-one connection between the sizes of the difference sets in $B_{EBM}$ and the number of consecutive blocks in $B$ under any ordering $\sigma$. Since for any $B$, $m_{0}$, $m_{1}$, and $m_{01}$ are fixed, finding the optimal ordering $\sigma^*$ that minimizes $ds(B_{EBM})$ also minimizes $cb(B)$, completing the proof that COP is NP-hard.

The connection of COP to CBMP directly instructs how to order view collections. There are known, and highly practical, constant-time approximation algorithms for CBMP, which can be integrated into a system. We integrated the algorithm from reference [17] into Graphsurge. The algorithm, which we refer to as CBMP$_{1,5}$ takes as input an $m \times k$ boolean matrix $B$. In our setting $m$ is the number of edges and $k$ is the number of views in the view collection. CBMP$_{1,5}$ then reduces $B$ into an $m \times (k + 1)$ matrix $0B$ by padding a 0 column, and then transforming $0B$ into a $((k + 1)^2$ clique $G^{0B}$, where each column (so each view in our case) is a node, and the weight between the nodes is the Hamming distance of the columns they represent. Reference [17] shows that $G^{0B}$ satisfies the triangle inequality and the entire transformation from $B$ to $G^{0B}$ is approximation preserving. Therefore, solving TSP, with the well known Christofides algorithm [13] yields a 1.5-approximation to CBMP. This also leads to 3-approximation for COP. To see this, consider any input $C_{EBM}$ to COP. We can find a triangle inequality and the entire transformation from $B$ to $G^{0B}$ is approximation preserving. Therefore, solving TSP, with the well known Christofides algorithm [13] yields a 1.5-approximation to CBMP. This also leads to 3-approximation for COP. To see this, consider any input $C_{EBM}$ to COP and any ordering $\sigma$ for $C_{EBM}$. Because in each row $r$, there are either $2cb(r, \sigma) - 1$ or $2cb(r, \sigma)$ differences, $cb(C_{EBM}, \sigma) \leq ds(C_{EBM}, \sigma) \leq 2cb(r, \sigma)$. Therefore a 1.5-approximation algorithm for CBMP guarantees a 3-approximation for COP.

Algorithm 1 shows our full collection ordering optimizer. Given an EBM $C_{EBM}$, we construct $G^{0C_{EBM}}$ using a TD program that performs the padding and then in an embarrassingly parallel way find the Hamming distances between each view. Then we collect the $G^{0C_{EBM}}$ in a single worker and run Christofides’s algorithm in a single TD worker. Recall that the size of $G^{0C_{EBM}}$ is $(k + 1)^2$ but this quadratic complexity is on the number of views $k$, which is a query-dependent parameter. So we expect it to be small in practice, e.g., the largest view collection we use in our evaluations contain a few hundred views.

**5. COLLECTION SPLITTING**

Even after we find a good ordering that minimizes the sizes of the difference sets generated, running each view differentially may not be the most performant choice. Intuitively, differential computation keeps track of the computation footprint of a given analytics computation $A$ on the latest view $GV$. Specifically, it keeps track of the outputs of
which we randomly add 500 edges and remove 500 edges to construct an initial view. We take 10M edges from the Orkut social network graph and illustrate this on a more realistic application in Section 7. We will also demonstrate these factors through a controlled experiment. We will also demonstrate how large are the difference sets? We next demonstrate these factors on two consecutive views on the Orkut social network graph.

Each iteration of the computation footprint of $A$ on $GV_{i+1}$ is different from $A$ on $GV_i$ and $GV_{i+1}$ are very different, specifically in $C_{1,K}$, running PageRank differentially also starts to be the better option. That is the size of the differences also determines whether running views differentially vs from scratch is the better option. Naturally, there can be cases, when an adversarial minor update can lead to large changes in the computational footprint of $A$ between $GV_i$ and $GV_{i+1}$ even if $A$ is stable but we cannot predict such updates, at least for a black box computation $A$, without actually first fixing the computation.

We have implemented an adaptive optimizer that decides whether to run each view $GV_i$ in a view collection differentially or from scratch. We call the operation of running a view $GV_i$ from scratch as splitting because this is effectively equivalent to running $GV_{i+1}$, $GV_{i+2}$, ..., $GV_k$ as another view collection differentially (in absence of further splittings). Our optimizer observes two simple runtime metrics to make its splitting decisions: (1) Each time the system decides to split the collection at $GV_i$ and run $GV_i$ from scratch, we measure how long it took to compute $A$ on $GV_i$ from scratch and what was the size of $C_i$. (2) Each time the system decides to run a $GV_i$ differentially we keep track of how long it took to run $GV_i$ differentially and what was the size of $\delta C_i$. Then, for each $GV_i$, we use two simple linear models to estimate how long it would take to rerun $GV_i$ from scratch and differentially given, respectively, the sizes of $GV_i$ and $\delta C_i$, and pick the faster estimated option. Specifically:

1. Run $GV_i$ from scratch and $GV_i$ differentially and keep track of $(|GV_i|, st_i)$, for scratch time, and $(\delta C_2, dt_i)$, for differential time.

2. Then for each other view $GV_i$ for $i = 3, ... k$, estimate the run time of running $GV_i$ from scratch or differentially using the collected $st_j$ times and the size of $|GV_i|$ and the $dt_j$ points and the size of $\delta C_i$.

In our actual implementation, instead of deciding whether to split the collection for each view, we make a decision for a set of $\ell$ views at a time (10 by default). This is because if we decide to run all of the next $\ell$ views differentially, feeding them to DD makes DD’s data indexing code run faster, which has performance benefits. We will demonstrate that

---

**Algorithm 1: Collection Ordering Optimizer**

**Input:** Edge Boolean Matrix $B_{m \times k}$, W workers

**Output:** A column ordering $\sigma$

**Variables:**
- $W$: Number of workers
- $B_i$: Edge Boolean Matrix
- $U$: Unit matrix
- $D_i$: Graph surger observed by $i$
- $\sigma^*$: tsp_christofides($G$)
- $\delta C_i$: Adversarial minor update

**Algorithm: Partition $B_{m \times k}$ into $W$ files $B_i$**

**Begin**

- At each worker $w_i$, $0 \leq i < W$:
  - $C_i \leftarrow \{0\} B_i$
  - $U \leftarrow \text{unit matrix}$
  - $D_i = C_i^T (U - C_i) + (U - C_i)^T C_i$
  - Shuffle $D_i$ to worker $w_0$

**Shuffle**

- Broadcast $\sigma^*$ to all workers $w_i$

**Receive**

- $D_i$ from all workers $w_i$

**Broadcast**

- $G \leftarrow \text{complete graph with } |V| = (k + 1)$ and $|E| = (k + 1)^2$ induced from adj. matrix $D$

**Table 2: Runtimes of Bellman-Ford’s algorithm and PageRank on two view collections in two ways: (i) diff-only; and (ii) scratch. View collections contain 1K- and 3.5M-size difference sets.**

| Difference Sets | Algorithm | 1K    | PR    | 3.5M  | PR    |
|----------------|-----------|-------|-------|-------|-------|
|                | BF        | 1.4s  | 13.5s | 13.0s | 25.7s |
|                | PR        | 136.2s| 281.9s| 193.2s| 193.2s|

---

We have implemented an adaptive optimizer that decides whether to run all of the next $\ell$ views differentially or from scratch. We call the operation of running a set of $\ell$ views differentially and what was the size of $GV_{\ell-1}$. The sizes of the difference sets are picked to obtain a collection with highly similar and highly different views, respectively. We then run Bellman Ford's algorithm and PageRank on both collections in two ways: (i) **diff-only**: runs the collection only differentially; and (ii) **scratch**: runs each view in the collection from scratch.

Table 2 shows the runtimes. First notice that on $C_{3,5}$, while it is better to run Bellman Ford differentially, it is better to run PageRank from scratch. This is because PageRank is a less stable algorithm than Bellman-Ford. For example, assume $GV_{\ell+1} = GV_i \cup \{u \rightarrow v\}$, so the views differ by a single edge addition, and consider differentially fixing the first iteration of Bellman Ford. At a high-level this additional result in 1 difference in the JoinMsg operator. In vertex-centric terms, the addition will result in $u$ sending 1 more extra message to $v$ containing $u$’s current distance (assuming $u$ is the source). The rest of $u$'s messages in the first iteration of $G_i$ are not affected. In contrast, in PageRank, $u$ sends a message of $1/\text{deg}(u)$ to its neighbors so all of the messages that $u$ sends might change. Second observe that when the views are sufficiently similar, specifically in $C_{1,K}$, running PageRank differentially also starts to be the better option.
our optimizer is both able to adapt to running computations differentially or from scratch, when either option is superior, and can even outperform both options in some cases by selectively splitting collections in a subset of the views.

We next discuss an important question: How much faster can running an algorithm \( A \) differentially on a view collection \( C \) be compared to running \( A \) on each view from scratch (and vice versa)? A high-level answer to this question should instruct the benefits we can expect from adaptively splitting. Consider a \( k \)-view collection \( C \), where each view is identical. This is conceptually the best case for running \( A \) on \( C \) differentially, where after the first view, the rest of the views are computed instantaneously. Therefore differentially computing \( A \) can be \textit{k factor better} than running \( A \) from scratch. Importantly as the number of views increase, we can expect to see an increasing amount of benefits. Interestingly, the situation is not similar in the reverse direction. The worst case for running \( A \) on \( C \) differentially is if each view was completely disjoint, i.e., \( \delta C_i = \{ - GV_{i-1} \cup GV_i \} \). So we effectively completely remove \( GV_{i-1} \) and add \( GV_i \). Therefore when running \( A \) on \( GV_i \) differentially, \( DD \), to the first approximation, will “undo” computation for \( GV_{i-1} \) and then run \( A \) from scratch differentially. So we effectively compute \( A \) on each view twice, assuming undoing the computation takes similar amount of effort as doing it. Instead, running \( A \) on each view from scratch computes \( A \) on each view once. So we should expect a bounded, around \( 2x \), slow down to running computations differentially even in this worst case. This is an important robustness property of performing computations differentially. However, it is still important to perform our splitting optimization because: (i) we have observed up to \( 4x \) improvements of running computations from scratch (we will present up to \( 2.7x \) improvements in our experiments), so there is still a significant performance gain to obtain over pure differential computation; and (ii) some unstable computations consistently perform better when computed from scratch and our splitting optimization automatically detects those cases.

6. AGGREGATE VIEWS

We are implementing \textsc{Graphsurge} to be a full-fledged view-based graph analytics system. To allow a wider range of applications to be developed on the system, we support \textit{aggregate views}, which are views introduced in the Graph OLAP literature \cite{12, 35}. An aggregate view over an input graph \( G \) describes a particular high-level connectivity summary of \( G \). Specifically, aggregate views group sets of nodes in \( G \) into super-nodes and aggregate the original edges between nodes into super-edges, with the semantics that each edge \((u, v)\) in the original graph becomes part of an aggregation of a super-edge between the super-nodes that \( u \) and \( v \) belong to. GVDL contains group by constructs on nodes to define super-nodes, and aggregate constructs on nodes and edges to define aggregate properties on super-nodes and super-edges. The resulting query is evaluated in TD using a dataflow that consists of aggregation operators.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
create view NY-Dr-CA-Lawyer on Calls \\
\hspace{1cm} nodes group by [ \\
\hspace{2cm} (profession='Doctor' and city='NY'), \\
\hspace{2cm} (profession='Lawyer' and city='LA'), \\
\hspace{2cm} (profession='Teacher' and city='DC') ]
\hspace{1cm} aggregate count(*) \\
create view City-Calls-City on Calls \\
\hspace{1cm} nodes group by city aggregate num-phones: count(*) \\
\hspace{1cm} edges aggregate total-duration: sum(duration)
\end{tabular}
\caption{Example GVDL aggregate view queries.}
\end{table}

\textbf{Example 5.} Listing 4 shows two aggregate views. The first is an aggregate view showing the total calls made between doctors in NY, lawyers in LA, and teachers in DC (so a triangle). The second is the city-calls-city view that captures the connectivity of calls between cities, where city super-nodes have a \textit{num-phones} property, for the number of phone calls registered in the city, and super-edges have a \textit{total-duration} property for the total duration of calls made between cities.

Supporting aggregate views allows users to start analytics on very large graphs, construct smaller graphs by creating aggregate views, and perform further graph analytics on these smaller graphs, all in the same system. We plan to extend the system with new features as we encounter other view-based graph analytics applications.

7. EVALUATION

We next present our experiments. In Section 7.2, we start by empirically demonstrating the possible performance gains of running computations differentially across views vs running them from scratch. In Section 7.3, we present the benefits of our collection splitting optimization. Section 7.4 evaluates the benefits of our collection ordering optimization. In Section 7.5, we review several baseline comparisons between \( DD \) and the GraphBolt \cite{24} and Tegra \cite{19} systems that were previously reported in these references. For completeness of our work, Section 7.6 presents scalability experiments, showing that we can scale across compute nodes in clusters, with good scalability.

7.1 Experimental Setup

\textbf{Datasets:} We evaluate \textsc{Graphsurge} on 5 different real-world graphs, that we describe below:

- \textbf{Stack Overflow} \cite{2} (SO, \(|V| = 2M, |E| = 63M\)) is a temporal dataset where every edge has an associated unix timestamp indicating its creation time.
- \textbf{Paper Citations} (PC, \(|V| = 172M, |E| = 605M\)) is a paper-to-paper citation graph constructed from the Semantic Scholar Corpus \cite{9} (version 2019-10-01). The vertices have 2 associated properties: the year of publication and the count of co-authors.
- \textbf{Com-Livejournal} \cite{1} (LJ, \(|V| = 3M, |E| = 34M\)) is a social network graph containing a list of ground-truth communities representing social groups that a subset of the users are part of. Users can be part of multiple communities.
- \textbf{Com-Wiki-Topcats} \cite{3} (WTC, \(|V| = 1M, |E| = 28M\)) is a web graph whose vertices can belong to one or more communities representing the category of a web page.
- \textbf{Twitter} \cite{22} (TW, \(|V| = 42M, |E| = 1.5B\)) is a large social network graph.

\textbf{Computations:} We use 5 different graph analytical computations to evaluate \textsc{Graphsurge}: (i) weakly connected components (WCC); (ii) strongly connected components (SCC), which implements the doubly-iterative Coloring algorithm \cite{27}; (iii) breadth-first search (BFS); (iv) PageRank (PR); and (v) multiple pair shortest path (MPSP). For
We expect differential-dataflow two ways: diff-only selection, turning our splitting and ordering optimizers off, in We evaluate the performance of our algorithms on each collection, the reverse comparison is bounded. We

Recall also our observation from Section 5 that while differentially computing $A$ can be unboundedly faster than running from scratch, the reverse comparison is bounded. We start by demonstrating this intuition empirically. We model

(i) $C_{\text{sim}}$: are a set of similar view collections that each start with a 5-year window of the graph, from May 2008 to May 2013, which forms the first view. Then we set a time window of size $w$ of 1 day, 1 month, 6 mosths, 1 year, and 2 years, and expand the initial window by $w$, so each view $GV_i$ includes $GV_{i-1}$ plus an additional number of edges for a larger $w$-size window. This generates 5 collections. $C_{\text{sim},1d}$, where $w$ is 1 day, contains the most similar and largest number of views. $C_{\text{sim},2y}$ is the least similar and contains the fewest number of views.

(ii) $C_{\text{no}}$: are a set of non-overlapping, so highly different views, where we start with a window of the graph from May 2008 till December 2008, then we completely slide the window by a window of size $w$ of 6 months, 1, 2, 3, and 4 years. This generates 5 collections, all of which are completely non-overlapping. The window size $w$ allows us to create collections with increasingly more views.

We evaluate the performance of our algorithms on each collection, turning our splitting and ordering optimizers off, in two ways: diff-only and scratch, which were described in Section 5. We expect diff-only to be more performant than scratch in each $C_{\text{sim}}$ collection, but increasingly more as $w$ gets smaller and there is a larger number of views. We expect scratch to be more performant in each $C_{\text{no}}$ collection, but we do not expect to see increasingly more gains as the number of views increases. Figures 6 and 7 show our results for the $C_{\text{sim}}$ and $C_{\text{no}}$ collections, respectively. Observe that in $C_{\text{sim}}$ collections, indeed as $w$ gets smaller, we see an increasing factor on benefits for diff-only varying from $1.5x$ to $13.7x$. The only exception is PageRank, which we observed is not as stable as the rest of our algorithms. In contrast, in the $C_{\text{no}}$ collection, we see up to $2.5x$ performance improvements for scratch but we do not observe improved factors with increasing number of views.

7.3 Benefits of Collection Splitting

We next evaluate GraphSurge’s adaptive splitting optimizer continuing our previous set up. We refer to this configuration as adaptive. We still keep our ordering optimizer off to only study the behavior of our adaptive optimizer, which we refer to as adaptive. We reran the previous experiment with adaptive. The adaptive bar in Figures 6 and 7 show our results. In 38 out of 40 of these experiments, adaptive is able to perform as good as almost as good as the better of diff-only or scratch. Note that in these experiments, it is always better to either run the computations with one of diff-only or scratch, so we do not expect adaptive to outperform both of these strategies. Importantly, in almost all cases, we adapt to the better strategy. The only exceptions are on running PageRank on $C_{\text{sim}}$ when $w$=1 and 2 years, where the number of views is 6, and 4, respectively, so adaptive does not get many instances to accurately estimate how differential and running from scratch behaves.

Next we created view collections in which adaptive can outperform both diff-only and scratch. Specifically, we created 3 different view collections on the PC citation dataset, each containing a different mix of edge additions and deletions combinations between their views.

(i) $C_{\text{sh}}$: for slide contains the papers published in the oldest decade in the dataset, which is 1936, 1945], as the first view. Then we slide this 10 year window by 5 year, generating a total of 16 views: [1941, 1950], ..., [2011, 2020]. Each view effectively adds and removes 5 years of new papers to its predecessor.

(ii) $C_{\text{ex-sh-slide}}$: for expand-shrink-slide contains views of papers published in varying number of year windows. First is the 1995, 2000] window, which expands through only additions to [1995, 2005], then shrinks to [2000, 2005], and then slides to [2005, 2010] all by one year windows.

(iii) $C_{\text{aut}}$: for authors contains the Cartesian product of two sets of windows on two properties. First is a 5 year non-overlapping window from [1996, 2000] to [2016, 2020]. The other is a window for the number of authors on the papers, that expands from [0, 5] to [0, 25] in windows of size 5. For example, the view [1996, 2000]x[0, 5] is the view that contains all papers written between 1996 and 2000 containing at most 5 authors and their citations. This collection contains views that generates a sequence of addition-only differences as the number of authors window expands, and then a non-overlapping view, when the year window slides, creating a potential splitting point.

Table 3 shows the runtimes of WCC, BFS, SCC, and PR on our view collections. Observe that across all of these 12 experiments, adaptive almost matches or outperforms, by up to $1.8x$, the better of diff-only and scratch. On $C_{\text{aut}}$ adaptive is able to pick the splitting points where the year window slides and consistently outperforms diff-only and scratch when running all algorithms. Finally, we note that, unlike our ordering optimization, which we evaluate in the next section, our adaptive splitting optimizer’s overheads are negligible as it consists of running small amount of arithmetic operations during runtime and no data processing.

7.4 Benefits of Collection Ordering

The goal of our next experiment is to study the performance gains of our collection ordering optimization. In this experiment, we develop a perturbation analysis application
on our graphs with ground truth communities, namely comlivejournal and wiki-topcats. We construct view collections by taking the largest $N$ communities and remove each $k$ combination of these $N$ communities to perturb the graphs in a variety of ways. Specifically we construct two collections for two $N, k$ combinations: $C_{10, 5}$ sets $N=10$ and $k=5$, and $C_{7, 4}$ sets $N=7$ and $k=4$. Note that this is an application where finding a good manual order is difficult, as each view removes possibly millions of edges, and there are hundreds of views in the collection. Therefore as a baseline, we will use random collection orderings.

We first turned our adaptive splitting optimizer off to isolate the benefits due to collection ordering only and compared the performance of the order that Graphsurge picks, which we call Ord., with 3 random orderings, which we call $R1$, $R2$, and $R3$. We executed WCC, BFS, and MPSP for graph algorithms. Figures 8 and 9 shows our results with the no adapt. bar. Table 4 presents the amount of total edge differences in our edge difference sets. Observe that: (i) our optimizer’s order generates between 9.5x to 16.8x fewer differences than the random order; and (ii) our ordering optimization improves performance consistently and between 1.7x to 37.4x across our experiments. Note that we are assuming an environment where the view collections are created once while graph algorithms may be run many times, therefore we are focusing on the effect of ordering on the performance of algorithms. However, for reference, Table 4 also reports the times it takes Graphsurge to compute the collections, with and without ordering in row CCT, for collection creation time. The difference between the random orders’ CCT and $GS_{ns}$ is the overhead of ordering, which ranged from 1.1x to 1.7x over the collection creation times without ordering.

We next turned the adaptive splitting optimization on to measure performance benefits in the full system. We expect benefits of ordering to decrease as adaptive splitting is designed to improve performance when a collection’s ordering does not have highly overlapping consecutive views. Figures 8 and 9 shows our results with the with adapt. bar. First, observe that adaptive splitting improves the performance of all configurations, again demonstrating its robustness. Second, although our improvement factors due to ordering broadly decrease (with the exception of MPSP), we still see improvements ranging from 1.5x to 12.4x (ignoring MPSP where improvements increase to 47.1x).

### 7.5 Baseline Temporal Systems

We next review two systems, GraphBolt [24] and Tegra [19], that have provided baseline comparisons against DD. These are two alternative systems on top of which one can develop Graphsurge. We discuss the pros and cons of these alternative architectures for Graphsurge.
Graphsurge is a graph streaming system that is designed to maintain a computation as a stream of updates arrive. As such, we can develop a Graphsurge-like system on top of GraphBolt by feeding our view collections as an evolving graph to GraphBolt instead of DD. The primary difference between DD and GraphBolt, and the reason we chose DD, is that GraphBolt requires users to write different maintenance code in functions such as `retract` or `propagatedelta` to program how to maintain their analytics computations. Therefore we would give up the generality of DD to share computation for arbitrary analytics computations. This type of programming is particularly not suited for a system-like Graphsurge because Graphsurge is not designed for continuous analytics. Users do not analyze a dynamic graph. Instead, they analyze static views over static graphs. Moreover, writing specialized dynamic versions of some algorithms, such as the doubly-iterative SCC algorithm, are very challenging, and we are unaware of any systems work other than DD that has provided actual implementations.

At the same time, DD’s generality, as expected, comes at performance costs. This was demonstrated in Figures 8 of reference [24], where they showed that by providing PageRank-specific maintenance logic, they can maintain PageRank an order of magnitude faster than DD. Interestingly, they also showed in Figure 9, that on SSSP, DD was an order of magnitude faster, but for implementation-specific reasons explained in reference [24]. We were able to reproduce these relative performances (not absolute runtimes) in these figures in our setting (omitted due to space restrictions). Importantly, readers should expect that writing algorithm-specific maintenance code should in principle be faster than using DD’s black-box maintenance logic.

Tegra [19] is a recent system, developed on top of Apache Spark [34], that is designed to perform ad-hoc window-based analytics on a dynamic graph. Specifically, Tegra allows users to tag arbitrary snapshots of their graphs with timestamps. The system has a technique for sharing arbitrary computation across snapshots through a differential computation-like computation maintenance logic. However, the focus of the system is not on efficient computation sharing but on retrieving arbitrary snapshots that have been tagged in past very quickly. Although, Tegra is another alternative system on top of which a Graphsurge-like system can be developed, reference [19] reports its performance to be significantly slower than DD (Figure 14) for incrementally maintaining computations. The system is closed-sourced and we could not obtain the code to reproduce these results.

### GraphBolt

GraphBolt [24] is a graph streaming system that is developed on top of Ligra [29] and designed to maintain a computation as a stream of updates arrive. As such, we can develop a Graphsurge-like system on top of GraphBolt by feeding our view collections as an evolving graph to GraphBolt instead of DD. The primary difference between DD and GraphBolt, and the reason we chose DD, is that GraphBolt requires users to write different maintenance code in functions such as `retract` or `propagatedelta` to program how to maintain their analytics computations. Therefore we would give up the generality of DD to share computation for arbitrary analytics computations. This type of programming is particularly not suited for a system-like Graphsurge because Graphsurge is not designed for continuous analytics. Users do not analyze a dynamic graph. Instead, they analyze static views over static graphs. Moreover, writing specialized dynamic versions of some algorithms, such as the doubly-iterative SCC algorithm, are very challenging, and we are unaware of any systems work other than DD that has provided actual implementations.

At the same time, DD’s generality, as expected, comes at performance costs. This was demonstrated in Figures 8 of reference [24], where they showed that by providing PageRank-specific maintenance logic, they can maintain PageRank an order of magnitude faster than DD. Interestingly, they also showed in Figure 9, that on SSSP, DD was an order of magnitude faster, but for implementation-specific reasons explained in reference [24]. We were able to reproduce these relative performances (not absolute runtimes) in these figures in our setting (omitted due to space restrictions). Importantly, readers should expect that writing algorithm-specific maintenance code should in principle be faster than using DD’s black-box maintenance logic.

### Tegra

Tegra [19] is a recent system, developed on top of Apache Spark [34], that is designed to perform ad-hoc window-based analytics on a dynamic graph. Specifically, Tegra allows users to tag arbitrary snapshots of their graphs with times-
Temporal and Streaming Graph Analytics Systems:

SAMS [31] is a system to execute a single algorithm on multiple snapshots. However, the system does not have any computation sharing capabilities similar to DD. That is running a WCC algorithm on $k$ views would result in rerunning WCC $k$ times from scratch. Instead, the system is optimized for sharing accesses to the same parts of the graph to increase data locality across these $k$ computations.

Delta-Graph [20] is a system designed for temporal analysis but the system is designed primarily for retrieval of views of a dynamic graph in arbitrary timestamps and not for performing analytics.

Gelly Streaming [4] is a library on top of Flink [11] to program pure streaming computations with a graph API. Users have to implement their own streaming, computation maintenance, and operator synchronization logic as a stream of edges arrive at Flink. Similar to the temporal graph analytics systems, it does not provide any general computation sharing capabilities. Therefore it is not appropriate to use as an execution layer for Graphsurge, as it would require developing a DD-like layer on top of it.

9. CONCLUSIONS AND FUTURE WORK

We presented the design and implementation of Graphsurge, an open-source view-based graph analytics system, developed on top of the TD system and its DD layer. Graphsurge allows users to define arbitrary filtered views over their graphs, organize these views into view collections, and perform arbitrary graph analytics using a DD-based analytics API. Graphsurge is motivated by real-world applications, such as perturbation analysis or analysis of the evolution of large-scale networks, that require capabilities to analyze multiple, sometimes thousands of filtered views of static input graphs efficiently. We presented two optimization problems, the collection ordering and splitting problems, for which we described efficient algorithms and studied the performances of our optimizations. Graphsurge’s approach for computation sharing is based on differential computation. As future work, we are interested in studying modifications one can make to the internals of DD to share computations more efficiently, for example using techniques from incremental versions of specific graph algorithms [8, 24], or using a mix of differential and specialized non-differential operators in analytics programs.
10. REFERENCES

[1] SNAP: Network datasets: LiveJournal social network. https://snap.stanford.edu/data/com-LiveJournal.html.

[2] SNAP: Network datasets: Stack Overflow temporal network. https://snap.stanford.edu/data/stackexchange.txt.lz4.

[3] SNAP: Network datasets: Wiki-topcats network. https://snap.stanford.edu/data/wiki-topcats.html.

[4] Gelly Streaming, 2020. https://github.com/vasia/gelly-streaming.

[5] An Implementation of Differential Dataflow Using Timely Dataflow on Rust, 2020. https://github.com/TimelyDataflow/timely-dataflow.

[6] A Modular Implementation of Timely Dataflow in Rust, 2020. https://github.com/TimelyDataflow/differential-dataflow.

[7] M. Abadi, F. McSherry, and G. D. Plotkin. Foundations of Differential Dataflow. In FOSSaCS, 2015. https://doi.org/10.1007/978-3-662-46678-0_5.

[8] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. PVLDB, 11(6), 2018. https://doi.org/10.14778/3184470.3184473.

[9] W. Ammar, D. Groeneveld, C. Bhagavatula, I. Beltagy, M. Crawford, D. Downey, J. Dunkelberger, A. Elgohary, S. Feldman, V. Ha, R. M. Kinney, S. Kohlmeier, K. Lo, T. C. Murray, H.-H. Ooi, M. E. Peters, J. L. Power, S. Skjönsberg, L. L. Wang, C. Wilhelm, Z. Yuan, M. van Zuylen, and O. Etzioni. Construction of the literature graph in semantic scholar. In NAACL-HLT, 2018. https://api.semanticscholar.org/corpus.

[10] E. Bullmore and O. Sporns. Complex Brain Networks: Graph Theoretical Analysis of Structural and Functional Systems. Nature Reviews Neuroscience, 10(3), 2009. https://doi.org/10.1038/nrn2575.

[11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4), 2015. http://sites.computer.de/dbull/a15dec/p28.pdf.

[12] J. M. F. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun. Kaskade: Graph Views for Efficient Graph Analytics. arXiv:1906.05162, 2019. http://arxiv.org/abs/1906.05162.

[13] N. Göbel. Optimising Distributed Dataflows in Interactive Environments. Master’s thesis, ETH Zürich, 2019. https://www.research-collection.ethz.ch/record/20.500.11850/343045.

[14] S. Haddadi and Z. Layouni. Consecutive Block Minimization is 1.5-Approximable. Information Processing Letters, 108(3), 2008, https://doi.org/10.1016/j.ipl.2008.04.009.

[15] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph OLAP: Towards Online Analytical Processing on Graphs. In IEEE ICDM, 2008. doi:10.1109/ICDM.2008.30.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. 3rd edition edition, 2009. http://mitpress.mit.edu/books/introduction-algorithms.

[17] J. P. G. Sterbenz, E. K. Çetinkaya, M. A. Hameed, A. Jabbar, and J. P. Rohrer. Modelling and Analysis of Network Resilience. In COMSNETS, 2011. https://doi.org/10.1109(COMSNETS.2011.5716502).

[18] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. ACM SIGPLAN Notices, 48(8), 2013, https://doi.org/10.1145/2517349.2522738.

[19] J. Shun, G. E. Blelloch, and M. A. Hameed. Efficient Graph Analytics. In IEEE ICDE, 2013. https://doi.org/10.1109/ICDE.2013.6544892.

[20] L. T. Kou. Polynomial Complete Consecutive Information Retrieval Problems. SIAM Journal on Computing, 6(1), 1977. https://doi.org/10.1137/0206004.

[21] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval Over Historical Graph Data. In IEEE ICDE, 2013. https://doi.org/10.1109/ICDE.2013.6544892.

[22] H. Kwak, C. Lee, H. Park, and S. B. Moon. What Is Twitter, A Social Network or a News Media? In WWW, 2010. https://doi.org/10.1145/1772690.1772751.

[23] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. TKDD, 1(1), 2007. https://doi.org/10.1145/1217299.1217301.

[24] M. Mariappan and K. Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In EuroSys, 2019. https://doi.org/10.1145/3302424.3303974.

[25] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In CIDR, 2013. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[26] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In ACM SOSP, 2013. https://doi.org/10.1145/2517349.2522738.

[27] S. Orzan. On Distributed Verification and Verified Distribution. PhD thesis, Free University of Amsterdam, 2004. https://www.cs.vu.nl/en/Images/SM_Orzan_5-11-2004_tcm210-258582.pdf.

[28] S. Sahu, A. Mchedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. The VLDB Journal, 2019. https://doi.org/10.1007/s00778-019-00548-x.

[29] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. ACM SIGPLAN Notices, 48(8), 2013, https://doi.org/10.1145/2517327.2442530.

[30] J. P. G. Sterbenz, E. K. Çetinkaya, M. A. Hameed, A. Jabbar, and J. P. Rohrer. Modelling and Analysis of Network Resilience. In COMSNETS, 2011. https://doi.org/10.1109(COMSNETS.2011.5716502).
[32] K. Xirogiannopoulos, V. Srinivas, and A. Deshpande. GraphGen: Adaptive Graph Processing using Relational Databases. In GRADES, 2017. https://doi.org/10.1145/3078447.3078456.

[33] G. Yadav and S. Babu. NEXCADE: Perturbation Analysis for Complex Networks. PLOS ONE, 7(8), 2012, https://doi.org/10.1371/journal.pone.0041827.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[35] P. Zhao, X. Li, D. Xin, and J. Han. Graph Cube: On Warehousing and OLAP Multidimensional Networks. In ACM SIGMOD, 2011. https://doi.org/10.1145/1989323.1989413.