Optimizing Differentially-Maintained Recursive Queries on Dynamic Graphs

Khaled Ammar
University of Waterloo, BorealisAI
Waterloo, ON, Canada
khaled.ammar@uwaterloo.ca

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

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

M. Tamer Özsu
University of Waterloo
Waterloo, ON, Canada
tamer.ozsu@uwaterloo.ca

ABSTRACT

Differential computation (DC) is a highly general incremental computation/view maintenance technique that can maintain the output of an arbitrary and possibly recursive dataflow computation upon changes to its base inputs. As such, it is a promising technique for graph database management systems (GDBMS) that support continuous recursive queries over dynamic graphs. Although differential computation can be highly efficient for maintaining these queries, it can require prohibitively large amount of memory. This paper studies how to reduce the memory overhead of DC with the goal of increasing the scalability of systems that adopt it. We propose a suite of optimizations that are based on dropping the differences of operators, both completely or partially, and recomputing these differences when necessary. We propose deterministic and probabilistic data structures to keep track of the dropped differences. Extensive experiments demonstrate that the optimizations can improve the scalability of a DC-based continuous query processor.

PVLDB Reference Format:
Khaled Ammar, Siddhartha Sahu, Semih Salihoglu, and M. Tamer Özsu. Optimizing Differentially-Maintained Recursive Queries on Dynamic Graphs. PVLDB, 15(11): 3186 - 3198, 2022.
doi:10.14778/3551793.3551862

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/khaledammar/optimized-DC.

1 INTRODUCTION

Graph queries that are recursive in nature, such as single pair shortest path (SPSP), single source shortest path (SSSP), variable-length join queries, or regular path queries (RPQ), are prevalent across applications that are developed on graph database management systems (GDBMS). Many of these applications require maintaining query results incrementally, as the graphs stored in GDBMSs are dynamic and evolve over time. For example, millions of travellers use navigation systems to find the fastest route between two points on a map. To keep the route information fresh, these systems need to continuously update their SPSP query results as road conditions change. Similarly, several knowledge graphs, such as RefinitivGraph [1] contain billions of connections between real-world entities, such as companies, banks, stocks, and managers. A Refinitiv product, World-Check Risk Intelligence[1], searches for direct and indirect connections between entities to help companies and banks comply with mandatory regulations. Since these graphs are frequently updated by new facts, these applications require the queries to be continuously evaluated.

Many GDBMSs have capabilities to evaluate one-time versions of recursive queries over static graphs, but generally do not support incrementally maintaining them. As such, in dynamic graphs, existing systems require rerunning these queries from scratch at the application layer. A GDBMS that can incrementally maintain recursive queries inside the system would lead to easier and more efficient application development. In this paper we investigate the use of *differential computation* (DC) [22], a new incremental maintenance technique, to maintain the results of recursive queries in GDBMSs. DC is designed to maintain arbitrarily cyclic (thus, recursive) dataflow programs [22, 23].

Unlike using a specialized incremental derivation rule, DC starts from a dataflow program that evaluates the one-time version of the query. By keeping track of the differences to the inputs and outputs of the operators across different iterations, called *timestamps* in DC terminology, DC maintains and propagates the changes between operators as the original inputs to the dataflow are updated. This makes DC more general than other techniques, as it is agnostic to the underlying dataflow computation.

However, DC can have significant memory overhead [17], as it may need to monitor a high number of input and output differences between operators. For example, Table 1 shows the performance and memory overhead of the DC implementation of the standard Bellman-Ford algorithm for maintaining the results of SSSP queries on the Skitter internet topology dataset [18]. In the experiment, we modify the graph with 100 batches of 1 random edge insertion each, and provide the system with 10GB memory to store the generated differences. The table also shows the performance of a baseline

[1] https://www.refinitiv.com/en/products/world-check-kyc-screening

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551862

Table 1: Execution time (in seconds) for an SPSP workload using different number of queries

Number of Queries	10	20	30	40
Scratch	6.1K	13.6K	20.7K	28.3K
Differential Computation	0.2	OOM	OOM	OOM

that re-executes the Bellman-Ford algorithm from scratch after each update, thus not requiring any memory for maintaining these queries. Although the differential version of the algorithm is about five orders of magnitude faster, it cannot maintain more than 10 concurrent queries due to its large memory requirement. This limits the scalability of systems that adopt DC.

In this paper, we study how to reduce the memory overheads of DC to increase its scalability when maintaining the popular classes of recursive queries mentioned above. Our optimizations are broadly based on *dropping differences* and instead recomputing them when necessary. We focus on optimizing the differential version of a common subroutine in graph algorithms where vertices aggregate their neighbours' values iteratively and propagate their own values to neighbours until a stopping condition, such as a fixed point, is reached. Variants of this subroutine with different aggregation, propagation, and stopping conditions can be used to evaluate all of the recursive queries we focus on in this paper. This routine consists of `Join` operator and an aggregation operator, e..g, a `Min`, and has been given different terms in literature, such as *propagate-AndAggregate* [30] or *iterative matrix vector multiplication* [15]. We refer to it as *iterative frontier expansion* (IFE).

In this work, we start with the base version implementation of DC as in the differential dataflow (DD) [22] and its precursor Naiad system [23]. We propose two main optimizations: Join-On-Demand (JOD) (Section 4) that completely drops the output differences of the `Join` operator of the IFE dataflow and only computes these differences when DC needs to inspect them; and (2) two *partial difference dropping* optimizations (Section 5) that drop some of the differences in the output of the aggregation operator in IFE.

Our partial difference dropping optimization offers users a knob to drop a certain percentage of the system's differences. We begin by describing a baseline deterministic optimization Det-Drop that explicitly keeps track of the vertex and timestamp of each dropped difference. We show that although Det-Drop reduces the memory consumption of a system, it also has inherent limitations in terms of scalability improvements, as the additional state it keeps is proportional to the amount of differences that it drops. We then propose a probabilistic approach Prob-Drop that addresses this shortcoming by leveraging a probabilistic data structure, specifically a Bloom filter. Prob-Drop may attempt to reconstruct a non-existing difference due to false negatives but it more effectively reduces the memory consumption, so a system using Prob-Drop needs to drop fewer differences to meet same memory budgets as Det-Drop. Finally, we describe an optimization that uses the degree information of each vertex to choose which differences to drop as opposed to dropping them randomly.

We demonstrate that JOD reduces the number of differences up to 8.2× in comparison to vanilla DC implementations. We also show that exploiting the degree information to select the differences to drop can improve the performance of partial dropping optimizations (Det-Drop or Prob-Drop) by several orders of magnitude. We further show that Prob-Drop achieves up to 1.5× scalability relative to Det-Drop when selecting the differences to drop based on degrees. Our optimizations overall can increase the scalability of our differential algorithms by up to 20× in comparison to DD, while still outperforming a baseline that reruns computations from scratch by several orders of magnitude.

2 RELATED WORK

Broadly, there are two approaches to maintaining the results of a computation over a dynamic graph: (i) using a computation-specific specialized solution; or (ii) using a generic incremental computation/view maintenance solution that is oblivious to the actual computation, at least for some class of computations. DC falls under the second category. Below, we review both approaches.

2.1 Specialized Techniques and Systems

There is extensive literature dating back to 1960s on developing specialized incremental versions of (aka *dynamic*) graph algorithms that maintain their outputs as an input graph changes. Many of the earlier work focuses on versions of shortest path algorithms, in particular all pairs shortest paths computation [6–9, 19, 26, 27]. These works aim at developing fast algorithms that can, in worst-case time, be faster than recomputing shortest paths upon a single update, e.g., when the edge weights are integer values.

On the systems side, there are several graph analytics systems that enable users to develop incremental versions of a graph algorithm. GraphBolt [20] is a recent shared-memory parallel streaming system that can maintain dynamic versions of graph algorithms. GraphBolt requires users to write explicit maintenance code that generic systems such as DD do not require. iTurboGraph [17] focuses on incremental neighbour-centric graph analytics with an objective to reduce the overhead of large in-memory intermediate results in systems like GraphBolt and DD.

Broadly, programming specialized algorithms or GraphBolt-like systems can be more efficient than generic solutions. For example, several references have demonstrated this difference between DD and GraphBolt [20, 29]. In contrast, generic solutions such as DD, which we focus on in this work, are fundamentally different and have the advantage that users can program arbitrary static versions of their algorithms, which will be automatically maintained. Therefore they are suitable as core incremental view maintenance techniques to integrate in general data management systems.

2.2 Generic Techniques and Systems

When an input graph is modeled as a set of relations and a graph algorithm is modeled as a query over these relations, maintaining graph computation can be modeled as *incremental view maintenance*, where the view is the final output of the query. Traditional incremental view maintenance (IVM) techniques for recursive SQL and Datalog queries have focused on variants of incremental maintenance approaches [13] such as Delete-and-Rederive, which consists of a set of delta-rules that can produce the changes in the outputs of queries upon changes to the base relations. This contrasts with DC as it does not store intermediate computations to speed up processing. Interestingly, the only incremental open-source Datalog implementation we are aware of does not use the Delete-Rederive

Table 2: Commonly used notations and acronyms.

Acronym	Description
D and δD	Vertex distance collections and their delta.
E and δE	Edges collections and their delta.
J and δJ	The output of `Join` operator and their delta.
$C_{\langle G_k, i \rangle}$	Collection C at timestamp $\langle$ graph version G_k and iteration i $\rangle$.
C^v	C's partition by a key/vertex ID v.
IFE	Iterative Frontier Expansion.
DC	Differential Computation.
VDC	Our Vanilla implementation for DC.
DD	Differential Dataflow system.
JOD	Join-on-Demand optimization
Det-Drop	Partial difference dropping optimization using a deterministic data structure.
Prob-Drop	Partial difference dropping optimization using a probabilistic data structure.

maintenance algorithm but uses DC [28]. This work compiles Datalog programs into DD programs, so ultimately uses vanilla DD, which we optimize and use as a baseline in our work.

Tegra [14] is a system developed on top of Apache Spark [32], that is designed to perform ad-hoc window-based analytics on a dynamic graph. Tegra allows the creation of arbitrary snapshots of graphs and executes computations on these snapshots. The system has a technique for sharing arbitrary computation across snapshots through a computation maintenance logic similar to DC. However, the system is optimized for retrieving arbitrary snapshots quickly instead of sharing computation across snapshots efficiently.

There has been several systems work that use the generic incremental maintenance capabilities of DC. GraphSurge [29] is a distributed graph analytics system that lets users create multiple arbitrary views of a graph organized into a *view collection* using a declarative *view definition language*. Users can then run arbitrary computations on these views using a general programming API that uses DD as its execution engine, which allows Graphsurge to automatically share computation when running across multiple views. References [31] implements a DC-based Software Defined Network Controller that incrementally updates the routing logic as the underlying physical layer changes. Similarly, RealConfig [33] is a *network configuration verifier* uses DD to incrementally verify updates to a network configuration without having to restart from scratch after every change.

3 PRELIMINARIES

In this section we first review the graph and query models used in the paper. Then we summarize the IFE recursive algorithmic subroutine and differential computation. Table 2 shows the notations and abbreviations that are used throughout the paper.

3.1 Graph and Query Model

We consider *property graphs*, so vertices and edges can have attributes. Formally, a graph $G = (V, E, P_V, P_E)$, where V is the set of vertices, E is the set of directed edges, P_V is the set of properties over vertices, and P_E is the set of properties over edges. Our continuous queries compute properties of vertices, which we refer to as their *states*. We will not explicitly model states but these can be thought of as temporary properties in P_V. For an edge e, we maintain two properties: $label(e)$, and $weight(e)$. If G is unweighted, the the weights of each edge is set to 1.

We focus on three recursive queries in this paper: SPSP, K-hop, and RPQ. K-hop is the query in which we are given a source vertex s and output all reachable vertices from s that are at a distance (in terms of hops) of $\leq k$ for a given k. Each one of these queries can interact with different parts of our graph model. The edge properties that a recursive query needs to access and the vertex states for this computation will be clear from context.

In a dynamic graph setting, an initial input graph G_0 may receive several batches of updates. Each batch is defined as a list of edge insertions or deletions $\delta E = [(u, v, label, weight, +/-)]$, which includes an edge, and its $label$ / $weight$, and a +/- to indicate, respectively, an insertion or a deletion (updates appear as one deletion and one insertion). We do not consider vertex insertions or deletions because these implicitly occur in our algorithms through explicit edge insertions and deletions. G_k refers to the actual set of edges in a graph G after G receives its k'th batch of updates δE_k (so the union of G_0 and the k batches of updates).

The problem of incremental maintenance of a recursive query Q is to report the changes to the output vertex states of Q after every batch of updates. These batches can be thought of as output in the form of $(v, state(v), +/-)$, for a vertex v and a new vertex state $state(v)$ and +/- indicating addition or removal of a state.

3.2 Iterative Frontier Expansion as a Dataflow

Iterative Frontier Expansion (IFE) is a standard subroutine for implementing many graph algorithms solving many computational problems, including graph traversal queries like SPSP, SSSP, RPQs. At a high-level, the computation takes as input the edges (possibly with properties) of a graph G and an initial set of vertex states, and, iteratively, aggregates for each vertex the states of its neighbours to compute a new vertex state, and propagates this state to its neighbours. These iterations continue until some stopping criterion is met, e.g., a fixed point is reached and the vertex states converge. Figure 1a shows the template IFE dataflow that consists of two operators, `ExpandFrontier`, that expands the frontiers and the `Stop` operator that determines when to stop the query execution.

We use and optimize variants of this basic IFE dataflow to evaluate the queries we consider. As an example, Figure 1b shows a specific instance of the IFE dataflow implementing the standard Belman-Ford algorithm for evaluating an SSSP query where vertex states are latest distances from a source vertex s. `ExpandFrontier` operator is implemented with two operators, `Join` and `Min`. For each vertex v in the frontier, `Join` sends possible new distances to v's outgoing neighbours (considering v's latest distance and possible weights on the edges). For each vertex u of v's outgoing neighbours', the new value is computed with a `Min` operator that computes the smallest received distance for u considering u's latest known distance. For different variants of shortest-path queries, RPQs, and variable-length join queries, we use the IFE template dataflow with always the same `Join` operator, but possibly different aggregator implementations and different `Stop` conditions.

3.3 Differential Computation Overview

DC [22] is a general technique to maintain the outputs of arbitrarily nested dataflow programs as the base input collections change.

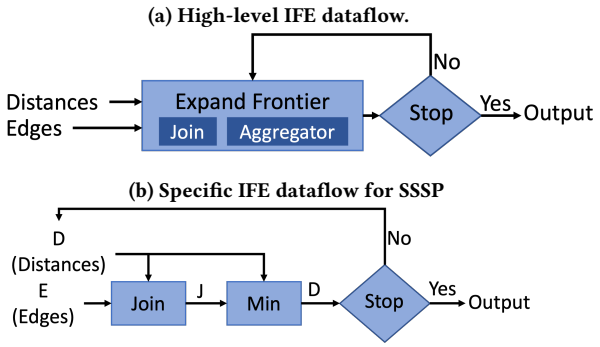


Figure 1: Template IFE dataflow and a specific Bellman-Ford algorithm's dataflow implementation.

Dataflow programs consist of operators, such as `Join` or `Min` in Figure 1b, that take input and produce output *data collections*, which are tables storing tuples. For example, in the IFE dataflow, the edges in an input graph are stored as (`src`, `dst`) tuples in the `Edges` (E) data collection. We will refer to collections, such as `E`, that are inputs to the dataflow as *base collections*, and other collections that are outputs of an operator as *intermediate collections*.

We review DC through an example. Consider the IFE instance from Figure 1b implementing the Bellman-Ford algorithm and running it on the input graph shown in Figure 2. Given this iterative dataflow computation, DC computes the input and output data collections of each operator as *partially ordered timestamped difference sets* and maintains these difference sets as the original input collections to the entire dataflow (in this case `Edges` (E) and `Distances` (D)) change. Timestamps can be multi-dimensional. For example, in the above computation, the timestamps are two dimensional, the first is *graph-version* and the second is *Bellman-Ford iteration*, which we will refer to it as *IFE iteration*, represented as a $\langle G_k, i\rangle$ pair. Collections, e.g., D, can change for two separate reasons: (1) changes in the graph (E), such as inserting an edge, or (2) changes in distances (D) during the computation of IFE iterations.

More generally, for each data collection C, let C_t represent the contents of C at a particular timestamp t, and let δC_t be the *difference set* that stores the "difference tuples" (differences for short) for C at t. Differences are extended tuples with + or - multiplicities. For base data collections, such as E, +/- indicate external insertions or deletions to them. For intermediate data collections, these may not have as clear an interpretation. Instead, the + or -'s are assigned to tuples to ensure that summing all the δC_t prior to a particular timestamp t gives exactly C_t. Sum of two difference sets adds the multiplicities for the differences with the same tuple values and if a sum equals 0, then the tuple is removed from the collection. Consider an operator with one input and one output collections, I and O, respectively. DC ensures that for each collection and operator the following equations hold:

$$I_t = \sum_{s \le t} \delta I_s \;\Rightarrow\; \delta I_t = I_t - \sum_{s<t} \delta I_s \tag{1}$$

$$O_t = Op(\sum_{s \le t} \delta I_s) \;\Rightarrow\; \delta O_t = Op(\sum_{s \le t} \delta I_s) - \sum_{s<t} \delta O_s \tag{2}$$

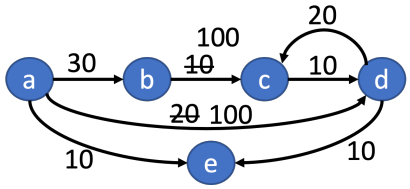


Figure 2: A dynamic graph with two updates: (i) $a \to d$ from 20 to 100 in G_1; and (ii) $b \to c$ changes from 10 to 100 in G_2.

DC uses Equations 1 and 2 to compute the differences to store in δI_t and δO_t for each timestamp. Then, DC uses these difference sets to *reassemble* correct contents of I_t and O_t at each timestamp when needed during its maintenance procedure (explained momentarily).

Suppose a system has maintained the Bellman-Ford dataflow differentially for k many updates to its base collection E; that is, the system has computed the differences for each base or intermediate collection for timestamps $\langle G_0, 0\rangle, \ldots, \langle G_k, max\rangle$, where max is the maximum number of iterations that the dataflow ran on any of $G_0, \ldots, G_k$. Given a new, $k+1$'st set of updates to the base collections, DC maintains the dataflow's computation by computing a new set of differences for collections at some of the timestamps $t = \langle G_{k+1}, i\rangle \mid i \in \{0 \ldots max\}$ by rerunning some of the operators at these timestamps. If on G_{k+1}, the Bellman-Ford dataflow computation requires more than max iterations to converge, then the system generates difference sets for timestamps $\langle G_{k+1}, i\rangle \mid i > max$.

We next explain DC's maintenance procedure. Suppose that the operators work on partitions of collections. In our example, the partitioning of the collections would be by vertex IDs and each operator would perform some computation per a vertex ID. Let C_t^v indicate the contents of C_t's partition for key v. DC reruns an operator Op at different timestamp τ according two rules:

- *Direct rerunning rule:* if Op's input I has a difference at τ for a particular key v, i.e., δI_τ^v is non-empty, DC reruns Op (on key v) at timestamp τ. That is DC reassembles $I_\tau^v = \sum_{t \le \tau} \delta I_t^v$ and executes Op on I_τ^v, which computes a new O_τ^v. Then, DC computes the difference set δO_τ^v as $\delta O_\tau^v = O_\tau^v - \sum_{t<\tau} \delta O_t^v$.
- *Upper bound rule:* For correctness, Op may need to be executed on later timestamps than τ for v even if there is no immediate differences in I at those timestamps. Specifically, DC finds every timestamps $t_f \not\le \tau$ in which Op's input has differences for key v and reruns Op on timestamps that are least upper bounds of such t_f and τ.

Importantly, if no difference is detected to vertex v's partitions of inputs of an operator for timestamps from $\langle G_{k+1}, 0\rangle$ to $\langle G_{k+1}, max\rangle$, no operator needs to rerun on v. For many dataflow computations, the effects of many updates in graphs can be localized to small neighbourhoods, and DC automatically detects the vertices in this neighbourhood on which operators need to rerun. As an example, Table 3 shows the full difference trace for each collection in the IFE dataflow implementing Bellman-Ford algorithm in the example dynamic graph in Figure 2 that has two updates: (i) an update on (a, d) from 20 to 100, at timestamp <G_1, 0>; and (ii) an update on (b, c) edge from 10 to 100 at timestamp <G_2, 0>. These updates are modeled as differences in collection E at these timestamps, which are omitted in the figure and can be found in the longer version of our paper [3]. Reference [2] formally proves that applying this

Table 3: Differences in our running example (excluding δE).

Graph Updates →

IFE iterations ↓

		G_0	G_1	G_2
0	δJ	$+(a,0), +(b,\infty), +(c,\infty), +(d,\infty), +(e,\infty)$	$\varnothing$	$\varnothing$
	δD	$+(a,0), +(b,\infty), +(c,\infty), +(d,\infty), +(e,\infty)$	$\varnothing$	$\varnothing$
1	δJ	$+(b,30), +(d,20), +(e,10)$	$-(d,20), +(d,100)$	$\varnothing$
	δD	$-(b,\infty), +(b,30), -(d,\infty), +(d,10), -(e,\infty), +(e,10)$	$-(d,20), +(d,100)$	$\varnothing$
2	δJ	$+(c,40), +(c,40), +(e,30)$	$-(c,40), +(c,120), -(e,30), +(e,110)$	$-(c,40), +(c,130)$
	δD	$-(c,\infty), +(c,40)$	$\varnothing$	$-(c,40), +(c,120)$
3	δJ	$+(d,50)$	$\varnothing$	$-(d,50), +(d,130)$
	δD	$\varnothing$	$-(d,100), +(d,50)$	$-(d,50), +(d,100)$
4	δJ	$\varnothing$	$-(c,120), +(c,70), -(e,110), +(e,60)$	$-(c,70), +(c,120), -(e,60), +(e,110)$
	δD	$\varnothing$	$\varnothing$	$\varnothing$

simple rule to decide which operators to rerun correctly maintains any dataflow computation.

4 COMPLETE DIFFERENCE DROPPING: JOIN-ON-DEMAND

When maintaining IFE with DC, the memory overheads of storing the difference sets for the output of the `Join` operator (`J`) is generally much larger than those for the output of the following aggregation operator (`D`). Consider the IFE implementation of SPSP, where edges have weights and vertex states represent shortest distances to a source vertex. Suppose at a particular iteration i of the IFE at a specific graph version G_k, a vertex v's state is (v, d_v) and v has $deg(v)$ many outgoing edges, e.g., $(u_1, w_1), \ldots (u_{deg(v)}, w_{deg(v)})$. Then to simulate v propagating possible new shortest distances to its outgoing neighbours, `J` would contain $deg(v)$ many tuples at timestamp $\langle G_k, i\rangle$: $(u_1, d_v + w_1)$, ..., $(u_{deg(v)}, d_v + w_{deg(v)})$. Similarly, the partition J^u of `J` contains one tuple for each of u's incoming neighbours. When maintaining IFE differentially, `J`'s size is commensurate with the number of edges in G, which can be much larger than `D`, whose size is commensurate with the number of vertices in G.

Example 1. *Observe that in Table 3, δD has two differences for vertex d at timestamp $\langle G_1, 1\rangle$, $-(d, 20)$ and $+(d, 100)$. These changes lead to four differences in δJ because d has two outgoing edges, one to c and the other to e.*

The goal of JOD is to avoid storing any difference sets for `J`, i.e., to completely drop δJ, and regenerate J^u for any u on demand when DC requires running the aggregation operator (in our example `Min`) on u at a particular timestamp. We first describe an unoptimized version of JOD, then describe an optimization called *eager merging* that reduces the timestamps to regenerate J^u, which is the optimized JOD we have implemented.

4.1 JOD

Recall that DC reruns `Min` on a vertex u at timestamp $t = \langle G_{k+1}, i\rangle$ if (1) δD^u_t or δJ^u_t are non-empty (direct rule); or (2) t is an upper bound of τ_1 and τ_2 that satisfy the following conditions (upper bound rule): (i) $\tau_1 \in \mathrm{T}_1 = \{\langle G_{k+1}, i'\rangle | i' < i\}$ and $\delta D^u_{\tau_1}$ and/or $\delta J^u_{\tau_1}$ are non-empty; and (ii) $\tau_2 \in \mathrm{T}_2 = \{\langle G_{k'}, i\rangle | k' < k+1\}$ and $\delta D^u_{\tau_2}$ and/or $\delta J^u_{\tau_2}$ are non-empty. If δJ are dropped, how can we correctly decide when to rerun `Min` and recompute the needed dropped δJ for these reruns to ensure we correctly differentially maintain IFE? DC$^{\text{JOD}}$ is our modified version of DC maintenance subroutine that has this guarantee, which works as follows. In the below description, when `Min` is rerun on u at timestamp t, J^v_t is constructed by inspecting for each incoming neighbour w of u, D^w_t and E^w_t and performing the join. Note that we do not drop the differences related to `D` and `E`.

DC$^{\text{JOD}}$:

- *δE Direct Rule: For each $(u, v, l, p, +/-) \in \delta E_{k+1}$, since there is a difference in $\delta E^u_{\langle G_{k+1},0\rangle}$, there is also a difference in $\delta J^v_{\langle G_{k+1},0\rangle}$. So we rerun `Min` on v in $\langle G_{k+1}, 0\rangle$ (direct rule).*
- δD Direct Rule: Each time `Min` reruns on u at a timestamp $\langle G_{k+1}, i\rangle$, we check if it generates a difference for $\delta D^u_{\langle G_{k+1},i+1\rangle}$. If so, this implies there is a difference in $\delta J^v_{\langle G_{k+1},i+1\rangle}$ for each outgoing neighbour v of u. Therefore we schedule `Min` on v at timestamp $\langle G_{k+1}, i+1\rangle$ (direct rule).
- *Upper Bound Rule:* Each time we schedule to rerun `Min` on a vertex v, either by δE or δd Direct Rule at timestamp $\langle G_{k+1}, i+1\rangle$, by the upper bound rule, we schedule to rerun `Min` on v at timestamp $\langle G_{k+1}, j\rangle$ s.t. $j > i+1$ if either of these two conditions are satisfied: (i) there is a non-empty $\delta D^v_{\langle G_h,j\rangle}$ s.t $h < k+1$; and (ii) there is an incoming neighbour w of v with a non-empty $\delta D^w_{\langle G_h,j\rangle}$ s.t., $h < k+1$.

In the longer version of our paper [3], we prove inductively, starting from $\langle G_{k+1}, 0\rangle$ to $\langle G_{k+1}, max\rangle$, that the above procedure reruns `Min` on every vertex v in the timestamps that vanilla DC would rerun and produces the correct differences for `D`.

Example 2. *We next demonstrate applications of JOD's rerunning rules on our running example. Consider the first update in our running example at timestamp $\langle G_1, 0\rangle$, which updates the weight of edge (a, d) from 20 to 100. By the δE Direct Rule of JOD, we rerun `Min` on d at timestamp $\langle G_1, 0\rangle$. Further by JOD's Upper Bound Rule, we also schedule to run d at timestamp $\langle G_1, 2\rangle$ because $\delta D^c_{\langle G_0,2\rangle}$ is non-empty and c is an incoming neighbour of d (condition (ii)). Note that rerunning `Min` on d at timestamp $\langle G_1, 0\rangle$ creates a difference for $\delta D^d_{\langle G_1,1\rangle}$. By the δD Direct Rule, we further schedule to rerun `Min` on c and e, which are the outgoing neighbours of d, at timestamp $\langle G_1, 1\rangle$.*

4.2 Eager-Merging

The naive implementation of JOD can be expensive because the number of possible timestamps $\langle G_h, i\rangle$ to inspect, where $h < k+1$, can grow unboundedly large as batches of edge updates continue to arrive. The eager merging optimization we describe next, which extends a periodic merging optimization of the DD system (explained momentarily), reduces the number of these timestamps.

Consider the point at which a new set of updates to graph version G_k has arrived and the system has finished maintaining the

Table 4: Differences in D on our running example with eager-merging when maintaining the computation for $\langle G_2, 2\rangle$.

Graph Updates →

IFE iterations ↓	G_0	G_1	G_2
0			$+(a,0), +(b,\infty), +(c,\infty), +(d,\infty), +(e,\infty)$
1			$+(b,30), +(d,100), +(e,10)$
2		$+(c,40)$	
3		$+(d,50)$	

computation for G_k. So there are $k \times max$ many different timestamps in the computation so far. Let us think of these timestamps in a 2D grid with columns as graph version indices and rows as IFE iterations as in Table 3.

Observe that as more updates arrive to the system, the timestamps will increase in the graph version dimension to G_{k+1}, G_{k+2}, etc, so more columns will be added to this grid. Consider reassembling the contents of some collection C at timestamp $\langle G_{k+1}, 0\rangle$. To do so, DD has to sum the differences in $\delta C_{Row_0} = \{\delta C_{\langle G_0,0\rangle}, ..., \delta C_{\langle G_k,0\rangle}\}$. To reassemble C at timestamp $\langle G_{k+1}, 1\rangle$, DD has to sum the difference sets in δC_{Row_0} and $\delta C_{Row_1} = \{\delta C_{\langle G_0,1\rangle}, ..., \delta C_{\langle G_k,1\rangle}\}$, etc. Observe that once the (k+1)'st graph updates have arrived, the system will never have to re-execute an operator at timestamps $\langle G_h, i\rangle$ where $h < k+1$. Instead of computing δC_{Row_j} multiple times for each possible $\{C_{\langle G_{k+1},j\rangle} | j > i\}$, the original DD periodically unions the individual difference sets in δC_{Row_j} into a single difference set $\delta C_{\langle G_{k+1},j\rangle}$. This allows DD to reassemble collections faster and store the difference sets more compactly.

Instead of periodic merging, we eagerly merge the differences along the graph-version dimension as we run DC's maintenance procedure for $\langle G_{k+1}, 0\rangle$ to $\langle G_{k+1}, max\rangle$. That is, as soon as DC finishes maintaining $\langle G_{k+1}, i\rangle$, we merge the difference sets for D for timestamps $\langle G_k, i\rangle$ and $\langle G_{k+1}, i\rangle$. This guarantees that for any vertex, we only need to keep one-dimensional timestamps, i.e., only for *IFE iteration*. Table 4 shows the states of the differences stored in the system with eagerly merging differences and the DC algorithm is in the process of maintaining the computation at timestamp $\langle G_2, 2\rangle$. Differences at grey cells have been merged to the right most cell on the row. In presence of eager merging, whenever JOD needs to investigate if $\delta D_{\langle G_h,i\rangle} \mid h < k+1$ is non-empty for any vertex, we only need to inspect timestamps with $h = k$.

We end this subsection with a discussion of another benefit of eager merging. Eager merging allows dropping all differences with negative multiplicities in the difference sets for D. This is because in the algorithms we consider, vertices take one unique state at each iteration of IFE. Therefore in one-dimensional timestamps, the change in the state of a vertex from s to s' at iteration i, is always represented with two differences: (i) one with positive multiplicity with s'; and (ii) one with negative multiplicity for s. In absence of negative multiplicities, we can also avoid doing any summations when computing the state of a vertex at timestamp i, i.e., D_i^v. Instead we can find the latest iteration $i^* \leq i$ in which vertex v has a (positive) difference and return it.

5 PARTIAL DIFFERENCE DROPPING

We next investigate optimizations that partially drop the differences in D. When we apply JOD, D is the only data collection for which we store differences, except for the original edges in the graph. Partial dropping the differences in D allows trading off scalability with query performance. Specifically, the memory overhead to store D decreases, yet it also decreases performance because when DC needs to reassemble the contents of D at a timestamp t, the dropped differences need to be recomputed. In this section, we will describe optimizations with different scalability/performance tradeoffs. Throughout this section we assume running DC^{JOD} with eager merging and use single dimensional timestamps to refer to data collections, such as D_i, instead of $D_{\langle G_k,i\rangle}$.

A partial dropping optimization has two key components:

- *Dropped Difference Maintenance*: When DC^{JOD} accesses D_i^v, the system needs to identify if a difference was dropped with key/vertex ID v at timestamp i. Therefore, the system needs to maintain the dropped vertex ID-timestamp pair information.
- *Selecting the Differences to Drop:* The system also needs to decide which differences to drop and which ones to keep.

We describe alternative approaches to both components.

A third important decision is to choose how many differences to drop given a memory constraint. At a high-level, the answer to this question is clear: drop as little as possible without violating the memory constraint. In practice however estimating this amount may be challenging because each update to the graph changes the amount of differences needed to maintain registered queries. Further, a system needs to estimate and plan for newly registered or deregistered queries. In such dynamic scenarios, systems can adopt adaptive techniques that determine how many differences to drop from each query by observing the stored differences. This is a future topic for a rigorous within study. Within the context of this paper, we will assume a user-define probability p that drops each difference with probability p (see Section 5.2).

5.1 Dropped Difference Maintenance

One natural approach to maintaining the dropped vertex ID-timestamp pairs (VT pairs for short) is to store them explicitly in a separate data structure `DroppedVT`. We discuss two possible designs for this data structure. We first present a straightforward deterministic data structure, and discuss its scalability bottlenecks. Then, we propose a probabilistic data structure, which can address this scalability bottleneck but possibly leading to spurious re-computations of undropped differences. In our evaluations, we show that, despite this possible performance disadvantage, our probabilistic approach can still be more performant as it can drop fewer differences than our deterministic approach under limited memory settings.

5.1.1 Deterministic Difference Maintenance (Det-Drop). Det-Drop uses a hash table to implement `DroppedVT`. During DC^{JOD}, when D_i^v is needed, we perform the following $\texttt{AccessD}_i^v\texttt{WithDrops}$ procedure. Before we describe our procedure, recall from Section 4.2 that we do not store differences with negative multiplicities for D when we eagerly merge differences, so we do not need to do any summation to compute D_i^v. We only need to find and return the latest iteration $i^* \leq i$ for which there is a difference for v.

$\texttt{AccessD}_i^v\texttt{WithDrops}$:

1. Let δD be the index that stores the difference sets for D. We check δD for the latest iteration $g^* \leq i$, if any, for which the system has stored a difference for v.

Figure 3: Degree-Drop Strategy for dropping differences

2. Check DroppedVT for the latest iteration $d^* \leq i$, if any, for which the system has dropped a difference for vertex v.
3. If a $d^* > g^*$ exists, recompute the stored difference at d^* and return this value. Otherwise, return the value at δD at g^*.

Note that to recompute a dropped difference at timestamp d^* in step 3 we rerun the aggregation operation, e.g., Min, for vertex v at iteration $d^* - 1$. This procedure is similar to how we rerun Min operator for vertices at different timestamps as part of the DC^{JOD} procedure. Then using $J^v_{d^*-1}$ and $D^v_{d^*-1}$ we rerun Min and compute $D^v_{d^*}$. However when we access $D^v_{d^*-1}$, we recursively call $\texttt{AccessD}^v_i\texttt{WithDrops}$, as there may be dropped differences for v or one of its incoming neighbours w at timestamp $d^* - 1$. Therefore, this may lead to further recomputations, which may further cascade.

Example 3. *Consider the running example. Suppose that after the first update, the system decides to drop the difference* $+(b, 30)$ *at iteration 1. Consider now the arrival of the second update where the weight of* (b, c) *changes from 10 to 100. To maintain the computation differentially,* Min *is rerun on* c *at* $\langle G_2, 1\rangle$ *(due to* δE *Direct Rule) and then due to the Upper Bound Rule on every timestamp in which* c *has a difference.* c *already has a difference that is not dropped at iteration 2, so* c *is scheduled to rerun at iteration 2. We further check if* c *has any dropped differences at iterations 3 and 4. Since it does not, we do not schedule* c *to rerun at these differences. Then when rerunning* c *at 2, we need both* b*'s and* d*'s distances at iteration 1 and check if they have any differences. We see that* d *has stored difference but* b *does not, so we check if* b *has a dropped difference at 1. Since it does, we recompute that difference by rerunning* b *at iteration 1.*

Explicitly keeping track of all dropped VT pairs requires keeping additional state that is proportional to the number of differences that are dropped, which limits its scalability. Note that a difference is simply a triple that consists of a VT pair plus a vertex state (e.g., distance). Suppose we need d bytes to store VT pairs and s bytes to store the actual state in a difference. Then, for each dropped $d + s$ bytes, we have to keep d bytes in DroppedVT. This means that even if we partially drop 100% of differences, there is a hard limit of $\frac{d}{d+s}$ on the scalability benefits we can obtain from deterministically dropping differences. Our next optimization overcomes this limitation by using a probabilistic data structure.

5.1.2 Probabilistic Difference Maintenance (Prob-Drop) . Prob-Drop drops the entire difference, i.e., both the VT pair and the state, and uses a probabilistic data structure to maintain the dropped VT pairs. Probabilistic data structure, such as Bloom [4] or Cuckoo filters [11], have the advantage that their sizes can remain much smaller than the amount of data they store. Prob-Drop requires a probabilistic data structure that never returns false negatives because if a VT pair was dropped and the structure returns false when queried, we may ignore this difference and reassemble incorrect states for vertices during DC^{JOD}. However, the structure can return false positives, because false positives can only lead to unnecessarily recomputing a vertex state, but the recomputed vertex state will still be correct.

We use a Bloom filter[2], into which we insert the dropped VT pairs. Using a Bloom filter requires minor modifications to the $\texttt{AccessD}^v_i\texttt{WithDrops}$ procedure from Section 5.1.1. Specifically, in the second step, the procedure needs to check the Bloom filter for each potentially dropped difference at iteration $d \in (g^*, i]$ starting from i to see if a VT pair for (v, d) was dropped. If the answer is negative, then processing moves to the next d until we arrive at g^*. In this case, the value from D for iteration g^* (obtained from step 1) is the correct value of v at iteration i. If the answer is positive for an iteration $d^* \in (g^*, i]$, then the value of pair (v, d^*) is recomputed.

In our evaluations, we show that Prob-Drop can increase the scalability of a GDBMS more than Det-Drop because its size does not grow as the system drops more differences. Furthermore, in some settings, the system does not need to drop as many differences in Prob-Drop as in Det-Drop to reach a certain scalability level (in our evaluations this is the number of concurrent queries).

5.2 Selecting the Differences To Drop

The second component of a partial difference dropping optimization is to decide which differences to drop. A baseline heuristic is to drop each difference uniformly at random. We next show a more optimized technique that use the degree information of vertices to select the differences to drop.

5.2.1 Degree-based Difference Dropping. A GDBMS using DC to maintain continuously running recursive queries can exploit the fact that the dataset is a graph, therefore partitioning keys are vertex IDs. Intuitively, when executing the recursive algorithms we consider, high degree vertices are used frequently when computing the states of other vertices, i.e., they will be accessed more by DC when maintaining the input IFE dataflow. Therefore, dropping their differences can lead to frequent vertex state re-computations. Similarly, vertices with low-degrees are relatively less frequently accessed by DC. Based on this intuition, we implement a heuristic that takes two thresholds τ_{min} and τ_{max}, for minimum and maximum degrees, respectively, and a probability parameter p. Then, our heuristic performs the following for a difference with a VT pair ⟨vertex, iteration⟩ pair ($\langle v, i\rangle$) assuming that $deg(v)$ is the degree of vertex v (Figure 3):

- If $deg(v) < \tau_{min}$, drop the difference.
- If $deg(v) > \tau_{max}$, do not drop the difference.
- Otherwise drop the difference with probability p.

We found empirically that setting τ_{min} as 2 and τ_{max} as the top 80th degree percentile of the input graph is reasonable for the graphs we used in the experiments. We note that more sophisticated properties, such as betweenness centrality of vertices, can also be used to decide the differences to drop. A practical advantage of using degrees is that, degree information is readily available in adjacency list indices, which are ubiquitously used in GDBMSs.

6 EVALUATION

6.1 Experimental Setup

We run all experiments on a Linux server with 12 cores and 32 GB memory, unless mentioned otherwise. For each experiment, we report the total time, in single-threaded execution, needed to update

[2]We use the Bloom filter implementation from https://github.com/lemire/bloofi

Table 5: Datasets

Name	$\|E\|$	$\|V\|$	Max. Degree	Avg. Degree	Avg. In-Degree
LiveJournal (LJ)	69M	4.8M	4K	8.5	14.2
Skitter (SK)	11M	1.7M	35K	8.2	12.6
Patents	16.5M	3.8M	704	2.3	4.7
Orkut	117.2M	3	29.6K	17.7	34.4
LDBC SNB	77.6M	7.2M	20.8K	7.3	9.8

the graph and the query answer after applying a batch of updates. For each dataset, we shuffle the edges, and split the dataset such that 90% of the data is used as an initial graph, while the remaining 10% models the dynamism in the graph consisting of the update to the graph. We use a default batch size of 1, because differential computation is more suitable for near real-time dynamic graph updates than for infrequent updates. We evaluate the effects of batch size on the performance of DC in the longer version of our paper [3]. We use insertion-only workloads in our main experiments. In the longer version of our paper [3], we also present experiments that use workloads with different amounts of deletions for our main experiments.

6.1.1 Datasets. We use a combination of real and synthetic graphs summarized in Table 5[3]. The four real graphs are **Skitter**, **LiveJournal**, **Patents**, and **Orkut**, all obtained from [18]. **Skitter** represents an internet topology from several scattered sources to millions of destinations on the internet and its vertices are strongly connected. **LiveJournal** and **Orkut** [18] represent social network interactions with a vertex degree distribution that follows power-law. **Patent** [18] represents a citation graph for all utility patents granted between 1975 and 1999. In order to experiment with weighted SPSP queries, we created weighted versions of both graphs by adding a random integer weight between 1 and 10 uniformly at random to each edge. **LDBC SNB** [10] is a synthetic graph that models dynamic interactions in social network applications. This graph has edge labels that are used in RPQ queries. LDBC SNB includes several types of entities, such as persons or forums. Each edge has a label such as Knows or ReplyOf. We use a scale factor of 10 that generates a graph of 7.2M vertices and 77.6M edges.

6.1.2 Workloads. We use SPSP, K-hop, and several popular RPQ queries as our main query workloads. We run SPSP and K-hop on the weighted and unweighted versions of the real datasets, respectively. For SPSP query generation, we pick a random pair of vertices in the graph. For K-hop, we pick a random set of vertices and set the value of maximum hops $K = 5$ to make it a 5-hop query.

RPQ queries require edge labels, so this experiment is conducted only on the LDBC dataset. We use a set of RPQ templates used in real-world workloads as defined in reference [5] which were used to study streaming RPQ evaluation in reference [24]. We use the following RPQ query templates:

- $Q_1 = a^*$
- $Q_2 = a \circ b^*$
- $Q_3 = a \circ b \circ c \circ d \circ e$

We used Likes, Knows, ReplyOf, and hasCreator, to construct queries from these templates in the LDBC SNB dataset.

[3]Reported degrees are for the initial loaded graphs in the experiments.

SPSP, K-hop, and RPQs are queries that can be supported in high-level languages of GDBMSs. These are the main queries that motivate our work. However our optimizations are applicable to other computations based on IFE. To demonstrate this, we implemented the differential versions of standard weakly connected components (WCC) algorithm, which is based on iteratively propagating and keeping track of minimum vertex IDs, and PageRank (PR) (ran a fixed 10 number of iterations) in our setting.

6.1.3 Baselines and Different GraphflowDB Configurations. We implement our optimizations inside the continuous query processor (CQP) of GraphflowDB [16], which is a shared memory GDBMS. We extended the CQP of GraphflowDB to implement a baseline DC and our optimizations to maintain the recursive queries we cover (see our longer paper [3] for the details of our implementation). We call the GraphflowDB configurations for different configurations of DC as: VDC, JOD, Det-Drop, or Prob-Drop.

We compare our proposed optimizations with three baselines: DD, Scratch, and DC. DD is an implementation of our workloads in the Differential Dataflow system [21], which is the reference implementation of differential computation. Scratch is a baseline extension of GraphflowDB's CQP to support our queries by simply executing each query from scratch after every batch of changes. Scratch represents a baseline GDBMS's performance that does not support continuous queries. We use an IFE-like label propagation algorithm for K-hop queries and RPQs. We note that this algorithm is identical to what is referred to as the "incremental" fixed point algorithm in the original Differential Dataflow paper [22] (see Figure 1 in the reference). This term is used to indicate that only the vertices whose values are updated in a particular iteration propagate their labels in that iteration (as opposed to all vertices).

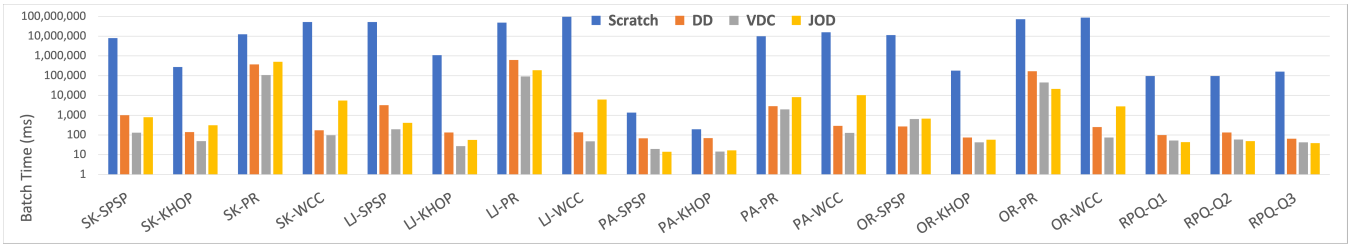
VDC is the vanilla differential computation implementation in GraphflowDB. The difference between VDC and DD is that the former is our single machine implementation using Java while the latter is a distributed system implemented in Rust. In Section 6.2, we verify that VDC behaves similar to DD (and even outperforms it in terms of runtime); therefore, we use VDC as a suitable baseline for our optimizations that is implemented inside the same GDBMS. VDC ingests and stores the input graph in the same way, uses similar data structure to store the differences, and the same programming language as the following GraphflowDB configurations:

(1) JOD: The DC version that implements join-on-demand optimization from Section 4;
(2) Det-Drop: Integrates deterministic partial dropping optimization on top of JOD as discussed in Section 5.1.1;
(3) Prob-Drop: Integrates probabilistic partial dropping optimization on top of JOD as discussed in Section 5.1.2.

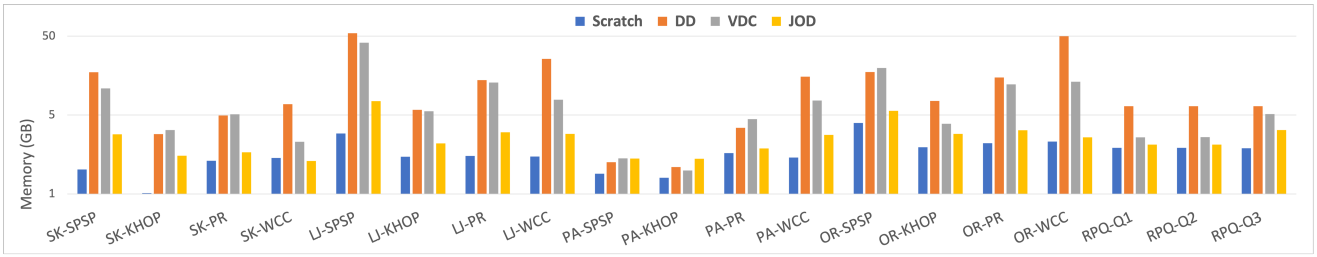
We also evaluate different versions of Det-Drop and Prob-Drop to evaluate our degree-based difference dropping optimization.

6.2 Baseline Evaluation

Our first set of experiments measure the performances of Scratch, DD, and VDC. Our goals are: (i) to obtain baseline measurements for our optimized DC implementations; and (ii) to validate that VDC is competitive with DD to justify its use as a more suitable baseline than DD for our optimizations. In this experiment, we ran SPSP, K-hop queries, WCC and PR on Skitter, LiveJournal, Patents,



(a) Total Batch Time (ms)



(b) Memory (GB)

Figure 4: Comparison between Scratch, DD, VDC, and join-on-demand (JOD).

and Orkut datasets, and all three RPQ queries on LDBC dataset. For SPSP, K-hop and RPQ workloads, we used 10 queries. In each experiment, we simulated dynamism by using 100 insertion-only batches, with 1 edge in each batch.

Our results are shown in Figure 4 (ignore the JOD charts for now). As shown in the figure Scratch, as expected, is several orders of magnitude slower than VDC and DD but also has the smallest memory overheads. Scratch is most competitive with VDC and DD in PR, though still over an order of magnitude slower. This is expected because as also observed in prior work [29], during differential maintenance, the changes in PR are harder to localize to small neighbourhoods as in other computations, i.e., small changes are more likely to change the PR values of larger number of vertices. We observe that VDC is slightly faster than DD while using comparable memory. We expect VDC to be faster than DD because DD assumes a distributed setting where messaging involves network protocols, even though we are running DD in a single machine setting. Instead, VDC assumes a shared memory setting avoiding such communication.

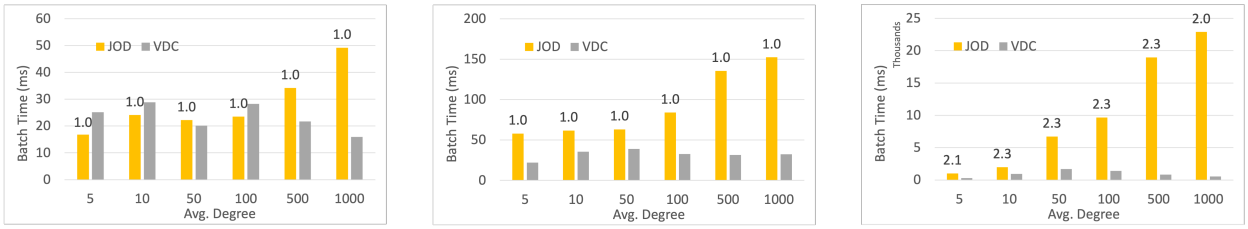
The longer version of our paper [3] repeats these experiments with two different update workloads that include deletions: (i) where 25 of the batches are deletions; and (ii) where 50 of the batches are deletions. We observe that the performance tradeoffs our optimizations offer are broadly similar across these different update workloads. Note that this is expected as the amount of updates we ingest is relatively minor compared to the number of edges we start with, which recall comprise 90% of all edges in each dataset. Overall these results confirm that VDC is a more suitable baseline for analyzing the effects of our optimizations than DD. In the remainder, we use VDC and Scratch as the main baselines to evaluate our proposed optimizations on top of VDC.

6.3 Join-On-Demand

Our next set of experiments aim to study the performance and memory benefits and overheads of JOD. JOD is guaranteed to reduce the memory overhead of a system implementing vanilla differential computation, e.g. DD or VDC. However, in terms of performance, JOD has both computation overheads and benefits. On the one hand using JOD reduces the work done by vanilla differential computation for storing differences. However, as updates arrive, JOD requires re-computing the join on demand by reading the states of in-neighbours' of vertices at different timestamps to inspect if some δJ partitions are non-empty. This is less performant than materializing δJ difference sets and inspecting them to see if they are non-empty. Our goal is to answer: *What is the net effect of these performance benefits and costs? What governs this net effect?*

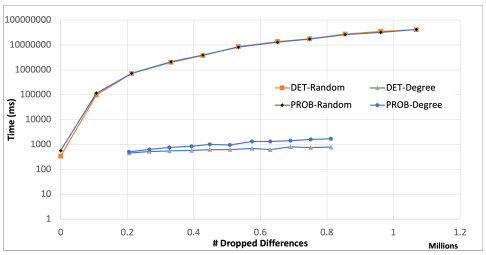
Our hypothesis is that JOD computation overhead increases proportionally with the average degree of the input graph. This is because, given a vertex v, looping through v's incoming neighbours to re-compute the join at a timestamp t should increase with the number of neighbours of v. At the same time the benefits of JOD from not storing the differences depends on how many differences are produced by the `Join` operator. This depends partially on average degree but also on the average number of times the state of a vertex changes during a computation. For example, readers can see that in the full difference trace of our running example, which is presented in Table 3, there is a new δJ difference only when the state of a vertex changes. As we will momentarily demonstrate, this number is quite small and does not necessarily grow as the average degree increases on our computations. Therefore as the average degree increases, we expect that JOD's overhead to increase faster than its benefits, and we should eventually see VDC outperforming JOD in terms of performance.

In our first experiment, we rerun our baseline experiments from Section 6.2 with JOD. The average in-degrees Orkut, Skitter, LiveJournal, Patents, and LDBC (for the subgraph containing `Knows` edges) are respectively, 34.4, 12.6, 14.2, 4.7, and 4.7. So expect VDC to be faster than JOD by larger margins on Orkut and Skitter and smaller margins on Patents and LDBC. Our results are shown in Figure 4. As expected, we observe that JOD uses significantly less memory (between 1.2× to 5.5×) than VDC irrespective of the input graph or query. In terms of performance, we find as expected that VDC is faster than JOD on Orkut (1.3x on k-hop) and Skitter (4.6× on K-hop) and even slower than JOD on Patents (2.4x on SPSP) and on LDBC RPQs (by a factor of 1.2×).

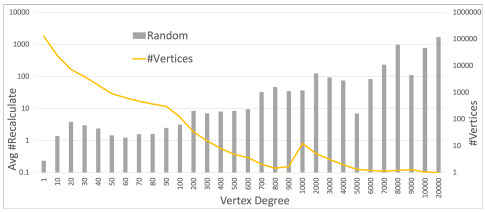


(a) RPQ-Q1 on the Knows subgraph of LDBC. (b) K-hop on the Knows subgraph of LDBC. (c) SPSP on the Knows subgraph of LDBC.

Figure 5: Comparison of VDC and JOD when running RPQ-Q1, K-hop, and SPSP as we increase the average vertex degree in the Knows subgraph of LDBC. Numbers on top of the are the average number of differences in δD per vertex.



(a) Performance of difference selection policies.



(b) Average number of dropped differences re-computed per vertex.

Figure 6: Comparison of Random and Degreebased difference dropping when running 10 K-hop queries.

Although the previous experiment provides support for our hypothesis, the degree differences between the input graphs we used are still relatively close to each other and we did not control for the queries we used across these datasets. We next perform a more controlled experiment. Using LDBC, we systematically increase the average degree of the Knows subgraph from its original value (4.7 to 20) to 100, 500, and 1000 and run all of our algorithms SPSP, K-hop, and RPQ query Q1 on each version of these graphs. We increase the average degree by adding random edges that connect vertices in this subgraph.

Our results are shown in Figure 5. As we expect, when the average degrees are small, specifically for RPQ queries, JOD either outperforms or is competitive with VDC, but as the degrees get large, VDC consistently outperforms JOD. The numbers on top of the VDC and JOD bars in Figure 5 are the average number of differences in δD per vertex for vertices that have non-zero differences, measured at the end of the experiment. Note that this number is always 1.0 for K-hop and Q1 as vertex values in these computations take only one value assigned at the first iteration in which a vertex becomes reachable.

6.4 Selecting the Differences To Drop

We next evaluate the effectiveness of the two strategies we discussed in Section 5.2 for selecting which differences to drop in our partial dropping optimization. We refer to these as: (i) Random which randomly selects the differences with a probability p; and (ii) Degree drops differences based on vertex degrees. As we discussed in Section 5.2.1, we expect Degree to outperform Random.

We run 10 K-hop queries over Skitter with 100 insertion-only batches of size 1 using DET-DROP and PROB-DROP with both Random and Degree selection strategies. In total, we have 4 system configurations. For Degree, we set τ_{min} to 2 and τ_{max} to the 80^{th} percentile of the vertex degrees. We increase the dropping probability p for DET-DROP and PROB-DROP starting from 0 to 100% and plot the total number of dropped differences on the x-axis and the runtime on the y-axis. Figure 6 shows our results. First, observe that as expected all of the lines in the figure go up, i.e., as we drop more differences the performance of each system configuration gets slower. Note that in JOD storing fewer differences potentially leads to performance advantage as we have to maintain less differences. This advantage does not exist for partial dropping optimizations because they still have to store and maintain auxiliary data structures to maintain the dropped differences. So dropping differences primarily has a performance cost, as it can lead the system to recompute those dropped differences. Second observe that as we expect, configurations with Degree (the two bottom lines), irrespective of if we use DET-DROP and PROB-DROP, are between 3 to 5 orders of magnitudes faster than the configurations with Random (two top lines). Note that the lines with Random have a bigger span on x-axis because there are limits to the minimum and maximum number of differences that configurations with Degree can drop. For example, at the minimum when $p = 0$, the configurations with Degree still drops all differences of vertices with degree $< \tau_{min}$, whereas Random can drop as few as 0 differences.

We perform further analyses using a micro-benchmark to better explain the performance difference between Random and Degree. We first fix the drop probability p (= 0.1), a workload (10 K-hop queries) and a dataset (Skitter with 100 batch of 1 edge insertions). We then use DET-DROP with Random selection policy and count for each vertex v the number of times DET-DROP re-computed a dropped difference with key v, i.e., how many times DET-DROP has accessed D^v at some point, but v's state had to be re-computed because a difference was dropped in DroppedVT. Then we bucket vertices by their degree, where for each degree bucket (e.g., $[1-10)$) we plot the average number of re-computations for each vertex in that bucket. Figure 6b shows our results. The bar charts use the left y-axes and represent the average number of re-computations for vertices with different degree buckets, where a tick in the x-axes represents a bucket with the next tick. The line chart uses the right y-axes and plots the vertex degree distribution in the graph.

As shown in Figure 6b, the degree distribution follows a power-law distribution, as is commonly the case in real world graphs. The average number of re-computations per vertex follows the opposite trend where vertices with smaller degrees on average lead to fewer re-computations, e.g, vertices with degree more than 2000 lead to more than 1000 re-computations on average, while those with degrees $[1, 10)$ lead to less than 1 re-computations. Since the memory saving of dropping 1 difference is the same regardless of the vertex degree, as done by our `Degree` strategy, it is more efficient to drop more differences from vertices with smaller degrees.

6.5 Difference Maintenance

Our next set of experiments focus on evaluating Det-Drop and Prob-Drop. In the experiments reported in Figure 6a, we evaluate the performances of Det-Drop and Prob-Drop when both drop exactly the same number of differences when using `Degree` and `Random` selection policies. They behave similarly when using the same selection strategy, with Det-Drop slightly more performant, which is expected as Prob-Drop may perform spurious re-computations due to false positives. However, Det-Drop and Prob-Drop do not have similar memory footprints when they drop the same number of differences: Prob-Drop's approach is more efficient than Det-Drop. We next provide a more systematic evaluation of the scalability and performance tradeoffs of these techniques under `Degree` policy, which as we established outperforms `Random`.

Our experiment analyzes how much Det-Drop and Prob-Drop increases the system scalability in terms of the number of concurrently maintained queries relative to VDC for a given memory budget for SSSP, K-hop, and RPQ queries. We omit PageRank and WCC from these experiments, as these are batch computations and we cannot increase the number of queries for these. For completeness, we also evaluate the performances of JOD and Scratch. To simulate a fixed memory budget environment, we give each system configuration 10GB memory for storing differences and/or additional data structures, e.g., to manage dropped VT pairs. We repeat our experiment from Section 6.2 with the same datasets and query combinations. However, we now increase the number of queries systematically until the system runs out of memory.

Figure 7 shows our results. We use the maximum scalability level of VDC, which is the configuration with the highest memory overheads, as the lowest number of queries we use and increase the number of queries in the system from this point on. That is why VDC appears as a single grey point in our charts. For Det-Drop and Prob-Drop, for each number of queries q, we find the lowest dropping probability p_{det} for Det-Drop and p_{prob} for Prob-Drop that can support q queries and report their performances with these levels. Note that here we are assuming an ideal setting in which a system is able to find this lowest dropping probability. Although this may be challenging in practice, this allows us to evalaute the most performant versions of Det-Drop and Prob-Drop for the given query level. We show p_{det} that we use for Det-Drop under the Det-Drop line, and the p_{prob} that is used for Prob-Drop above the Prob-Drop line.

We make several observations. First, as in Figure 4, we see that JOD can increase the number of queries that could be concurrently run by $2.3\times - 10\times$ over VDC. Second, we observe that increasing the number of queries with partial dropping optimizations can increase the run time super-linearly beyond a particular point where increasing scalability requires increasing the dropping probability, which leads to more differences to be re-computed. However, we see that partially dropping differences can still increase the number of concurrent queries by up to $20\times$ relative to VDC while still outperforming Scratch by several orders of magnitude. Third, we compare the performances of Det-Drop and Prob-Drop. As mentioned earlier, Det-Drop does not incur any spurious re-computations due to false positives but has to drop more differences than Prob-Drop to scale to more queries (as it has a higher memory overhead for storing the dropped VT pairs). We see that this advantage and disadvantage overall balance out for the scalability levels both Det-Drop and Prob-Drop can handle, i.e., they perform similarly at these scalability levels. However, Prob-Drop can consistently scale to higher levels than Det-Drop (up to $1.5\times$).

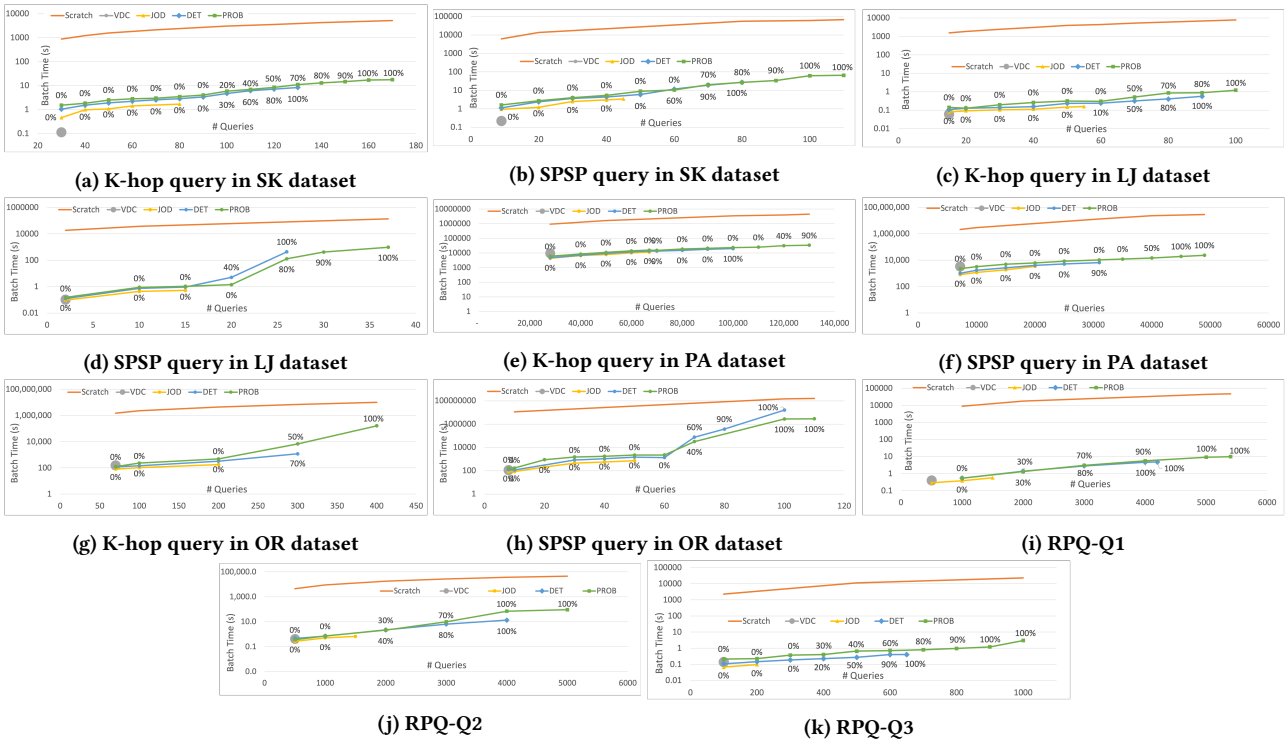
Finally, we performed a similar experiment for PR and WCC, for which we can only run one "query". We used LJ and picked a memory budget of 2.75GB for PR and 2GB for WCC, which requires less memory and picked the lowest drop probabilities at which these budgets were enough for Det-Drop and Prob-Drop. Figure 8 shows our results, with the necessary drop percentages presented on top of the bars. We find that on PR Det-Drop requires 100% dropping rate and takes 369 seconds to complete while Prob-Drop requires 90% dropping rate and takes 268 seconds to complete[4]. On WCC, Det-Drop requires 90% dropping rate and takes 11.9 seconds to complete while Prob-Drop requires 70% dropping rate and takes 11.5 seconds to complete. Overall, similar to our previous experiments, Prob-Drop needs to drop fewer differences to successfully complete the experiment and leads to better performance.

6.6 Further Applications of Diff-IFE

Our previous experiments so far focused on demonstrating the performance tradeoffs that our optimizations offer when evaluating continuous recursive queries using Diff-IFE. Our final set of experiments do not evaluate our optimizations. Instead, we aim to demonstrate further applications of Diff-IFE in systems. Specifically, we show that we can improve our Scratch baseline for SPSP queries through using and differentially maintaining a popular shortest path index, called *landmark indices* [12, 25]. A landmark index is a single-source shortest distance index, i.e., it stores the shortest path distance from a "landmark" vertex to the rest of the vertices. We use landmark indices to prune the search space of Scratch. Specifically, in the shortest path query from s to d, the sum of the distances of s to l and d to l give an upper bound ℓ_u on the shortest distance between s and d. Similarly, the difference between the v to l distance and d to l distance give a lower bound ℓ_b on the distance from u to d. If v is visited at distance k in the Bellman-Ford algorithm, and $k + \ell_b$ is greater than ℓ_u, then we can avoid traversing v as it cannot be on the shortest path from s to d.

We used all of our datasets, except LDBC, and picked the 10 highest-degree nodes as the landmarks and implemented an optimized version of Scratch in which as updates arrive at the graph, we first maintain these 10 landmark indices using Diff-IFE. Then, we run each registered query using our landmark-enhanced Scratch,

[4]Recall that 100% dropping rate does not mean all differences are dropped as we do not drop any differences for vertices over 13 degree in LiveJournal dataset.



(a) K-hop query in SK dataset (b) SPSP query in SK dataset (c) K-hop query in LJ dataset

(d) SPSP query in LJ dataset (e) K-hop query in PA dataset (f) SPSP query in PA dataset

(g) K-hop query in OR dataset (h) SPSP query in OR dataset (i) RPQ-Q1

(j) RPQ-Q2 (k) RPQ-Q3

Figure 7: Number of queries maintained by Scratch, DC, JOD, Det-Drop, and Prob-Drop under a limited memory budget of 10GB. The large dot in bottom left of each figure is DC.

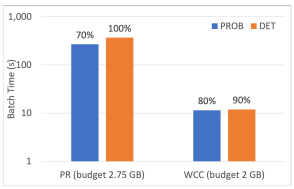


Figure 8: Comparison of Det-Drop and Prob-Drop when running PageRank and WCC on LJ under limited memory.

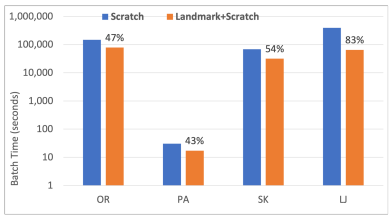


Figure 9: Comparing Scratch vs. Scratch-landmark on 100 queries and 100 batches of updates. Numbers on orange bars are the runtime improvements of Scratch-landmark.

which we call Scratch-landmark, and compare this to our baseline Scratch. We registered 100 random SPSP queries in our system and measured the end-to-end time of 100 batches of single edge insertions. Our results are shown in Figure 9. The reported times for Scratch-landmark include both the time to maintain the index and then (non-differentially) evaluate each query. As shown in the figure, by using and differentially maintaining landmark indices, we can reduce Scratch time between 43% to 83% (albeit now using additional memory to store both the index and the differences to differentially maintain the index).

7 CONCLUSIONS

Differential computation is a generic novel technique to maintain arbitrary recursive dataflow computations. As such, it is a promising technique to integrate into data management systems that aim to support continuous recursive queries. We studied the problem of how to increase scalability of differential computation through optimizations that are based on dropping differences.

An important future work venue is to design more advanced algorithms than vanilla IFE that can be maintained efficiently with differential computation. One example is to differentially maintain shortest path algorithms that use indices. In Section 6.6, we demonstrated how Diff-IFE can be used to maintain landmark indices, but this algorithm only enhanced our baseline Scratch algorithm with an index and does not evaluate queries differentially. It is less clear how to design a differential shortest path algorithm that uses an index that also needs to be updated as updates arrive at the system. One possibility is to develop two separate dataflows: (1) that maintains the indices; (2) that uses the updates to E and the updates to indices as base collections, and uses both the index and edges, e.g, by joining E, distances D, and the indices, to find shortest paths. No prior work we are aware of has proposed such algorithms and developing them is an important research topic.

REFERENCES

[1] 2022. REFINITIV Knowledge Graph. https://solutions.refinitiv.com/KnowledgeGraphs.

[2] Martín Abadi, Frank McSherry, and Gordon D Plotkin. 2015. Foundations of Differential Dataflow. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–83.

[3] Khaled Ammar, Siddhartha Sahu, Semih Salihoglu, and M. Tamer Özsu. 2022. Optimizing Differentially-Maintained Recursive Queries on Dynamic Graphs. https://doi.org/10.48550/ARXIV.2208.00273

[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the maze of Wikidata query logs. In *Proc. 28th Int. World Wide Web Conf.* 127–138.

[6] TimothyM. Chan. 2005. All-Pairs Shortest Paths with Real Weights in O(n 3/log n) Time. In *Algorithms and Data Structures*, Frank Dehne, Alejandro López-Ortiz, and Jörg-Rüdiger Sack (Eds.). Lecture Notes in Computer Science, Vol. 3608. Springer, 318–324. https://doi.org/10.1007/11534273_28

[7] Timothy M. Chan. 2012. All-pairs Shortest Paths for Unweighted Undirected Graphs in O(Mn) Time. *ACM Trans. Algorithms* 8, 4, Article 34 (Oct. 2012), 17 pages. https://doi.org/10.1145/2344422.2344424

[8] Camil Demetrescu and Giuseppe F Italiano. 2001. Fully dynamic all pairs shortest paths with real edge weights. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 260–267.

[9] Camil Demetrescu and Giuseppe F. Italiano. 2004. A New Approach to Dynamic All Pairs Shortest Paths. *J. ACM* 51, 6 (2004), 968–992. https://doi.org/10.1145/1039488.1039492

[10] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.

[11] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *ACM CoNEXT*. 75–88.

[12] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory.. In *SODA*, Vol. 5. Citeseer, 156–165.

[13] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. 2013. *Datalog and recursive query processing*. Foundations and Trends in Databases.

[14] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *NSDI*. 337–355. https://www.usenix.org/conference/nsdi21/presentation/iyer

[15] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM*.

[16] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD*.

[17] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. 2021. ïTurboGraph: Scaling and Automating Incremental Graph Analytics. In *SIGMOD*. 977–990.

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[19] Peter S Loubai. 1967. A network evaluation procedure. *Highway Research Record* 205 (1967).

[20] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference*. Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

[21] Frank McSherry. 2022. Differential Dataflow. https://github.com/frankmcsherry/differential-dataflow.

[22] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proc. 6th Biennial Conf. on Innovative Data Systems Research*.

[23] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proc. 24th ACM Symp. on Operating System Principles*. 439–455.

[24] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *SIGMOD*. 1415–1430. https://doi.org/10.1145/3318464.3389733

[25] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast Shortest Path Distance Estimation in Large Networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009)*. Association for Computing Machinery, New York, NY, USA, 867–876. https://doi.org/10.1145/1645953.1646063

[26] VV Rodionov. 1968. The parametric problem of shortest distances. *U. S. S. R. Comput. Math. and Math. Phys.* 8, 5 (1968), 336–343.

[27] Liam Roditty and Uri Zwick. 2011. On dynamic shortest paths problems. *Algorithmica* 61, 2 (2011), 389–401.

[28] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog.. In *Datalog*. 56–67.

[29] Siddhartha Sahu and Semih Salihoglu. 2021. Graphsurge: Graph Analytics on View Collections Using Differential Computation. In *SIGMOD*. 1518–1530. https://doi.org/10.1145/3448016.3452837

[30] Semih Salihoglu and Jennifer Widom. 2014. HelP: High-Level Primitives For Large-Scale Graph Processing. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*.

[31] Christian Stuecklberger. 2016. *Expressing the Routing Logic of a SDN Controller as a Differential Dataflow*. Master's thesis. ETH Zürich.

[32] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[33] Peng Zhang, Yuhao Huang, Aaron Gember-Jacobson, Wenbo Shi, Xu Liu, Hongkun Yang, and Zhiqiang Zuo. 2020. Incremental Network Configuration Verification. In *HotNets*. 81–87. https://doi.org/10.1145/3422604.3425936