Systems and Algorithms for Dynamic Graph Processing

by

Khaled Ammar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Vasiliki Kalavri Assistant Professor, Department of Computer Science, Boston University
Supervisor(s):	M. Tamer Özsu University Professor, David R. Cheriton School of Computer Science, University of Waterloo
	Semih Salihoglu Associate Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Members:	Grant Weddell Associate Professor, David R. Cheriton School of Computer Science, University of Waterloo
	Jimmy Lin Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Patrick Lam Associate Professor, Electrical and Computer Engineering, University of Waterloo

Author's Declaration

This thesis consists of material, all of which I authored or co-authored: see the Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based on the peer-reviewed joint work [9, 10] supervised by Prof. Semih Salihoglu and Prof. M. Tamer Özsu, in which I am the first author.

Abstract

Data generated from human and systems interactions could be naturally represented as graph data. Several emerging applications rely on graph data, such as the semantic web, social networks, bioinformatics, finance, and trading among others. These applications require graph querying capabilities which are often implemented in *graph database management systems* (GDBMS). Many GDBMSs have capabilities to evaluate one-time versions of recursive or subgraph queries over static graphs – graphs that do not change or a single snapshot of a changing graph. They generally do not support incrementally maintaining queries as graphs change. However, most applications that employ graphs are dynamic in nature resulting in graphs that change over time, also known as *dynamic graphs*.

This thesis investigates how to build a generic and scalable incremental computation solution that is oblivious to graph workloads. It focuses on two fundamental computations performed by many applications: recursive queries and subgraph queries. Specifically, for subgraph queries, this thesis presents the first approach that (i) performs joins with worst-case optimal computation and communication costs; and (ii) maintains a total memory footprint almost linear in the number of input edges. For recursive queries, this thesis studies optimizations for using *differential computation* (DC). DC is a general incremental computation that can maintain the output of a recursive dataflow computation upon changes. However, it requires a prohibitively large amount of memory because it maintains differences that track changes in queries input/output. The thesis proposes a suite of optimizations that are based on reducing the number of these differences and recomputing them when necessary. The techniques and optimizations in this thesis, for subgraph and recursive computations, represent a proposal for how to build a state-of-the-art generic and scalable GDBMS for dynamic graph data management.

Acknowledgements

All praise and thanks are to the one true God, Allah the Almighty, the most merciful and the most knowledgeable. Without Allah's help, this thesis could not be possible. Prophet Mohammed PBUH said, "He who does not thank people does not thank God"; the following is an attempt to thank all those who made this thesis possible and supported me during my long journey.

I am deeply grateful to my supervisor, Prof. Tamer Özsu, for his support during my Ph.D. and for teaching me many things on the personal, professional, and academic levels. When I proposed the idea of Dynamic Graphs, I did not know much about the topic, but I was interested to learn. Prof. Özsu supported me in this journey and shared his experience and knowledge with me until I could develop my own ideas and could even argue with him sometime. When a course project had the potential to become a funded startup, he supported me and asked me to choose freely if I wanted to exercise this option. When a full-time job opportunity showed up, he supported me and let me choose what was best for myself and my family. He taught me to be knowledgeable, diligent, innovative, humble, and kind to everyone around me. I am indebted to his support and honoured to be his student.

I want to thank my co-supervisor, Dr. Semih Salihoglou, whom I consider a friend and a colleague. Since he joined the University of Waterloo, I have learned many theoretical and engineering concepts from him. Semih introduced me to the theory behind WCOJ, which became a significant part of my thesis. We participated in several pair programming and whiteboard brainstorming sessions that benefited me. I feel privileged to have worked with Tamer and Semih, and I will always be grateful for their contributions to my academic and professional growth.

Thanks to my thesis committee members, prof. Jimmy Lin, Prof. Grant Weddell, Prof. Vasiliki Kalavri, and Prof. Patrick Lam, for their great feedback and comments. I am especially grateful for their time and effort in reviewing my work. Their insightful and constructive criticism has been instrumental in shaping my thesis.

I want to thank all faculty members at the Data Systems Group. They created an environment where students can learn, grow, and make lifetime friends. I appreciate many discussions with Khuzaima about distributed systems. Ken Salem was a great mentor when I participated in organizing the SIGMOD programming contest and was an amazing hiking partner at the Golden Gate national recreational area. Ihab Ilyas and Jimmy Lin have been great mentors with whom I like discussing new ideas and career choices. I also thank Charlie Clarck, Ashraf Abounaga, Lukas Golab, Grant Weddell, and David Toman for many interesting discussions.

Late Prof Kamel was a great support for me. He was a visionary leader in the field of AI and pattern recognition. He believed in my potential and provided me opportunities to succeed in this field. While working with his team, I had the first opportunity to build a real distributed cluster for machine learning applications.

I am deeply grateful to IBM for sponsoring the first few years of my Ph.D. through the IBM Ph.D. CAS Fellowship. During this fellowship, I worked at the IBM Center of Advanced Studies in Canada under Calisto Zuzarte's supervision to build a distributed version of DB2's RDF engine. I also had a great internship with Mauricio Hernandez and Rajasekar Krishnamurthy at the IBM Almaden research center in California. Thanks to all the friends I met during this time: Christina Christodoulakis, Eva Sitaridi, Nikos Katsipoulakis, Siddhartha Banerjee, Ricardo Santana Pineda, Geli Fei, Alan Akbik, Ashraf Bah, Jose Lugo-Martinez and many more.

Many friends made this journey more enjoyable. Mohamed Feteiha was a great help in finding housing before I came to Waterloo and hosted my family and me on our first night at Waterloo. I hope I do not forget anyone, but many thanks to Mohamed Sabri, Abdullah Rashwan, Abdullah el-Sayed, Mohamed Sadek, Hytham Atia, Yasser Atwa, Amine Mhedhbi, Anil Pacaci, Gunes Aluc, Iman Elghandour, Ahmed Farahat, Siddhartha Sahu, Mina Farid, Mustafa Korkmaz, Zeynep Korkmaz, Michael Joseph Mior, Muhammad Badrah, Hani Kashef, Ahmed Elroby.

I am grateful to my parents and siblings, Samia, Waleed and Walaa, for their unconditional love and support. Their encouragement and belief in me have been a constant source of motivation throughout this entire journey. They had always offered wisdom and support when I needed it most. Their guidance and advice have been invaluable, helping me navigate through life's challenges and obstacles with grace and resilience.

Finally, I want to express my appreciation to my wife, Doaa, and our kids, Hamza and Razan. They witnessed all the ups and downs of this journey and supported me the most. Doaa always offered her continuous love and encouragement even when life was tough. Her patience and understanding helped me to stay focused and motivated, even in the face of adversity. She has been my biggest cheerleader, and we have celebrated every milestone together. I am truly blessed to have such an amazing partner, and I could not have accomplished this without her.

Dedication

This thesis is dedicated to my parents, wife, and kids!

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large volumes of data generated from human interaction with software systems that support daily applications in areas such as commerce, entertainment and social networking have given rise to what is commonly referred to as the "Big Data Problem". Most of this data can be represented as graphs which represent the relationships between two entities. Applications that rely on graph data include the semantic web (i.e., RDF), bioinformatics, finance and trading, and social networks, among others. Graphs naturally model complicated structures, such as protein interaction networks, product purchasing, business transactions, relationships and interactions in social or computer networks, and web page connections. The size and complexity of these graphs raise significant data management and data analysis challenges.

Much of the existing literature focuses on static graphs – graphs that do not change or a single snapshot of a changing graph. However, most applications employ graphs that are dynamic in nature. These graphs change over time and are known as *dynamic graphs*. For example:

- Twitter users write 500 million tweets per day[1]. Every tweet updates graphs that model relationships between users, posts, interests, and locations, among others.
- Facebook has about 2.89 billion monthly active users in 2022[2]. Collectively, these users generate many terabytes of logs every day.

[1] https://www.internetlivestats.com/twitter-statistics/
[2] https://sproutsocial.com/insights/facebook-stats-for-marketers/

- The web graph has more than 190 million websites, and 250 thousand new websites are created every day[3].
- Business transaction graphs for several online or retail stores have billions of transactions per year.

Software systems that can manage large graphs, called *Graph Database Management Systems* (DGBMS), are designed to process a variety of workloads. Two of the important workloads tackled in this thesis are recursive navigation queries and subgraph queries. A recursive query initializes vertices with an initial value. Then each vertex collects information for its in-neighbour, computes a new value, then distributes this value to its out-neighbours. This process continues until a stopping condition is satisfied. An example is a single source shortest path query that computes the distance from a specific vertex in the graph to every other vertex and a single pair shortest path query which finds the minimum distance between two specific vertices in a graph. A subgraph query finds all instances of a specific subgraph pattern query in the input graph. An example is finding all triangles or all 4-clique instances in a graph.

Many GDBMSs have the capabilities to evaluate recursive and subgraph queries over static graphs. As noted earlier, many real-life graphs are dynamic, and in those cases, the systems resort to re-executing the query, which has unacceptable performance. This thesis addresses the following question: *How should a modern DGBMS query processor be designed to efficiently evaluate recursive and subgraph queries over dynamic graphs?*

The thesis makes two claims. First, the query processor should evaluate queries over dynamic graphs incrementally. Without incremental graph processing, a query processor would process a query from scratch after every change or at a slower frequency. Processing queries from scratch can be very slow, and it is unrealistic to keep the query answer up-to-date with graph changes in most applications. Second, the query processor should incorporate differential computation (DC) and incremental versions of worst-case optimal multiway join algorithms for effective incremental evaluation. DC is a new incremental maintenance technique that is generic and offers several orders of magnitude faster execution than running queries from scratch. On the other hand, worst-case optimal join algorithms can guarantee that the search space will never grow more than the maximum query size. For cyclic subgraph queries, this can increase the speed of a query processor by an order of magnitude and reduce its memory overhead by several orders of magnitude. The thesis addresses several technical problems arising from these claims, as discussed in the next section.

[3] https://siteefy.com/how-many-websites-are-there/

1.1 Graph and Query Model

In *property graphs*, vertices and edges can have attributes. Formally, a graph G is defined as a four-tuple $G = (V, E, P_V, P_E)$, 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. A graph query is *continuous* when the answer to this query needs to be updated frequently as the graph changes, which means that the query engine should *continuously* answer this query. Continuous queries compute properties of vertices, which are referred to as their *states*. States will not be modelled explicitly, but these can be considered temporary properties in P_V. For an edge e, two properties are maintained: $label(e)$, and $weight(e)$. If G is unweighted, the weights of each edge are set to 1.

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 deletion (updates appear as one deletion and one insertion). Vertex insertions or deletions are not considered because these could implicitly occur in the proposed 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).

Many graph queries in practice are join-heavy, so they could be expressed as relational algebra expressions that primarily use the join operator. Therefore, this thesis will adopt the relational view of graph queries when discussing optimizations. This thesis discusses two broad query types: recursive and subgraph queries.

1.1.1 Recursive Queries

Many graph queries 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`). Each one of these queries can interact with different parts of the graph model referenced in Section 1.1. The edge properties that a recursive query needs to access and the vertex states for this computation will be clear from the context.

Recursive queries are often supported in the query languages of GDBMSs [70]. However, existing GDBMS do not have the capabilities to maintain them incrementally. As such, in dynamic graphs, existing systems require rerunning these queries from scratch when updates occur. A GDBMS that can incrementally maintain recursive queries would lead to more accessible and efficient application development.

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.

1.1.2 Multiway join Queries

A subgraph query can be seen as a multiway join on replicas of an edge table of the input graph. Subgraph queries, i.e., finding instances of a given subgraph in a larger graph, are fundamental computations performed by many applications and supported by many software systems that process graphs. Example applications include finding triangles and larger clique-like structures for detecting related pages in the World Wide Web [29] and finding *diamonds* for recommendation algorithms in social networks [35]. In addition to GDBMSs [56, 78], multiway joins are supported by RDF engines [57, 89], as well as many other specialized graph processing systems [3, 49, 75].

The problem of incremental maintenance of a subgraph query Q is to report the newly created/deleted subgraphs after every batch of updates. These subgraphs can be thought of as output in the form of $([V], +/-)$, for a sorted list of vertices V that conform to the subgraph query Q and +/- indicating addition or removal of a subgraph.

1.2 Thesis Scope

This thesis proposes optimizations for algorithms and systems that maintain subgraph and recursive graph queries over dynamic graphs. As mentioned above, the computations to evaluate these queries can be modeled as executing multiway or recursive join-heavy queries. The algorithm/system, in this context, receives the complete input graph before it starts executing, and the graph sees updates to its structure over time. Dynamic algorithms continuously and incrementally update the answer as the input dataset changes rather than computing it from scratch after every change. In this thesis, the input graph is represented by an *edge* and a *vertex* relation; the graph query is modelled as a join query over these relations. The fundamental challenge is incrementally executing these join queries in a graph database.

Several proposed algorithms and system optimizations for dynamic graph processing are proposed in this thesis. They are described briefly below and more fully in the individual chapters dedicated to each topic.

1.2.1 Differential Computation for Recursive Queries

Many popular traversal workloads are recursive queries, such as `SPSP`, `SSSP`, and `RPQ`. In recursive queries, each vertex iteratively aggregates its neighbours' values, computes its own (new) value, then propagates the new value to neighbours until a stopping condition, such as a fixed point, is reached. These queries are commonly solved by a subroutine that consists of the `Join` operator and an aggregation operator, e.g., a `Min` operator, and has been given different terms in literature, such as *propagateAndAggregate* [70] or *iterative matrix-vector multiplication* [40]. This subroutine is referred to as *iterative frontier expansion* (IFE) in this thesis and will be reviewed in Section 2.1.1.

Maintaining IFE over dynamic graphs is similar to maintaining materialized views in relational database systems. There is a vast body of work on incrementally maintaining views that contain selection, projection, and joins [64]. Traditional *incremental view maintenance* (`IVM`) techniques for recursive SQL and Datalog queries have focused on variants of incremental maintenance approaches [33] such as Delete-and-Rederive [34], which consists of a set of delta-rules that can produce the changes in the outputs of queries upon changes to the base relations. These rules can be highly inefficient as they first delete all derivations of updated/removed facts and then rederive them using the updated facts, only to finally detect whether any deletions and/or additions affect the final query result.

The approach taken in this thesis is to use dataflows that contain recursive join operators, and maintain these dataflows using Differential Computation (DC) [51], a new incremental maintenance technique, to maintain the results of recursive queries in GDBMSs. DC is designed to maintain arbitrarily recursive dataflow programs [51, 55]. 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 in the inputs and outputs of the operators across different iterations, DC maintains and propagates the changes between operators as the original inputs to the data flow 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 [44], as it may need to monitor a high number of input and output differences between operators. For example, Table 1.1 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 [46]. This experiment modifies the graph with 100 batches of 1 random edge insertion each and provides the system with 10GB memory to store the generated differences. The table also shows the performance of a baseline that re-executes the Bellman-Ford algorithm from scratch after each update, thus not requiring any memory for

Table 1.1: Execution time (in seconds) for an SPSP workload, on Skitter dataset, using a scratch algorithm, which re-executes a standard non-incremental Bellman-Ford algorithm, vs. a DC version, which keeps track of changes. DC is more than five orders of magnitude faster but fails with out-of-memory when the number of queries increases.

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

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. Several optimizations have been proposed in Chapter 2 to increase DC's scalability measured by the number of maintained recursive queries up to 20×.

1.2.2 Worst-Case Optimal Joins for Subgraph Queries

`Join` is a fundamental operator that appears in every subgraph query at Table 1.2. Computing joins is considered *worst-case optimal* for a query Q if its computation and memory cost is not asymptotically larger than the maximum possible output size for the given size of the relations in Q; this quantity is called AGM bound [11] and referred to as $MaxOut_Q$ in this thesis. Traversal queries could be computed optimally using the typical *Binary Join* [84] (BJ) algorithm (also known as edge-at-a-time computation) that performs a series of pairwise joins. BJ has been adopted in most existing relational database systems. However, BJ is not optimal for subgraph queries like finding all occurrences of triangles (`Triangle` query). For example, BJ will do $O(N^2)$ computations in the worst-case, to compute `open-tri`, which is worse than the $MaxOut_Q$ quantity of $N^{3/2}$ for a `Triangle` query:

$$\texttt{open-tri}(x,y,z) := R(x,y) \circ S(y,z)$$
$$\texttt{Triangle}(x,y,z) := \texttt{open-tri}(x,y,z) \circ T(x,z)$$

The approach proposed in this thesis for multiway join queries, such as subgraph workloads, is to design algorithms inspired by a family of worst-case join algorithms called *Generic Join* (GJ) [58, 59, 80]. Specifically, a new algorithm called DELTA-GJ is proposed by this thesis. DELTA-GJ is based on the incremental view maintenance (`IVM`) techniques

Table 1.2: Query workloads represented with join operations ($\circ$). Note that R, S, T, U are replicas of the edge table.

Workload	Syntax
Triangle	$\texttt{Triangle}(x, y, z) := R(x, y) \circ S(y, z) \circ T(x, z)$.
Diamond	$\texttt{Diamond}(x, y, z, w) := R(x, y) \circ S(y, z) \circ T(x, w) \circ U(w, z)$.
SSSP	$\texttt{SSSP}(x;\ y : int) := R(\text{start}, x);\ y = 1$. $\texttt{SSSP}(x;\ y : int)^* := S(w, x) \circ \texttt{SSSP}(w);\ y = \langle MIN(w) \rangle + 1$.

[14, 34], which derive a set of delta queries $dQ1, ..., dQn$ (originally referred to as *delta rules* [34]) for Q and evaluate each dQi as the input tables change to maintain the result of Q.

Note that although the idea of delta queries is not new, the evaluation of dQi using a worst-case join algorithm (GJ) is novel and has theoretical implications that do not exist for existing incremental view maintenance algorithms [14, 34]. Specifically, Chapter 3 presents proof that under insertion-only workloads, if the delta queries are evaluated using the worst-case optimal GJ algorithm with specific attribute orderings, the total computation done to maintain the original join query incrementally is worst-case optimal up to constants that depend on Q.

Existing literature has focused on implementations and optimization of GJ in the single node setting [59]. The challenges of developing distributed versions of GJ and of Delta-GJ are much less understood. To fill this gap, Chapter 3 focuses on the distributed implementation of GJ and Delta-GJ. This contrasts with the treatment of DC in Chapter 2. Note that while DC was originally developed assuming a distributed setting, the implementations and optimizations in Chapter 2 are based on a single-node implementation. However, many of these optimizations directly apply to the distributed implementations of DC.

Chapter 3 proposes the BigJoin and Delta-BigJoin algorithms to process multiway join in static and dynamic in distributed settings. Together, they can find complex subgraphs very efficiently on dynamic graphs with up to 64B edges on a cluster of 16 machines with minimum memory overhead and linear performance scalability.

1.3 Thesis Contribution

The contributions of this thesis are as follows. First, the thesis proposes two distributed multiway join algorithms for subgraph queries. The following algorithms are implemented on Timely Dataflow system:

- BigJoin algorithm expands GJ to run in distributed share-nothing environments and expands the definition of worst-case optimal to include network communication. Using batching and pipelining, the proposed algorithm can balance the workload to all machines in the cluster. Section 3.6.2 shows that the costs per worker decrease linearly as additional workers are introduced.

- Delta-BigJoin, which uses BigJoin instead of GJ, is the first algorithm that (i) performs worst-case optimal computation and communication, (ii) maintains a total memory footprint linear in the number of input edges, and (iii) reduces per-worker computation, communication, and memory requirements linearly as the number of workers increases. Note that it can also be easily used in both existing distributed bulk synchronous parallel systems, such as MapReduce [21] and Spark [86], as well as streaming systems, such as Storm [79] and Apache Flink [17].

Second, this thesis proposes several optimizations to increase DC's scalability, and reduce its memory overhead by dropping some of the maintained differences. These algorithms are:

- Complete Difference Dropping (CDD) optimization increases the efficiency of a typical differential computing system to maintain 7x more queries. CDD has two main components: Join-on-demand (JOD), which avoids materializing the output of the `Join` operator and early merging which is an implementation optimization. JOD leads to fundamental changes in how differential computation should be processed. Theorem 2.2.1 and its proof shows that the new algorithm is correct.

- Partial Difference Dropping (PDD) optimization increases the efficiency of a typical differential computation up to 20X. This optimization has a knob that allows developers to balance the trade-off of reducing a query's memory overhead at the expense of its performance. It uses the properties of real graphs to decide which differences to drop and uses a deterministic/probabilistic data structure to store which difference has been dropped.

1.4 Thesis Structure

The rest of the thesis is structured as follows:

- Chapter 2 proposes algorithms and data structures that reduce the overhead of DC during the execution of traversal, and more generally recursive, queries.

- Chapter 3 proposes an algorithm (Delta-GJ) that extends the GJ algorithm to process dynamic graphs with worst-case guarantees. It also expands the worst-case definition in GJ to include network communication, then introduces BigJoin. After that, it introduces a new distributed version of Delta-GJ (called Delta-BigJoin).
- Chapter 4 concludes the thesis, explains its limitations and describes possible future work.

Chapter 2

Optimizing Recursive Query Execution[1]

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 GDBMS. Many of these applications require maintaining query results incrementally, as the graphs stored in GDBMSs are dynamic and evolve over time. 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. This chapter investigates the use of *differential computation* (DC) [51] to maintain the results of recursive queries in GDBMSs. DC is a general incremental computation approach for arbitrary recursive dataflow programs [51, 55]. Section 2.1.2 presents a technical overview of DC.

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 in 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, this generality comes with a significant memory overhead that limits its ability to scale in maintaining multiple queries. For example, as shown in Table 1.1, the differential version of the algorithm is about five orders of magnitude faster but it cannot maintain more than 10 concurrent

[1] This work has been published as [10].

queries due to its large memory requirement.

This chapter studies techniques for reducing the memory overhead of DC to increase its scalability when maintaining the popular classes of recursive queries mentioned above. These optimizations are broadly based on *dropping differences*, i.e., avoiding explicitly keeping track of all changes to inputs of operators, and instead recomputing some of them from scratch when necessary. The focus is on optimizing the differential version of a common subroutine that is called Iterative Frontier Expansion (IFE). In IFE, each vertex iteratively aggregates its neighbours' values, computes its own value, then propagates it to neighbours until a stopping condition, such as a fixed point, is reached. Section 2.1.1 discusses IFE and shows that this subroutine consists of a `Join` operator and an aggregation operator, e.g, a `Min` operator.

Two optimization categories have been proposed to reduce the memory overhead of DC while computing queries represented as IFE: First, Join-On-Demand (JOD) (Section 2.2) 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 second is *partial difference dropping* optimization (Section 2.3) that drops some of the differences in the output of the aggregation operator in IFE.

Partial difference dropping optimization offers developers a knob to drop a certain percentage of the system's differences. To re-compute these differences when necessary, the system needs to know for which key/vertex and timestamp a difference was dropped. There are two approaches proposed to achieve this: (1) a deterministic approach (Det-Drop) that explicitly keeps track of the vertex and timestamp of each dropped difference; and (2) a probabilistic approach (Prob-Drop) that addresses this shortcoming by leveraging a probabilistic data structure, specifically a Bloom filter. Det-Drop reduces the memory consumption of a system, but it also has inherent limitations in terms of scalability improvements, as the additional state it keeps is proportional to the number of differences it drops. 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 achieve the same amount of memory as Det-Drop. Finally, there is an optimization that uses the degree information of each vertex to choose which differences to drop as opposed to dropping them randomly.

The rest of this chapter is structured as follows. Section 2.1.1 introduces Iterative Frontier Expansion (IFE), and Section 2.1.2 reviews the technical details of *differential computation* (DC). Then, Section 2.2 and Section 2.3 explain the complete dropping and partial dropping optimizations. Section 2.4 reviews the implementation details. Finally, a suite of experiments are conducted to study the effectiveness and trade-offs of all opti-

mizations in Section 2.5. These demonstrate the following:

- JOD reduces the number of differences up to 8.2× in comparison to vanilla DC implementations.
- 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.
- Prob-Drop achieves up to 1.5× scalability relative to Det-Drop when selecting the differences to drop based on degrees.
- This suite of optimizations 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.

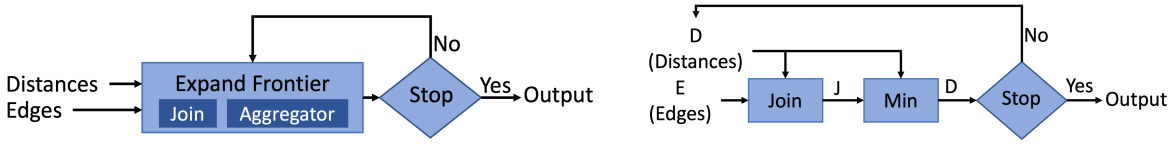
2.1 Related Work

This section starts by introducing Iterative Frontier Expansion (IFE) which represents the standard subroutine in the thesis for implementing recursive graph queries. Then, it describes Differential Computation (DC) which is a general technique to maintain the outputs of dataflow programs.

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

2.1.1 Iterative Frontier Expansion (IFE)

Iterative Frontier Expansion (IFE) is a standard subroutine for implementing many graph algorithms used in various computational problems, including graph traversal queries like `SPSP`, `SSSP`, `RPQ`, which are often supported in the query languages of GDBMSs [70]. 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



(a) High-level IFE dataflow.

(b) Specific IFE dataflow for SSSP

Figure 2.1: Template IFE dataflow and a specific Bellman-Ford algorithm's dataflow implementation.

neighbours. Those vertices that receive data from their neighbours form the frontier for the next iteration. These iterations continue until some stopping criterion is met, e.g., a fixed point is reached and the vertex states converge. Figure 2.1a shows the template IFE dataflow that consists of two operators, `ExpandFrontier`, which expands the frontiers, and `Stop`, which determines when to stop the query execution.

This chapter uses and optimizes variants of this basic IFE dataflow to evaluate recursive queries. As an example, Figure 2.1b shows a specific instance of the IFE dataflow implementing the standard Belman-Ford algorithm for evaluating an `SSSP` query where vertex states are the latest distances from a source vertex s. The `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, `RPQ`s, and variable-length join queries, this IFE template dataflow is always used with the same `Join` operator, but possibly different aggregator implementations to compute new vertex states and different stop conditions (e.g., variable-length K-hop queries stop the dataflow after k many iterations and not when a fixed point is reached).

2.1.2 Differential Computation (DC)

Differential Computation (DC) [51] is a general technique to maintain the outputs of arbitrarily nested dataflow programs as the base input collections change. Dataflow programs consist of operators, such as `Join` or `Min` in Figure 2.1b, 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. Data collections, such as `E`, that are input to the dataflow are referred to as *base*

collections, and other collections that are outputs of an operator (possibly the final output of the dataflow) are referred to as *intermediate collections*.

Consider the IFE instance from Figure 2.1b implementing the Bellman-Ford algorithm and running this IFE on the input graph shown in Figure 2.2 from a source vertex a. 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*, later on referred to 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 adding, deleting, or updating 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 `Edges` in IFE, $+$ and $-$ indicate insertions or deletions to the base data collections. For intermediate data collections that are generated by operators, these may not have as clear an interpretation, at least for dataflows with multi dimensional timestamps. 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. 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:

$$I_t = \sum_{s \leq t} \delta I_s \;\Rightarrow\; \delta I_t = I_t - \sum_{s < t} \delta I_s \tag{2.1}$$

$$O_t = Op(\sum_{s \leq t} \delta I_s) \;\Rightarrow\; \delta O_t = Op(\sum_{s \leq t} \delta I_s) - \sum_{s < t} \delta O_s \tag{2.2}$$

DC uses Equations 2.1 and 2.2 to compute which 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 (specifically during its maintenance procedure, which will be explained momentarily).

Suppose a system has maintained the Bellman-Ford dataflow differentially for k many

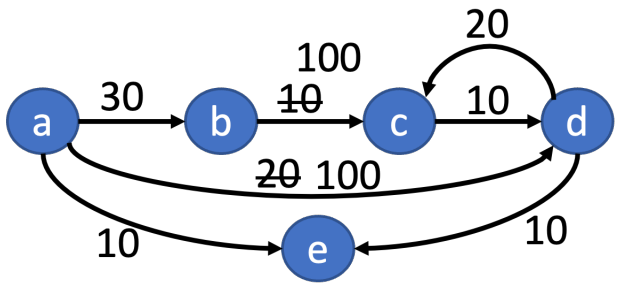


Figure 2.2: A dynamic graph with two updates: (i) $a \rightarrow d$ from 20 to 100 in G_1; and (ii) $b \rightarrow c$ changes from 10 to 100 in G_2.

updates to its base collection `Edges`; 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 $G_0, ..., G_k$. 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 $t = \langle G_{k+1}, i\rangle \mid i \in \{0...max\}$ 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 $\langle G_{k+1}, i\rangle \mid i > max$.

DC's maintenance procedure is as follows. Suppose that the operators work on partitions of collections, as in many dataflow systems. In the above 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:

1. *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 first reruns Op (on key v) at timestamp τ. That is DC first reassembles $I_\tau^v = \sum_{t \leq \tau} \delta I_t^v$. Then executes Op on I_τ^v, which computes a new O_τ^v. Finally, it computes the difference set δO_τ^v as $\delta O_\tau^v = O_\tau^v - \sum_{t<\tau} \delta O_t^v$.

2. *Upper bound rule:* For correctness, Op may need to be executed on later timestamps than τ for v, too even if there are no immediate differences in I at those timestamps. Specifically, DC finds every timestamp $t_f \mid t_f \geq \tau$ in which Op's input has differences for key v and reruns Op on all timestamps that are more than, upper bound for, t_f and τ.

Abadi et al. [1] formally prove that applying this simple rule to decide which operators to rerun correctly maintains any dataflow computation, establishing that DC is a very generic incremental computation maintenance technique.

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 2.1 shows the full difference trace for each collection in the IFE dataflow implementing Bellman-Ford algorithm in the example dynamic graph in Figure 2.2 that has two updates.

Each raw in Table 2.1 represents IFE iteration, while columns represent graph versions. Initially, at graph version G_0 and iteration 0, collection δE adds differences that represent all edges in the graph and collection δD adds a difference to represent the initial source vertex, a, with distance 0 and distance ∞ for all remaining vertices. As iterations progress and graph version change, more differences are added/removed as described in Equation 2.1 and 2.2.

2.1.3 Generic Techniques and Systems for Computations on Dynamic Graphs

When an input graph is modelled as a set of relations and a graph algorithm is modelled as a query over these relations, maintaining graph computation can be modelled 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 [33] 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. These rules can be highly inefficient as they first delete all derivations of updated/removed facts and then re-drive them again using the updated facts, only to finally detect whether any deletions and/or additions affect the final result. This contrasts with DC because it does not store intermediate computations to speed up processing. Interestingly, the only available incremental open-source Datalog implementation does not use the Delete-Rederive maintenance algorithm but uses DC [68]. This work compiles Datalog programs into DD programs, so it ultimately uses vanilla DD, which this thesis optimizes and uses as a baseline.

Tegra [38] is a system developed on top of Apache Spark [86], that is designed to perform ad-hoc window-based analytics on a dynamic graph. Tegra allows the creation of

Graph Updates →

IFE iterations ↓

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

Table 2.1: Full trace of differences in the SPSP example from Figure 2.2

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. A performance comparison [38] of Tegra versus DD reports Tegra's performance to be significantly slower than DD for incrementally maintained streaming computations.

There have been several systems work that uses the generic incremental maintenance capabilities of DC. GraphSurge [69] 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 share computation when running across multiple views automatically. Stuecklberger [72] implement a DC-based software-defined network controller, which represents the routing logic of a network as a dataflow graph. DD allows the system to incrementally update the routing logic as the underlying physical layer changes. Similarly, RealConfig [88] is a *network configuration verifier* that is used to detect if changes to a network configuration could lead to a potential network outage. RealConfig uses DD to incrementally verify updates to a network configuration without having to restart from scratch after every change.

2.1.4 Specialized Techniques and Systems for Computations on Dynamic Graphs

There is extensive literature dating back to the 1960s on developing specialized incremental versions of (aka *dynamic*) graph algorithms that maintain their outputs as input graph changes. Many of the earlier work focuses on versions of shortest path algorithms, in particular all pairs shortest paths computation [22, 66, 23, 19, 18, 65]. 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. Fan et al. [28] present theoretical results on the foundations of such algorithms. Specifically, they show that the cost of performing six specific incremental graph computations, such as regular path queries and strongly connected components algorithms, cannot be bounded by only the size of the changes in the input and output. Then, they develop algorithms that have bounded guarantees in terms of the work performed to maintain the computation.

On the systems side, there are several graph analytics systems that enable users to develop incremental versions of a graph algorithm. GraphBolt [50] is a shared-memory parallel streaming system that can maintain dynamic versions of graph algorithms. Graph-

Bolt requires users to write explicit maintenance code in functions such as `retract` or `propagateDelta` that generic systems such as DD do not require. As graph updates arrive, the system executes these functions, and if a user has provided a dynamic algorithm with provable convergence guarantees, the system will correctly maintain the results.

Another system is iTurboGraph [44] which 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. iTurboGraph keeps graph data on disk as streams and models the graph traversal as an enumeration of walks to avoid maintaining large intermediate results in memory. They avoid expensive random disk access by adopting the nested graph windows approach [43]. Instead, our proposed solutions keep the intermediate results in memory and drop these differences to reduce memory overhead.

Broadly, programming specialized algorithms or GraphBolt-like specialized systems can be more challenging for users than programming a generic solution such as DD as users need to design and write dynamic versions of algorithms. At the same time, as is expected, such systems can be more efficient than generic solutions. For example, several references have demonstrated this difference between DD and GraphBolt [50, 69]. In contrast, generic solutions such as DD, which is used as an example of the standard DC implementation, are fundamentally different. They have the advantage that users can program arbitrary static versions of their algorithms, and the system can automatically maintain them. Therefore these systems are suitable as core incremental view maintenance techniques to integrate into general data management systems, such as GDBMSs in our context, that aim to support large classes of queries.

2.2 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 the output of the `Join` operator (`J`) would include a possible new shortest distances to its outgoing neighbours. It 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 $\texttt{J}^u$ 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 2.1, 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. This section starts by describing a naive version of JOD, then describes a useful implementation optimization called *eager merging* that reduces the timestamps to regenerate J^u, which is the optimized JOD used in the rest of this chapter.

2.2.1 Naive 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)
2. t is an upper bound of τ_1 and τ_2 that satisfy the following conditions (upper bound rule):
   (a) $\tau_1 \in \mathrm{T}_1 = \{\langle G_{k+1}, i'\rangle | i' < i\}$ and $\delta D^u_{\tau_1}$ and/or $\delta J^u_{\tau_1}$ are non-empty; and
   (b) $\tau_2 \in \mathrm{T}_2 = \{\langle G_{k'}, i\rangle | k' < k+1\}$ and $\delta D^u_{\tau_2}$ and/or $\delta J^u_{\tau_2}$ are non-empty.

If δJ are dropped, how can DC correctly decides when to rerun `Min` and to recompute the needed dropped δJ for these reruns to ensure it correctly differentially maintains IFE? $\mathrm{DC}^{\mathrm{JOD}}$ is the modified version of DC maintenance subroutine that has this guarantee, which works as follows. In the below description, when `Min` reruns 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 $\mathrm{DC}^{\mathrm{JOD}}$ does not drop the differences related to `D` and `E`.

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

- *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 `Min` reruns on v in $\langle G_{k+1}, 0\rangle$ (direct rule).*

- δD Direct Rule: Each time `Min` reruns on u at a timestamp $\langle G_{k+1}, i\rangle$, DC$^{\text{JOD}}$ checks if this run 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 DC$^{\text{JOD}}$ schedules to rerun `Min` on v at timestamp $\langle G_{k+1}, i+1\rangle$ (direct rule).

- *Upper Bound Rule:* Each time DC$^{\text{JOD}}$ schedules 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, DC$^{\text{JOD}}$ schedules 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., $h < k+1$.

The next theorem shows that DC$^{\text{JOD}}$ correctly maintains the IFE dataflow. Starting from $\langle G_{k+1}, 0\rangle$ to $\langle G_{k+1}, max\rangle$, it shows 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`.

Theorem 2.2.1. *The subset of timestamps that DC^{JOD} re-computes `Min` on any key/vertex ID subsumes the timestamps that DC re-computes `Min` and correctly generates the same set of differences for `D`.*

Proof. Assume for simplicity that there is a global max iteration on which the IFE computations run. By induction on timestamps t, and taking the base cases $\langle G_0, 0\rangle$ to $\langle G_0, max\rangle$, the behaviour of DC$^{\text{JOD}}$ simply follows DC. This directly follows the computation that is performed when running a static version of IFE on an input graph. In this case, the behaviour is that `Min` runs on all vertices in $\langle G_0, 0\rangle$ and then for each vertex v at $\langle G_0, 0\rangle$ if one of its incoming neighbour's states change.

As induction hypothesis, assume that for each vertex v, DC$^{\text{JOD}}$ runs `Min` on a larger subset of timestamps and correctly generates the same set of differences for the data collection `D` (and `E`, whose maintenance is independent of the JOD optimization) until $\langle G_k, max\rangle$. This proof shows that for each key v, if DC runs `Min` in timestamp $\langle G_{k+1}, i\rangle$, so does DC$^{\text{JOD}}$. Note that for $t_0 = \langle G_{k+1}, 0\rangle$, this is true because if v is re-computed by DC in t_0 it is because there is change in $\delta J^v_{t_0}$, which can only occur if there is an edge (u, v) in δE_{t_0}, i.e., one of v's incoming edges must have had an edge update that triggered DC to rerun `Join` which must trigger a difference in $\delta J^v_{t_0}$. The first step of DC$^{\text{JOD}}$ ensures that for each edge (u, v) in δE_{t_0}, `Min` is re-computed for v in t_0. This is used as a base case, then, by induction only on the second component of the timestamps in $\langle G_{k+1}, i\rangle$. This assumes that from $\langle G_{k+1}, 0\rangle$ to $\langle G_{k+1}, i\rangle$, the theorem is true and proven for $\langle G_{k+1}, i+1\rangle$.

Now consider $t_{i+1} = \langle G_{k+1}, i+1 \rangle$ to prove the contraposition of the claim: if DC$^{\text{JOD}}$ does not re-compute `Min` on u, then neither will DC. Note that DC$^{\text{JOD}}$ does not run on u if two conditions are satisfied: (1) none of u's incoming neighbours v had a non-empty $\delta D^u_{\langle G_{k+1}, i \rangle}$. If there was then it schedules u to re-compute, which means DC cannot execute `Min` on v as an application of the direct rule. (2) none of u's incoming neighbours w_1, ..., w_2 has a non-empty $\delta D^w_{\langle G_h, j \rangle}$ s.t., $h < k+1$. Next, the proof shows that if this condition is true, then all such $\delta J^u_{\langle G_h, j \rangle}$ are non-empty, so DC cannot have triggered the upper bound rule either.

Finally, to prove this, a third induction starting from $\delta J^u_{\langle G_0, j \rangle}$ to $\delta J^u_{\langle G_k, j \rangle}$ is used. For the base case of $\delta J^u_{\langle G_0, j \rangle}$, observe that:

$$\delta J^u_{\langle G_0, j \rangle} = J^u_{\langle G_0, j \rangle} - J^u_{\langle G_0, j-1 \rangle} = \emptyset$$

This is true because this is a 1 dimensional case and $J^u_{\langle G_0, j \rangle}$ and $J^u_{\langle G_0, j-1 \rangle}$ are computed buy the `Join` operator using the same of incoming neighbour states (recall that our assumption is that all $\delta D^w_{\langle G_0, j \rangle}$ are empty). Suppose now that $\delta J^u_{\langle G_0, j \rangle}$ up to $\delta J^u_{\langle G_h, j \rangle}$ are empty. The proof shows that $\delta J^u_{\langle G_{h+1}, j \rangle}$ is empty:

$$\delta J^u_{\langle G_{h+1}, j \rangle} = J^u_{\langle G_{h+1}, j \rangle} - J^u_{\langle G_{h+1}, j-1 \rangle} - \sum_{\ell=0,\ldots,h} \delta J^u_{\langle G_\ell, j \rangle}$$

By induction hypothesis the last summation is non-empty. Note further that $J^u_{\langle G_{h+1}, j \rangle}$ and $J^u_{\langle G_{h+1}, j-1 \rangle}$ are the same set because they are computed by the `Join` operator using the same of incoming neighbour states for u, since it is assumed that all $\delta D^w_{\langle G_h, j \rangle}$ are empty for each incoming neighbour w of u, completing the proof. □

Note that there can be timestamps in which DC$^{\text{JOD}}$ unnecessarily re-compute `Min`, but by the correctness argument of DC in Abadi et al. [1], any timestamp that DC avoids rerunning a computation is guaranteed to produce empty differences, so these spurious re-computations cannot affect the correctness of DC$^{\text{JOD}}$, so as a corollary of Theorem 2.2.1, DC$^{\text{JOD}}$ correctly maintains the IFE dataflow. A simple example of spurious rerun is the simple case in the SPSP example is when a vertex u has two incoming edges, say from w_1 to w_2. Suppose for purpose of demonstration w_1 and w_2 start with states 0 initially (so at a timestamp $\langle G_0, 0 \rangle$), and suppose further that edges (w_1, u) has a weight of 10 and (w_2, u) has a weight of 2. In this case, the $J^u_{\langle G_0, 1 \rangle}$ would contain $(u, 10, +)$ and $(u, 20, +)$. Suppose in G_1, the weights of these edges are swapped. The original DC would not re-compute `Min` at timestamp $\langle G_1, 1 \rangle$ on u, because there is no difference directly to u's input (nor through

an upper bound rule), where as DC$^{\text{JOD}}$ would. This is because DC$^{\text{JOD}}$ would immediately schedule `Min` to execute on u because w_1 (or w_2's) states changed at $\langle G_1, 0\rangle$ and u is an outgoing neighbour of w_1.

Example 2. *This example demonstrates an application of* JOD*'s rerunning rules on this chapter's running example. Consider the first update in the running example at timestamp* $\langle G_1, 0\rangle$*, which updates the weight of edge (a, d) from 20 to 100. By the Direct Rule of* JOD*, DC^{JOD} reruns* `Min` *on d at timestamp* $\langle G_1, 0\rangle$*. Further by JOD's Upper Bound Rule, DC^{JOD} 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, DC^{JOD} further schedules to rerun* `Min` *on c and e, which are the outgoing neighbours of d, at timestamp* $\langle G_1, 1\rangle$*.*

2.2.2 Eager-Merging

The naive implementation of DC$^{\text{JOD}}$ 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 extends a periodic merging optimization of the DD system (explained momentarily), which 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 computation for G_k. So there are $k \times max$ many different timestamps in the computation so far. It is possible to think of these timestamps in a 2D grid with columns as graph version indices and rows as IFE iterations as in Table 2.1.

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.

Graph Updates →

IEF 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		$+(d,50)$	

Table 2.2: Differences in D on our running example **with eager-merging** when maintaining the computation for $\langle G_2, 2\rangle$.

Instead of periodic merging, DC$^{\text{JOD}}$ eagerly merges the differences along the graph-version dimension as DC$^{\text{JOD}}$ runs DC's maintenance procedure for $\langle G_{k+1}, 0\rangle$ to $\langle G_{k+1}, max\rangle$. That is, as soon as DC$^{\text{JOD}}$ finishes maintaining $\langle G_{k+1}, i\rangle$, DC$^{\text{JOD}}$ merges 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, DC$^{\text{JOD}}$ only needs to keep one-dimensional timestamps, i.e., only for *IFE iteration*. Table 2.2 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 DC$^{\text{JOD}}$ needs to investigate if $\delta D_{\langle G_h, i\rangle} \mid h < k+1$ is non-empty for any vertex, DC$^{\text{JOD}}$ needs to inspect timestamps with $h = k$.

There is one more benefit of eager merging. Eager merging allows dropping all differences with negative multiplicities in the difference sets for D. This is because in the considered algorithms, 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. For example, once DC^{JOD} with eager merging finishes maintaining the computation for all timestamps for graph version G_2, the distances stored for vertex d will be $\{(1,100,+),(3,100,-),(3,50,+)\}$, where the first values in these tuples are the timestamps, which are represented only with an IFEiteration number. These differences can be stored as $\{(1,100),(3,50)\}$, and the $(3,100,-)$ would be implied. In absence of negative multiplicities, it is possible to avoid doing any summations when computing the state of a vertex at timestamp i, i.e., D_i^v. Instead it is possible to find the latest iteration $i^* \leq i$ in which vertex v has a (positive) difference and return it.

2.3 Partial Difference Dropping (PDD)

This section investigates optimizations that partially drop the distance differences in collection D at an IFE subroutine. When applying the JOD optimization, D is the only data collection for which DC^{JOD} stores 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. This section describes partial dropping optimizations with different scalability/performance trade offs. Note that throughout the rest of this chapter, it is assumed that $\text{DC}^{JOD+PDD}$ runs JOD optimization with eager merging and uses a single dimensional timestamp to refer to data collections, such as D_i, instead of $D_{\langle G_k,i\rangle}$. It also uses PDD optimization to reduce the size of collection D.

A partial dropping optimization has two key components:

- *Dropped Difference Maintenance*: When $\text{DC}^{JOD+PDD}$ 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.

This section describes 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 unregistered 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 topic is discussed as part of future work. Based on the context of this chapter, there is a user-define probability p that drops each difference with probability p as discussed in Section 2.3.2.

2.3.1 Dropped Difference Maintenance

One natural approach to maintaining the dropped VertexID/timestamp pairs (VT pairs for short) is to store them explicitly in a separate data structure called `DroppedVT`. There are two possible designs for this data structure. A straightforward deterministic data structure is a hash table. This section starts by discussing its scalability bottlenecks. Then, it proposes a probabilistic data structure, which can address this scalability bottleneck but possibly leading to wasted re-computations of differences that do not exist. The evaluation section shows that, despite this possible performance disadvantage, the probabilistic approach can still perform better than the deterministic one because probabilistic can drop fewer differences than deterministic approach under limited memory settings.

Deterministic Difference Maintenance (Det-Drop)

Det-Drop uses a hash table to implement `DroppedVT`. During $DC^{JOD+PDD}$, when D_i^v is needed, it performs the following $\texttt{AccessD}_i^v\texttt{WithDrops}$ procedure. Before describing the procedure, recall from Section 2.2.2 that DC^{JOD} does not store differences with negative multiplicities for D when differences are merged eagerly, so it does not need to do any summation to compute D_i^v. DC^{JOD} only needs to find and returns 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. δD is checked for the latest iteration $g^* \leq i$, if any, for which the system has stored a difference for v.
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 $DC^{JOD+PDD}$ reruns the aggregation operation, e.g., `Min`, for vertex v at iteration $d^* - 1$. This procedure is similar to how DC^{JOD} reruns `Min` operator for vertices at different timestamps. Then using $J^v_{d^*-1}$ and $D^v_{d^*-1}$ $DC^{JOD+PDD}$ reruns `Min` and compute $D^v_{d^*}$. However when it accesses $D^v_{d^*-1}$, it recursively call $\texttt{AccessD}_i^v\texttt{WithDrops}$, 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.

Figure 2.3: Degree-Drop Strategy for dropping differences

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. Note that* c *already has a difference that is not dropped at iteration 2, so* c *is scheduled to rerun at iteration 2.* $DC^{JOD+PDD}$ *further checks if* c *has any dropped differences at iterations 3 and 4. Since it does not,* $DC^{JOD+PDD}$ *does not schedule* c *to rerun at these differences. Then when rerunning* c *at 2,* $DC^{JOD+PDD}$ *needs both* b*'s and* d*'s distances at iteration 1 and check if they have any differences.* $DC^{JOD+PDD}$ *sees that* d *has stored difference but* b *does not, so* $DC^{JOD+PDD}$ *checks if* b *has a dropped difference at 1. Since it does,* $DC^{JOD+PDD}$ *recomputes that difference by rerunning* b *at iteration 1.*

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

Probabilistic Difference Maintenance (Prob-Drop)

Prob-Drop drops the entire difference, i.e., both the VT pair and the state, then uses a probabilistic data structure to maintain the dropped VT pairs. Probabilistic data structures, such as Bloom [15] or Cuckoo filters [27], 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, $DC^{JOD+PDD}$ may ignore this difference and

reassemble incorrect states for vertices during execution. 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.

This chapter uses a Bloom filter[2], into which $DC^{JOD+PDD}$ inserts the dropped VT pairs. Using a Bloom filter requires minor modifications to the $\texttt{AccessD}_i^v\texttt{WithDrops}$ procedure from Section 2.3.1. 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 $DC^{JOD+PDD}$ arrives 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. Note that $DC^{JOD+PDD}$ does not need to verify whether the returned d^* was a true or false positive because regardless of the reason, the re-computed result is the correct value of v at iteration d^* (and also at iteration i).

The evaluation section (Section 2.5) shows 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 (represented as the number of concurrent queries in the evaluation section).

2.3.2 Selecting the Differences To Drop

The second component of a PDD optimization is to decide which differences to drop. A baseline heuristic is to drop each difference uniformly at random. This section proposes a more optimized technique that uses the degree information of vertices to select the differences to drop.

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 considered in this chapter, 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

[2]A Bloom filter implementation from https://github.com/lemire/bloofi

can lead to frequent vertex state re-computations. Similarly, vertices with low-degrees are relatively less frequently accessed by DC. Based on this intuition, the proposed heuristic takes two thresholds τ_{min} and τ_{max}, for minimum and maximum degrees, respectively, and a probability parameter p. Then, $DC^{JOD+PDD}$ 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 2.3):

- 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.

Empirical studies showed that setting τ_{min} as 2 and τ_{max} as the top 80th-degree percentile of the input graph is reasonable for the graphs used in the evaluation section. More sophisticated properties, such as the 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.

2.4 Implementation

Upon an update δE_{k+1} to the graph, the implementation of DC, DC$^{\text{JOD}}$, and DC$^{\text{JOD+PDD}}$ keeps track of a "frontier", which is the list of vertices and iteration numbers during which the aggregation operator should be re-computed. These are stored as an array of hash sets, to remove duplicate additions of a vertex into this set, where there is a set for each IFE iteration until max. Recall that max is the maximum number of iterations IFE has executed in any of the graph versions. Since DC$^{\text{JOD}}$ and DC$^{\text{JOD+PDD}}$ do eager merging, in this case max is the maximum IFE iteration for G_k.

These implementations store the differences in vertex states (i.e., the output of the aggregation operator) also in a hash table where the keys are vertex IDs and the value is a list of pairs $\langle i, s_v^i \rangle$ that is sorted by i, where s_v^i is the new state of vertex v in IFE iteration i. Recall again that because of eager merging, timestamps are one dimensional. To check for the state of a vertex v at iteration i, DC$^{\text{JOD}}$ find the latest available iteration $i^* \leq i$ in v's sorted list using binary search.

For PDD approaches discussed in Section 2.3, DC$^{\text{JOD+PDD}}$ uses a separate data structure (`DroppedVT`) to store the dropped vertex-timestamp pairs. Det-Drop uses a hash table, such that the key is vertex-id and the value is a sorted list of dropped iterations. When DC$^{\text{JOD+PDD}}$ needs to check if a pair ($\langle v, i \rangle$) exists in `DroppedVT`, it finds the list of iterations using the vertex-id (v) and then search for the latest dropped iteration $d^* \leq i$ in the list. For Prob-Drop the hash table is replaced by a Bloom filter. Each object in this Bloom filter is 8-bytes object and is constructed by concatenating vertex-id and iteration number together using binary operations. Searching in the Bloom filter requires constructing a search object first, using binary operations, and then check if the Bloom filter contains this object.

2.5 Evaluation

2.5.1 Experimental Setup

All experiments are run on a Linux server with 12 cores and 32 GB memory. Each experiment reports the total time, in single-threaded execution, needed to update the graph and the query answer after applying a batch of updates. For each dataset, the edges are shuffled, then the dataset is split so 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. The 90% ratio is helpful to ensure that the input graph has enough edges that represent its properties which might change over time, such as densification and shrinking diameter [47].

The default batch size is 1, because differential computation is more suitable for near real-time dynamic graph updates than for infrequent updates. An evaluation of the effects of batch sizes on the performance of DC is available in Ammar et al. [10]. It shows that as the batch size increases, the benefits of DC decreases until a certain point when Scratch could be faster. Note that all experiments, by default, use insertion-only workloads. The only exception is Section 2.5.2, which presents experiments that use workloads with different amounts of deletions and show that the effect of deleting edges is minimal in the proposed optimizations.

Datasets

This evaluation section uses a combination of real and synthetic graphs that are summarized in Table 2.3[3]. The four real graphs are **Skitter**, **LiveJournal**, **Patents**, and **Orkut**, all obtained from [46]. **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** [46] represent social network interactions with a vertex degree distribution that follows power law. **Patent** [46] represents a citation graph for all utility patents granted between 1975 and 1999. In order to experiment with weighted SPSP queries, weighted versions of real graphs have been created by adding a random integer weight between 1 and 10 uniformly at random to each edge. **LDBC SNB** [25] 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. This version has a scale factor of 10 which generates a graph of 7.2M vertices and 77.6M edges.

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	3M	29.6K	17.7	34.4
LDBC SNB	77.6M	7.2M	20.8K	7.3	9.8

Table 2.3: Datasets

Workloads

The main workload queries are SPSP, Khop, and several popular RPQ queries. SPSP and Khop run on the weighted and unweighted versions of the real datasets, respectively. For each SPSP query generation, a random pair of vertices in the graph have been selected. For Khop, a random set of vertices have been selected, and the value of maximum hops K is 5 to make it a 5-hop query.

RPQ queries require edge labels, so RPQ experiments are conducted only on the LDBC dataset. These queries leverage RPQ templates that have been frequently used in real-world workloads as defined in Bonifati et al. [16] which were used to study streaming RPQ

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

evaluation in Pacaci et al. [61]. There are only two self-relationships in LDBC SNB: Knows and ReplyOf. Self-relationships here refer to an edge label that can exist consecutively in an arbitrary path. Therefore, some templates that expect more than two self-relationships cannot be used in LDBC SNB. The following RPQ query templates are used:

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

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

SPSP, Khop, and RPQ are queries that can be supported in high-level languages of GDBMSs. Although these are the main queries that motivate this chapter, the proposed optimizations are applicable to other computations based on IFE. To demonstrate this, differential versions of the 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) have been implemented.

Baselines and Different GraphflowDB Configurations

All optimizations have been implemented inside the continuous query processor (CQP) of GraphflowDB [41], which is a shared memory GDBMS. This CQP has been extended to implement a baseline DC and all proposed optimizations to maintain the recursive queries investigated in this chapter. During this section, the following GraphflowDB versions for different configurations of DC are used: VDC, JOD, DET-DROP, or PROB-DROP.

All proposed optimizations have been compared with three baselines: DD, SCRATCH, and VDC. DD is an implementation of all workloads in the Differential Dataflow system [24], which is the reference implementation of differential computation. SCRATCH is a baseline extension of GraphflowDB's CQP to support 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. For all queries, an IFE-like label propagation algorithm is used. Note that this algorithm is identical to what is referred to as the "incremental" fixed point algorithm in the original Differential Dataflow [51] (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 a single-machine implementation using Java while the latter is a distributed system implemented in Rust. Note that both implementations are not part of this thesis's contribution. Section 2.5.2 verifies that VDC behaves similarly to DD (and even outperforms it in terms of runtime); therefore, VDC is used as the main baseline for all optimizations that are implemented inside the same GDBMS. VDC ingests and stores the input graph in the same way, uses a similar data structure to store the differences, and the same programming language as the following GraphflowDB configurations:

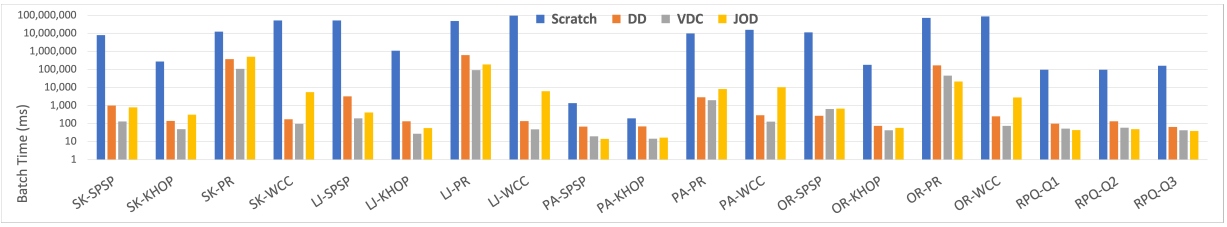
1. JOD: A DC version that implements join-on-demand optimization from Section 2.2;
2. Det-Drop: Integrates deterministic partial dropping optimization on top of JOD as discussed in Section 2.3.1;
3. Prob-Drop: Integrates probabilistic partial dropping optimization on top of JOD as discussed in Section 2.3.1.

There is also a comparison between different versions of Det-Drop and Prob-Drop to evaluate the degree-based difference dropping optimization.

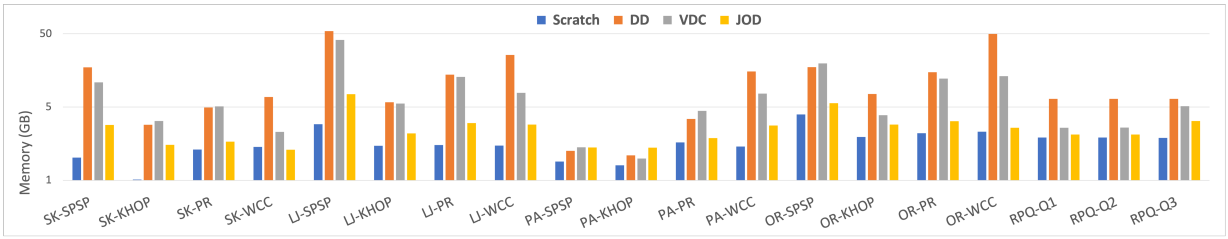
2.5.2 Baseline Evaluation

The first set of experiments measures the performances of Scratch, DD, and VDC. There are two goals: (i) to obtain baseline measurements for the proposed DC optimizations; and (ii) to validate that VDC is competitive with DD to justify its use as a more suitable baseline than DD for all optimizations. This experiment ran SPSP, K-hop queries, WCC and PR on Skitter, LiveJournal, Patents, and Orkut datasets, and all three RPQ queries on the LDBC dataset. For SPSP, K-hop and RPQ workloads, there are 10 registered queries to be monitored. Each experiment simulates dynamism by using 100 insertion-only batches, with 1 edge in each batch.

Results are shown in Figure 2.4 (ignore the JOD charts for now). As shown in 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



(a) Total Batch Time (ms)

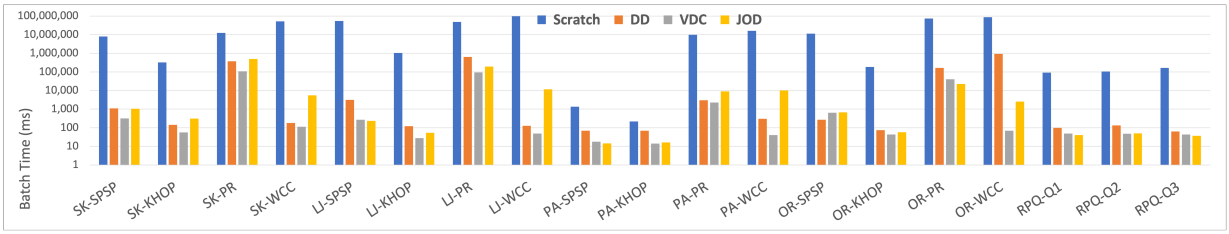


(b) Memory (GB)

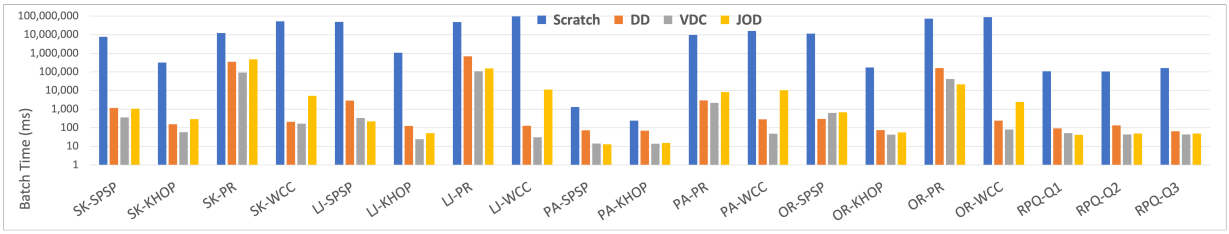
Figure 2.4: Comparison between SCRATCH, DD, VDC, and join-on-demand (JOD).

in PR, though still over an order of magnitude slower. This is expected because as also observed in prior work [69], 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 a larger number of vertices. VDC is slightly faster than DD while using comparable memory because DD assumes a distributed setting where, by default, all messaging and communication between IFE operators involves network overhead. In contrast, VDC assumes a shared memory setting, avoiding such communication overhead.

The following section 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. The performance trade-offs offered by the proposed optimizations offer are broadly similar across these different update workloads. Note that this is expected as the amount of ingested updates 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 the proposed optimizations than DD. In the rest of this chapter, VDC and SCRATCH will be used as the main baselines to evaluate all proposed optimizations on top of VDC.



(a) Total Batch Time (ms) with 25% batches



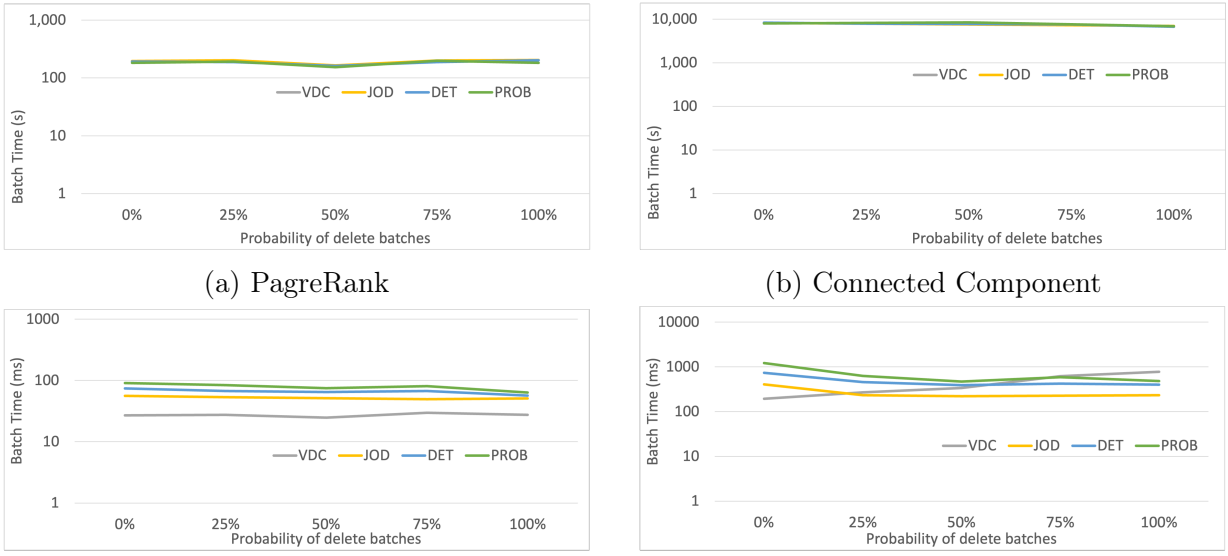
(b) Total Batch Time (ms) with 50% batches

Figure 2.5: Comparison between Scratch implementation (SCRATCH), Differential Dataflow (DD), vanilla DC implementation on top of Graphflow (VDC), and join-on-demand (JOD) using different ratios of edge deletions.

Impact of delete batches on DC optimizations

All experiments, so far, have assumed edge addition. This section evaluates the impact of deleting batches. Figure 2.5 shows the baseline experiments when 25% and 50% of batches are deleting edges. These figures are very similar to the original baseline figure (Figure 2.4), where all batches are edge additions.

All workloads have been examined for further analysis with different probabilities of delete batches (0%, 25%, 50%, 75%, 100%). Similar to previous experiments, different queries ran with 100 batches, each with 1 edge. Figure 2.6 does not have "Scratch" because it is several orders of magnitudes slower than other approaches. In general, changing the ratio of batches with delete does not change the previously reported results regarding JOD, DET-DROP, and PROB-DROP. An important observation, however, is that for SPSP query VDC is getting slower as the delete probability increases while JOD, DET-DROP, and PROB-DROP are getting faster. Note that SPSP is a weighted query and typically has a large number of iterations and a large number of differences. Deleting edges have the opposite impact for VDC, which does not use early-dropping, in comparison to the remaining implementations. From one side, deleting edges in VDC leads to adding more negative multiplicities, which then adds more overhead to be stored and maintained. On



(a) PagreRank

(b) Connected Component

(c) K-hop query

(d) SPSP query

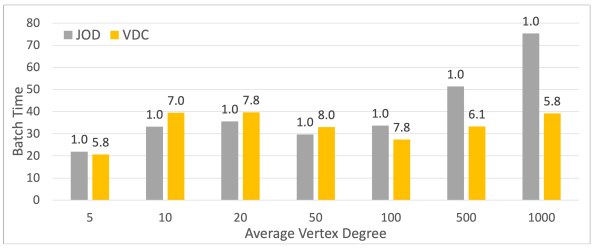
Figure 2.6: Changing the probability of deleting batches while running different queries on the LiveJournal dataset.

the other side, deleting edges with an implementation that uses early-dropping leads to reducing the number of differences.

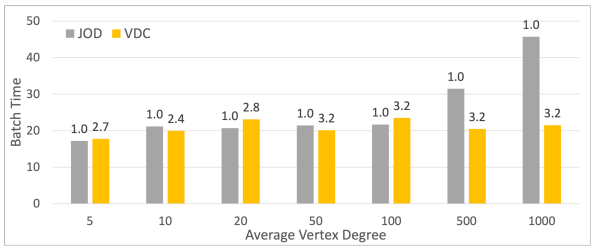
2.5.3 Join-On-Demand

The next set of experiments aims 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 overheads and benefits. On the one hand, using JOD reduces the work done by vanilla differential computation for storing differences. On the other hand, as updates arrive, JOD requires re-computing the join on demand by reading the states of in-neighbours' vertices at different timestamps to inspect if some δJ partitions are non-empty. This should be slower than materializing δJ difference sets and inspecting them to see if they are non-empty. The goal of this experiment is to answer: *What is the net effect of these performance benefits and costs? and What governs this net effect?*

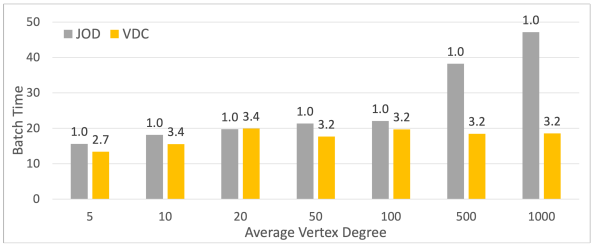
The hypothesis is that JOD computation overhead increases proportionally with the average degree of the input graph. This is because, given a vertex v, the overhead of



(a) SPSP on the `Knows` subgraph of LDBC.



(b) K-hop on the `Knows` subgraph of LDBC.



(c) RPQ-Q1 on the `Knows` subgraph of LDBC.

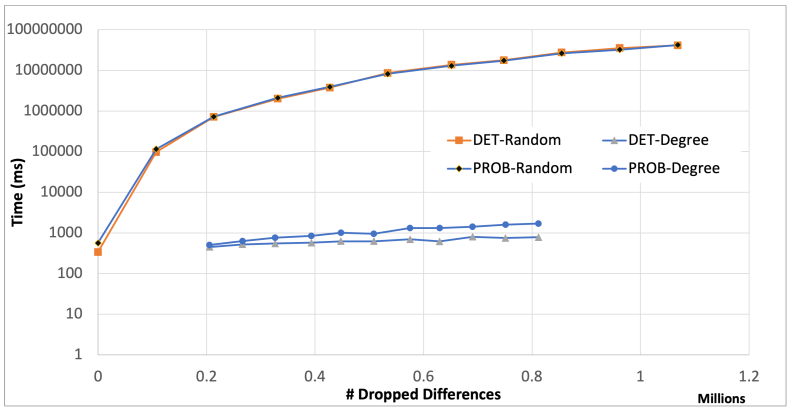
Figure 2.7: Comparison of VDC and JOD when running RPQ-Q1, K-hop, and SPSP as the average vertex degree is increased in the `Knows` subgraph of LDBC. Numbers on top of the are the average number of differences in δD per vertex.

looping through v's incoming neighbours to re-compute the join at timestamp t should increase with the number of neighbours of v. At the same time, the benefits of JOD from not storing the differences depend on how many differences are produced by the `Join` operator. This depends partially on the average degree but also on the average number of times the state of a vertex changes during a computation. For example, in the full difference trace of the running example, Table 2.1, there is a new δJ difference only when the state of a vertex changes. This number is quite small and does not necessarily grow as the average degree increases. Therefore as the average degree increases, it is expected that JOD's overhead to increase faster than its benefits, and eventually VDC will outperform JOD in terms of speed.

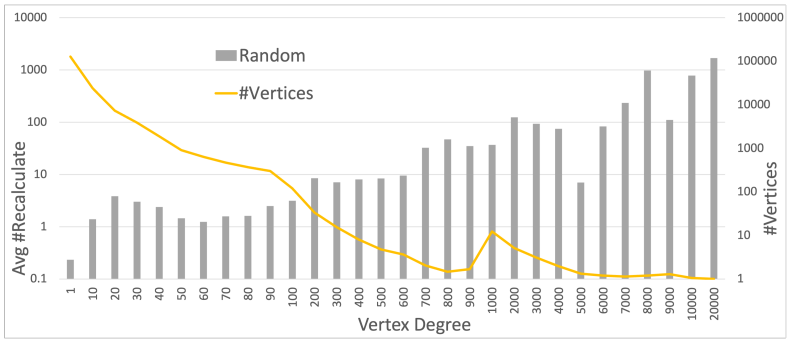
The first experiment reruns the baseline experiments from Section 2.5.2 with JOD. The average in-degrees of Orkut, Skitter, LiveJournal, Patents, and LDBC (for the subgraph containing `Knows edges`) are 34.4, 12.6, 14.2, 4.7, and 4.7 respectively. So it is expected that VDC to be faster than JOD by larger margins on Orkut and Skitter and smaller margins on Patents and LDBC. Results are shown in Figure 2.4. As expected, JOD uses significantly less memory (between 1.2× to 5.5×) than VDC irrespective of the input graph or query. In terms of performance, as expected, VDC is faster than JOD on Orkut (1.3x on k-hop) and Skitter (4.6× on K-hop) but slower than JOD on Patents (2.4x on SPSP) and on LDBC RPQs (by a factor of 1.2×).

Although the previous experiment supports the initial hypothesis, the degree differences between the input graphs are still relatively close to each other, and it was not possible to control the queries across different datasets. The following experiment is more controlled. In this experiment, the `Knows` subgraph was extracted from the LDBC dataset. Then several versions of this dataset were created by increasing the average degree from its original value of 4.7, to 20, 100, 500, and 1000. SPSP, K-hop, and RPQ queries Q1 are run on each version of these graphs. The average degree was increased by adding random edges that connect vertices in this subgraph. Results are shown in Figure 2.7.

As expected, irrespective of the query, when the average degrees are small, 4.7 or 20, JOD either outperforms or is competitive with VDC, but as the degrees get large, e.g., 100 and above in this setting, VDC consistently outperforms JOD. The numbers on top of the DC and JOD bars in Figure 2.7 are the average number of differences maintained by the aggregation operation in vertices with non-zero differences. Note that this number does not even necessarily increase as the degree increases and remains small relative to the average degree. It can even decrease in SPSP, primarily because SPSP converges faster when the degrees are larger, i.e., the number of SPSP iterations decreases, so the number of different differences vertices get can decrease.



(a) Performance of difference selection policies.



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

Figure 2.8: Comparison of RANDOM and DEGREE-based difference dropping when running 10 K-hop queries.

2.5.4 Selecting the Differences To Drop

Next is the evaluation of the effectiveness of the two strategies proposed in Section 2.3.2 for selecting which differences to drop in PDD optimizations, which are:

1. RANDOM, which randomly selects the differences with probability p;
2. DEGREE drops differences based on vertex degrees.

As mentioned in Section 2.3.2, it is expected that DEGREE to outperform RANDOM.

This experiment runs 10 `Khop` 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, there are 4 system configurations. For DEGREE, τ_{min} is set to 2 and τ_{max} to the 80^{th} percentile of the vertex degrees in the input graph. The drop probability p for DET-DROP and PROB-DROP has, then, been increased from 0% to 100%. The experiment then plots the total number of dropped differences on the x-axis and the runtime on the y-axis in Figure 2.8. There are two important observations.

First, all of the lines in the figure go up, i.e., as more differences are dropped, the performance of each system configuration gets slower. This is expected because PDD optimization has to store and maintain auxiliary data structures to maintain the dropped differences. Therefore, dropping differences primarily has a performance cost, as it can lead the system to recompute those dropped differences. Note that this is not the case with JOD, where storing fewer differences potentially leads to a performance advantage because fewer differences are maintained.

Second, configurations with DEGREE (the two bottom lines), irrespective of using 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 the maximum number of differences that configurations with DEGREE can drop. For example, at the minimum when $p = 0\%$, the configurations with DEGREE still drop all differences of vertices with degree $< \tau_{min}$, whereas RANDOM can drop as few as 0 differences.

Further analysis using a micro-benchmark is conducted to explain better the reason for the performance difference between RANDOM and DEGREE. This analysis used a configuration that uses DET-DROP with RANDOM selection policy, a fixed drop probability p (= 10%), a workload (10 `Khop` queries) and a dataset (Skitter with 100 batches of 1 edge insertions). For each vertex v the number of times DET-DROP re-computed a dropped difference with key (vertex-id) v is recorded, i.e., how many times DET-DROP has accessed

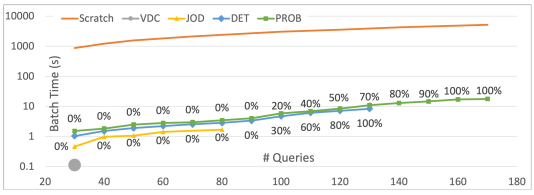
D^v at some point, but v's state had to be re-computed because a difference was dropped and then recorded in `DroppedVT`. Then, vertices were bucketed by their degree, where for each degree bucket (e.g., $[1-10)$) Figure 2.8b plot the average number of re-computations for each vertex in that bucket. Note that 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 2.8b, the degree distribution follows a power-law distribution, as is commonly the case in real-world graphs [26, 47]. 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 degrees more than 2000 lead to more than 1000 re-computations on average, while those with degrees $[1, 10)$ lead to fewer than 1 re-computations. Since the memory saving of dropping 1 difference is the same regardless of the vertex degree, the DEGREE strategy is more efficient because it drops more differences from vertices with smaller degrees.

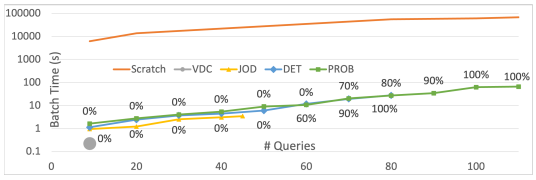
2.5.5 Difference Maintenance

The next set of experiments focus on evaluating DET-DROP and PROB-DROP. Figure 2.8a shows the performance of DET-DROP and PROB-DROP when both drop exactly the same number of differences using DEGREE and RANDOM selection policies. They behave similarly when using the same selection strategy, with DET-DROP slightly faster, which is expected because PROB-DROP may perform unnecessary 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 memory efficient than DET-DROP. Next is a more systematic evaluation of the scalability and performance trade-offs of these techniques under the DEGREE policy, which outperforms RANDOM.

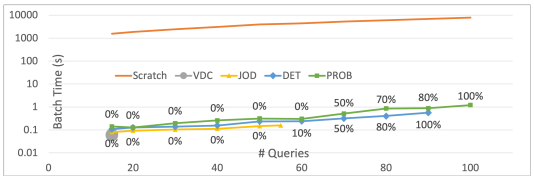
The following experiment (Figure 2.9) analyzes how much DET-DROP and PROB-DROP increase the system scalability in terms of the number of concurrently maintained queries relative to VDC for a given memory budget for SSSP, K-hop, and RPQ queries. Note that PageRank and WCC queries have been omitted from these experiments because these are batch computations and there is only one possible query/result for each graph. For completeness SCRATCH and JOD are also represented in this experiment. To simulate a fixed memory budget environment, this experiment assumes a system with a maximum memory of 10GB. This memory could be used for storing the graph data structure, differences



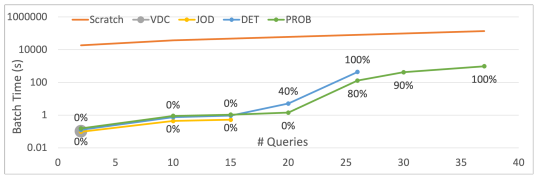
(a) K-hop query in SK dataset



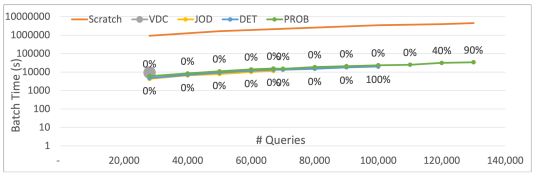
(b) SPSP query in SK dataset



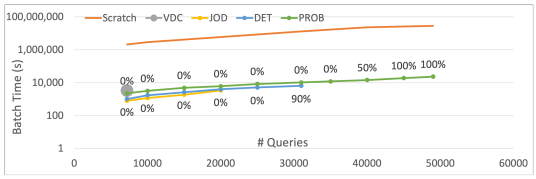
(c) K-hop query in LJ dataset



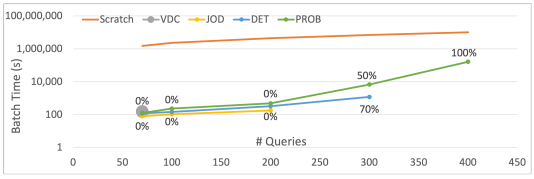
(d) SPSP query in LJ dataset



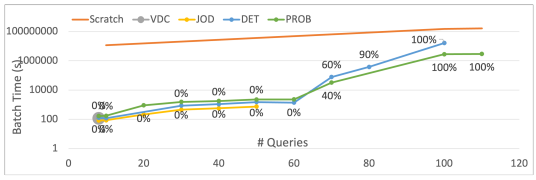
(e) K-hop query in PA dataset



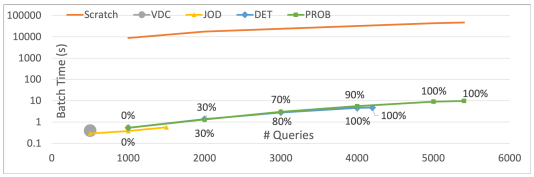
(f) SPSP query in PA dataset



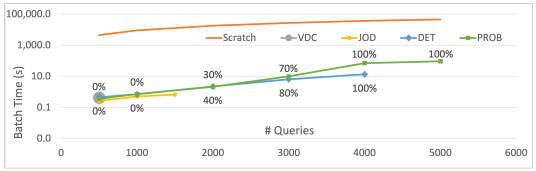
(g) K-hop query in OR dataset



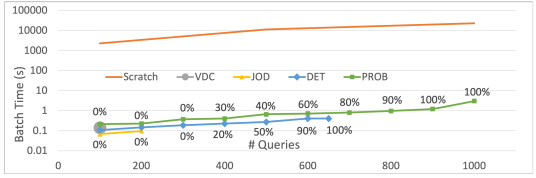
(h) SPSP query in OR dataset



(i) RPQ-Q1



(j) RPQ-Q2



(k) RPQ-Q3

Figure 2.9: Number of queries maintained by Scratch, DC, JOD, Det-Drop, and Prob-Drop under a limited memory budget of 10GB. The large dot in the bottom left of each figure is DC.

and/or additional data structures, e.g., to manage dropped VT pairs. For consistency, this experiment uses the same queries, datasets, and batches from Section 2.5.2. However, the number of queries systematically increases until the system runs out of memory.

Figure 2.9 uses the maximum scalability level of VDC, which is the maximum number of queries that can fit in the memory budget (10GB), as the lowest number of queries, and then increases the number of queries in the system from this point on. That is why VDC appears as a single grey point in all charts. For DET-DROP and PROB-DROP, for each number of queries q, this experiment finds 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 there is an assumption of an ideal setting, such that there is a system that is able to find this lowest dropping probability. This system is used to evaluate the most performant versions of DET-DROP and PROB-DROP for the given query level. In practice, finding this probability is challenging and left as future work. In Figure 2.9 p_{det} that is used for DET-DROP is reported under the DET-DROP line, and the p_{prob} that is used for PROB-DROP is reported above the PROB-DROP line.

This experiment leads to a few critical observations. First, as in Figure 2.4, JOD can increase the number of queries that could be concurrently run by $2.3\times - 10\times$ over VDC. Second, increasing the number of queries with partial dropping optimizations typically increases the run time super-linearly beyond a particular point. After this point, increasing scalability requires increasing the dropping probability, which leads to more differences to be re-computed. However, PDD can increase the number of concurrent queries by up to $20\times$ relative to VDC while still outperforming SCRATCH by several orders of magnitude. Third, the advantage and disadvantages of DET-DROP and PROB-DROP 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 $1.5\times$). That is because, as mentioned earlier, DET-DROP does not incur any unnecessary 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).

The last experiment in this section uses PR and WCC workloads, for which there is only one possible "query", using the LJ dataset. The memory budget for PR is 2.75GB, and the memory budget for WCC is only 2GB. Figure 2.10 shows the lowest drop probabilities at which these budgets were enough for DET-DROP and PROB-DROP, with the necessary drop percentages presented on top of the bars. Note that for PR, DET-DROP requires 100% dropping rate and takes 369 seconds to complete while PROB-DROP requires 90% dropping

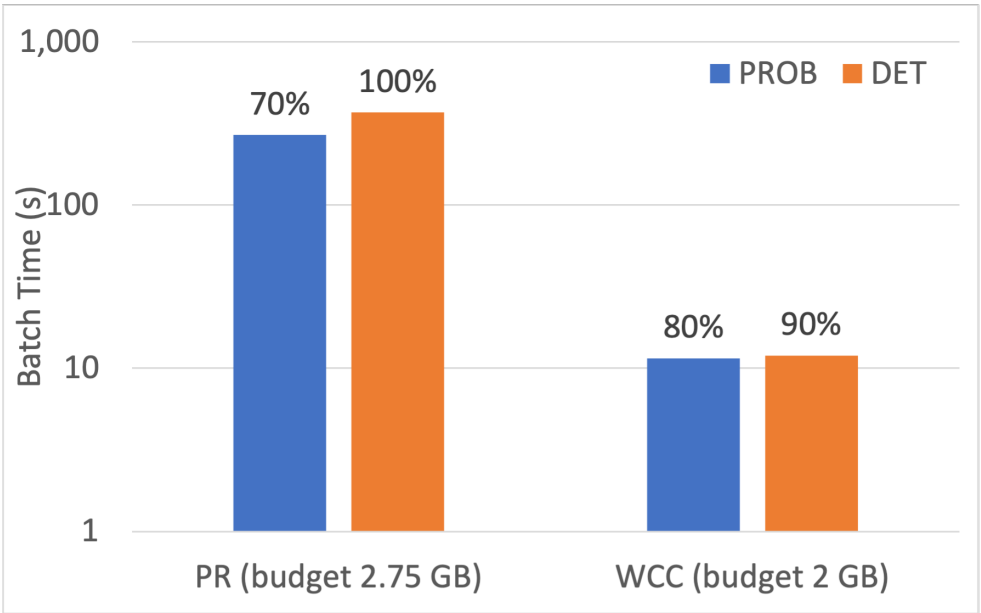


Figure 2.10: Comparison of Det-Drop and Prob-Drop when running PageRank and WCC on LJ under limited memory. The probabilities on top of each bar represent the lowest drop probabilities at which a budget of 2.75GB for PR and a budget of 2GB for WCC are enough for query execution.

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 previous experiments, Prob-Drop needs to drop fewer differences to successfully complete the experiment, which sometimes leads to better performance.

2.5.6 Further Applications of Diff-IFE

The previous experiments have focused on demonstrating the performance tradeoffs that the JOD, Det-Drop, and Prob-Drop optimizations offer when evaluating continuous recursive queries using IFE module. This section aims to demonstrate further applications of IFE in general systems. Specifically, it shows that it is possible to improve the

[4]Recall that 100% dropping rate does not mean all differences are dropped. The Degree approach does not drop any differences for vertices with a degree over 13 in the LJ dataset.

performance of SCRATCH baseline for SPSP queries through using and differentially maintaining a popular shortest path index, called *landmark indices* [32, 63]. 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.

In this section, landmark indices are used 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 l to d gives an upper bound ℓ_u on the shortest distance between s and d. Similarly, the difference between the v to l distance and l to d distance gives a lower bound ℓ_b on the distance from v to d. If v is visited at distance k in the Bellman-Ford algorithm, and $k + \ell_b$ is greater than ℓ_u, then the algorithm can avoid traversing v as it cannot be on the shortest path from s to d.

This experiment uses all datasets except LDBC and picks the 10 highest-degree nodes as landmarks. It uses an optimized version of SCRATCH which is called SCRATCH-landmark. In this optimization, as updates arrive at the graph, the system first maintains these 10 landmark indices using JOD and then run each registered query using the landmark-enhanced SCRATCH.

Figure 2.11 compares SCRATCH and SCRATCH-landmark using 100 SPSP queries and measured the end-to-end time of 100 batches of single-edge insertions. 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, SCRATCH-landmark 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).

2.6 Conclusions

Differential computation is a highly generic novel technique to maintain arbitrary dataflow computations, particularly iterative ones that can implement recursive subroutines. As such, it is a promising technique to integrate into data management systems that aim to support continuous queries that require recursive computations. Differential computation is based on a simple computation maintenance procedure that stores the input and output differences of operators in partially ordered timestamps and reruns these operators at different timestamps when changes to their inputs are detected. Although its simple routine makes it very generic and highly efficient, the amount of storage required by differential computation can be very large for some queries, limiting its scalability.

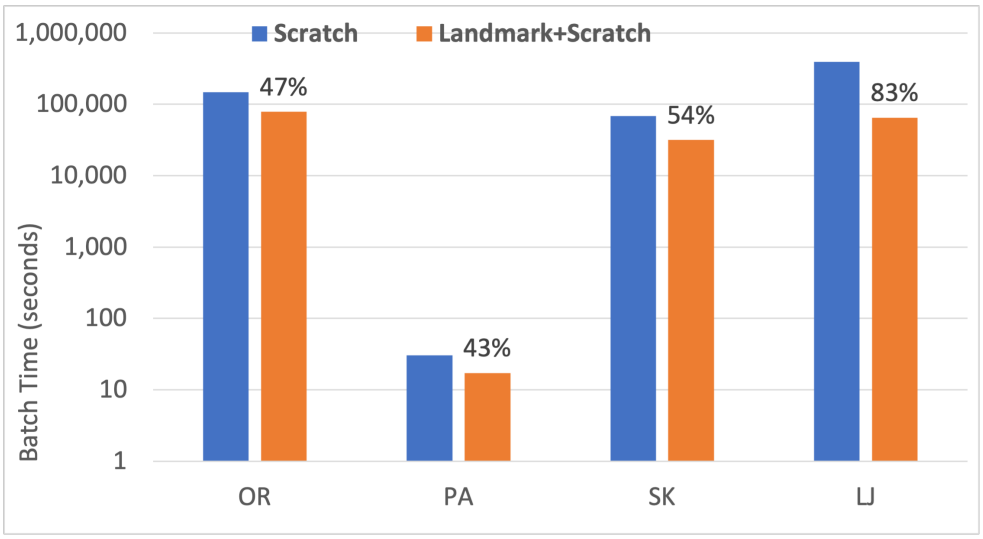


Figure 2.11: Comparing SCRATCH vs. SCRATCH-landmark on 100 queries and 100 batches of updates. Numbers on orange bars are the runtime improvements of SCRATCH-landmark.

This chapter studied the problem of how to increase the scalability of differential computation, focusing on a standard dataflow subroutine whose variants can evaluate many recursive graph queries, as supported in the query languages of GDBMSs. The proposed optimizations (complete dropping and partial dropping) are generally based on dropping differences either completely or partially. They are best suited for applications for which updates are relatively small, and evaluation of the query upon each update is critical. As in any materialized view maintenance application, applications that do not require getting updates on the query results may choose to rerun the query periodically instead. Rerunning the query periodically from scratch can be done by the application or supported and delegated to the system and could indeed outperform methods, including differential computation-based methods described in this chapter, if the frequency of query re-computation is low enough and updates are very infrequent.

The scalability and performance trade-offs of these optimizations have been studied. This chapter showed that these optimizations can increase the scalability of vanilla differential computation by more than an order of magnitude while still outperforming a baseline that reruns queries from scratch.

Complete dropping optimization, a.k.a. JOD, avoids storing any difference sets for `J` in IFE. This means JOD completely drops δJ, and regenerates J^u for any vertex u on demand when DC requires running the aggregation operator (in our example `Min`) on u at a particular timestamp. Eager merging implementation helps JOD by guaranteeing that for

any vertex, there are only one-dimensional timestamps, i.e., only for *IFE iteration* instead of two dimensions (*Graph version,IFE iteration*). Furthermore, eager merging allows dropping all differences with negative multiplicities in the difference sets for `D` because vertices take one unique state at each iteration of IFE. The experimental results show 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, JOD and VDC could be faster based on the graph degree. When the average degrees are small, 4.7 or 20, JOD either outperforms or is competitive with VDC, but as the degrees get large, e.g., 100 and above in this setting, VDC consistently outperforms JOD.

Partial dropping, PDD, partially drops some differences from `J` in IFE by keeping track of their keys and timestamps but not the actual difference values and recomputes these differences when differential computation's maintenance procedure needs them. Instead of selecting random differences to drop, the chapter proposes a degree-based approach which leads to several orders of magnitude faster execution than the random approach. The Det-Drop and Prob-Drop approaches keep track of differences in timestamps using a deterministic data structure, e.g. hash table, or probabilistic data structure, e.g. bloom filter, respectively. Together, JOD and PDD increase the scalability of vanilla differential computation up to 20× while being several orders of magnitude faster than Scratch.

Chapter 3

Optimizing Fixed-Length Subgraph Query Execution[1]

Subgraph queries, i.e., finding instances of a given subgraph in a larger graph, are a fundamental computation performed by many applications and supported by many software systems that process graphs. Example applications include finding triangles and larger clique-like structures for detecting related pages in the World Wide Web [29] and finding diamonds for recommendation algorithms in social networks [35]. Example systems include graph databases [56, 78], Resource Description Framework (RDF) engines [57, 89], as well as many other specialized graph processing systems [3, 49, 75]. As the scale of real-world graphs and the speed at which they evolve increase, applications need to evaluate subgraph queries efficiently both offline and in real time. Variable length subgraph queries, like Regular Path Queries (RPQ), could be answered using the approaches discussed in the previous chapter. This chapter focuses on fixed-length subgraph queries.

Recall that this thesis adopts the relational view of graph queries, in which any subgraph query can be seen as a multiway join on replicas of an edge table of the input graph. Also, recall that a join algorithm is *worst-case optimal* for a query Q if its computation cost is not asymptotically higher than the AGM bound (referred to as $MaxOut_Q$) of Q [11], which is the maximum possibly output size for the given size of the relations in Q.

This chapter proposes a new version of *Generic Join* (GJ) [58], called Delta-GJ, that supports dynamic graphs. Delta-GJ requires memory that is linear in the size of the changes to the graph and is worst-case optimal for insertion-only workloads in terms of

[1]This work has been published as [9]

computation costs. Section 3.4.1 has proof that under insertion-only workloads, DELTA-GJ is worst-case optimal.

This chapter also explores distributed implementation of GJ because, unlike BJ which is used in recursive queries, the previous work on GJ has already provided several single-node implementations of WCOJ algorithms [3, 41]. Distributed GJ (called BIGJOIN) expands GJ to run in distributed share-nothing environments and also expands the definition of worst-case optimal to include network communication. Specifically, it shows optimizations to balance the workload of the machines in the cluster and make algorithms provably skew-resilient, i.e., guarantee that the costs per worker decrease linearly as additional workers are introduced. Recall that in a single-node setting, a join algorithm is *worst-case optimal* for a query Q if its computation and memory cost is not asymptotically larger than the AGM bound (referred to as $MaxOut_Q$) of Q [11], which is the maximum possible output size for the given size of the relations in Q. To expand this definition for distributed settings, this chapter expands the worst-case optimal definition to include communication costs. Indeed, a naive "distributed" algorithm can send all of the input to one worker w^* and use a sequential worst-case join algorithm. This algorithm would achieve all of the optimality guarantees but without balancing the workload in the cluster. Instead, this chapter shows optimizations to balance the workload of the cluster workers and make these algorithms provably skew-resilient, i.e., guarantee that the costs per worker decrease linearly as additional workers are introduced.

The rest of this chapter is organized as follows. Section 3.1 reviews the required preliminaries, and related work for this chapter, including GJ, existing distributed subgraph algorithms, Massively Parallel Computation (MPC) model [13, 12, 45], and Timely Dataflow. Section 3.2 uses the MPC model to introduce BIGJOIN for static graphs that achieve the expected theoretical guarantees based on *worst-case optimal* definition. BIGJOIN achieves cumulative worst-case optimality and, in real-world data sets and queries, achieves good workload-balance and low per-worker memory (Section 3.2). However, on adversarial inputs, it can lead to a single worker performing most of the work. This theoretical shortcoming is resolved in another algorithm called BIGJOIN-S (Section 3.3). Specifically, BIGJOIN-S is the first distributed join algorithm that has worst-case communication and computation costs and achieves workload balance across workers on every query. In addition, BIGJOIN-S achieves these guarantees with as low as $O(\frac{\text{IN}}{w})$ memory per worker where IN is the input size and w is the number of workers. After that, Section 3.4 introduces DELTA-GJ with a running example and then proposes a distributed version of DELTA-GJcalled DELTA-BIGJOIN.

Finally, Section 3.5 presents how BIGJOIN and DELTA-BIGJOIN algorithms are implemented in Timely Dataflow. Then, Section 3.6 summarizes the evaluation of the pro-

```
P_0={}
for (j = 1... m):
    P_j={}
    for (p ∈ P_{j-1}):
        // ∩ below is performed starting from smallestE Ext^i_j(p)
        ext_p = ∩Ext^i_j(p)
        P_j = P_j ∪ ext_p
```

Figure 3.1: Pseudo-code of GJ.

posed algorithms. Algorithms evaluations include comparisons against an optimized single-threaded algorithm, an existing shared-parallel system, and two existing distributed systems specialized for evaluating subgraph queries. Evaluations show that DELTA-BIGJOIN can monitor complex cyclic subgraphs very efficiently on graphs with up to 64B edges on a cluster of 16 machines using very low memory.

3.1 Related Work

3.1.1 Generic Join (GJ)

Ngo et al. [59] and Veldhuizen [80] proposed worst-case optimal join algorithms for equi-join queries called, respectively, *NPRR* and *Leapfrog TrieJoin*. These algorithms are instances of a more general algorithm called *Generic Join* [58] (GJ). As applied to subgraph queries, these algorithms adopt a *vertex-at-a-time* evaluation technique. Specifically, on a query that involves $\{v_1, ..., v_m\}$ query vertices, these algorithms first find all of the (v_1) answer vertices that can end up in the output. Then they find all sets of (v_1, v_2) vertices that can end up in the output and so on until the final output is constructed. When extending a partial subgraph to a new vertex v_i, all of the edges that are incident on v_i are considered and intersected. For example, on the `Triangle` query (x, y, z), these algorithms would first find all x vertices and then (x, y) edges that can possibly be part of a triangle. Then the algorithms extend these edges to (x, y, z) triangles by intersecting x's incoming and y's outgoing edges. Compared to *Binary Join* [84] (BJ) algorithm (also known as edge-at-a-time), which finds matching edges in the query one at a time, these algorithms will never generate intermediate data larger $MaxOut_Q$.

Figure 3.1 shows the pseudo-code of GJ using relational tables notations. Given a

query $Q(v_1, \ldots, v_m)$ with m attributes and n relations (R_1, $\ldots$,R_n), GJ consists of the following three high-level steps:

1. **Global Attribute Ordering:** GJ first orders the query attributes. For simplicity, one can assume the order is $v_1, .., v_m$. Everything that follows remains correct if the attributes are arbitrarily ordered.

2. **Extensions Indices:** Let a *prefix j-tuple* be any fixed values of the first $j < m$ attributes. For each table replica R_i and j-tuple p only some values for attribute v_{j+1} exist in R_i. Let the *extension index* Ext^i_j map each j-tuple p to values of v_{j+1} matching p in R_i:
$$Ext^i_j : (p = (v_1, .., v_j)) \rightarrow \{v_{j+1}\} .$$
Extension indices need three properties for the theoretical bounds of GJ. For a given p, the algorithm must be able to:
   - retrieve the size $|Ext^i_j(p)|$ in constant time;
   - retrieve the contents of $Ext^i_j(p)$ in time linear in its size
   - check that a value e of attribute v_{j+1} exists in $Ext^i_j(p)$ in constant time.

   Throughout this chapter, $Ext^i_j(p \bullet e)$ refers to the operation of checking of value e in $Ext^i_j(p)$. Many data structures, such as hash tables, satisfy these properties.

3. **Prefix Extension Stages:** GJ iteratively computes intermediate results $P_1 \ldots P_m$, where P_j is the result of Q when each relation is restricted to the first j attributes in the common global order. GJ starts from the singleton relation P_0 with no attributes, determines P_{j+1} from P_j using the extension indices, and ultimately arrives at $P_m = Q$. Specifically, for each *prefix* j-tuple $p \in P_j$, GJ determines the (possibly empty) set of $(j+1)$-tuples extending p as follows:
   (a) identify the size of $Ext^i_j(p)$ of each relation R_i containing v_{j+1};
   (b) sort all extension sets ($Ext^i_j(p)$) based on their size;
   (c) intersect all $Ext^i_j(p)$ ordered by size; starting from the smallest set. In general performing this intersection in time proportional to the size of the smallest set ensures worst-case optimal run-time.

The following is a theorem from [58] expressed using the notations adopted in this thesis:

Theorem 3.1.1. *[58] For any query Q comprising relations $R_1 \dots R_n$ and attributes $v_1 \dots v_m$, and any ordering of attributes, if Ext^i_j indices satisfy the three properties discussed earlier,* GJ *runs in time $O(mnMaxOut_Q)$.*

The proof of this theorem can be found in Ngo et al. [58]. Note that the $m \times n$ factor in front of $MaxOut_Q$ is a constant that depends only on the number of relations and attributes in the query but not the number of tuples in the relation. The following is an example that shows how to execute GJ when evaluating a subgraph query like `Triangle`.

Example 4. *Consider evaluating the triangle query $Q(v_1, v_2, v_3) := R_1(v_1, v_2)$, $R_2(v_2, v_3)$, $R_3(v_3, v_1)$, where each R_i is an exact replica of the edges in the input graph in Figure 3.2.* GJ *executes as follows starting from $P_0 = \{\epsilon\}$:*

1. *P_1:* GJ *extends $\{\epsilon\}$ to $P_1 = \epsilon \times (Ext^1_1[\epsilon] = \{1, 2, 3, 4, 5, 6, 7\} \cap Ext^3_1[\epsilon] = \{1, 6, 7, 8, 9, 10, 11\})$. Ext^1_1 and Ext^3_1 correspond to indices over R_1 and R_3, respectively. In this case, neither set is smaller than the other and* GJ *is free to choose arbitrarily. This intersection produces $P_1 = \{(1), (6), (7)\}$.*

2. *P_2:* GJ *extends each prefix p in P_1 with valid v_2 producing $p \times (Ext^1_2[p] \cap Ext^2_2[p])$. This is done by considering the sizes of $Ext^1_2[p]$ and $Ext^2_2[p]$, which for (1) are $Ext^1_2[(1)] = \{6\}$ and $Ext^2_2[(1)] = \{1, 2, 3, 4, 5, 6, 7\}$. The former index is smaller, and so* GJ *starts from the set $\{6\}$ and intersects it with $Ext^2_2[(1)]$, producing $(1, 6)$. Other extensions in P_2 are $(6, 7)$ and $(7, 1)$.*

3. *P_3: Finally,* GJ *extends each of these three prefixes using Ext^2_3 and Ext^3_3, again starting from the smaller of the candidate extensions for each prefix. For example, when extending $(1, 6)$, $Ext^2_3 = \{7, 8, 9, 10, 11\}$ is intersected with $Ext^3_3 = \{7\}$ giving the triangle $(1, 6, 7)$. Similarly $(6, 7)$ and $(7, 1)$ give the outputs $(6, 7, 1)$ and $(7, 1, 6)$, respectively.*

3.1.2 Massively Parallel Computation Model

Massively Parallel Computation (MPC) [13, 12, 45] is an abstract model of distributed bulk synchronous parallel systems. It has been previously used to understand and analyze the performance of distributed multiway join queries [13, 12, 45]. In this model, there are w workers (or machines) in a cluster and the input data is assumed to be equally distributed among the workers arbitrarily. The computation is broken down into a series of *rounds*,

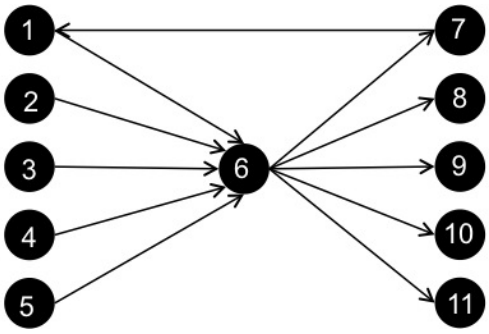


Figure 3.2: Example input graph.

where in each round the workers first perform some local computation and then send each other messages. The complexity of algorithms are measured in terms of three parameters: (1) r: the number of rounds; (2) L: the maximum *load* or messages any of the workers receives in any of the rounds; and (3) C: the total communication, i.e., sum of the loads across all rounds.

To evaluate algorithms balance between workers, this chapter extends typical MPC with a fourth parameter M that measures the memory that an algorithm uses. Let M_k^t be the local memory that worker k requires in round t, excluding the output tuples. In this setting, M_k^t represents the load L of worker k in round t and the amount of input data worker k has indexed. M is then the the maximum cumulative memory the algorithm requires in any round: $\max_{t=1,\ldots,r} \Sigma_{k=1,\ldots,w} M_k^t$. This model assumes output tuples are written to a storage outside the cluster and do not stay in memories of workers. This is because any correct algorithm incurs this cost.

For simplicity, similar to prior work [6, 13, 45] the unit of communication and memory will be tuples and prefixes, instead of bits, assuming that tuples and prefixes are all the same size.

3.1.3 Timely Dataflow (TD)

Timely Dataflow [55, 77] is a distributed data-parallel dataflow system, in which one connects dataflow *operators* describing computation using dataflow edges describing communication. The operators are *data-parallel*, meaning that their input streams may be partitioned by a provided key, and their implementations may be distributed across multiple workers. All operators are distributed across all workers, and each worker is responsible

for the execution of some fraction of each operator, which allows our algorithms to share indices (of the underlying relations) between operators.

TD is a dataflow system in the sense that computation occurs in response to the availability of data, rather than through centralized control. The *timely* modifier corresponds to the extension of each operator with information about logical progress through the input streams, roughly corresponding to *punctuation* or *watermarks* in traditional stream processing systems. Note that operators can delay processing inputs with some timestamps until others have finished, which can be used to synchronize the workers and ensure that the work queues of downstream operators have drained, an important component of ensuring a bounded memory footprint.

3.1.4 Distributed Subgraph Queries Algorithms

Existing distributed approaches that can be used to evaluate general subgraph queries can be broadly grouped into two classes of algorithms. The first group consists of edge-at-a-time approaches [20, 30, 37, 57, 74, 75, 87] that correspond to binary join plans in relational terms. As discussed earlier (Section 3.1.1), this approach is provably suboptimal for subgraph queries.

The second group includes approaches that use variants of the Shares [5] or Hypercube [13, 12, 45] algorithm. Consider a distributed cluster with w workers and a query with n relations and m attributes, i.e., n is the number of edges, and m is the number of vertices in the query. Shares divides the m-dimensional output space equally over the w workers and replicate each tuple t of each relation to every worker that can produce an output that depends on t. Finally, each worker runs any local join algorithm on the inputs it receives to produce the outputs that belong to the worker's partition.

There are several advantages of Shares. For most queries and parallelism levels w (but not all), Shares' communication cost is less than the $MaxOut_Q$. In addition, in distributed bulk synchronous parallel systems, in which the computation is broken down into a series of *rounds*, Shares requires a very small number of rounds. However, Shares' cumulative memory requirement is $O(w^{1-\epsilon}\text{IN})$ and its memory requirement per worker is $O(\frac{\text{IN}}{w^{\epsilon}})$. Note that IN is the size of the input, and $\epsilon \in [0, 1]$ is a query-dependent parameter. This implies a super-linear cumulative memory growth and sub-linear scaling of per-worker memory (and workload) as w increases. For example, for the `Triangle` query, $\epsilon = 1/2$. Often ϵ is much smaller, and scaling becomes an increasingly resource-inefficient way to improve performance.

Hu et al. [36] proposed an algorithm for queries that involve only two relations based on sorting the relations on their join attributes. This contrasts with the hashing approach of Shares. The algorithm runs for a small number of rounds. It requires cumulative memory and communication as large as the actual output but does not generalize to more complex joins, e.g., involving three relations.

Afrati et al. [7] has introduced a multiround join algorithm called *GYM*, which takes as input a *generalized hypertree decomposition* (GHD) D of Q. The algorithm first computes several intermediate relations based on D in one round using Shares. Then the algorithm runs a distributed version of Yannakakis [84]'s algorithm for acyclic queries. Overall the algorithm runs for $O(n)$ rounds and incurs a communication and cumulative memory cost of $O(\text{IN}^w)$, where $w \geq 1$ is called the *width* of the GHD D. This amount of communication cost is always $O(MaxOut_Q)$ for acyclic queries because w is only 1. For any cyclic query, the memory requirements of GYM are superlinear in IN.

Finally, Joglekar and Ré [39] introduces DBP algorithm. DBP uses 3 rounds and takes $O(L\text{IN}^{DBP(L)})$, where L is a free parameter that indicates load per machine and $DBP(L)$ is a query-related parameter that is called the *degree-based packing bound* of the query for load L. Similar to GYM, for any L, DBP's communication is always $O(MaxOut_Q)$. Still, the algorithm can require a cluster memory that is superlinear in IN as it computes intermediate relations that can be large in size.

3.1.5 One-time Subgraph Queries

Most existing systems that evaluate general subgraph queries are based on the edge-at-a-time strategy, unlike BigJoin's vertex-at-a-time strategy.

EmptyHeaded [3] (EH) is a highly-optimized shared-memory parallel system evaluating subgraph queries on static graphs using GJ. EH evaluates queries using a mixture of GJ and binary join (BJ) plans. The EH optimizer considers *generalized hypertree decompositions* of the query, which join multiple subsets of the relations using GJ which are then joined using BJ. EH is highly optimized for evaluating queries on static graphs and spends a non-trivial amount of time preparing its indices, which vary their representation in response to the structural properties of the underlying data. Section 3.6.2 evaluates EH against BigJoin.

SEED [48] is a scalable sub-graph enumeration approach with several optimizations for evaluating undirected subgraph queries in the distributed setting. Section 3.6.4 discusses the implementation details of these optimizations and shows that they can improve the performance of BigJoin.

Arabesque [75] is a distributed system specialized for finding subgraphs in large graphs. In Arabesque, each distributed worker gets an entire copy of the graph and starts extending a partition of the vertices to form larger and larger subgraphs that are called *embeddings*, equivalent to prefixes in GJ's terminology. In Arabesque, prefixes are extended by considering the neighbours of individual vertices, rather than by intersecting the neighbourhoods of multiple vertices as GJ does. Unlike BigJoin, Arabesque is using the edge-at-a-time strategy discussed at the beginning of this chapter. Section 3.6.2 evaluates Arabesque's version (1.0.1-BETA), which runs on Giraph [31] against BigJoin.

PSgL [71] is another distributed subgraph enumeration system that is built on top of Giraph [31]. PSgL picks an order of the vertices (i.e., attributes), say $a_1, ..., a_m$ in Q, called a *traversal order*. It starts with candidate partial matches G_{psi} for a_1, then extends each G_{psi} to all neighbours of a_1 in Q (not just a_2). When matching a_j, the existence of edges (a_i, a_j) edges for $i < j$ will be checked, and if they exist, a_j will be extended to all neighbours $a_k > a_j$. This is similar to Arabesque's VertexInducedEmbeddings and is an edge-at-a-time strategy. Shao et al. [71] discussed techniques for picking good traversal orders, balancing workload among workers, and breaking internal symmetries in queries over undirected graphs, which can complement BigJoin on undirected graphs as well.

SPARQL [62] queries can express any subgraph query. TrinityRDF [87] and Spartex [2] are two distributed RDF engines that can evaluate any SPARQL query. The optimizers of both systems use edge-at-a-time strategies, which are provably not optimal in comparison to vertex-at-a-time strategies used in BigJoin.

3.1.6 Continuous Subgraph Queries

There is a vast body of work on incrementally maintaining views that contain selection, projection, joins, and group-by-aggregates, among others; Rada Chirkova and Jun Yang [64] is a survey of these techniques. Delta-BigJoin falls under the *algebraic technique* of representing updates to tables as delta relations and maintaining views through a set of relational algebraic queries. This approach has been extensively studied in previous work. Prior work on algebraic techniques ranges from addressing limitations of delta query-based techniques, e.g., when evaluating a top-k query [85], to techniques using higher-delta queries [8], e.g., delta queries of a query. When evaluating subgraph queries, these techniques do not yield theoretically optimal results and require materializing very large intermediate results.

Recently, a few surveys [73, 82] studied algorithms that monitor subgraph queries on dynamic graphs. There are two categories of incremental view maintenance (`IVM`)

algorithms that process continuous subgraph queries. The first category incrementally updates the query answer from changes in ΔG, similar to Delta-GJ. The second category uses an index I, and then computes ΔI from ΔG to answer the subgraph query.

The first category includes Delta-BigJoin and inc-LFTJ [81]. Similar to Delta-BigJoin, inc-LFTJ is based on the Leapfrog TrieJoin (LFTJ) worst-case optimal join algorithm [80]. Similar to GJ, LFTJ is based on doing intersections of multiple extension sets in time proportional to the size of the minimum-size set. Unlike our description of GJ, which uses hash-based indices, LFTJ uses tries to index the prefixes of the tuples in each input relation.

Algorithms in the second category create an auxiliary data structure and maintain it. SJ-Tree [20] divides a subgraph query into smaller sub-queries in a deep left tree and then joins them using BJ. This tree is considered a cache for the query answer, but it grows exponentially and limits SJ-Tree's ability to scale. Changes in the graph impact relations in the SJ-Tree in a bottom-up direction until the final result is updated. TurboFlux [42] uses a spanning tree of the query as an index which grows linearly with the query and graph size, making it significantly more efficient than SJ-Tree. Finally, SymBi [54] uses an auxiliary data structure based on a directed acyclic graph instead of a spanning tree. This data structure has a better pruning power than TurboFlux's data structure leading to a significant improvement in performance.

3.2 BigJoin Algorithm

Recall from Section 3.1.1 that the main process in GJ is extending prefixes P_j to P_{j+1}. Informally, the BigJoin algorithm follows the spirit of GJ, starting from the singleton relation P_0 and deriving relations P_j and ultimately $P_m = Q$. For each P_j, responsibility for the prefixes is partitioned across the workers by hashing the prefixes, ensuring an approximately balanced distribution. At this point, each prefix follows the course of the GJ algorithm, encountering each of the relevant extension indices Ext_j^i, whose elements are also distributed among the workers using a hash of their keys (the restrictions of tuples in R_i to the first i attributes in the global order). This section starts by describing a naive version of a core dataflow primitive, Figure 3.3, then uses it to develop BigJoin.

3.2.1 Dataflow Primitive

The core dataflow primitive starts from a collection of P_j tuples stored across w workers, and produces the P_{j+1} tuples across the same workers. Initially, this core primitive will closely tracks the GJ algorithm behaviour, starting from the full collection P_j and producing the full collection P_{j+1}. Then, this primitive will be modified to achieve both memory limits and workload balance across workers to achieve our theoretical bounds.

In bulk synchronous parallel (BSP) computation, a dataflow is described as a sequence of steps, where workers execute each step to completion and synchronize between steps (corresponding to a round in BSP terms). Naive execution of these steps may produce very large amounts of data between steps and require very large memory in the workers. Note that this process is similar to the GJ algorithm described in Section 3.1.1, where prefixes are extended using the extension index proposing the fewest candidates, followed by intersection with the extension indices of other relations.

Figure 3.3 shows the operators implementing this dataflow primitive. In the figure, the vertical lines annotated with s indicate synchronization points. The `Count`, `Proposal`, and `Intersect` are the dataflow operators implementing the steps above. These steps, executed in sequence would be a synchronous BSP implementation of the extension of P_j to P_{j+1}, which could be repeated until the algorithm arrives at $P_m = Q$. Specifically, the extending P_j to P_{j+1} follow these steps:

- *Initially*: The tuples of P_j are distributed among the w workers arbitrarily. Each prefix p is transformed into a triple $(p, \infty, \perp)$ capturing the prefix, the currently smallest candidate set size, and the index of the relation with that number of candidates.

- *Count minimization*: For each R_i binding attribute a_{j+1}, in order, workers exchange the triples by the hash of p's attributes bound by R_i, placing each triple at the worker with access to $Ext^i_j[p]$. Each worker updates each triple with the smallest count and introduces its own index if $|Ext^i_j[p]|$ is smaller than the recorded smallest count. Eventually, each triple is then output as input of the count minimization for the next relation. In the end, the dataflow has a collection of triples $(p, \textit{min-c}, \textit{min-i})$ indicating for each prefix the relation with the fewest extensions.

- *Candidate Proposal*: Each worker exchanges triples using a hash of p's attributes bound by $R_{\textit{min-i}}$. Each worker now produces for each triple $(p, \textit{min-c}, \textit{min-i})$ it has, and each extension e of p in $Ext^{\textit{min-i}}_j[p]$, a candidate $(j+1)$-tuple $(p \bullet e)$.

- *Intersection*: For each relation R_i binding attribute a_{j+1}, in order: Workers exchange the candidate $(p \bullet e)$ tuples by the hash of $(p \bullet e)$'s attributes bound by R_i. Each worker consults $Ext^i_j[p]$. If e exists $(p \bullet e)$ is produced as output otherwise it is discarded.

The random access working set of these operators are only the extension indices; all inputs and outputs are processed sequentially. Nonetheless, the sizes of the inputs and intermediate outputs to operators could be quite large requiring large memory/storage; the following two approaches are used to address this issue.

A batching optimization to reduce memory

Notice that the `Proposal` operator is the only operator that may produce more output than it consumes as input; it can increase the memory usage of the system. This issue could be fixed with a simple batching optimization. Instead of producing all of the proposals for each P_j prefix they have, each `Proposal` operator produces its candidate extensions in batches of B'. The remaining extensions are produced in the subsequent invocations. This may leave some prefixes only partially extended. To keep track of these partial extensions, the algorithm has to store $(p, \textit{min-c}, \textit{min-i}, \textit{rem-ext})$ quadruples where *rem-ext* is the *remaining extensions* metadata. Assuming $B = wB'$ ensures that the dataflow has at most B queued elements at any time across the workers, as the B proposals created by `Proposal` operators are retired before any more are produced. The `Count` and `Intersect` steps remain unchanged.

A streaming implementation

The above optimization leads us to a streaming implementation, in which the algorithm only executes *Proposal* steps when their output queues are empty, and only produce at most B' outputs when it does so. The *Count* and *Intersect* operators are run whenever their inputs are non-empty, completely draining the output of *Proposal* before it can produce more output. This ensures that the dataflow has at most B' queued elements at any time, as the B' proposals created by *Proposal* are retired before any more are produced. In practice, this streaming implementation can improve performance.

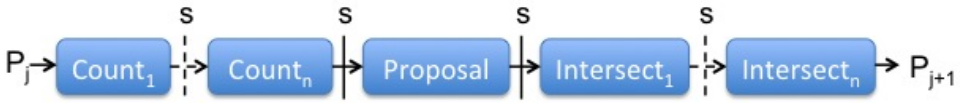


Figure 3.3: Dataflow Primitive.

3.2.2 BigJoin: Joins on Static Relations

Using the above dataflow primitive, it is possible to build a distributed algorithm that evaluates queries on static graphs, which is referred to as BigJoin, as follows. First, similar to GJ, query attributes are ordered arbitrarily and indices over each relation are built for each prefix of its attributes in the global order. Next, the dataflow extends each P_j to P_{j+1} for each attribute a_j, so that starting from an empty input tuple it produces streams of prefixes P_j, and eventually, $P_m = Q$. Finally, an empty tuple is used to start the computation, producing the stream of records from Q as output. The batching optimization described above is used, and when deciding which batch of P_j to P_{j+1} extensions to invoke next, the algorithm picks the largest j value such that at least one worker has B' prefixes to propose. The next observation analyses the costs of BiGJoin:

Observation 1. *Given a query Q over m attributes and n relations, the communication and computation cost of* BigJoin *equals that of computation of* GJ *and is $O(mnMaxOut_Q)$. Let B' be a batching parameter and let $B = wB'$, the cumulative memory* BigJoin *requires is $O(m\text{IN} + mB)$, and the number of rounds of computation* BigJoin *takes is $O(\frac{mnMaxOut_Q}{B'})$.*

Proof. Recall that each `Proposal` operator stores $(p, c, i, \textit{rem-ext})$ quadruples, where *rem-ext* is the metadata the operator keeps for prefix p to track the remaining amount of candidate extensions the operator has to do for p. This metadata is called the remaining *intersection work for p*. Assume each `Proposal` operator, for each P_j keeps track of the *cumulative intersection work* it has to do for the set of prefixes it has in P_j. Figure 3.4 shows the stages of BiGJoin's dataflow. Unlike Figure 3.3, it starts each dataflow from the `Proposal` operator and omit the dataflow extending P_0 to P_1. As discussed in Section 3.2, BiGJoin picks a P_j to extend to P_{j+1} prefixes, where each operator extends a subset of its prefixes up to at most B' extensions. Recall that BiGJoin picks the P_j with the largest j value (where j is from 1 to $m-1$) such that at least one `Proposal` operator has B' cumulative intersection work to do. Let P_{j^*} be the prefix set BiGJoin picks. If $j^* < m-1$ then the algorithm generates a batch of P_{j^*+1} prefixes. Otherwise if $j^* = m-1$, it produces and writes a batch of outputs. Assume throughout our analyses that the

operators of the dataflow run across different workers and use the terms operator and worker interchangeably. Moreover there is a synchronization between any two operators (not only the `Proposal` but also `Count` and `Intersect`) in the dataflow consisting of a round in MPC terms. This is actually needed in order to analyze our dataflow in MPC because in a single round of MPC, workers cannot perform computations on the data they receive. Therefore it is possible to assume there is a synchronization barrier between each arrow in Figure 3.4.

Observe that BiGJoin maintains two invariants: (1) At any point in time, each *Proposal operator* has at most $2B'$ prefixes (not candidate extensions) from each P_j; and (2) at each round of computation the amount of intermediate data due to candidate extensions being intersected is at most $O(wB')$. Initially both invariants hold because only one *Proposal operator* has the $P_0 = \{(\epsilon)\}$ prefix and its quadruple and other workers do not hold any prefixes.

At any point in time, since each `Proposal` operator extends at most B' candidates, and that the `Count` and `Intersect` operators do not generate more data than their inputs, and that there are w workers, the amount of intermediate data is bounded by wB', so the second invariant is satisfied. Note that the first invariant is also maintained because each `Proposal` operator w_i generates at most B' new P_{j^*+1} prefixes for itself. However, note that w_i must have less than B' P_{j^*+1} prefixes, because otherwise BiGJoin would have decided to extend P_{j^*+1} prefixes instead of extending P_{j^*} prefixes. This proves that the cumulative memory BiGJoin needs to store all of the P_j's is $O(mwB')$.

For BiGJoin's communication and computation costs, note that cumulatively BiGJoin performs exactly the same amount of computation as GJ. To see this, first note that the constant counting and test membership assumptions hold when `Intersect`, `Proposal` and `Count` operators perform appropriate operations on the Ext_j^i indices. Second, similar to GJ, for each prefix p, BiGJoin starts its intersections from the relation that has the minimum number of extensions. Even though BiGJoin can intersect the extensions of each prefix p in multiple batches, effectively each of the possible extensions gets intersected with at most n different relations incurring a computation cost of at most n. GJ similarly perform exactly same number of intersections for each of the candidate extensions of p. In addition, BiGJoin incurs an equivalent amount of communication cost when doing the intersections of each of the candidate extensions. This is because for each intersection the candidate extension is sent to another operator. Therefore, the cumulative computation and communication cost of BiGJoin is the same as the computation cost of GJ, which is $O(mnMaxOut_Q)$. For memory consumption, the two invariants above are satisfied. Therefore, beside the indexing cost, each worker holds at most $2B'$ P_j prefixes for each j. Therefore, the cumulative memory needed to store all of the P_j's is $O(mwB') = O(mB)$.

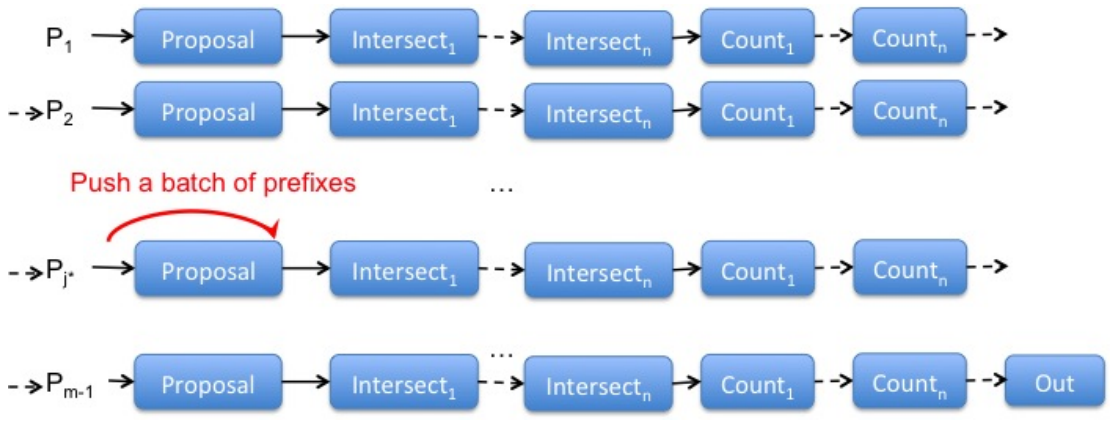


Figure 3.4: BiGJoin Dataflow.

The final part of this proof is to analyze the number of rounds of computation BiGJoin takes in MPC terms. There are at most $2n + 1$ operators in each dataflow primitive extending P_j to P_{j+1}, as there are at most n `Intersect` and n `Count` operators. Therefore an iteration of extending B' prefixes from one of the m P_j sets takes at most $2n+1$ rounds. Note that by the AGM bound, the size of each P_j set is at most $MaxOut_Q$. Therefore the number of rounds of computation is $O(\frac{mnMaxOut_Q}{B'})$, completing the proof. □

The proof of this observation is based on the fact that each operation that BigJoin does on each tuple corresponds to an operation in the serial execution of GJ and small enough batches can keep the memory footprint very low. In essence, BigJoin inherits its computation and communication optimality from GJ. Moreover, as demonstrated in Section 3.6, in practice BigJoin also achieves good workload-balance across the workers in the cluster. However, on adversarial inputs BigJoin cannot guarantee workload balance in theory, which will be addressed in the following section.

3.3 BigJoin-S: A Skew Resilient BigJoin

Although BigJoin is worst-case optimal in terms of its cumulative computation and communication costs and requires small cumulative memory, it can generate imbalances in terms of how these costs are distributed across the machines in the cluster. Therefore, it has an important theoretical shortcoming. Specifically, it does not guarantee that the workloads of the workers are balanced. Indeed, it is easy to construct skewed inputs where

most of the work could be performed by a single worker. Specifically, there are three sources of imbalance in BigJoin:

1. *Sizes of extension indices:* Recall that Ext^i_j are distributed randomly yet for each prefix p, a single worker stores the entire $Ext^i_j(p)$ (the a_{j+1} extensions of p). In graph terms, this corresponds to a single worker storing the entire adjacency list of a vertex. On skewed inputs, this may generate imbalances in the amount of data indexed at each worker.

2. *Number of Proposals:* After count minimization, each worker gets a set of $(p, \textit{min-c}, \textit{min-i}, \textit{rem-ext} = \textit{min-c})$ quadruples where p is a P_j prefix to extend. Even if each worker has to extend the same number of prefixes, each worker might have to do imbalanced amount of proposals of $(p \bullet e)$ candidate extensions because the counts might be very different.

3. *Number of Index Lookups:* When minimizing the counts of a P_j prefix p, producing the candidate proposals, or intersecting the $(p \bullet e)$ candidate extensions with Ext^i_j, prefixes and candidate extensions are routed to the worker that holds $Ext^i_j(p)$ based on the hash of p's attributes that are bound by R_i. If there are many prefixes whose bound attributes are the same, there may be an imbalance in the number of prefixes and extensions each worker receives. For example, consider a triangle query where all triangles involve some specific vertex v^*, then every P_j prefixes could be routed to a single worker to access v^*'s count.

Ammar et al. [9] modifies BigJoin, and proposes BigJoin-S, to ensure workload balance across workers[2]. Note that these modifications are only interesting for theoretical guarantees but do not lead to good performance in real-world graphs.

3.4 Delta-GJ Algorithm: Joins on Dynamic Relations

Since most real graphs are dynamic in nature, it is valuable to find newly emerged or deleted subgraphs based on graph changes. Since subgraph query Q can be expressed as multiway joins of the edges table of an input graph G, it is possible to use incremental view maintenance (IVM) techniques for *join views*, i.e., views corresponding to joins of multiple tables, to continuously detect changes to the output of Q. Doing so would detect the emerged and deleted subgraphs that match Q in G.

[2]The BigJoin-S algorithm is not part of this thesis.

As discussed in Section 3.1.1, *Binary Join* BJ has several inefficiencies when it is used to solve subgraph queries, specifically the cyclic ones, such as `Triangle`. Therefore, this section looks at using GJ or a similar algorithm to design an `IVM`-like algorithm that detects subgraph instances as they appear (or disappear) based on graph updates.

This section proposes DELTA-GJ which extends GJ to evaluate continuous subgraph queries. DELTA-GJ is based on IVM techniques from Blakeley et al. [14] and Gupta et al. [34] which derive a set of delta queries $dQ1, ..., dQn$ (originally referred to as *delta rules* [34]) for Q and evaluate each dQi as the input tables change to maintain the result of Q.

Although the idea of delta queries is not new, the evaluation of dQi using a worst-case join algorithm has theoretical implications that do not exist for typical incremental view maintenance algorithms that use delta queries, such as [14, 34]. Specifically, it is possible to show that under insertion-only workloads, if the delta queries are evaluated using the worst-case optimal GJ algorithm with specific attribute orderings, the total computation done to incrementally maintain the original join query is worst-case optimal up to constants that depend on the query Q.

This section starts by reviewing the technique of delta queries, and then describe the proposed DELTA-GJ algorithm, which evaluates these delta queries using the worst-case GJ algorithm.

3.4.1 Delta Join Queries

Let Q be a multiway join query and assume that one or more of the base tables are going to be updated at a particular point in time. Let ΔR_i, for each i, be a set of insertions or deleted tuples to relation R_i in this update. Assume that tuples in each ΔR_i are labeled such that we can tell the inserted tuples apart from the deleted ones. Let R'_i be $R_i \cup \Delta R_i$, where the union operation removes a tuple t in R_i if ΔR_i contains a deletion of t. Let Out and Out' be the output of Q before and after the updates to the base relations, respectively. Then consider the following n delta queries:

$$dQ_1 := \Delta R_1, R_2, R_3, ..., R_n$$

$$dQ_2 := R'_1, \Delta R_2, R_3, ..., R_n$$

$$dQ_3 := R'_1, R'_2, \Delta R_3, R_4, ..., R_n$$

$$...$$

$$dQ_n := R'_1, R'_2, R'_3, ..., \Delta R_n$$

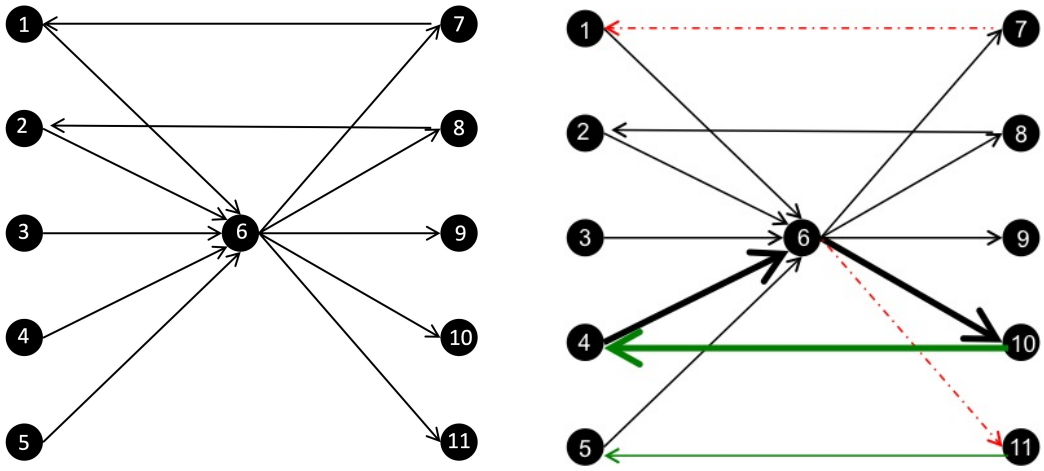
When evaluating each dQ_i if a deleted tuple from ΔR_i joins with tuples from the other $n-1$ relations and forms an output, dQ_i contains an output tuple that will be labelled as deleted. It can be shown [14, 34] that $Q' \setminus Q = dQ_1 \cup dQ_2 \cup ... \cup dQ_n$. That is, evaluating the union of the n queries above gives exactly the changes to the output of Q. The following example shows how this approach works.

Example 5. *Consider continuously evaluating the* `Triangle` *query from Section 3.1.1. Suppose initially the graph is as in Figure 3.2 with an extra edge (2, 8) (as shown in Figure 3.5a). This graph already has two groups of three symmetric (and directed) triangles (1, 6, 7), (6, 7, 1), (7, 1, 6) and (2, 6, 8), (6, 8, 2), (8, 2, 6). Consider adding edges (10, 4), (11, 5) and deleting the edges (6, 11) and (7, 1) from this graph. Figure 3.5b shows the new graph that forms after this update. In the figure, old edges are drawn in black colour, new edges are drawn in green colour, and the deleted edges are drawn in red colour with dashes. Note that in the new graph, there are again two groups of three symmetric triangles: (2, 6, 8), (6, 8, 2), (8, 2, 6), which already existed and (4, 6, 10), (6, 10, 4), (10, 4, 6), which emerged after the update. In Figure 3.5b, the edges of the emerged triangles are drawn thicker. It is expected to detect as part of the update three deleted triangles (1, 6, 7), (6, 7, 1), (7, 1, 6) and three newly emerged triangles (4, 6, 10), (6, 10, 4), (10, 4, 6).*

Figure 3.6 shows the edges, Δ edges, and edges' tables. Now consider executing the three delta queries that would correspond to the triangle query. Since each table in our example has the same name, we name the delta queries as $dQ1$, $dQ2$, and $dQ3$.

- $dQ1(v_1, v_2, v_3) : -\Delta edges(v_1, v_2), edges(v_2, v_3), edges(v_3, v_1)$*: The output of* $dQ1$ *is {(7, 1, 6)-, (10, 4, 6)+, (11, 5, 6)+}, where +/- indicate whether the output tuple is inserted or deleted, corresponding to whether the triangle has emerged or has been deleted. Notice that (11, 5, 6) is actually not a triangle that actually has emerged.* $dQ3$ *will have the (11, 5, 6)- output, so when the union of* $dQ1$ *and* $dQ3$ *outputs will not include the (11, 5, 6) tuple in the final changes to the output.*
- $dQ2(v_1, v_2, v_3) : -edges'(v_1, v_2), \Delta edges(v_2, v_3), edges(v_3, v_1)$*: The output of* $dQ2$ *is {(6, 7, 1)-, (6, 10, 4)+}.*
- $dQ3(v_1, v_2, v_3) : -edges'(v_1, v_2), edges'(v_2, v_3), \Delta edges(v_3, v_1)$*: The output of* $dQ2$ *is {(1, 6, 7)-, (4, 6, 10)+, (11, 5, 6)-}.*

The union of the outputs of $dQ1$, $dQ2$, *and* $dQ3$ *gives exactly { (1, 6, 7)-, (6, 7, 1)-, (7, 1, 6)-, (4, 6, 10)+, (6, 10, 4)+, (10, 4, 6)+} as expected.*



(a) Graph before updates (b) Graph after updates

Figure 3.5: Input graph to DELTA-GJ

a_i	a_j	+/-
6	11	-
7	1	-
10	4	+
11	5	+

(a) Δedges table

a_i	a_j
1	2
1	6
2	6
2	8
3	6
4	6
5	6
6	7
6	8
6	9
6	10
6	11
7	1

(b) edges table.

a_i	a_j
1	2
1	6
2	6
2	8
3	6
4	6
5	6
6	7
6	8
6	9
6	10
10	4
11	5

(c) edges' table.

Figure 3.6: Example input graph and its edge table.

Cost Analysis

This section shows a theoretical evaluation for DELTA-GJ using a theorem and its proof. The next chapter will present an empirical evaluation for the parallel implementation of GJ (called BIGJOIN) and the parallel implementation of DELTA-GJ (Called DELTA-BIGJOIN).

Under insertion-only workloads, DELTA-GJ's cumulative computation cost at any point in time is worst-case optimal up to constants that depend on the query. The following notations are important for the theorem and proof. Consider a query Q and consider a series of updates, $\Delta(1), \Delta(2), ...$, that modify the input relations R_i of Q. $\Delta R_i(z)$ is the

set of updates to R_i in $\Delta(z)$. $R_i(z)$ is the state of relation R_i after incorporating all of the updates until and including $\Delta R_i(z)$, i.e., $R_i(z) = \Delta R_i(1) \cup \Delta R_i(2) \cup ... \cup \Delta R_i(z)$. This leads to the following theorem.

Theorem 3.4.1. *Consider a query Q and a series of z updates that only consist of inserting tuples to the input relations of Q. Then the total computation cost of* Delta-GJ *is $O(mn^2 MaxOut_Q)$, where $MaxOut_Q$ is the AGM bound of Q on $R_i(z)$.*

Proof. Delta-GJ evaluates each dQ_i z times, one for each update. For each dQ_i it is possible to show that the cumulative cost of these z evaluations is less than running GJ on $R_i(z)$ with the same attribute ordering Delta-GJ uses for dQ_i. Let the query that GJ evaluates on the final relations be $Q(z) := R_1(z), ..., R_n(z)$. Note that by Theorem 3.1.1, the cost of running GJ with any attribute ordering is $O(mnMaxOut_Q)$. The claim of the theorem then follows from the fact that there are n delta queries. The three Delta-GJ cost factors when evaluating dQ_i z times are:

1. cost of indexing,
2. cost of computing P_{r_i} tuples, and
3. cost of computing the rest of $P_{r_i+1}, ..., P_m$.

For cost (1), note that the cost of indexing each tuple t in $\Delta R_i(1), ..., \Delta R_i(z)$ first into r_i Δ-$Ext^i_{v_{x_{ik}}}$ and then into r_i $Ext^i_{v_{x_{ik}}}$ indices does not asymptotically change the indexing cost of running GJ when evaluating Q on $R_i(z)$ (it only doubles the cost).

For cost (3), note that once P_{r_i} have been computed, the cost of Delta-GJ for extending any tuple $t \in P_{r_i}$ to P_j for $r_i < j \leq m$ is clearly less than the cost that GJ incurs when extending t when GJ fixes the same global attribute ordering as Delta-GJ. This is because by definition, $dQ_i := R_1(z), R_2(z), ..., dR_i, R_i(z-1), ..., R_n(z-1)$ and each relation in dQ_i is a subset of its corresponding relation in $Q(z)$

Finally for cost (2). Recall that Delta-GJ incurs a cost of $O(mn)$ for each tuple $t \in \Delta R_i$ to verify whether t is in P_{r_i} or not. Cumulatively across all of the z updates, these checks incur an extra cost of $O(mn|R_i(z)|)$. Moreover, cumulatively across all of the dR_i, this cost is $O(mn\text{IN})$. We note that in the best case $MaxOut_Q$ is $\Omega(\frac{\text{IN}}{n})$, since the largest relation is of size at least $\frac{\text{IN}}{n}$ and the worst-case output of any query is at least the size of its largest relation. Therefore $mn^2 MaxOut_Q$ is at least $mn\text{IN}$, which implies that $O(mn\text{IN})$ is a cost that is subsumed by $O(mn^2 MaxOut_Q)$, completing the proof. □

Finally, note that the meaning of worst-case optimality is not well defined when considering deletions. That is because if $\Delta(1)$ inserts a large number of tuples to each relation R_i. Then, $\Delta(2)$ removes all of these tuples. Then since the input relations will be empty, $MaxOut_Q$ after updating 2 will be ϕ, yet clearly, any IVM algorithm needs to detect the emergence and deletions of all of the tuples at the end of the first update. Therefore, the theoretical analysis of Delta-GJ under general workloads that consist of both insertions and deletions is omitted. However, if the definition of worst-case optimality for general workloads is evaluated for each update independently, and the target is to detect the emergence or deletions of subgraphs in the query output, then the above theory and proof suffice to show Delta-GJ worst-case optimality guarantees on general workloads.

3.4.2 Delta-BigJoin: Distributed Delta-GJ

To create a distributed version of Delta-GJ, Delta-BigJoin has a separate dataflow for each dQi that is a dQ_i-specific variation of the BigJoin dataflow from Section 3.2.2. By ordering the attributes of dQ_i starting with the attributes of R_i, we can seed the computation with the elements of ΔR_i, which is expected to be much smaller than the other relations in dQ_i. The remaining attributes are ordered arbitrarily.

Since there is an independent dataflow for each dQ_i, changes to any relation can be routed to the appropriate delta query dataflow. Note that Delta-BigJoin only needs to *maintain* the indices as changes occur, rather than fully rebuilding them. The resulting cost is proportional to the number of changes (for rebuilding indices) and the number of prefixes in the delta queries. Furthermore, for each tuple t, in parallel, in ΔR_i, the lookups in the extension indices to verify whether t is in P_{r_i} are replaced by distributed lookups, incurring at most $O(mn)$ communication, computation, and rounds. The following theorem and proof describe Delta-BigJoin's costs.

Theorem 3.4.2. *Consider a query Q and a series of z insertion-only updates to the input relations of Q. Let $\mathrm{IN}(z)$ denote $\sum_i |R_i(z)|$. Then, given a batch size B' on w workers such that $B = wB'$:*

- Delta-BigJoin*'s communication and computation cost under insertion-only workloads is $O(mn^2 MaxOut_Q)$;*
- Delta-BigJoin*'s cumulative memory is $O(mn\mathrm{IN}(z) + mB)$;*
- *The number of rounds of computation* Delta-BigJoin *takes is $O(mn^2 \frac{MaxOut_Q}{B'} + zmn^2)$.*

Proof. Note that by Thereom 3.1.1 BigJoin's communication and computation cost on any query Q is asymptotically same as the computation cost of running GJ on Q, which is $O(mnMaxOut_Q)$. By the same arguments in Theorem 3.4.1's proof, the communication and computation cost of Delta-BigJoin is therefore $O(mn^2MaxOut_Q)$.

The cumulative memory cost of Delta-BigJoin is never larger than the memory cost of running BigJoin on $Q(z)$ which is $O(mn\text{IN}(z) + mB)$. This is because at any point in time, the indices that Delta-BigJoin uses is at most as large as the indices that BigJoin uses on $Q(z)$. Similarly, the intermediate data that Delta-BigJoin generates on any dQ_i query is at most as large the intermediate data that BigJoin generates on $Q(z)$.

For the number of rounds, note that after each update, Delta-BigJoin runs an extra $O(mn)$ rounds of computation to compute P_{r_i} for each update, making a total of extra $O(zmn)$. The algorithm will extend prefixes and perform intersections using the same batch size of B' as BigJoin does. This means that Delta-BigJoin, for each dQ_i, computes the same P_{r_i} as BigJoin does on $Q(z)$. Therefore, for each dQ_i, cumulatively across all of the z updates, Delta-BigJoin runs the same number of rounds as BigJoin does, which is $\frac{mnMaxOut_Q}{B'}$, and an additional $O(zmn)$ rounds. Since there are n delta queries, the total number of rounds of computation is $O(mn^2\frac{MaxOut_Q}{B'} + zmn^2)$, completing the proof. □

3.5 Implementation

This section describes the implementations of BigJoin and Delta-BigJoin in Timely Dataflow. Although these implementations are tailored for evaluating subgraph queries, and the input relations are the same and consist of the edges of an input graph, which are represented as a binary relation, the underlying machinery nonetheless is suitable for more general queries. This section starts by developing the prefix extension dataflow primitive as a Timely fragment.

3.5.1 Prefix Extension in Timely Dataflow

The approach to prefix extension follows the primitive from Section 3.2.1: it will assemble a dataflow fragment that starts from a stream of prefixes of some number j of attributes, and produces as output the corresponding stream of prefixes resulting from the extension of each input prefix by the relations constraining the attribute a_{j+1} in terms of the first j attributes. As explained in Section 3.5.3, regarding Delta-BigJoin implementation, the

prefixes are tagged with a timestamp and a signed integer, reflecting the time of change and whether it is an addition or deletion, respectively.

Prefix extension happens through three methods acting on streams, corresponding to the three steps described in Section 3.2.1: count minimization, candidate proposal, and intersection. Each of these steps is implemented as a sequence of operators, each of which corresponds to one of the relations constraining attribute a_{j+1}. Each operator will consult some indexed form of the relation it represents, and requires the prefixes in its input stream to be shuffled by the corresponding attribute, so that prefixes arrive at the worker that store the appropriate fragment of the index. Importantly, the same partitioning is used for each relation and attribute in that relation, so that any number of uses of the relation in the query require only one physical instance of each index. In the case of graph processing, this means it only keeps a forward and reverse index, storing respectively the outgoing and incoming neighbours of each vertex.

Count Minimization

The implementation of this step is straightforward and follows the description in Section 3.2.1 directly. There is a sequence of operators and each one represents one relation $R_i(a_{j+1}, a_k)$ or $R_i(a_k, a_{j+1})$, where $k \leq j$. The operator takes (p, c, i) triples as input. Let $v^* = \Pi_{a_k} p$. The operator updates the count c if the size of v^*'s outgoing neighbours (if $R_i = R_i(a_{j+1}, a_k)$), or incoming neighbours (if $R_i = R_i(a_k, a_{j+1})$), is less than c. At the end the $(p, \textit{min-c}, \textit{min-i})$ triples have been identified but then send only p to a stream for R_i (explained next).

Candidate Proposal

This step is implemented by a single operator that divides its stream of input prefixes into one stream for each relation R_i, by the *min-i* index identified in the previous stage. Suppose $R_i = R_i(a_{j+1}, a_k)$. Then an input p whose *min-i* was i, where $v^* = \Pi_{a_k} p$, will be part of the stream for R_i and be extended to a tuple $(p \bullet \{e_1, ..., e_c\})$ containing the *set of candidate extensions*, which are v^*'s outgoing neighbours. When $R_i = R_i(a_k, a_{j+1})$, the incoming neighbours of v^* are used instead. This slightly deviates from the original description that flattened this tuple for simplicity of explanation and had c separate $(p \bullet e)$ candidate extensions. These extensions are sent through a single output stream for the next stage.

Intersection

The stream of pairs of prefix and candidate extensions go through a sequence of operators, one for each involved relation $R_{i'}$, each of which intersects the set of candidate extensions with an appropriate neighbour list of a vertex and removes those extensions that do not intersect. The result is a stream of pairs of prefix and valid extensions, successfully intersected by all relations. The extensions are flattened to a list of prefixes for the next stage except if they are the final outputs, they are output in their compact representation.

3.5.2 The BiGJoin Dataflow

The dataflow for enumerating subgraphs in a static graph applies a sequence of prefix extension stages, each corresponding to an attribute in the global attribute order. For simplicity, the global order is fixed so that the first two attributes are connected by an edge, which allows the algorithm to seed the stream of prefixes with length-two prefixes read from the edges themselves. This is equivalent to starting the extensions from P_2 instead of the empty tuple $() \in P_0$. All other attributes are extended using the prefix extension dataflow fragment described above.

The indices used by the workers are static, and is simply memory-maped in a pre-built index. For simplicity the whole graph is used, which means it is possible to easily vary the number of workers without changing the used index. One could alternately partition the graph and provide each worker with its own index, but the graphs used in the evaluation section for static computations are rather small. For larger graphs, such as those considered with Delta-BigJoin, the indices are built as part of the computation, distributing the data to only the workers that require it.

The execution of the BigJoin dataflow happens in batches, where some number of prefixes are fed into the dataflow and await their results before introducing more prefixes. As discussed in Section 3.2.1, this batching allows some control over the peak memory requirements.

3.5.3 The Delta-BigJoin Dataflow

The dataflow for finding subgraphs in a dynamically changing graph is more complex than for a static graph, along a few dimensions. First, as described in Section 3.4.2, it has an independent dataflows for each dQ_i. Each dataflow is responsible for changes to each

logical relation R_i in the query, i.e., one for each of the edges in the subgraph query. Second, although these dataflows may execute concurrently, the implementation will logically sequence them so that each dataflow computes the delta query as if it was executed in sequence (to resolve simultaneous updates correctly). Third, the index implementation will be more complicated, as it must support changes as well as the multi-versioned interaction required by the logical sequencing above.

There is a dataflow for each dQ_i, each of which use a different global attribute order as described in Section 3.4.2. Although there are several dataflows with different attribute orders, each operator only requires access to either the *forward* or *reverse* edge index.

Each delta query dataflow dQ_i computes changes in the outputs made to relation R_i with respect to the other relations. Recall that dQ_i uses the "new" versions $R'_i = R_i + \Delta R_i$ for $i < j$ and the "old" versions R_i for $i > j$. This has the effect of logically sequencing the update rules, so that they are correct even if there are simultaneous updates to the input relations, something that is expected in graph queries where the single underlying edges relation is re-used often. This use of new and old versions of the same index requires the implementation to be multi-versioned, in order to have a single copy of each index.

The index implementation is a multi-version index, which tracks the accumulation of (src, dst) pairs at various times and with various integer weights. The index can respond to queries about the outgoing and incoming neighbours for a given key v and a given timestamp. Updates at a particular timestamp τ are "finalized" when all tuples in the system have a timestamp greater than τ. This means these updates will participate in all future accumulations for v; this information comes from Timely Dataflow's progress tracking infrastructure.

The execution of the Delta-BigJoin dataflow proceeds with the stream of batches of updates to the graph supplied as an input. Each of the tuples moving through a delta query dataflow has both a logical timestamp and a signed integer weight. The former allows the algorithm to work with multiple logical times concurrently, and to remain clear on which version of an index the prefix should be matched against. The integer weight allows the index to represent both additions and deletions from the underlying relations.

3.6 Evaluation

This section shows an empirical performance study of the Timely Dataflow implementations of BigJoin (BigJoin-T) and Delta-BigJoin (Delta-BigJoin-T) using a variety of

subgraph queries and large-scale static and dynamic input graphs. After describing the experimental setup, this section is organized as follows:

- **Section 3.6.2** evaluates a reference computation (triangle finding) on several standard graphs using a few different systems, to establish a baseline for running time. For all systems, the capacity limitation has been discovered; they struggle to load graphs at the larger end of the spectrum.
- **Section 3.6.3** studies the scalability of BigJoin and Delta-BigJoin while increasing the number of workers both within a single machine as well as across multiple machines on a 64 billion-edge graph.
- **Section 3.6.4** demonstrates that several efficient optimizations that have been introduced in prior work can also be integrated into the proposed algorithms to improve their performance.
- **Section 3.6.5** studies the effects of the batch size on performance and memory usage. Unless specified explicitly, the default batch size in all experiments is $100,000$.

3.6.1 Experimental Setup

Table 3.1 reports statistics of the graphs used for evaluation. The sizes range from the relatively small but popular LiveJournal graph, with 68 million edges, up three orders of magnitude to the relatively large Common Crawl graph, with 64 billion edges. The abbreviations used for the datasets are given in parentheses in Table 3.1. There are five queries used in this evaluation:

- `triangle`:= $e(a_1,a_2),e(a_1,a_3),e(a_2,a_3)$
- `4-clique`:= $e(a_1,a_2),e(a_1,a_3),e(a_1,a_4),e(a_2,a_3),e(a_2,a_4),e(a_3,a_4)$
- `diamond`:= $e(a_1,a_2)$, $e(a_2,a_3),e(a_4,a_1)$, $e(a_4,a_3)$
- `house`:= $e(a_1,a_2),e(a_1,a_3),e(a_1,a_4),e(a_2,a_3),e(a_2,a_4),e(a_3,a_4)$, $e(a_2,a_5),e(a_3,a_5)$[3]
- `5-clique`:= $e(a_1,a_2),e(a_1,a_3),e(a_1,a_4),e(a_1,a_5),e(a_2,a_3),e(a_2,a_4)$, $e(a_2,a_5),e(a_3,a_4)$, $e(a_3,a_5),e(a_4,a_5)$

[3]This is query $q6$ from the SEED [48] and is a 5-clique with two missing edges from one node.

Name	Vertices	Edges
LiveJournal (LJ) [46]	4.8M	68.9M
Twitter (TW) [76]	42M	1.5B
UK-2007 (UK) [76]	106M	3.7B
Common Crawl (CC) [83]	1.7B	64B

Table 3.1: Graph datasets used in our experiments.

Note that the Common Crawl dataset has prohibitively large number of instances of each query. For example, there are approximately more than 2.36×10^{16} diamonds in Common Crawl, and enumerating all of them explicitly would take a prohibitively long time for any correct system. Instead, the focus is on *incremental maintenance* of these queries, which can fortunately be performed without the initial computation of all answers.

For all experiments except one a local cluster of up to 16 machines is used. All machines have 2x Intel E5-2670 @2.6GHz CPU with 16 physical cores in total. Most machines have 256 GB memory, but a machine with 512 GB memory to accommodate single-machine experiments was occasionally used. Each machine has 10 Gigabit network interface. For experiments using EmptyHeaded (see Section 3.6.2), an AWS machine similar to the local cluster machines (r3.8xlarge) was used along with another machine with 1TB memory (x1.16xlarge) to accommodate EmptyHeaded triangle query on the TW graph.

In all experiments, each Timely worker was assigned to one CPU core. The description of each experiment explicitly states how the workers are located, i.e., within a single machine, across machines, or both.

3.6.2 Baseline measurements

The goal of this section is to assess whether BigJoin-T has relatively good absolute performance when evaluating queries in static graphs. There are three possible baselines considered: (1) a single threaded implementation (Section 3.6.2); (2) a shared-memory parallel system, EmptyHeaded, (Section 3.6.2); and (3) a distributed system, Arabesque, (Section 3.6.2). All of these implementations operate only on static graphs. None of these implementations are capable of working with the largest graph, and not all of them can evaluate our smaller graphs either. Note that there is no baseline comparison with Shares [5] because Shares-like algorithms have been designed primarily for theoretical analysis. There is no baseline available system that could be used as a baseline, implementing the Shares algorithm.

Figure 3.7: BiGJoin and Delta-BiGJoin counting triangles in the Twitter graph, plotted with the time it takes our single-threaded implementation. Both approaches outperform the single-threaded implementation with small number of cores, and continue to improve from there. The Delta-BiGJoin performance lags slightly behind, as it uses more complex data structures to support updates.

COST

Configuration that Outperforms a Single Thread (COST [52]) is a metric to evaluate the parallelism overheads of an algorithm or a system. Specifically, COST of a parallel algorithm A solving a problem P is the number of cores that the algorithm needs to outperform an optimized single-threaded algorithm solving P. A small COST indicates that the system itself introduces little overhead, and the benefits of scaling are immediately realized.

To measure the COST of BigJoin while solving the `Triangle` query, an optimized single-threaded triangle enumeration algorithm, based on GJ in Rust [67], is used. Figure 3.7 shows the single-threaded GJ, BigJoin-T, and Delta-BigJoin-T for the `Triangle` query. Note that Delta-BigJoin-T can find all triangles in a static graph, starting from an empty graph and then considering all graph edges as updates to the Delta-BigJoin-T. However, a dynamic algorithm is expected to be slower than loading the whole graph first and then finding triangles. The COST of the two implementations is 2 and 4 cores, respectively.

Query	EH (R)	EH (I)	BigJoin-T (R)	BigJoin-T (I)
Triangle-LJ	1.2**s**	150.3**s**	6.5s	1.9s
Diamond-LJ	31.7**s**	150.3**s**	712.3s	1.9s
Triangle-TW	213.8**s**	4155**s**	588s	34.4s

Table 3.2: Comparison against EmptyHeaded. "(R)" and "(I)" indicate runtime and index size, respectively. EmptyHeaded's absolute performance is better on a single machine. However, the index building time can be non-trivial.

EmptyHeaded

EmptyHeaded's technical details was discussed in Section 3.1.5. As mentioned before, its optimizer uses *generalized hypertree decompositions* of the query to join multiple subsets of the relations using GJ and then join these subsets using BJ. However, for the queries studied in this paper, EH uses a pure GJ plan like BigJoin based simply on attribute ordering.

To guarantee a fair comparison with EH, this experiment runs using the AMI machine provided by EH team. An initial experiment was performed using a machine with a similar configuration as our cluster machines[4], however, EH ran out of memory when running the triangle query on TW. Therefore, this experiment ran using an x1.16xlarge AWS machine with 64 cores and 976 GB memory. The TW and LJ datasets and the triangle and diamond queries were used. Unfortunately, EH ran out of memory on the diamond query on TW. Table 3.2 reports two metrics for EH and BigJoin-T: (1) the runtime; and (2) the time to index the input data.

Note that the goal of this experiment is to evaluate EH as a high-quality reference implementation. It is expected that BigJoin-T performs worse than EH due to lack of specific optimizations for static datasets, such as compacting dense extension lists into bit vectors. In exchange, BigJoin-T is able to distribute across multiple machines and respond to changes in input, but this generality comes at a price. This experiment also evaluates EH's index build time and memory footprint, something EH is explicitly not optimized for, which combined with a lack of distribution limits EH evaluation on larger datasets.

[4] An r3.8xlarge AWS machine with 244 GB memory and 32 cores.

Query	Arabesque (R)	Arabesque (I).	BigJoin-T (R)	BigJoin-T (I)
Triangle	69.0s	1.46B	**3.4s**	**38M**
4-clique	273.7s	18.7B	**21.8s**	**350M**

Table 3.3: Comparison against Arabesque. BigJoin-T is faster and considers fewer candidate subgraphs than Arabesque.

Arabesque

Arabesque has been discussed in Section 3.1.5. This experiment was run using Arabesque's version (1.0.1-BETA) which runs on Giraph. On the local cluster, Arabesque was only able to load the LJ dataset and ran out of memory on other datasets. Throughout this experiment, the triangle and 4-clique queries were used with 8 machines, each running one Arabesque worker, and each worker using 16 cores. Both run-time and intermediate prefixes were measured. The code for triangle and 4-clique queries was provided by the authors of the system but was amended to not output any intermediate prefixes or final output[5].

Table 3.3 reports the running times, in comparison to BigJoin-T, as well as the number of intermediate results considered, which partly explain the running times. Arabesque considers roughly 30× more prefixes than BigJoin-T, which manifests as between 10x and 20x higher running times.

3.6.3 Capacity and Scaling

When a graph fits in the memory of a single machine, the straightforward parallelization strategy of replicating the graph to each machine should work very well in practice. That is why one of BigJoin's primary goals is to scale to graphs (and datasets) whose collected indices does not fit in the memory of a single machine.

Very large graphs can contain prohibitively many instances for even the simplest queries. For example, it is estimated that there are over 9 trillion triangles and 23 quadrillion (2.3×10^{16}) diamonds in Common Crawl[6]. The goal is therefore not to evaluate BigJoin when computing all subgraph instances, but Delta-BigJoin's throughput and capacity

[5]Note that this code used *VertexInducedEmbeddings* of Arabesque, which extend prefixes by one vertex but internally by considering each edge separately.

[6]These estimates are based on the number of triangles and diamonds found per edge in the incremental experiments, which are 143 and over 368K, respectively.

when maintaining these queries under updates. Note that DELTA-BIGJOINdoes not require that BIGJOIN runs first on the initial dataset; it only requires to index the initial dataset and can immediately start reporting changes in subgraph query matches as a function of changes to the source dataset.

For this experiment, the `Triangle` query and the Common Crawl dataset were used. This dataset has 64B edges and is roughly 1,000x larger than LJ, 50x larger than TW, and 20x larger than UK. Note that when each node ID requires 4 bytes, the graph requires ≈512GB written as a list of edges (u, v), and ≈ 256GB as an adjacency list. DELTA-BIGJOIN indexes edges in both directions, therefore, it requires ≈512GB.

In this experiment, various fractions of the edges in the graph are initially loaded, ranging from one-sixteenth to the full graph, and evaluate DELTA-BIGJOIN-T on a range of one to sixteen machines, each machine uses 14 workers only. The total number of cores/workers range from 14 to 224. Each subset of the graph results in a scaling curve as the number of machines increases. Note that DELTA-BIGJOIN-T requires an increasing number of machines to start the experiment as the size of the subsets grow. The number of edges indexed on each machine, the peak memory required, and the throughput of changes (both input and output) are tracked for each configuration.

Figures 3.8, 3.9, 3.10 show the scaling results on this large graph. For each fixed subset of the graph, additional workers both improve the throughput and reduce the per-machine index size and memory requirements. The plot of maximum index size (across all machines) indicates that as the amount of data and number of workers are doubled, the maximum size stays roughly fixed and at 8 billion, which is roughly sizes of the total tuples divided by the number of machines. This indicates effective balance despite some vertices with very high degree (the largest out-degree is ≈45 million). With the exception of the smallest dataset on the largest number of machines, throughput increases and peak memory requirements decrease with further machines; however, as the work gets progressively more thin (one sixteenth of the graph spread across 224 workers) system overheads do begin to emerge.

Table 3.4 also reports the throughput and the peak memory required when running the diamond and 4-clique queries when loading the full graph and using 224 workers. Note that there is a substantially lower throughput of input changes, but a relatively similar throughput for output changes. That is, each input edge changed results in substantially more subgraph matches changed, and it is the volume of output that limits DELTA-BIGJOIN-T throughput.

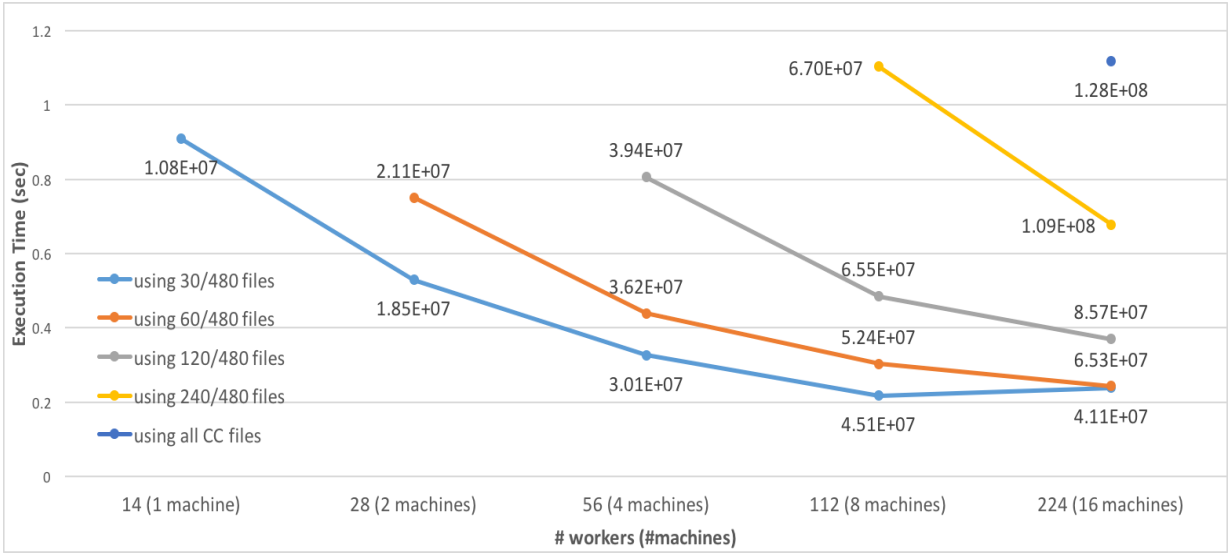


Figure 3.8: Scaling while increasing machines (and workers) and the initial graph input. Each line represents an experiment where the system pre-load an indicated fraction of the CC dataset, and then performs 20 rounds of 1M input edge updates for a triangle-finding query. This figure shows the **execution time**. Data points are average times to perform twenty batches of one million updates. The numbers by each data point report the number of output changes per second (triangles changed). The computation processes roughly 1M updates per-second, reporting between 10M and 100M changed triangles per second.

Query	Average Time	Output Throughput	Max. Mem.
4-clique	226.378 **s**	$46,517,875$ /**s**	108.4 **GB**
Diamond	276.587 **s**	$26,681,430$ /**s**	92.6 **GB**

Table 3.4: Common Crawl experiments. Sixteen machines load 64 billion edges, index them, and track motifs in 20 batches of $10K$ random edge changes. Although the input throughput is much lower than for triangles, the *output* throughput remains relatively high at tens of millions of observed subgraph changes per second.

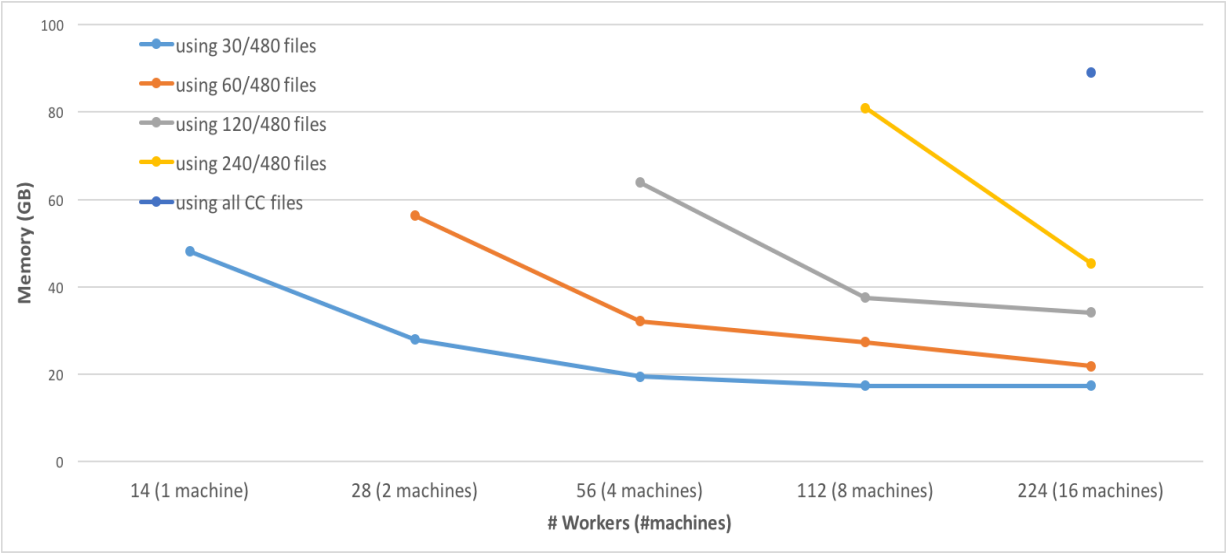


Figure 3.9: Scaling while increasing machines (and workers) and the initial graph input. Each line represents an experiment where the system pre-load an indicated fraction of the CC dataset, and then performs 20 rounds of 1M input edge updates for a triangle-finding query. This figure shows **maximum memory**, in gigabytes per machine. The peak occurs in initial index building rather than steady-state execution. The maximum does increase as the workers and input size are doubled, but this appears to be due to skew in data loading.

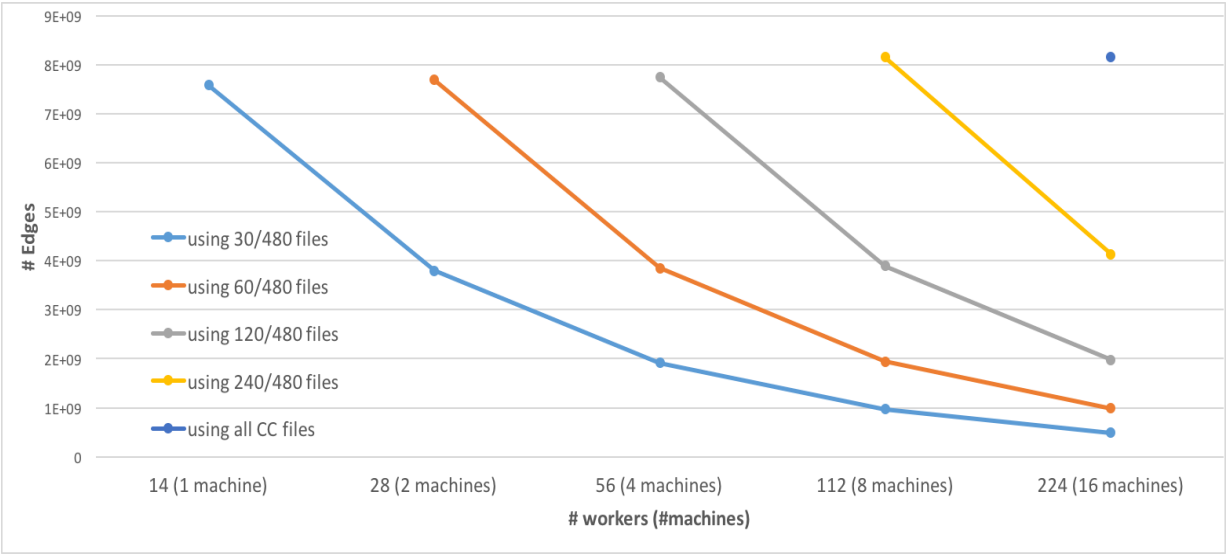


Figure 3.10: Scaling while increasing machines (and workers) and the initial graph input. Each line represents an experiment where the system pre-load an indicated fraction of the CC dataset, and then performs 20 rounds of 1M input edge updates for a triangle-finding query. This figure shows the **maximum index size per machine**, in total index tuples per machine. Index size decrease roughly linearly with additional machines at each scale.

3.6.4 Generality and Specializations

This section shows that BigJoin-T and Delta-BigJoin-T can employ existing optimizations from subgraph queries and multiway joins literature. The goal is two-fold: First, to compare these algorithms to a similar implementation in the literature known as SEED [48], which develops efficient optimizations for evaluating in undirected subgraph queries in the distributed setting. Second, implementing one of their optimizations demonstrates that the BigJoin approach can take as input general relations instead of the binary `edge(`a_i`,` a_j`)` relations that have been used so far. The following three optimizations have been implemented:

- **Symmetry Breaking:** SEED imposes constraints on vertex IDs to break symmetries. For example, the `4-clique` query might be constrained such that $a_1 < a_2 < a_3 < a_4$. One can be more efficient by first ordering by *degree*, and then by ID if there are ties. This allows finding each undirected four-clique once instead of 24 times, for each permutation of the vertices in the clique. This is commonly accomplished by giving new IDs to the vertices so that they are ordered by degree, and edges point from vertices with lower ID to higher ID. This optimization has been incorporated by transforming the input dataset, and supporting inequality constraints (which are just filters applied to intermediate prefixes).

- **Triangle Indexing:** SEED builds index structures over small non-trivial subgraphs, such as triangles. These indices provide more direct access to relevant vertex IDs reflecting multiple constraints already imposed. The ideas are similar to the recent FAQ work [4], which identifies some common subqueries in larger queries (for example, triangles in a four-clique query) and materializes these subqueries. This optimization has been incorporated by first finding all the triangles in the graph and then writing these as a ternary relation $\texttt{tri}(a_i, a_j, a_k)$. Since BigJoin-T supports general relational queries and can index general relations, it is possible to index $\texttt{tri}(a_i, a_j, a_k)$ by (a_i, a_j) and provide efficient random access to vertices a_k that complete a triangle with (a_i, a_j). Using the `tri` relation, `4-clique` query simplifies to:

  $$\texttt{tri}(a_1, a_2, a_3), \texttt{tri}(a_1, a_2, a_4), \texttt{tri}(a_1, a_3, a_4).$$

  This rewriting reduces the complexity of the query, and results in fewer intermediate prefixes explored. It is important to note that this is not precisely the same optimization SEED does. SEED indexes triangles by a_1 so that full neighbourhoods of each vertex is available, revealing large cliques at once. The implemented optimization is

Query	SEED-O	BigJoin-T	BigJoin-T-SYM	BigJoin-T-SYM-TR
4-clique	60s	54.0s	43.4s	13.3s
house	1013s	370.0s	294.3s	74.1s
5-clique	1206s	2861.1s	2153.2s	315.7s

Table 3.5: Comparison with SEED, against three BigJoin-T variants including several optimizations: breaking symmetry by renaming vertices by degree (-SYM) and then re-using pre-computed triangles (-TR). BigJoin-T's absolute performance is comparable to optimized approaches, and improves as optimizations are applied.

closer in spirit to the FAQ work, but demonstrates the utility of supporting general relations in evaluating subgraph queries.

- **Factorization:** The `house` query is amenable to a technique called *factorization* [60], which expresses parts of the query results as Cartesian products. In the `house` query, (a_2, a_3, a_4, a_5) form a clique and the missing edges are (a_1, a_4) and (a_1, a_5). A system can first computes the triangle (a_2, a_3, a_4) and then perform two independent extensions to the lists of a_1 and a_5 values. As these two variables do not constrain each other, they can be left as lists rather than flattened into the list of their cartesian product. SEED proposes a similar optimization (named SEED+O) in which large cliques are kept as cliques, rather than explicitly enumerating all bindings to variables. This optimization can only be utilized for the `house` query.

Table 3.5 compares SEED+O (SEED with clique optimizations) measurements taken from their paper with three variations of BigJoin: (i) vanilla BigJoin-T, (ii) BigJoin-T with symmetry breaking (BigJoin-T-SYM), and (iii) BigJoin-T with symmetry breaking and triangle indexing (BigJoin-T-SYM-TR). All of `house` query measurements also contain the factorization optimization. This experiment used 10 machines with 16 cores, which is a cluster setup similar to the one used in the SEED paper. Table 3.5 demonstrates two things: (1) BigJoin's algorithm implementations have the flexibility to employ several optimizations from prior work to become more efficient; and (2) The results of the 4-clique and 5-clique queries demonstrate that these implementations are initially competitive with SEED using the same resources, and when incorporating some of their optimizations, they can even outperform it.

3.6.5 Senitivity to Batch Size

This section evaluates the effects of the batch size on BigJoin and Delta-BigJoin. Batch size affects two aspects of these algorithms. First, very small batch sizes can impede parallelism. As an extreme example, consider finding all instances of a subgraph in a graph with a batch size of 1. Then at least initially only one worker in the cluster will do count minimization, candidate proposals, and intersections. Second, with larger batch sizes, the algorithm is expected to use more cluster memory. Therefore it is expected that as batch sizes get larger, runtime improves because the algorithm can parallelize better but after reaching a large enough batch size, it is expected the algorithm reaches a stable runtime but uses more memory.

To test this, the triangle query has been used and while running Delta-BigJoin-T on the UK graph using 16 workers on 1 machine and using batch sizes of 10, 100, 1K, 10K, 100K, 1M, and 10M. The experiment starts by loading the dataset, then ran Delta-BigJoin-T using a total of 10M edges and different batch size. The results are shown in Figure 3.11. The numbers on top of the points indicate the maximum memory usage[7]. As shown in the figure, indeed as batch size increases the runtime initially improves and then remains the same around after batch size of 10K. As expected, larger batch sizes lead to more cluster memory usage. Note that the increase in the memory usage is very small for batch sizes less than or equal to 100K because the intermediate data that the algorithm generates with these batch sizes is insignificant compared to the size of the input graph. Batch size is a useful parameter to balance memory usage and speedup.

3.7 Conclusion

Subgraph queries, i.e., finding instances of a given subgraph in a larger graph, are a fundamental computation performed by many applications and supported by many software systems that process graphs. This chapter proposes an efficient approach (Delta-BigJoin) to monitor changes in subgraph queries as the graph changes. It also has several theoretical and implementation contributions. From a theoretical perspective: (1) it expands the definition of AGM bound and the definition of worst-case optimal to include communication costs in distributed systems; (2) it proposes a new distributed algorithm, BigJoin, which meets the new worst-case optimal definition; (3) it proposes a new algorithm BigJoin-S,

[7]Memory has been measured using an operating system tool which reports a snapshot of memory usage every second instead of the average memory usage every second. This explains the small approximation and inaccuracy in the reported memory size.

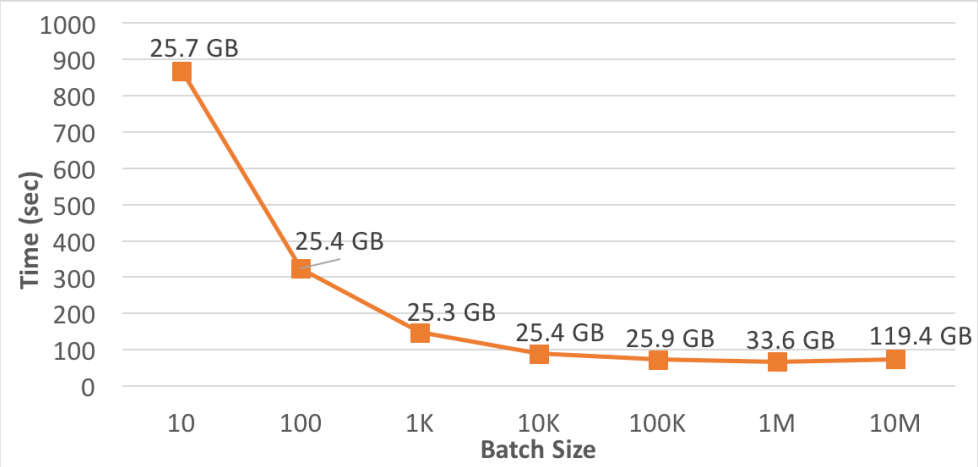


Figure 3.11: Effects of batch size. Note that the maximum memory usage in small batches is very close to the index size (25.1 GB).

which handles potential possible imbalance between machines in the case of skewed input datasets.

From an implementation perspective, it shows how to implement Delta-BigJoin using Timely Dataflow to dynamically find subgraph queries in large dynamic graphs. The evaluation section shows that BigJoin and Delta-BigJoin are significantly faster than existing distributed systems and require cluster memory that is linear in the size of the inputs. The concluding experiment, Table3.4 shows that Delta-BigJoin, with 16 machines can have a throughput up of 46 million 4-cliques per second in a common crawl graph.

Chapter 4

Conclusions and Future Work

Graph data has been growing rapidly. Most existing GDBMSs can support shortest path queries, regular path queries, and subgraph matching queries. However, as an input graph changes, these systems typically need to run these queries from scratch to update their answer repeatedly. This thesis addresses the following question: how should the modern GDBMS be designed to have an efficient query processor for evaluating queries on dynamic graphs?

More specifically, this thesis looks at two challenging query types: recursive queries and subgraph queries. Recursive queries, such as single pair shortest path (SPSP), single source shortest path (SSSP), variable-length join queries, or regular path queries (RPQ) are challenging to maintain on dynamic graphs. Subgraph queries find instances of a given subgraph in a larger graph. These are fundamental computations performed by many applications and supported by many software systems that process graphs.

This thesis proposes a solution for each query type. For recursive queries, Chapter 2 shows how to use DC to maintain recursive queries efficiently, up to 5 orders of magnitude faster, while minimizing its memory overhead using complete and partial difference dropping approaches. For subgraph queries, Chapter 3 shows an implementation of worst-case optimum join algorithms that can process static and dynamic graphs in distributed environments.

Chapter 2 demonstrates how to integrate DC into a prototype single-node GDBMS, and discusses the issues that arise from this. A vanilla implementation of DC is 5 orders of magnitude faster than running queries from scratch. However, it quickly fails with out-of-memory errors when maintaining a small number of queries. There are two proposed optimizations to reduce the memory overhead of DC. The first proposed optimization is

complete difference dropping (Section 2.2) which avoids explicitly storing `Join` results and compute them on demand when needed. Experiments on several real and synthetic data sets show that complete dropping uses up to 5× less memory in comparison to vanilla DC. The second optimization is partial difference dropping (Section 2.3) which chooses certain differences to drop. Partial dropping, together with complete dropping, can increase the scalability of a vanilla DC implementation up to 20× more maintained queries.

Chapter 3 addresses implementing worst-case-optimum-join (GJ) algorithms for dynamic graphs and in distributed settings. GJ have been implemented in single node systems for static graphs only. Section 3.2 demonstrates how to expand the definition of *worst-case optimal* for distributed settings to include communication costs and memory distribution among workers. The quantitative evaluation shows that BigJoin is competitive but slightly worse than a single-node GJ system (EmptyHeaded). This is because BigJoin lacks specific optimizations for static datasets, such as compacting dense extension lists into bit vectors, that exist in EmptyHeaded. Section 3.4.2 shows how to use BigJoin to process dynamic graphs in a distributed environment; with 16 machines Delta-BigJoin can have a throughput up to 46 millions 4-cliques per second when processing updates to the common crawl graph.

4.1 Future Work

Although this is not executed within the scope of this thesis, TD and DD infrastructure is ideal for developing a modern distributed GDBMS for dynamic graphs. Building an end-to-end system on top of TD and DD is a major engineering undertaking, but an interesting line of investigation. The result can be a state-of-the-art scalable system for dynamic graph data management. Materialize[1] is an example of how this system may look like for an SQL engine.

Figure 4.1 shows how the proposed algorithms and optimizations may fit in this system. The base component in this query engine is TD (Section 3.1.3). Both Delta-BigJoin and DD are built on top of TD. The partial differential drop (PDD) optimization, can be easily extended to support a distributed system. However, the complete differential dropping (CDD) optimization for DC has been implemented, assuming a single node and shared memory settings. More specifically, it assumes full access to the DC indices. An interesting future work direction is to use an efficient distributed index without adding significant overhead to CDD.

[1] https://materialize.com/docs/overview/

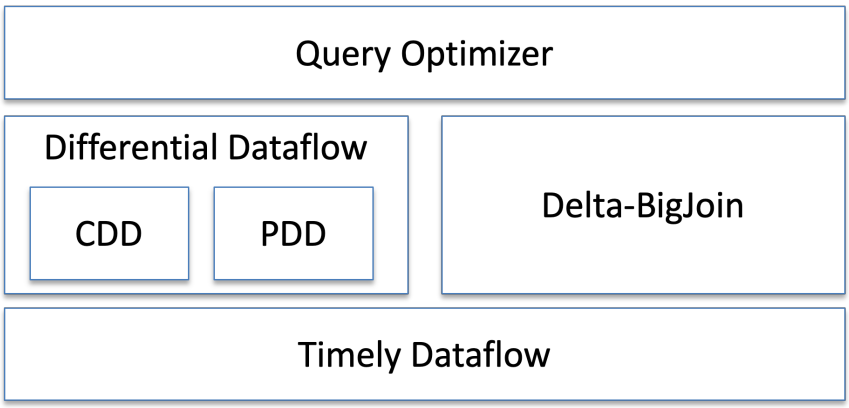


Figure 4.1: High-level architecture of a modern GDBMS supporting recursive and subgraph queries

A query optimizer should be able to (1) decide which approach to use, and (2) divide one query into several sub-queries and then decides to either use the Delta-BigJoin algorithm or the IFE abstraction with DD to monitor each sub-query. As discussed in Section 3.1, there are a few proposals in the literature to achieve this [3, 53].

Aside from building a distributed system, there are several directions that could represent exciting future work as described below.

4.1.1 Recursive Queries

There are three future work directions related to maintaining recursive queries:

1. System Optimizations: The proposed optimizations have several configurations. These could be either estimated using heuristics, such as the minimum/maximum threshold for the degree-based dropping approach in Figure 2.3, or manually optimized, such as the size of bloom filters. Optimizing these parameters can add significant value to the proposed approach.

2. Self-Managed System: In partial dropping, a system user has to do an exhaustive search to find the minimum dropping probability to accommodate a certain number of queries. There is a good opportunity to use regression models to estimate this probability using graph properties, query properties, and the system configuration as features.

3. DC with Indexes: Section 2.5.6 demonstrated how Diff-IFE could be used to maintain landmark indices. However, this algorithm only enhanced the SCRATCH baseline algorithm with an index that is maintained deferentially. This proposed algorithm does not evaluate the 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. There is no prior work that has proposed such an algorithm, and developing it is an important research topic.

4.1.2 Subgraph Queries

There are three future work directions related to maintaining subgraph queries:

1. BIGJOIN-S algorithm is theoretically skew-resilient, but the initial experiments showed that its overhead exceeds its benefits. The first future work direction is to study the possible imbalance in real-world graphs and design more efficient workload-balanced versions of BIGJOIN and DELTA-BIGJOIN with proper guards against data skew.

2. Some subgraph queries can have internal symmetries. For example, when evaluating the 4-clique query, some delta queries, e.g., $dQ2$ and $dQ3$, may compute the same prefixes due to the internal symmetry of the query. An interesting future direction is to automatically exploit such symmetries to share computations across multiple dataflows of delta queries.

3. Proposed algorithms assume monitoring one subgraph query, but in many cases, there is a need to monitor multiple subgraph queries. When processing multiple queries, there could be opportunities to find common sub-queries that should be evaluated once instead of evaluating them multiple times with each query. Identifying these common subqueries and implementing a system to execute multiple queries efficiently is an interesting future direction.

References

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

[2] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. SPARTex: A Vertex-Centric Framework for RDF Data Analytics (Demonstration). *Proc. VLDB Endowment*, 8(12), 2015.

[3] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 431–446, 2016.

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 13–28, 2016.

[5] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. and Data Eng.*, 23(9):1282–1298, 2011.

[6] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. *Proc. VLDB Endowment*, 6(4), 2013.

[7] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround distributed join algorithm. In *Proc. 20th Int. Conf. on Database Theory*, 2017.

[8] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endowment*, 5(10), 2012.

[9] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *Proc. VLDB Endowment*, 11(6):691–704, 2018.

[10] Khaled Ammar, Siddhartha Sahu, Semih Salihoglu, and M. Tamer Özsu. Optimizing differentially-maintained recursive queries on dynamic graphs. *Proc. VLDB Endowment*, 15(11):3186–3198, 2022.

[11] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM Journal on Computing*, pages 739–748, 2013.

[12] P. Beame, P. Koutris, and D. Suciu. Skew in Parallel Query Processing. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 212–223, 2014.

[13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):1–58, 2017.

[14] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. *ACM SIGMOD Rec.*, 15(2):61–71, June 1986.

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

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

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38:28–38, 2015.

[18] Timothy M. Chan. All-pairs shortest paths for unweighted undirected graphs in o(mn) time. *ACM Trans. Algorithms*, 8(4):34:1–34:17, 2012.

[19] TimothyM. Chan. All-pairs shortest paths with real weights in o(n 3/log n) time. In Frank Dehne, Alejandro López-Ortiz, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, volume 3608 of *Lecture Notes in Computer Science*, pages 318–324. Springer, 2005.

[20] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proc. 18th Int. Conf. on Extending Database Technology*, 2015.

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

[23] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[24] Differential Dataflow. https://github.com/frankmcsherry/differential-dataflow. Last accessed: 2022-08-01.

[25] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC social network benchmark: Interactive workload. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 619–630, 2015.

[26] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, August 1999.

[27] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[28] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 155–169, 2017.

[29] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans M. Coetzee. Self-Organization and Identification of Web Communities. *Computer*, 35(3):66–70, March 2002.

[30] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous Pattern Detection Over Billion-edge Graph Using Distributed Framework. In *Proc. 30th Int. Conf. on Data Engineering*, 2014.

[31] Giraph. http://giraph.apache.org. Last accessed: 2022-08-01.

[32] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proc. 16th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 156–165, 2005.

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

[34] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *ACM SIGMOD Rec.*, 22(2):157–166, 1993.

[35] Gupta, Pankaj and Satuluri, Venu and Grewal, Ajeet and Gurumurthy, Siva and Zhabiuk, Volodymyr and Li, Quannan and Lin, Jimmy. Real-time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endowment*, 7(13):1379–1380, August 2014.

[36] Xiao Hu, Yufei Tao, and Ke Yi. Output-optimal Parallel Algorithms for Similarity Joins. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2017.

[37] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. Knowl. and Data Eng.*, 23(9), 2011.

[38] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: efficient ad-hoc analytics on evolving graphs. In *NSDI*, pages 337–355, 2021.

[39] Manas Joglekar and Christopher Ré. It's all a matter of degree: Using degree information to optimize multiway joins. In *Proc. 19th Int. Conf. on Database Theory*, 2016.

[40] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proc. 9th IEEE Int. Conf. on Data Mining*, pages 229–238, 2009.

[41] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 1695–1698, 2017.

[42] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 411–426, 2018.

[43] Seongyun Ko and Wook-Shin Han. Turbograph++ a scalable and fast graph analytics system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 395–410, 2018.

[44] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. iturbograph: Scaling and automating incremental graph analytics. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 977–990, 2021.

[45] Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-Case Optimal Algorithms for Parallel Query Processing. In *Proc. 19th Int. Conf. on Database Theory*, pages 8:1–8:18, 2016.

[46] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[47] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proc. 11th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 177–187, 2005.

[48] Longbin Lai and Lu Qin and Xuemin Lin and Ying Zhang and Lijun Chang. Scalable distributed subgraph enumeration. *Proc. VLDB Endowment*, 10(3):217–228, 2016.

[49] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146, 2010.

[50] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference*, 2019.

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

[52] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at What Cost? In *Proc. 15th Workshop on Hot Topics in Operating Systems*, 2015.

[53] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment*, 12(11):1692–1704, 2019.

[54] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. Symmetric continuous subgraph matching with bidirectional dynamic programming. *Proc. VLDB Endowment*, 14:1298–1310, 2021.

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

[56] Neo4j. http://neo4j.com. Last accessed: 2022-08-01.

[57] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *Proc. VLDB Endowment*, 19(1):91–113, 2010.

[58] H. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *ACM SIGMOD Rec.*, 42(4):5–16, 2014.

[59] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 111–124, 2012.

[60] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Rec.*, 45(2):5–16, September 2016.

[61] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. Regular path query evaluation on streaming graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1415–1430, 2020.

[62] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[63] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, page 867–876, 2009.

[64] Rada Chirkova and Jun Yang. Materialized Views. *Foundations and Trends in Databases*, 2012.

[65] VV Rodionov. The parametric problem of shortest distances. *USSR Computational Mathematics and Mathematical Physics*, 8(5):336–343, 1968.

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

[67] Rust. https://www.rust-lang.org. Last accessed: 2022-08-01.

[68] Leonid Ryzhyk and Mihai Budiu. Differential datalog. In *Proc. 3rd International Workshop on the Resurgence of Datalog in Academia and Industry*, pages 56–67, 2019.

[69] Siddhartha Sahu and Semih Salihoglu. Graphsurge: Graph analytics on view collections using differential computation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1518–1530, 2021.

[70] Semih Salihoglu and Jennifer Widom. Help: High-level primitives for large-scale graph processing. In *Proc. of Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2014.

[71] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel Subgraph Listing in a Large-scale Graph. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2014.

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

[73] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. An in-depth study of continuous subgraph matching. *Proc. VLDB Endowment*, 15(7):1403–1416, 2022.

[74] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endowment*, 5(9):788–799, 2012.

[75] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proc. 26th ACM Symp. on Operating System Principles*, pages 425–440, 2015.

[76] The Laboratory for Web Algorithmics. https://law.di.unimi.it/datasets.php. Last accessed: 2022-08-01.

[77] Timely Dataflow. https://github.com/frankmcsherry/timely-dataflow. Last accessed: 2022-08-01.

[78] Titan: Distributed Graph Database. http://thinkaurelius.github.io/titan. Last accessed: 2022-08-01.

[79] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 147–156, 2014.

[80] Todd L. Veldhuizen. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[81] Todd L. Veldhuizen. Incremental Maintenance for Leapfrog Triejoin. *CoRR*, abs/1303.5313, 2013.

[82] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. A survey of continuous subgraph matching for dynamic graphs. *Knowledge and Information Systems*, pages 1–45, 2022.

[83] Web Data Commons. http://www.webdatacommons.org/hyperlinkgraph. Last accessed: 2022-08-01.

[84] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. *Proc. VLDB Endowment*, 1981.

[85] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient Maintenance of Materialized Top-k Views. In *Proc. 19th Int. Conf. on Data Engineering*, pages 189–200, 2003.

[86] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[87] Zeng, Kai and Yang, Jiacheng and Wang, Haixun and Shao, Bin and Wang, Zhongyuan. A Distributed Graph Engine for Web Scale RDF Data. *Proc. VLDB Endowment*, 6(4):265–276, 2013.

[88] Peng Zhang, Yuhao Huang, Aaron Gember-Jacobson, Wenbo Shi, Xu Liu, Hongkun Yang, and Zhiqiang Zuo. Incremental network configuration verification. In *Proc. of the 19th ACM Workshop on Hot Topics in Networks*, pages 81–87, 2020.

[89] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *Proc. VLDB Endowment*, 4(8): 482–493, 2011.