Cardinality Estimation in Streaming Graph Data Management Systems

by

Kerem Akillioglu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Kerem Akillioglu 2024

Author's Declaration

I hereby declare that I am the sole author of this 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.

Abstract

Graph processing has become an increasingly popular paradigm for data management systems. Concurrently, there is a pronounced demand for specialized systems dedicated to streaming processing that are essential to address the continual flow of data and the inherent dynamism in streaming data. Yet, the lack of a standardized, general-purpose query framework specifically for streaming graphs is a notable gap in existing technologies. This shortfall emphasizes the necessity for a more comprehensive solution for processing and analyzing streaming graph data efficiently in real time. Enhancing this solution is crucially dependent on improving the query processing pipeline, especially on cardinality estimation and query optimization, both of which are key factors in ensuring optimal system performance.

In this thesis, a novel cardinality estimation technique, called *GraphSketch*, that is tailored for streaming graph database management systems (GDBMS) is proposed. *GraphSketch* is a sketch-based framework designed to concisely summarize streaming graphs, enabling both accurate and efficient cardinality estimations. The thesis delves into the theoretical foundations of *GraphSketch*, outlining its conceptual design and the specific methodologies employed in its construction. Additionally, the thesis elaborates on the suitability of *GraphSketch* for streaming systems, highlighting its capability for incremental updates, which are pivotal in maintaining efficiency in the rapidly evolving environment of streaming data.

Acknowledgements

I am profoundly grateful to my supervisor, M. Tamer Özsu, for his exceptional guidance and mentorship throughout my academic path. His influence extends beyond supervision, serving as an inspiration and a role model that shaped my aspirations and research approach. His commitment to precision profoundly influenced my work, leading me to a more focused and meticulous research methodology. It was under his mentorship that I discovered my love for research, a passion that continues to drive me. I am incredibly fortunate to have had the privilege of learning from him, and his encouragement remains a key motivator in my ongoing academic endeavors.

Additionally, I thank Anil Pacaci for his significant contributions to this research. Over the years, Anil has been more than a collaborator; he has been a mentor and a cherished colleague. His involvement in the research for this thesis has been pivotal and has greatly enriched my experience and the quality of my work. I extend my heartfelt thanks to Angela Bonifati, whose guidance and expertise have been invaluable in my research journey. I would also like to thank my committee members, Grant Weddell, and Khuzaima Daudjee, for their advice, help, and support.

I extend my deepest gratitude to my family, whose unwavering support and love have been the backbone of my journey through graduate studies. My sisters, Ece and Merve, alongside my parents, Demet and Mehmet, have been the pillars of unconditional love and steadfast support throughout my life. Their presence has been irreplaceable, offering me strength and encouragement at every step. I am profoundly thankful for their enduring love and the sense of home they provide.

My graduate studies experience was significantly enhanced by the constant support and cheerful presence of my dear friends - Bugra, Dilan, Elif Nur, Elif Su, Gizem, Halil Suat, Lucas, Ophélia, Salih, Shri, Suraj. Their support has always lifted my spirits and kept me going, especially during tough times. I want to extend a special thanks to Sinan, my top childhood friend and eventual roommate in Waterloo, with whom I've shared 21 years of friendship. We've had endless conversations about everything, and thanks to these talks, he's probably the only political science student who knows the differences between relational and graph databases.

Lastly, I would like to express my heartfelt appreciation to the friends I have made during my time at Waterloo: Benson, Sairaj, Kriti, Fadhil, Anurag, Amine, Zeynep, Rafael, Lap Chi, Shubankar, and countless others whose names I regrettably cannot mention individually. Thank you for your unwavering friendship and support throughout my journey at Waterloo.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background Information and Related Work 6

2.1 Streaming Graph Semantics 6

2.1.1 Precise Definition of Objectives 9

2.1.2 Handling High Edge Arrival Rates in Streaming Graphs 9

2.1.3 Emphasis on Real-Time Processing for Streaming Graphs 10

2.1.4 Unboundedness 10

2.1.5 Time-Based Sliding Window Model 10

2.1.6 Stream Generation 11

2.2 Cardinality Estimation Techniques for Relational Systems 11

2.3 Cardinality Estimation Techniques for Graph Data Management 13

2.4 System R Cardinality Estimation Technique 13

3 GraphSketch Design 16

3.1 GraphSketch Construction Example . . . 17

3.2 GraphSketch Construction and Cardinality Estimation . . . 19

3.3 Incremental Updates of the GraphSketch . . . 33

3.3.1 Window Management in Streaming Environment . . . 34

3.3.2 Baseline Approach . . . 35

3.3.3 Updating GraphSketch Incrementally . . . 36

3.4 Comparison with System R's Method . . . 39

4 Implementation and Evaluation 45

4.1 Implementation . . . 45

4.2 Source Code . . . 46

4.3 Experimental Platform . . . 46

4.4 Evaluation . . . 47

4.5 Dataset . . . 47

4.6 Workloads . . . 52

4.7 Accuracy Experiments . . . 53

4.7.1 Limitations . . . 55

4.7.2 Accuracy Equivalence of GraphSketch and System R . . . 56

4.7.3 Impact of Bucket Count on GraphSketch Accuracy . . . 56

4.7.4 Impact of GraphSketch's Bucket Count on Query Plans . . . 63

4.8 Query Latency Experiments . . . 65

4.9 Efficiency Experiments . . . 69

5 Conclusion 72

5.1 Future Research Directions . . . 73

References 74

List of Figures

2.1 A chain query with two edges. 15

3.1 Example graph. 17
3.2 GraphSketch-3 visualization of edge label A. 19
3.3 Source-Source Pattern 20
3.4 Source-Target Pattern 20
3.5 Target-Source Pattern 20
3.6 Target-Target Pattern 20
3.7 A chain query with three edges. 20
3.8 Logical plan for the query in 3.7 21
3.9 A second example graph. 22

4.1 SGrafitto's Query Processing Pipeline. 46
4.2 Edge Label Distribution: First Three Months of StackOverflow Graph . . . 48
4.3 Edge Label Distribution: Synthetic Graph 49
4.4 Source Vertex Distribution: First Three Months of StackOverflow Graph . 49
4.5 Source Vertex Distribution: Synthetic Graph 50
4.6 Target Vertex Distribution: First Three Months of StackOverflow Graph . 50
4.7 Target Vertex Distribution: Synthetic Graph 51
4.8 A chain query with six edges. 52
4.9 A cycle graph query with six edges. 52

4.10 A star query with six edges. 53
4.11 Evaluation Workflow for Accuracy Experiments. 54
4.12 Accuracy Plots of Chain Queries 60
4.13 Accuracy Plots of Cycle Queries 61
4.14 Accuracy Plots of Star Queries 62
4.15 Comparative Analysis of Chain and Star Query Latency Between GS-900 and GS-1 Plans 66
4.16 Comparative Analysis of Star Query Latency Between GS-900 and GS-1 Plans 67
4.17 Comparative Analysis of Chain Query Latency Between GS-900 and GS-1 Plans 68
4.18 Frequency Histograms for Latency Experiments 71

List of Tables

3.1 Edge table for the example graph 18
3.2 Edge table for the example graph in Figure 3.9. 23

4.1 Comparative Analysis of Average Q-error: GS-1 vs. System R 56
4.2 Average Q-error Across Different Bucket Counts 57
4.3 Changes in Query Plans Across Different Bucket Counts 63
4.4 Comparison of Graph Sketch Construction Times 69

Chapter 1

Introduction

"You can't step into the same stream twice because it's always flowing."
— Frederick Jay Rubin, The Creative Act

Graphs are widely used in various applications such as bioinformatics, software engineering, e-commerce, finance, trading, and social networks [48]. The ability to process and query graph-structured data is made possible through graph database management systems (GDBMS). In this thesis, the focus is on GDBMSs that process streaming graph data. As the name implies, stream data consists of a continuous stream of data items arriving at a processing centre, usually in real-time, or sorted by a timestamp. Streaming GDBMSs with real-time and windowed processing capabilities are being investigated to better handle this data.

Streaming GDBMSs have a similar query processing pipeline to traditional relational or GDBMSs, although their requirements are different. Traditional GDBMSs, such as Neo4j[1], JanusGraph[2], and TigerGraph[18] offer limited query capabilities for streaming scenarios due to their reliance on snapshot models and lack of support for complex path navigations. These constraints, as discussed in the context of G-CORE [4], point to a need for more advanced query languages that can offer full composability and algebraic closure, which are critical for effective query optimization in streaming environments. Also, distributed graph processing engines like Pregel [41], GraphX [22], and PowerGraph [21] focus on

[1] https://neo4j.com

[2] https://janusgraph.org

static graphs while streaming GDBMSs must handle the continuous influx of data and the evolving nature of graph structures in real-time.

Existing streaming systems, primarily designed for relational streams, do not fully accommodate the requirements of streaming graph data [49]. The absence of a uniform, general-purpose query framework for streaming graphs reveals a gap in current technologies. This gap underscores the need for a more adaptable and comprehensive solution capable of efficiently processing and analyzing streaming graph data in real time.

Recognizing these gaps and the emerging challenges in streaming GDBMSs, it becomes crucial to understand the details of query processing within these systems. A query is processed through multiple steps, and there can be multiple logical query plans for a query and multiple physical execution plan for each logical query plan. A query plan is defined in terms of logical and physical operators and are typically represented as a tree of these operators. The equivalent execution plans may differ greatly in terms of performance. Hence, the query optimizer plays a central role in query processing, as it aims to avoid inefficient plans and pick an efficient plan from the set of possible execution plans for a given query. In order to make the query selection progress more systematic and methodical, most modern query optimizers employ a cost-based model. These optimizers compare numerous possible execution plans for a query and assign a cost to each plan based on various factors such as the required memory space, input/output operations, and CPU time. The optimizer then selects the plan with the lowest estimated cost to execute the query. Cost-based query optimizers rely on estimations of the size of intermediate data sets following the execution of each operator in the query plan in order to choose efficient execution plans. Cost-based query optimizers have three key components: (i) cardinality estimation, (ii) cost model, and (iii) plan enumeration.

- Cost model helps optimizers to assign the estimated costs for queries. Assigned costs also consider all the sub-plans in the queries with the sum of the costs of all operators.
- Cardinality estimation refers to estimating the number of results returned for each operator in the query plan and is used by the query optimizer to estimate the cost of executing different execution plans.
- Plan enumeration techniques are used to find the equivalent query plans. The naive approach of exhaustive search of all equivalent plans is NP-hard, thus plan enumeration considers heuristics to reduce the space of plans that are considered.

Cardinality estimation is generally considered to be the most critical component of the query optimizer and has been called the “Achilles heel” of optimizers [40]. Enhancing

the accuracy of cardinality estimations may not necessarily result in better query plans. Although reducing the error of cardinality estimation is a necessary condition,it is not sufficient – ensuring that "better" (i.e., one with lower error) estimation improves the performance of a query plan is far more challenging. [38].

The focus of this thesis is cardinality estimation in streaming GDBMSs. Most of the existing systems typically employ a popular technique that involves utilizing statistics about the underlying inputs, and making assumptions about the independence and uniformity of these statistics. Wrong statistics or oversimplifying assumptions lead to inaccurate estimations and hence to the selection of a highly sub-optimal execution plans [11].

The field of cardinality estimation has been subject to intensive research, with the majority of existing techniques being designed for use in relational DBMSs. Although it is possible to map graph-structured data to relational systems, such techniques fail to fully exploit the graph structure of the data, leaving ample room for optimization.

In order to capture the requirements of graph structure in the streaming setting, Pacaci et. al. [50] introduce a general-purpose query processing framework for streaming graphs that consist of (i) a Streaming Graph Query (SGQ) model and Streaming Graph Algebra (SGA) with well-founded semantics, and (ii) a prototype streaming graph query processor as a practical implementation of the proposed framework. SGQ and SGA establish the basis of the systematic study of query processing issues over streaming graphs. In particular, the rich plan space provided by SGA operators and their transformation rules provides the fundamental machinery for cost-based optimization of SGQ, which is the focus of this research. This thesis is conducted within the context of this framework.

Existing literature, by and large, study query optimization in the context of ad-hoc queries in the snapshot model where queries are transient and the data is persistent. In the snapshot model, query optimizers find an efficient plan for a given query by navigating the space of equivalent plans guided by a cost model. The cost model is used to estimate the resource usage (execution time, network cost etc.) required to successfully complete the execution of the given query using a set of statistics available about the underlying dataset. SGQs that we target in this research pose unique challenges to this traditional architecture:

1. SGQs are continuously evaluated over unbounded streams, so the evaluation of a query is never completed;

2. data arrivals occur at high velocity, so heavy-duty techniques for query optimization are not likely to work

3. underlying data is ever changing so existing techniques for collecting and maintaining statistic are not applicable; and
4. the recursive nature of SGQs with complex subgraph patterns and path navigations require additional statistics such as the path length (i.e., the recursion depth).

As described earlier, SGA provides the fundamental machinery for building a cost-based query optimization framework for SGQ. An important component of this framework is the cost model that is used to estimate the runtime performance of individual SGA operators and the query plans that consists of these operators. The proposed framework employs the unit-time cost model first proposed by Kang et. al [33] where the optimizer estimates the processing cost of an operator (or a plan) per unit application time. Cost of an operator (plan) per unit-time depends on the arrival rates of its input streaming graphs and per-tuple processing cost. It has been shown that the rate of the output streaming graph of an operator (plan) can be computed using following streaming graph characteristics: (i) input arrival rates, (ii) the length and distribution of validity intervals, and (iii) operator selectivities.

The initial prototype of the proposed SGQ optimization framework makes a number of simplifying assumptions regarding these streaming graph characteristics. First, it assumes that input arrival rates and their interval lengths are known and do not change during the lifespan of a query. Consequently, an optimal plan for a given SGQ does not change over time. Second, it is assumed that the vertices of input streaming graphs have a uniform degree distribution, simplifying the selectivity computation for operator predicates [49]. Edges represented by tuples in a streaming graph form a graph, and the degree of a vertex is the number of relationships it has over this graph. However, it is known that real-world graphs rarely have uniform degree distributions and characteristics of streaming graphs fluctuate over time as the graph evolves. Consequently, these assumptions results in sub-optimal optimization decisions, in particular, they diminish the accuracy of cost estimations. The primary objective of this thesis is to improve the cost model of the proposed SGQ optimizer framework by addressing the limitations arising from the above assumptions. The improvements targeted in this thesis are restricted to cardinality estimation.

In this thesis a cardinality estimation technique, called *GraphSketch* is developed that more accurately estimates cardinalities of persistent queries over streaming graphs. The contributions of the thesis can be summarized as follows:

- We analyze the existing cardinality estimation techniques and demonstrate that none of these techniques are sufficient to capture the requirements for streaming graphs.

- We develop a novel algorithm and model to estimate characteristics of streaming graphs to improve the accuracy of cost estimations performed by SGQ optimizers. In particular, we study efficient and accurate estimations of following streaming graph characteristics:

  1. the streaming rate and distribution of validity intervals over time;
  2. degree distributions of the snapshot graph induced by input streaming graphs; and
  3. selectivity estimations for SGA operator predicates.

- We integrate our benchmark with the prototype query optimizer. This generates an optimized query plan while utilizing our new cardinality estimation method. This testbed is used to verify the correctness of the optimizer's output and measure the latency of query execution.

This thesis is organized as follows. Chapter 2 introduces related works along two main background domains: cardinality estimation techniques for relational systems and cardinality estimation techniques for graph data management. Chapter 3 discusses the design of our proposed cardinality estimation method. Chapter 4 presents the evaluation section and our experimental results. Finally, Chapter 5 presents the conclusions and discusses the potential future work.

Chapter 2

Background Information and Related Work

This chapter begins with providing background on streaming data processing semantics, followed by an overview of well-known traditional cardinality estimation techniques, and an examination of related work in this domain. Next, we describe prevalent cardinality estimation techniques used in graph-based database management systems, including XML and RDF. The chapter concludes with a detailed discussion of cardinality estimation in streaming GDBMSs, highlighting the significance and relevance of System R's cardinality estimation technique in this context.

2.1 Streaming Graph Semantics

Before delving into the core concepts of this thesis, we define the streaming graph model that is used in the thesis. These definitions form the basis of the discussions and methodologies that follow.

Definition 1 (Graph). *A directed labeled graph is a quintuple $G = (V, E, \Sigma, \psi, \phi)$ where V is a set of vertices, E is a set of edges, Σ is a set of labels, $\psi : E \rightarrow V \times V$ is an incidence function, and $\phi : E \rightarrow \Sigma$ is an edge labeling function.*

Building upon the concept of a graph, we next consider paths within these structures and how they are labeled, further enriching our understanding of graph dynamics.

Definition 2 (Path and Path Label). *Given vertices* $u, v \in V$*, a path* p *from* u *to* v *in graph* G *is a sequence of edges* $u \xrightarrow{p} v : \langle e_1, \ldots, e_n \rangle$ *such that for each edge* $e_i \in E$*, the endpoints* $x_i, y_i \in V$ *satisfy* $y_i = x_{i+1}$ *for* $i \in [1, n)$*. The label sequence of a path* p *is defined as the concatenation of edge labels, i.e.,* $\phi_p(p) = \phi(e_1) \cdot \ldots \cdot \phi(e_n) \in \Sigma^*$*.*

Next, we define the time domain, an essential aspect in streaming data, which lays the groundwork for understanding how streaming graphs evolve over time.

Definition 3 (Time Domain). *Let* $\mathbb{T} = (T, \leqslant)$ *denote a discrete, totally ordered time domain, where* $t \in T$ *is a timestamp representing a specific time instant. In this thesis, non-negative integers are used to represent timestamps.*

With the time domain established, we now introduce the concept of streaming graph edges, which are fundamental to the notion of streaming graphs.

Definition 4 (Streaming Graph Edge). *A streaming graph edge (sge) is a quadruple* (src, trg, l, t) *where* src *and* trg *are vertices,* l *represents the label of the sge, and* $t \in T$ *is the event (application) timestamp assigned by the external data source.*

Extending from individual edges, we consider the stream of these edges as it forms over time, leading us to the definition of an input graph stream.

Definition 5 (Input Graph Stream). *An input graph stream is a continuously growing sequence of streaming graph edges* $S_I = \langle sge_1, sge_2, \ldots \rangle$ *where each* $sge_i = (src_i, trg_i, l_i, t_i)$ *represents an edge* $e \in E$ *labeled* $l_i \in \Sigma$ *between vertices* $src_i, trg_i \in V$*. The sges are non-decreasingly ordered by their timestamps.*

Closely linked to the concept of streaming graph edges is the notion of validity intervals, which delineate the timeframes during which these edges are considered relevant.

Definition 6 (Validity Interval). *A validity interval is a half-open time interval* $[ts, exp)$ *consisting of all distinct time instants* $t \in T$ *for which* $ts \leqslant t < exp$*. Validity intervals represent the period during which sges are considered valid.*

Continuing from the previously established definitions, we delve deeper into the nuances of timestamps and validity intervals, and their implications for the streaming graph data. Timestamps play a pivotal role in the context of streaming graph edges (sges). They are typically employed to denote the precise moment at which the interaction represented by an sge occurs, as referenced in various studies [55, 51, 39]. In contrast, validity intervals are

leveraged to represent the duration for which an sge retains its relevance or validity. The use of these intervals contributes to a more compact representation, significantly streamlining the semantics of operators and dissociating the specification of window constructs from their implementation.

Time-based sliding windows are instrumental in the assignment of these validity intervals. The specific windowing parameters of a query dictate the intervals, thereby aligning the data processing with the temporal dynamics inherent to the query's nature. This detailed exploration of timestamps and validity intervals underscores their significance in managing and interpreting streaming graph data. It sets the stage for a comprehensive understanding of the dynamic and temporal aspects of graph streams, which are crucial for the development and evaluation of queries within the proposed framework.

Definition 7 (Window). *A window indexed by k, denoted as W_k, over a streaming graph is a finite multi-set of streaming graph edges represented as a range $[W_{b_k}, W_{e_k})$, where W_{b_k} and W_{e_k} are the beginning and end borders of the window, respectively.*

Definition 8 (Time-based Sliding Window). *A time-based sliding window with window size $|W_k|$ and slide parameter β is a window that moves forward every β time units. At any time point t, the end border W_{e_k} is defined as $\lfloor t/\beta \rfloor \cdot \beta$, and the beginning border W_{b_k} is $W_{e_k} - |W_k|$.*

Definition 9 (Graph Snapshot). *A graph snapshot at a given time point t, denoted as $G_{W,t}$, is a pair of vertex and edge sets $G = (V, E)$ forming a graph constituted by the sges within the corresponding window W_k. For ease of reference, the graph snapshot $G_{W,t}$ and its corresponding window W_G are used interchangeably throughout this thesis.*

Definition 10 (Streaming Graph Tuple). *A streaming graph tuple (sgt) is a quintuple $sgt = (src, trg, l, [ts, exp), D)$ where src and trg are vertices in the graph. l is the label of the sgt, $[ts, exp) \in T \times T$ is a half-open time interval, representing the validity of the tuple, and D is the payload associated with the sgt.*

The streaming graph tuple generalizes standard graph edge representations (as delineated in Definition 4) to include not only input graph edges but also derived edges and paths. Derived edges refer to new edges resulting from operator and query outputs, which may not be a part of the original input graph. Paths, in this context, are sequences of edges that also emerge as a result of operator and query outputs.

In our notation, $E_I \subset E$ denotes the set of edges that are part of the input graph. The function $\phi(E_I)$ represents a fixed set of labels, specifically reserved for these input graph

edges. The payload D of an sgt, in cases where the sgt represents a path, embodies the path p (i.e., a sequence of edges). In other scenarios, where the sgt represents an edge, D is simply the edge e itself.

2.1.1 Precise Definition of Objectives

Now that the streaming graph model is precisely defined, we can more accurately define the objectives of this thesis. We focus on developing methodologies for efficiently and accurately estimating key characteristics of streaming graph data. Our objectives are as follows:

- **Estimation of Streaming Rate and Validity Interval Distribution:** We aim to accurately determine the rate at which new data arrives (streaming rate) and how the validity of data changes over time. This involves analyzing the temporal distribution of data validity intervals, which is critical for understanding the dynamics of the streaming data and for optimizing data processing strategies.
- **Analysis of Degree Distributions in Snapshot Graphs:** Another crucial aspect is to study the degree distributions within snapshot graphs, which are generated for each window. Understanding these distributions will enable us to better comprehend the structural properties of the graph at different time intervals. This analysis is vital for tasks such as anomaly detection, graph pattern recognition, and optimization of graph querying processes.
- **Identification of Equivalent Query Plans:** Finally, we seek to identify and evaluate various query plans that are equivalent in terms of their execution outcomes but may differ in their performance characteristics. By analyzing these plans, we aim to establish methods to optimize query execution in terms of efficiency and accuracy, thereby enhancing the overall performance of streaming graph data management systems.

2.1.2 Handling High Edge Arrival Rates in Streaming Graphs

This thesis acknowledges the challenges identified in existing literature, where streaming graphs, often seen in social networks and e-commerce, are characterized by high edge arrival rates and their unbounded nature. Such traits present considerable difficulties for conventional graph database management systems, which are typically not designed to accommodate such intensive data flows [58, 20, 55, 52].

2.1.3 Emphasis on Real-Time Processing for Streaming Graphs

Building upon the challenges outlined in existing works, this thesis focuses on streaming graphs that necessitate real-time processing. This requirement marks a significant departure from the more traditional static graph model, where the graph does not change and, more importantly, is fully accessible for queries. In contrast, streaming graphs continuously evolve and there is a need for processing updates without the entire graph available [13, 60, 62, 68, 66, 67, 9, 7, 34, 36, 57].

2.1.4 Unboundedness

Bounded data, by definition, is finite and possesses a distinct beginning and end. This type of data is commonly linked with batch processing methods. Consequently, this data is transferred to the database periodically, which could be weekly, monthly, or even annually. Analytical processes are then executed on this data to generate insights and outcomes through a batch procedure.

Unbounded data, conversely, is characterized as infinite, lacking a distinct start or termination point. This type of data is usually connected with stream processing methodologies. For instance, sensors persistently gather real-world data, such as temperature, speed, and location parameters. This data collection process is uninterrupted and operates continuously around the clock. Streaming graphs are unbounded, requiring appropriate techniques to deal with this characteristic.

2.1.5 Time-Based Sliding Window Model

To effectively manage unbounded streams in graph data, this thesis adopts the time-based sliding window model, aligning with approaches found in existing literature [51]. This model utilizes a fixed-size window that slides at predetermined intervals to accommodate new edges and the expiration of old ones. Adopting this windowing technique is crucial for bounding memory usage and ensuring that recent data is prioritized [20].

In the context of this thesis, the window size is essentially a measure of the 'freshness' of the data – only data points that have timestamps within the window size from the most recent data point are considered 'fresh' and thus kept within the window. This approach ensures that analysis and processing are focused on the most relevant and current data.

Sliding windows in data processing are grouped by two principal semantics: implicit and explicit. The implicit window approach is distinguished by its ongoing addition of

new results to the query output as new streaming graph tuples (sgts) arrive. It uniquely maintains the validity of previously reported results, even as they become outdated and the window progresses. This approach ensures that, in scenarios devoid of explicit edge deletions, the query results are monotonic.

The implicit model facilitates the preservation of monotonicity in query results, leading to the generation of an append-only result stream, particularly in cases where explicit deletions are not present. Although users or applications have the capability to explicitly remove previously arrived edges, the standard operational framework of this environment anticipates the automatic elimination of tuples upon the expiration of the window. This thesis aligns with the implicit model, offering an effective and adaptable implementation of the time-based sliding window model, central to the analysis and processing of streaming data.

2.1.6 Stream Generation

We assume that streaming graph tuples (sgts) originate from a single source and follow the order of their respective source timestamps τ_i, establishing their ordering in the stream. This assumption stems from the idea that handling unordered data or events arriving out of sequence would require more sophisticated data structures or approaches. The challenge of handling out-of-sequence delivery is deferred to future work.

2.2 Cardinality Estimation Techniques for Relational Systems

A substantial amount of research has been conducted on estimating the cardinality of queries in relational DBMSs. Traditional methods for cardinality estimation can be considered under six strategies: (1) summary tables, (2) wavelets, (3) histogram-based methods, (5) sketching based methods and (5) sampling-based methods and (6) other methods.

1. Summary tables: This approach utilizes materialized tables that represent pre-computed aggregate queries [1, 25]. This method is limited since it is not possible to get summary tables for all possible user queries.

2. Wavelets: Wavelets are mathematical functions that cut up data into different frequency components. In data synopsis context, this process computes a set of values, namely wavelet coefficients, which represent a compact data summary [10].

3. Histogram based methods: A histogram is a special type of column statistic that provides more detailed information about the data distribution in a table column. A histogram sorts values into "buckets," and provides accurate estimates of the distribution of column data. Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with non-uniform data distributions [29, 17].

4. Sketching based methods: Sketching models aim to count distinct values (e.g., HyperLogLog, [27]) or frequency of tuples (e.g., Count Min [16]) over a data stream. This approach operates by hashing each element in the stream into a data structure called a "sketch". Then, the sketch is used to estimate the number of distinct elements at query time. Summary of data streams differs from sampling, in that sampling provides answers using only those items which were selected to be in the sample, whereas the sketch uses the entire input, but is restricted to retain only a small summary of it [65].

5. Sampling-based methods: Given a dataset, a small number of elements are selected at random and a variety of statistics are computed over the sample, such as the sample mean and variance. These statistics are then used to estimate the value of the query result and provide bounds on the accuracy of the estimate [15]. Sampling-based methods require no assumptions about the fit of the data to a probability distribution. Unlike histogram based methods, they do not require storing and maintaining detailed statistics about the base data of the system. However, it has been proven that almost the entire dataset needs to be sampled to be confident in obtaining a highly accurate estimate [43]. Yet, sampling algorithms can still be useful when the underlying distribution of the dataset is known.

6. Other methods: The methods listed above provide data summary and reduction. The methods we list as "other methods" do not have a similar approach, such as graph-based model, genetic programming, and online processing. Graph-based model generates a graph that summarizes both the join-distribution and the value-distribution of a relational database [61]. Genetic programming approach corresponds to rewriting the initial query to minimize the elaboration costs and maximizing the accuracy [54]. On-line processing approach continuously provides an estimate of the final answer to each aggregate query and shows a preview of it during the elaboration [31].

2.3 Cardinality Estimation Techniques for Graph Data Management

Existing literature on graph query optimization predominantly focus on subgraph queries [3, 44, 56]. These models do not handle path navigation queries that are abundantly found in existing query logs [9]. Sparqling Kleene [24] incorporates reachability queries into subgraph patterns by building a FERRARI [60] reachability index for every label in the graph and uses the index size for cardinality estimation during query planning. Fletcher et al. [19] propose a path specific histogram that can be used for path cardinality estimation for RPQs. Similarly, unit-cost matrix proposed by Nguyen et al [47] encode frequency of all length-2 paths in the input graph and use these frequencies to estimate the cardinality of complex path expressions. However, all these works require offline processing of the input graph to produce the underlying index structure and therefore cannot be used in the streaming context. μ-RA [30] extends relational algebra with a fixpoint recursion operator and provides a set of transformation rules to manipulate recursive queries, enabling query optimizers to consider queries with recursion.

All these focus on static graphs, and to the best of our knowledge, there is no work that studies optimization of path navigation queries in the streaming model. Stream summaries or sketches are used to provide approximate answers over data streams and they can be used for estimating stream characteristics for query optimization. Existing literature on stream summarization, by and large, focus on one-dimensional streams (i.e., the input is a simple tuple) and cannot maintain the topology of the graph [14, 42, 69]. The use of graph summaries and sketches for query planning has been investigated in the context of static graphs, but high ingestion rates and ever-changing nature of streaming graphs makes maintaining these summaries a challenging problem. There has been recent interest in graph summarization in the streaming model [8, 63], and we plan to investigate their use for the optimization of SGQs. In particular, a structural graph summary can be used to estimate graph characteristics such as degree distributions, path-length etc. to be used during cost estimation.

2.4 System R Cardinality Estimation Technique

As noted above, there is no work that tackles cardinality estimation over streaming graphs. This makes it difficult to find a baseline to compare with *GraphSketch*. Therefore, we've modified System R's technique to fit the unique needs of streaming GDBMS, and used it in our benchmark studies.

System R's method [59] is considered the classical approach to cardinality estimation. It employs a set of statistics to assign a selectivity factor (F) for each condition in the predicate list. This factor roughly corresponds to the expected fraction of tuples satisfying the predicate. The estimation technique relies on the number of distinct elements in the relation columns and requires a propagation method up the query plan tree. We discuss how we extend the statistics propagation for intermediate steps in Chapter 4.

This technique is built upon several key assumptions: uniform distribution of column values, independence of values in different columns, and inclusion. These assumptions play a crucial role in simplifying the estimation process and making it computationally feasible, though they may not always hold true in real-world data scenarios.

1. Uniformity Assumption: System R's uniformity assumption considers that values in a database column are uniformly distributed across the possible range of values. This means that every distinct value in a column has approximately the same frequency of occurrence.

   Under this assumption, the selectivity of a predicate (like `column = value`) can be approximated by dividing one by the number of distinct values in the column. This assumption simplifies the computation of selectivity factors but may lead to inaccurate estimations in the presence of skewed data distributions.

2. Independence Assumption: The independence assumption holds that the distribution of values in one column is independent of the distribution of values in another column. In other words, the presence or absence of a particular value in one column does not influence the distribution of values in another column.

   This assumption is particularly relevant for estimating the selectivity of queries with multiple predicates, especially those involving joins or compound predicates (e.g., `column1 = value1 AND column2 = value2`). Under the independence assumption, the selectivity of a compound predicate can be estimated as the product of the selectivities of its individual components.

3. Inclusion Assumption: The inclusion assumption suggests that the set of values in one column encompasses or includes the set of values in another column, especially in the context of join operations. This is often assumed when one of the columns serves as a foreign key referencing another table's primary key.

   This assumption affects the estimation of join operations, where the selectivity of a join predicate is based on the assumption that every value in the foreign key column has a corresponding value in the referenced primary key column.

While these assumptions simplify the process of cardinality estimation, they can also be the source of estimation errors. Real-world data often exhibit patterns of skewness, correlated columns, and varying inclusion relationships, which deviate from these idealized assumptions. Modern database systems and advanced cardinality estimation techniques strive to account for such complexities to improve the accuracy of their estimations.

To clarify how System R's cardinality estimation technique can be utilized in the context of graph data and query, consider the query in Figure 2.1. In this case, there are two "edge tables" (Table A and Table B) that are joined. We can assume, without loss of generality, that each table contains the source vertex id and target vertex id of the edge labeled accordingly. For this query, the join is between the target vertex column of Table A and the source vertex column of Table B.

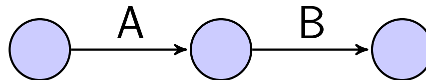


Figure 2.1: A chain query with two edges.

System R estimates the cardinality of the join through the following formula:

$$|A| \times |B| \times \frac{1}{\max(\text{distinct targets of } A, \text{distinct sources of } B)}$$

In Chapter 4, we will adapt System R to streaming graphs.

Chapter 3

GraphSketch Design

This chapter presents the design of a sketch-based estimation method, called *GraphSketch*, that aligns with the constraints and requirements of streaming GDBMSs. *GraphSketch* operates on windowed streams and constructs a data structure and performs cardinality estimation on each window. It groups each graph stream according to the edge labels, and summarize them into a concise summary structure. Cardinality estimation computation then uses these summaries.

GraphSketch compresses the graph and serves as a compact representation of the graph. Each streaming graph tuple (sgt[1]), as discussed in Section 2.1, contains a source vertex ID, a target vertex ID, and an edge label. When streaming sgts arrive, vertices incident to each edge are hashed into one of n buckets, where n is a design parameter. This approach translates the original graph's structure into a condensed graph within *GraphSketch*. Each sgt label has its own *GraphSketch*, which consists of three components (see the class definition in Listing 3.1): a matrix M of size $n \times n$ (i.e., `edgeSketch`), a vector holding the counts of source vertices incident on edges with that label (i.e., `distinctSourceCounts`), and a vector holding the same count for target vertices (i.e., `distinctTargetCounts`). The entry in the matrix $\texttt{edgeSketch}_r\texttt{[i][j]}$ (or $M_r[i, j]$ – we use both notations interchangeably) stores the number of edges from a vertex in bucket i to a vertex in bucket j with label r (we omit r in the remainder when it is not necessary).

This summarization leads to a trade-off between effectiveness and efficiency, a crucial factor for meeting streaming requirements. While *GraphSketch*'s detailed representations enhance the accuracy of cardinality estimation, it also increases processing time due to the complexity of construction and propagation mechanisms. The key advantage of this

[1] We use "edge" and "sgt" interchangeably since an sgt models an edge.

Listing 3.1: Sketch Class Representation

```
public class GraphSketch {
    public int bucketCount;
    private double[][] edgeSketch;
    private double[] distinctSourceCounts;
    private double[] distinctTargetCounts;
}
```

approach lies in finding the sweet spot where the accuracy loss is minimized while gaining significant improvements in processing time. We demonstrate this trade-off through empirical evaluation in Chapter 4.

In the remainder, we demonstrate *GraphSketch* construction by an example in Section 3.1, and more formally in Section 3.2, focusing on a single window. Section 3.2 also discusses cardinality estimation using *GraphSketch*. We present the extension to streaming (multiple windows) in Section 3.3. Finally, a comparison of *GraphSketch* and System R techniques are provided in Section 3.4.

3.1 GraphSketch Construction Example

In this section, we detail the process of constructing *GraphSketch*. We exemplify this process using a simple example graph with a single edge label given in Figure 3.1. The edge table for this graph is shown in Table 3.1.

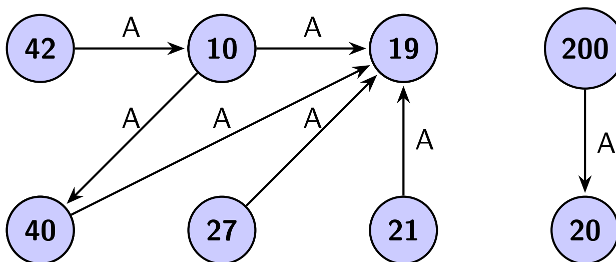


Figure 3.1: Example graph.

Suppose we aim to construct a *GraphSketch* with $n = 3$. We refer to such a sketch as $GraphSketch - 3$ or $GS - 3$. The process involves employing a simple hash function

Source Vertex	Edge Label	Target Vertex
42	A	59
59	A	16
37	A	16
27	A	19
21	A	19
200	A	20
10	A	40

Table 3.1: Edge table for the example graph

$f(x) = x \mod 3$, where x is the vertex ID, to assign each vertex to a bucket. For this example, vertex IDs 42, 27, and 21 are assigned to bucket 0; IDs 59 and 10 are assigned to bucket 1; and IDs 59 and 100 are assigned to bucket 2. Following this mapping, we count the number of edges originating from vertices in one bucket (source) and ending at vertices in another bucket (target). As an example, consider the first entry:

Source Vertex	Relation	Target Vertex
42	A	59

As vertex 42 is mapped to bucket 0 and vertex 59 is mapped to bucket 2, we indicate the presence of this edge by incrementing the entry $M[0, 2]$ in the graph sketch matrix by 1. The dimensions of M are 3×3 (since we have 3 buckets). The resulting graph sketch matrix M contains the number of edges between each pair of buckets, providing a compact representation of the graph data while preserving its topological characteristics. One notable point is that the total sum of counts in the matrix equals the number of edges in the *GraphSketch*. For the given data, we obtain the following edge matrix M_A:

$$M_A = \begin{bmatrix} 0 & 2 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

GraphSketch matrix M_A can be visualized as in Figure 3.2. This visualization demonstrates the relationships captured within the sketch. The numbers on the edges indicate the count of connections between vertices, and self-loops represent multiple connections among vertices in one bucket.

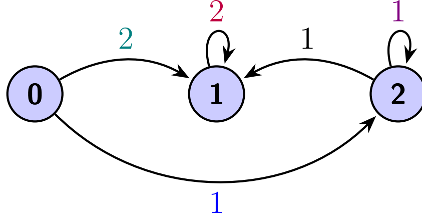


Figure 3.2: GraphSketch-3 visualization of edge label A.

As each sgt is processed, the number of unique source and target elements for each label are counted and stored in two arrays: `distinctSourceCounts` and `distinctTargetCounts`. Each entry i of these arrays stores the count of unique elements in bucket i that are source/-target vertices of an edge. Consider again the first entry in the edge table. For the given edge $42 \xrightarrow{A} 59$, vertex 42 is a distinct source element that is assigned to bucket 0. Therefore, the corresponding element at index 0 in the distinct source array `distinctSourceCounts` is incremented by 1. Similarly, as vertex 59 is a distinct target element assigned to bucket 1, the corresponding element at index 1 in the distinct target array `distinctTargetCounts` is incremented by 1.

After processing the entire dataset, we end up with the following arrays:

$$\text{Distinct Source Vertices} = [3, 2, 2]$$
$$\text{Distinct Target Vertices} = [0, 3, 2]$$

3.2 GraphSketch Construction and Cardinality Estimation

The query processor for which we are developing the cardinality estimation subsystem maps graph patterns to (self-)joins on the edge table. The cardinality estimation, therefore, produces estimates for these joins. There are four distinct graph patterns, so we explore four distinct forms of joins. The Source-Source pattern, denoted PatternType.SS, (Figure 3.3) has a vertex (v_1) that has two outgoing edges to v_2 and v_3. When this pattern is evaluated, the query processor performs a join over the edge table looking for edges where v_1 is the source and the edge labels match. This is called Source-Source, because v_1 is the source of both edges ($\overrightarrow{v_1, v_2}$ and $\overrightarrow{v_1, v_3}$). The other patterns and their execution are defined similarly. The Source-Target pattern, denoted PatternType.ST, (Figure 3.4)

has a vertex (v_1) that has two incoming edges from v_2 and v_3. Target-Source pattern (PatternType.TS), is the inverse of Source-Target, but since we are working with directed graphs, it is considered a separate pattern (Figure 3.5). Finally, the Target-Target pattern (PatternType.TT) has one vertex (v_1) that is the target of two edges (Figure 3.6). The evaluation of queries with the last three patterns follows the same approach that is specified for the Source-Source pattern.

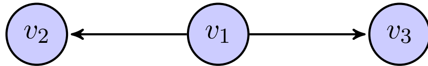


Figure 3.3: Source-Source Pattern

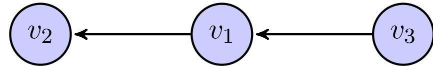


Figure 3.4: Source-Target Pattern

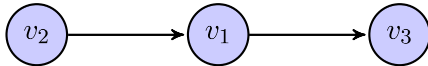


Figure 3.5: Target-Source Pattern

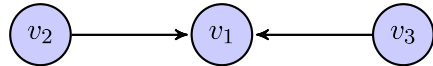


Figure 3.6: Target-Target Pattern

Figure 3.7: A chain query with three edges.

Consider the example query pattern shown in Figure 3.7, which is a chain query with three edges with different labels. The query processor evaluates this query as a sequence of join operations. The first join is over the target vertex of edge X[2] and the source vertex of edge Y, i.e., we are looking for matches between these two vertices. More precisely, the first join operation is between the sgts (edges) in the incoming graph stream with label X and sgts with label Y, where target of X and source of Y are both v_2, producing intermediate result I_{XY}. This is then followed by a join of the intermediate result with the sgts with label Z such that the target vertices of I_{XY} and the source vertex of Z are both v_3. This is a binary join depicted in Figure 3.8.

Our objective is to estimate the cardinality of this query plan. This is performed iteratively, starting from the leafs and ending at the root. Recall that the sum of counts in `edgeSketch` in *GraphSketch* equals the number of edges (modeled as sgts) that is the result of the computation at that node, which gives the cardinality of that computation.

[2]We are abusing the notation slightly by using the edge label to refer to the edge.

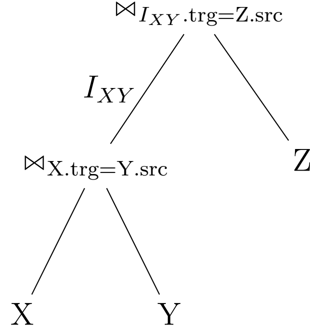


Figure 3.8: Logical plan for the query in 3.7

The first step is to construct the *GraphSketch* for each of the leaf nodes X, Y, and Z (i.e., constructing *GraphSketch* for each edge label), referred as *GraphSketch*(X), *GraphSketch*(Y), and *GraphSketch*(Z), respectively. These provide the cardinality of the sgts (edges) in the input stream with that specific label[3]. The algorithm to construct *GraphSketch* for a leaf node is given in Algorithm 1.

We demonstrate the construction by reference to the example graph given in Figure 3.9 whose edge table is given in Table 3.2.

Edge Label X:

$$M_{\text{X}} = \begin{bmatrix} 0 & 1 & 1 \\ 2 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

Distinct Source Node Counts: [2, 2, 1]
Distinct Target Node Counts: [3, 1, 1]
Total Edge Count: 6

Edge Label Y:

$$M_{\text{Y}} = \begin{bmatrix} 0 & 2 & 2 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

[3]This is analogous to treating a leaf node in relational query trees as `scan` operator.

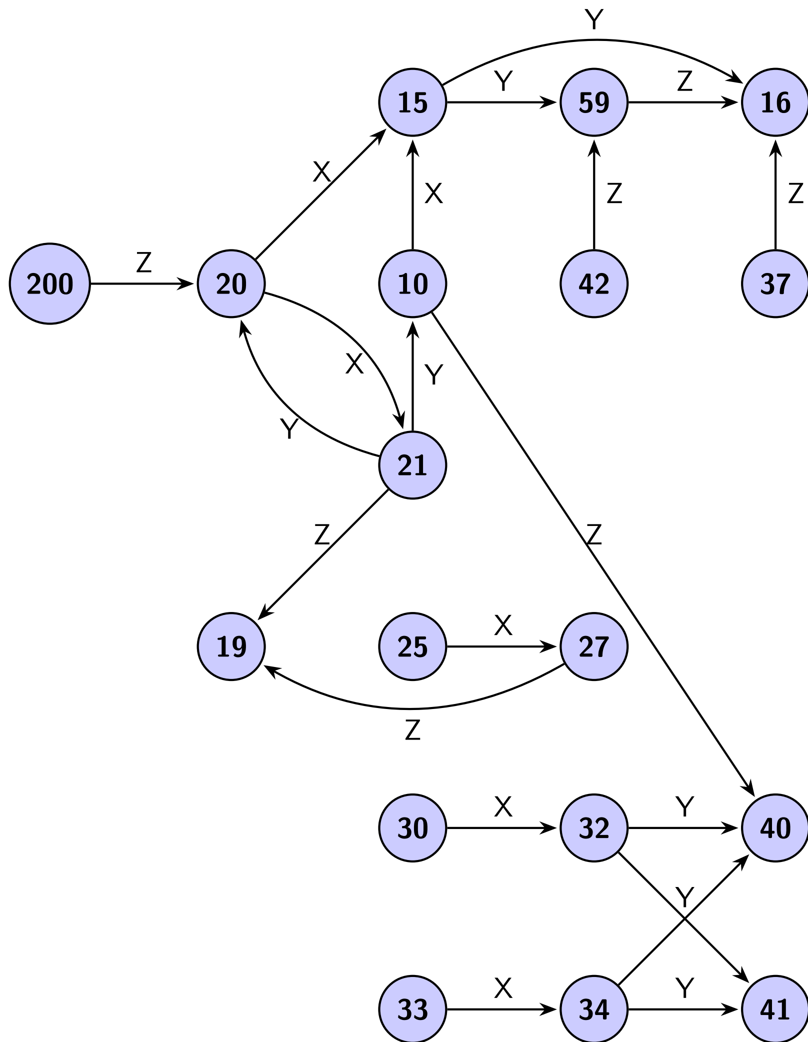


Figure 3.9: A second example graph.

Source Vertex	Edge Label	Target Vertex
10	X	15
20	X	15
20	X	21
25	X	27
30	X	32
33	X	34
15	Y	59
15	Y	16
21	Y	10
21	Y	20
32	Y	40
32	Y	41
34	Y	40
34	Y	41
42	Z	59
59	Z	16
37	Z	16
27	Z	19
21	Z	19
200	Z	20
10	Z	40

Table 3.2: Edge table for the example graph in Figure 3.9.

Algorithm 1 GraphSketch Construction for Leaf Nodes

```
1: function AssignToBucket(vertexID, bucketCount)
2:     return vertexID mod bucketCount
3: end function
4: procedure ConstructGraphSketch(n, edges)
5:     M ← an n × n zero matrix
6:     src_set, trg_set ← arrays of n empty sets each
7:     for each edge (srcID, trgID) in edges do
8:         src_bucket ← AssignToBucket(srcID, n)
9:         trg_bucket ← AssignToBucket(trgID, n)
10:        src_set[src_bucket].add(srcID)
11:        trg_set[trg_bucket].add(trgID)
12:        M[src_bucket, trg_bucket] ← M[src_bucket, trg_bucket] + 1
13:    end for
14:    for k ← 0 to n − 1 do
15:        M.update_dist_src_count_array(k, len(src_set[k]))
16:        M.update_dist_trg_count_array(k, len(trg_set[k]))
17:    end for
18:    return M
19: end procedure
```

Distinct Source Node Counts: [2, 1, 1]
Distinct Target Node Counts: [0, 3, 3]
Total Edge Count: 8

Edge Label Z:

$$M_z = \begin{bmatrix} 0 & 2 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Distinct Source Node Counts: [3, 2, 2]
Distinct Target Node Counts: [0, 3, 2]
Total Edge Count: 7

Once the *GraphSketch* for the leaf nodes are constructed, the process moves up in the query plan tree. The *GraphSketch* construction for a non-leaf node uses those of its children nodes. For example, performing $\mathrm{X} \bowtie_{\mathrm{X}.trg=\mathrm{Y}.src} \mathrm{Y}$ involves joining sgts (edges) whose labels are X and Y, respectively and where the target vertex of the sgts with label X is the same as the source vertex of sgts with label Y. This produces intermediate result I_{XY}. Estimating the cardinality of this join node requires constructing $GraphSketch(\mathrm{X} \bowtie_{\mathrm{X}.trg=\mathrm{Y}.src} \mathrm{Y})$ (we simplify this by referring to the intermediate result sketch $GraphSketch(I_{XY})$) that uses $GraphSketch(\mathrm{X})$ and $GraphSketch(\mathrm{Y})$. Moving up the query plan tree, estimating the cardinality of node $I_{XY} \bowtie_{I_{XY}.trg=\mathrm{Z}.src} \mathrm{Z}$ requires construction of $GraphSketch(I_{XY} \bowtie_{I_{XY}.trg=\mathrm{Z}.src} \mathrm{Z})$ using $GraphSketch(I_{XY})$ and $GraphSketch(\mathrm{Z})$. *GraphSketch* at the root node is used for the cardinality estimate for the entire query. This iterative process of *GraphSketch* construction for non-leaf nodes is shown in Algorithm 2.

Algorithm 2 Cardinality Estimation for Graph Patterns

1: **procedure** EstimateCardinality(joinPattern, leftSketch, rightSketch)
2: $cardinality \leftarrow 0$
3: **if** joinPattern.getPattern() = PatternType.TS **then**
4: **for** $i \leftarrow 0$ to leftSketch.bucketCount **do**
5: **for** $j \leftarrow 0$ to leftSketch.bucketCount **do**
6: **if** $leftEdgeSketch[i][j] > 0$ **then**
7: $leftEdgeCount \leftarrow leftEdgeSketch[i][j]$
8: **for** $k \leftarrow 0$ to rightSketch.bucketCount **do**
9: **if** joinPattern = PatternType.TS and $rightEdgeSketch[j][k] > 0$ **then**
10: $rightEdgeCount \leftarrow rightEdgeSketch[j][k]$
11: $distinctCountLeft \leftarrow leftSketch.getDistinctTrgFromBucket(j)$
12: $distinctCountRight \leftarrow rightSketch.getDistinctSrcFromBucket(j)$
13: $\Delta cardinality \leftarrow \frac{leftEdgeCount \times rightEdgeCount}{distinctCountLeft \times distinctCountRight} \times \min(distinctCountLeft, distinctCountRight)$
14: $cardinality \leftarrow cardinality + \Delta cardinality$
15: **end if**
16: **end for**
17: **end if**
18: **end for**
19: **end for**
20: **end if**
21: **end procedure**

Algorithm 2 offers a structured approach to estimating cardinality in graph join patterns by utilizing sketches. The example query in this section has a Target-Source pattern, where the join connects the target vertex of the left edge with the source vertex of the right edge. The algorithm refers to *leftEdgeSketch* and *rightEdgeSketch* – these are the `edgeSketch`es for the left and right inputs to that join operator, respectively. The notation *leftEdgeSketch*[i][j] in the algorithm refers to $M[i, j]$. The process begins by traversing nested loops to examine bucket pairs in the *leftEdgeSketch*. When encountering non-zero edge counts, these are captured in *leftEdgeCount*. The algorithm then examines the right sketch to establish a connection from $leftEdgeSketch[i][j]$ to $rightEdgeSketch[j][k]$, forming a topological sequence $[i] \rightarrow [j] \rightarrow [k]$ that aligns with the query pattern. This indicates the existence of a path from a vertex in bucket $[i]$ in the $leftEdgeSketch$ to a vertex in bucket $[k]$ in $rightEdgeSketch$ transitively through a vertex in bucket $[j]$ in both edge sketches (since this is the join condition).

Let us now consider how this computation is performed for the example query plan. The first join $\text{X} \bowtie_{\text{X}.tgr=\text{Y}.src} \text{Y}$ that produces I_{XY} is summarized in $GraphSketch(I_{XY})$ – we focus on its edge matrix $edgeSketch(I_{XY})[i][k]$, which is computed from $leftEdgeSketch[i][j] = edgeSketch(\text{X})$ and $rightEdgeSketch[j][k] = edgeSketch(\text{Y})$.

Consequently, the following calculations demonstrate the creation of the output sketch for the given example where $leftEdgeSketch$ and $rightEdgeSketch$ are abbreviated as $leftES$ and $rightES$, respectively:

$$leftES[0][1] \bowtie rightES[1][1] \Rightarrow edgeSketch(I_{XY})[0][1] = \left(\frac{1}{1}\right) \times \left(\frac{1}{1}\right) \times 1 = 1.0$$

$$leftES[0][1] \bowtie rightES[1][2] \Rightarrow edgeSketch(I_{XY})[0][2] = \left(\frac{1}{1}\right) \times \left(\frac{1}{1}\right) \times 1 = 1.0$$

$$leftES[0][2] \bowtie rightES[2][1] \Rightarrow edgeSketch(I_{XY})[0][1] = 1 + \left(\frac{1}{1}\right) \times \left(\frac{1}{1}\right) \times 1 = 2.0$$

$$leftES[0][2] \bowtie rightES[2][2] \Rightarrow edgeSketch(I_{XY})[0][2] = 1 + \left(\frac{1}{1}\right) \times \left(\frac{1}{1}\right) \times 1 = 2.0$$

$$leftES[1][0] \bowtie rightES[0][1] \Rightarrow edgeSketch(I_{XY})[1][1] = \left(\frac{2}{3}\right) \times \left(\frac{2}{2}\right) \times 2 = 1.333$$

$$leftES[1][0] \bowtie rightES[0][2] \Rightarrow edgeSketch(I_{XY})[1][2] = \left(\frac{2}{3}\right) \times \left(\frac{2}{2}\right) \times 2 = 1.333$$

$$leftES[2][0] \bowtie rightES[0][1] \Rightarrow edgeSketch(I_{XY})[2][1] = \left(\frac{2}{3}\right) \times \left(\frac{2}{2}\right) \times 2 = 1.333$$

$$leftES[2][0] \bowtie rightES[0][2] \Rightarrow EdgeSketch(I_{XY})[2][2] = \left(\frac{2}{3}\right) \times \left(\frac{2}{2}\right) \times 2 = 1.333$$

This results in the following $edgeSketch(I_{XY})$:

$$edgeSketch(I_{XY}) = M_{I_{XY}} = \begin{bmatrix} 0 & 2.0 & 2.0 \\ 0 & 1.33 & 1.33 \\ 0 & 1.33 & 1.33 \end{bmatrix} \tag{3.1}$$

with a total estimated edge count of 9.33.

Algorithm 2 specifies the *GraphSketch* construction and cardinality estimation when the query pattern type is Target-Source. Similar algorithms exist for the other three pattern types. The cardinality estimation process involves iterating over pairs of buckets in the *leftEdgeSketch* and examining the corresponding buckets in the *rightEdgeSketch* with respect to the query pattern. Specifically, for PatternType.TT, PatternType.ST, and PatternType.SS, the only change required is in the indexing of the *edgeSketch* matrices to align with the respective join conditions; and these changes are as follows:

For TS and TT pattern types:

- For a TS query, represented as $[i] \xrightarrow{\text{leftLabel}} [j] \xrightarrow{\text{rightLabel}} [k]$, and a TT query, represented as $[i] \xrightarrow{\text{leftLabel}} [j] \xleftarrow{\text{rightLabel}} [k]$, the logic is as follows:

```
leftEdgeCount = leftEdgeSketch[i][j];
if ( PatternType.TS && rightEdgeSketch[j][k] > 0) {
    rightEdgeCount = rightEdgeSketch[j][k];
    }
else if (PatternType.TT && rightEdgeSketch[k][j] > 0) {
    rightEdgeCount = rightEdgeSketch[k][j];
    }
}
```

For ST and SS pattern types:

- For an ST query, represented as $[i] \xleftarrow{\text{leftLabel}} [j] \xleftarrow{\text{rightLabel}} [k]$, and an SS query, represented as $[i] \xleftarrow{\text{leftLabel}} [j] \xrightarrow{\text{rightLabel}} [k]$, the logic is:

```
        leftEdgeCount = leftEdgeSketch[j][i];
        if (PatternType.ST && rightEdgeSketch[k][j] > 0) {
            rightEdgeCount = rightEdgeSketch[k][j];
            }
        else if (PatternType.SS && rightEdgeSketch[j][k] > 0)
          {
            rightEdgeCount = rightEdgeSketch[j][k];
            }
```

In all cases, the end cardinality estimation formula is:

$$cardinality+ = \left(\frac{leftEdgeCount}{distinctCountLeft}\right) \times \left(\frac{rightEdgeCount}{distinctCountRight}\right) \times \min(distinctCountLeft, distinctCountRight) \quad (3.2)$$

In the algorithmic implementation for populating the $GraphSketch(I_{XY})$, we traverse three nested loops with indices i, j, and k. This approach is straightforward for a Target-Source (TS) join pattern, where filling the matrix cell $[i][k]$ is intuitive and straightforward. However, for other patterns like Target-Target or Source-Source, the correct index determination in $GraphSketch(I_{XY})$ can be complex. To accommodate the variability inherent in different join patterns, we utilize the `getOutputIndex` function. This function intelligently adjusts to the specific join pattern and is described as Algorithm 3.

This algorithm adapts to various graph join patterns like TT, TS, ST, and SS. Depending on the pattern, it calculates the appropriate indices for the $edgeSketch(I_{X,Y})$ matrix. The logic within the function uses a `switch` statement to assign the right index based on the pattern type and the specific field (source or target) in the join operation. This dynamic approach ensures accurate cardinality estimations and updates to the $edgeSketch(I_{X,Y})$ matrix, accommodating different graph join scenarios.

Since $GraphSketch$ requires distinct counts for the vertices involved in the join operation, we lastly explain the distinct count calculation. Obtaining statistics is straightforward for the leaf nodes of the join tree as we can directly count distinct elements and edge occurrences without requiring any propagation mechanism. However, at intermediate levels, directly counting these elements or occurrences is not feasible. Instead of counting, we need a method to derive these values from the operator and its inputs. To estimate distinct counts, we first create a bound for our estimation and then employ a uniform sampling with a replacement model for estimation. This is depicted in Algorithm 4.

Algorithm 3 Output Index Determination For Graph Patterns

```
 1: function getOutputIndex(outputField, type, i, j, k)
 2:     index ← 0
 3:     if type = PatternType.TT then
 4:         index ← switch (outputField)
 5:             case SRC1: return i
 6:             case SRC2: return k
 7:             case TRG1, TRG2: return j
 8:     else if type = PatternType.TS then
 9:         index ← switch (outputField)
10:             case SRC1: return i
11:             case TRG1, SRC2: return j
12:             case TRG2: return k
13:     else if type = PatternType.ST then
14:         index ← switch (outputField)
15:             case SRC1, TRG2: return j
16:             case SRC2: return k
17:             case TRG1: return i
18:     else if type = PatternType.SS then
19:         index ← switch (outputField)
20:             case SRC1, SRC2: return j
21:             case TRG1: return i
22:             case TRG2: return k
23:     end if
24:     return index
25: end function
```

Note that the number of distinct counts varies for each bucket, and therefore this calculation should be done for each bucket separately. By the end of this process, we fill the arrays `distinctSourceCounts` and `distinctTargetCounts` in Listing 3.1. These arrays have `bucketCount` number of elements. Consider, again, the first join in example query given in Figure 3.8: X $\bowtie_{\mathrm{X}.trg=\mathrm{Y}.src}$ Y. The process starts with a loop iterating over each bucket in the $edgeSketch(I_{XY})$. We first retrieve the total number of elements inside the bucket to use it as a bound for our estimation. We compute the number of edges per bucket by adding incoming edges to that bucket and outgoing edges from that bucket. If we need to compute the number edges for bucket i, we retrieve the total sum of the elements in the columns and rows i of $edgeSketch$ matrix.

Algorithm 4 Distinct Count Calculation in Graph Sketch

1: **for** $i \leftarrow 0$ to GraphSketch.bucketCount **do**
2: $resultSize = edgeSketch.getBucketSize(i)$
3: $srcEstimate \leftarrow estBucketwiseDistCnt(leftSketch, rightSketch, joinPattern, outputSrc, i)$
4: $distSrcCount \leftarrow \min(srcEstimate, resultSize)$
5: $trgEstimate \leftarrow estBucketwiseDistCnt(leftSketch, rightSketch, joinPattern, outputTrg, i)$
6: $distTrgCount \leftarrow \min(trgEstimate, resultSize)$
7: $GraphSketch.setDistinctSrcForBucket(i, distSrcCount)$
8: $GraphSketch.setDistinctTrgForBucket(i, distTrgCount)$
9: **end for**

The *estBucketwiseDistCnt* function is employed to perform distinct count estimations (Algorithm 5). It has five arguments: $leftSketch$, $rightSketch$, and $joinPattern$ are obvious; i specifies the bucket number for estimation in *estBucketwiseDistCnt*, and $outputField$ determines whether to estimate distinct source or target counts. Last parameter is set by *pattern.getOutputTrg*() or *pattern.getOutputSrc*(), respectively, to suit specific join needs. Initially, the function assesses the join pattern and output fields. In cases where the join pattern is Source-Source, the function compares the number of distinct source counts from both the left and right inputs. It then returns the minimum of these counts, reflecting the smallest set of distinct elements involved in the join.

For other join patterns, this function adopts a different approach. It estimates the number of distinct counts by utilizing a uniform sampling with replacement model. This model is particularly effective in cases where direct calculation of distinct counts is not feasible due to the complexity or size of the data.

Algorithm 5 takes the product of the bucket sizes of the left and right *edgeSketch*es and assigns this to *productSize*, which reflects the total number of combinations resulting from this join. *productSize* is used as a bound for the estimation. Then, based on the indices, the algorithm picks the corresponding vertex counts from corresponding buckets. Lastly, it retrieves the size of the desired bucket. These three values are used as parameters to call function *estimatePopulationSize* that computes

(a) the *degree* of both the source and the target (reflecting the average number of connections per element and offering insights into the elements' connectedness),

(b) the *samplingRatio*, which is the ratio of the estimated cardinality to the *productSize*

(assessing the representativeness of our sample in the context of the join's total combinations),

(c) the probability of no edge being sampled *probNoEdgeSampled* (represents the probability of an individual edge not being in the sample), and

(d) the probability of an edge being sampled *probEdgeSampled* (represents the probability that any given vertex has at least one of its edges included in the sample).

Since the graph has *nVertices* vertices, *estimatePopulationSize* returns the product of *nVertices* and *probEdgeSampled* as the expected number of vertices that are effectively connected through the sampled edges. The result is an estimation of the size of the "active" or "connected" vertex population within the context of the sampled graph.

Algorithm 5 Estimate Bucketwise Distinct Vertex Count

```
1: function estBucketwiseDistCnt(leftSketch, rightSketch, joinPattern, outputField, i)
2:     resultSize ← edgeSketch.getBucketSize(i)
3:     leftSrc ← leftSketch.getDistinctSrcFromBucket(i)
4:     rightSrc ← rightSketch.getDistinctSrcFromBucket(i)
5:     leftTrg ← leftSketch.getDistinctTrgFromBucket(i)
6:     rightTrg ← rightSketch.getDistinctTrgFromBucket(i)
7:     leftPatternCount ← joinPattern.isFirstSource() ? leftSrc : leftTrg
8:     rightPatternCount ← joinPattern.isSecondSource() ? rightSrc : rightTrg
9:     if ((outputField.isFirst() and joinPattern.isFirstSource() == outputField.isSrc())
       or (not outputField.isFirst() and joinPattern.isSecondSource() == outputField.isSrc())) then
10:        return min(leftPatternCount, rightPatternCount)
11:    end if
12:    leftSize ← leftSketch.getBucketSize(i)
13:    rightSize ← rightSketch.getBucketSize(i)
14:    productSize ← multiply(leftSize, rightSize)
15:    originalVertexCount ← outputField.isFirst() ? (outputField.isSrc() ? leftSrc :
       leftTrg) : (outputField.isSrc() ? rightSrc : rightTrg)
16:    return estimatePopulationSize(productSize, originalVertexCount, resultSize)
17: end function
18: function EstimatePopulationSize(nEdges, nVertices, sampleSize)
19:    degree ← nEdges/nVertices
20:    samplingRatio ← sampleSize/nEdges
21:    probNoEdgeSampled ← Math.pow(1.0 − samplingRatio, degree)
22:    probEdgeSampled ← 1.0 − probNoEdgeSampled
23:    return nVertices × probEdgeSampled
24: end function
```

Applying our example data to Algorithm 4 yields the following results:

Distinct Source Node Counts: [2, 1, 1]
Distinct Target Node Counts: [2, 1, 1]

These counts reflect the distinct sources and targets in each bucket after processing. Additionally, we refer back to the edge sketch matrix we constructed earlier:

$$M_{XY} = \begin{bmatrix} 0 & 2.0 & 2.0 \\ 0 & 1.33 & 1.33 \\ 0 & 1.33 & 1.33 \end{bmatrix} \tag{3.3}$$

Finally, we join this resulting graph stream I_{XY} with Z to complete the three-edge query. After applying our algorithms, we retrieve the cardinality estimation result of 9.33.

This demonstrates the basic functionality of the proposed algorithm. By creating sketches of each graph stream and subsequently joining these sketches, the result of complex graph queries can be estimated efficiently. This methodology allows for querying even very large graphs in a practical and timely manner.

3.3 Incremental Updates of the GraphSketch

Up to now, we have not considered streaming graphs and considered the original edge table as static and fully available. Considering the streaming nature of the input, the computation (both the construction of *GraphSketch* and the computation of cardinalities) has to be done incrementally and fast. As noted earlier, we use sliding windows to manage streaming sgts. We perform the *GraphSketch* construction and cardinality estimation per window.

In this section, we first describe our approach to managing time-based sliding windows. Following this, we outline our baseline method. Then, we detail our method for performing incremental updates on *GraphSketch* structures with a focus on efficiency.

3.3.1 Window Management in Streaming Environment

We maintain a data structure that represents the sliding window into which new sgts are inserted and from which expired sgts are deleted. The processing of continuous data streams are managed through Algorithm 6.

Algorithm 6 Window Management in Streaming Environment

```
 1: global variables
 2:    WINDOW_SIZE                                    {total duration of the window}
 3:    SLIDE_INTERVAL                {number of time units for the window's shift}
 4:    WINDOW_START                                   {start timestamp of window}
 5:    WINDOW_END                                       {end timestamp of window}
 6: end global variables
 7: WINDOW_START ← 0                                {this is the initial window start}
 8: WINDOW_END ← WINDOW_START + WINDOW_SIZE
 9: currentWindowData ← initialize current window
10:
11: function slideWindow
12:     WINDOW_START ← WINDOW_START + SLIDE_INTERVAL
13:     WINDOW_END ← WINDOW_START + WINDOW_SIZE
14: end function
15:
16: while true do
17:     if sgt.timestamp ⩾ WINDOW_SIZE then           {sgt is the new arriving one}
18:         processCurrentWindow(currentWindowData)
19:         slideWindow()
20:         evictExpiredEdges(currentWindowData)
21:     end if
22:     currentWindowData.add(sgt)
23: end while
```

- **Initialization:** Algorithm 6 uses two global variables: `WINDOW_SIZE`, which specifies the total duration of the sliding window, and `SLIDE_INTERVAL` specifies when the window slides forward. The variable `WINDOW_START` is initialized to set the commencement time of the window, marking the beginning of the time frame for data processing. Furthermore, `WINDOW_END` is calculated as the sum of `WINDOW_START` and `WINDOW_SIZE`, effectively determining the window's termination point in time.

This endpoint signifies the closure of the current window's active period. Lastly, `currentWindowData` is initialized to establish a data structure for storing and managing the window's current content, providing a foundation for subsequent data processing and analysis within the defined time frame.

- **Processing sgts (edges):** The algorithm iterates over each sgt (edge) in the streaming data. Each sgt is added to the `currentWindowData` as they arrive.

- **Window Slide:** When an sgt's timestamp exceeds the `WINDOW_SIZE`, the algorithm slides the window by predefined variable `SLIDING_INTERVAL`. This is an easy way to detect that time has moved beyond the current `WINDOW_END`. Window movement is achieved through the $slideWindow()$ function, which updates the window's temporal boundaries through sliding the window forward.

- **sgt (edge) eviction:** $evictExpiredEdges(currentWindowData)$ function is called to remove edges that are no longer within the new window.

3.3.2 Baseline Approach

The baseline approach is to construct *GraphSketch* from scratch every time the window moves. The $processCurrentWindow()$ function (Algorithm 7) handles *GraphSketch* construction and query processing.

Algorithm 7 Baseline Window Processing and Sketch Construction

```
1: function processCurrentWindow(currentWindowData)
2:     edgesByLabel ← groupEdgesByLabel(currentWindowData)
3:     for each edge label do
4:         edgesForLabel ← edgesByLabel.get(label)
5:         ConstructGraphSketch(n, edgesForLabel)        {n is bucketCount}
6:     end for
7: end function
```

Algorithm 7 is straightforward and implements the techniques discussed in the previous two sections on the contents of the current window. Its operations are as follows:

1. Groups edges by labels using *groupEdgesByLabel*, segregating edges from `currentWindowData` by label.

2. Iterates over each label and for each label (assume there are n labels), it:

   - obtains edges for the corresponding label using $edgesByLabel.get(label)$,
   - updates the sketch with these edges by invoking the $ConstructSketch$ function, as detailed in Algorithm 1.

3.3.3 Updating GraphSketch Incrementally

Constructing sketches from scratch for each window certainly works, but can incur significant costs, particularly as the window duration increases. Thus, we develop an approach to incrementally maintain *GraphSketches* as windows move.

For ease of presentation, in the following, we assume that all labels are known. This allows us to describe processes and structures using an array. We maintain an array, `SketchList`, which comprises a *GraphSketch* for each label. Similar to the role of `currentWindow` in managing the window's state, `SketchList` maintains *GraphSketch* instances for each label. These *GraphSketch* instances are initially constructed as outlined in Algorithm 1, and then utilized consistently throughout the entire processing period. As windows move, `SketchList` is updated to reflect both the addition and deletion of edges. These changes are applied to the respective *GraphSketch* instances in `SketchList`, maintaining an up-to-date graph state representation for each label.

Algorithm 8 shows the revised window management; compared with Algorithm 6, the main difference is that $processCurrentWindow$ and $evictExpiredEdges$ now gets `SketchList` as a parameter in addition to `currentWindowData`. This is necessary, as it allows for the direct interaction between the window's current state and the corresponding *GraphSketch* instances. Through the use of `SketchList`, $processCurrentWindow$ integrates incoming edges into each *GraphSketch* instance based on `currentWindowData`. Similarly, $evictExpiredEdges$ utilizes `SketchList` to systematically remove edges that are no longer within the current window from their respective *GraphSketch* instances. After these operations, *GraphSketch* instances in `SketchList` align with the latest state of the window.

Algorithm 9 details the addition of new streaming edges to their respective *GraphSketch* instance in `SketchList` as they enter the window. Algorithm 11 describes the removal process for edges that are no longer within the window. These processes provide a method for the ongoing update of *GraphSketch* instances, differentiating from the original approach of constructing them from scratch as shown in Algorithm 7. The critical components are

Algorithm 8 Incremental Window Management

```
1: global variables
2:     WINDOW_SIZE                                  {total duration of the window}
3:     SLIDE_INTERVAL                {number of time units for the window's shift}
4:     WINDOW_START                                    {start timestamp of window}
5:     WINDOW_END                                        {end timestamp of window}
6: end global variables
7: WINDOW_START ← 0                                {this is the initial window start}
8: WINDOW_END ← WINDOW_START + WINDOW_SIZE
9: SketchList ← initialize GraphSketch instances
10: currentWindowData ← initialize current window
11: function slideWindow
12:     WINDOW_START ← WINDOW_START + SLIDE_INTERVAL
13:     WINDOW_END ← WINDOW_START + WINDOW_SIZE
14: end function
15:
16: while true do
17:     if sgt.timestamp ⩾ WINDOW_SIZE then          {sgt is the new arriving one}
18:         processCurrentWindow(currentWindowData, SketchList)
19:         slideWindow()
20:         evictExpiredEdges(currentWindowData, SketchList)
21:     end if
22:     currentWindowData.add(sgt)
23: end while
```

how these incremental updates are accomplished. These are described in Algorithms 10 for sgt additions, and 12 for sgt deletions.

Building upon the framework established in Algorithm 8, Algorithm 10 further elaborates on the specific procedures for incorporating new sgts (edges) into *GraphSketch* instances. It details how sgts, associated with particular labels, are incorporated into the corresponding *GraphSketch* instances. The addition process includes several steps:

1. **Vertex storage initialization.** The function begins by initializing a data structure, `vertexStorage`, to hold distinct source and target vertices for each bucket.

2. **sgt (edge) processing.** For each edge in the provided list `edgesForLabel`, the function performs the following:

Algorithm 9 Efficient Addition Algorithm

```
1: function processCurrentWindow(currentWindow, SketchList)
2:     edgesByLabel ← groupEdgesByLabel(currentWindow)
3:     for each edge label do
4:         edgesForLabel ← edgesByLabel.get(label)
5:         SketchList.get(label).add(edgesForLabel)
6:     end for
7: end function
```

- Computes hash values for both the source and target vertices of the edge.
- Increments the edge count between the corresponding source and target buckets in `edgeSketch`.
- Stores the vertices in `vertexStorage`, ensuring that each bucket maintains a record of distinct source and target vertices.

3. **Distinct count update.** The function concludes by updating the distinct source and target counts for each bucket. It calculates these counts based on the size of the vertex sets stored in `vertexStorage`.

Through these operations, the `add` function effectively integrates new edge data into the sketch, keeping it updated with the latest information from the streaming data.

Now consider edge deletions (Algorithm 11). It details how sgts, associated with particular labels, are deleted from the existing *GraphSketch* instances. The deletion process includes several steps:

1. **Edge (sgt) identification:** The function begins by identifying edges to be deleted with *getEdgesToDelete* method. It filters which edges in `currentWindowData` are outside the current window.
2. **Edge (sgt) categorization:** Then, these edges are categorized by their respective labels and stored in a map structure, denoted as `edgesByLabel = Map⟨Label, List⟨Edge⟩⟩`. In this structure, each label acts as a key, mapping to a corresponding list of edges that are identified for removal under that specific label.
3. **Edge (sgt) deletion:** The deletion process for each label involves three steps. First, the list of edges, `edgesForLabel`, for the specific label are retrieved. Next, the corresponding *GraphSketch* is accessed from the `SketchList`. Finally, the *delete* method is executed on the sketch with `edgesForLabel` to remove these edges.

Algorithm 10 Sketch Addition Algorithm

```
1: procedure ADD(edgesForLabel)
2:     vertexStorage ← new ArrayList<ArrayList<HashSet<Integer>>>()
3:     for i ← 0 to bucketCount − 1 do
4:         innerList ← new ArrayList<HashSet<Integer>>()
5:         for j ← 0 to 1 do
6:             innerList.add(new HashSet<Integer>())
7:         end for
8:         vertexStorage.add(innerList)
9:     end for
10:    for each edge in edgesForLabel do
11:        srcHashVal ← HASH(edge.getSourceID(), bucketCount)
12:        trgHashVal ← HASH(edge.getTargetID(), bucketCount)
13:        edgeSketch[srcHashVal][trgHashVal] + +
14:        vertexStorage.get(srcHashVal).get(0).add(edge.getSourceID())
15:        vertexStorage.get(trgHashVal).get(1).add(edge.getTargetID())
16:    end for
17:    for i ← 0 to bucketCount do
18:        distinctSourceCounts[i]+ = vertexStorage.get(i).get(0).size()
19:        distinctTargetCounts[i]+ = vertexStorage.get(i).get(1).size()
20:    end for
21: end procedure
```

4. **Window Update:** The function concludes by updating `currentWindow`. It removes `edgesToDelete` from the window, thereby ensuring that only the current active edges are retained.

The *delete* function in Algorithm 12 removes expired edge data from each *GraphSketch* instance to reflect the current state of the graph. The design of the *delete* function mirrors that of the *add* function, outlined in Algorithm 10. While the *add* function incorporates edges through addition, the *delete* function executes their removal through subtraction.

3.4 Comparison with System R's Method

Since we use System R as our baseline, it is helpful to highlight the differences of GraphSketch with System R. For the given dataset, join operation, and join tree, System R would

Algorithm 11 Efficient Deletion Algorithm

```
1: function evictExpiredEdges(currentWindowData, SketchList)
2:     edgesToDelete ← getEdgesToDelete(currentWindowData)
3:     edgesByLabel ← groupEdgesByLabel(edgesToDelete)
4:     for label ← 0 to n do
5:         edgesForLabel ← edgesByLabel.get(label)
6:         SketchList.get(label).delete(edgesForLabel)
7:     end for
8:     currentWindow.delete(edgesToDelete)
9: end function
```

operate as follows.

First, it processes the join between X.trg and Y.src:

$$|X| \times |Y| \times \frac{1}{\max(\text{distinct targets of } X, \text{distinct sources of } Y)}$$

This would produce the following value:

$$6 \times 8 \times \frac{1}{\max(5,4)} = 9.6$$

This value would be System R's cardinality estimate for the join between X.trg and Y.src. However, an additional step is also required to perform the join operation, and System R would require distinct count statistics for the intermediate stream I_{XY}.

For the leaves, obtaining statistics is straightforward: the distinct elements and edge occurrences can be counted. However, for non-leaf nodes, directly counting these elements or occurrences is impractical. Instead of counting, System R employs a uniform sampling with a replacement model. The statistics derivation is as follows:

Given Cartesian product $I_{XY} = |\text{X}| \times |\text{Y}| = 6 \times 8$, the sampling ratio is SamplingRatio $= \frac{\text{Estimated Cardinality}}{|I_{XY}|} = \frac{9.6}{48}$.

Next, the maximum distinct sources and targets are determined: maxDistinctSource $= \max(4,5) = 5$ and maxDistinctTarget $= \max(5,6) = 6$.

The degrees as $degree = \frac{\text{Number of Edges}}{\text{Number of Vertices}} = \frac{|I_{XY}|}{\max(\max(\text{DistinctSource}),\max(\text{DistinctTarget}))}$. Hence, for sources, Source Degree $= \frac{48}{5} = 9.6$, and for targets, Target Degree $= \frac{48}{6} = 8$.

Algorithm 12 Sketch Deletion Algorithm

```
1:  procedure DELETE(edgesForLabel)
2:      vertexStorage ← new ArrayList<ArrayList<HashSet<Integer>>>()
3:      for i ← 0 to bucketCount − 1 do
4:          innerList ← new ArrayList<HashSet<Integer>>()
5:          for j ← 0 to 1 do
6:              innerList.add(new HashSet<Integer>())
7:          end for
8:          vertexStorage.add(innerList)
9:      end for
10:     for each edge in edgesForLabel do
11:         srcHashVal ← HASH(edge.getSourceID(), bucketCount)
12:         trgHashVal ← HASH(edge.getTargetID(), bucketCount)
13:         edgeSketch[srcHashVal][trgHashVal] − −
14:         vertexStorage.get(srcHashVal).get(0).add(edge.getSourceID())
15:         vertexStorage.get(trgHashVal).get(1).add(edge.getTargetID())
16:     end for
17:     for i ← 0 to bucketCount do
18:         distinctSourceCounts[i]− = vertexStorage.get(i).get(0).size()
19:         distinctTargetCounts[i]− = vertexStorage.get(i).get(1).size()
20:     end for
21: end procedure
```

With this, System R is able to calculate for each vertex:

$$\text{Probability of No Edge Sampled} = (1 - \text{SamplingRatio})^{\text{degree}}$$

and (using the calculated distinct count formula):

$$\text{CalculatedDistFormula} = (1 - \text{Probability of No Edge Sampled}) \times \text{Number of Distinct Vertices}$$

For the example:

$$\text{Distinct Sources} = (1 - (0.8)^{9.6}) \times 5 \approx 4.41$$
$$\text{Distinct Targets} = (1 - (0.8)^{8}) \times 6 \approx 4.99$$

For the final step, the derived statistics are used to calculate the cardinality of the root node :

$$
\begin{aligned}
&|I_{XY}| \times |\mathrm{Z}| \times \frac{1}{\max(\text{distinct targets of } I_{XY}, \text{distinct sources of Z})} \\
&= 9.6 \times 7 \times \frac{1}{\max(4.99, 7)} \\
&= 9.6
\end{aligned}
$$

When *GraphSketch* operates with a single bucket, GS-1, it provides the coarsest level of detail, and produces the same result as System R. To demonstrate this equivalence, we reexamine a specific example. Let's start by constructing individual sketches for the leaf nodes.

Edge Label X:

$$M_X = \begin{bmatrix} 6 \end{bmatrix}$$

$$
\begin{aligned}
\text{Distinct Source Node Counts: } & [5] \\
\text{Distinct Target Node Counts: } & [5] \\
\text{Total Edge Count: } & 6
\end{aligned}
$$

Edge Label Y:

$$M_Y = \begin{bmatrix} 8 \end{bmatrix}$$

$$
\begin{aligned}
\text{Distinct Source Node Counts: } & [4] \\
\text{Distinct Target Node Counts: } & [6] \\
\text{Total Edge Count: } & 8
\end{aligned}
$$

GS-1 comprises a single-element matrix, representing the total count of edges with this label. For edge label X, the distinct source and target node counts are both 5, indicating

the diversity of connections in this edge set. Edge label Y has a sketch with 4 unique sources and 6 unique targets.

When the sketch for the first join X $\bowtie_{\text{X}.trg=\text{Y}.src}$ (Y) is computed and the estimation algoiritm is applied, we obtain the following sketch result:

$$leftES[0][0] \bowtie rightES[0][0] \Rightarrow edgeSketch(I_{XY})[0][0] = \left(\frac{6}{5}\right) \times \left(\frac{8}{4}\right) \times 4 = 9.6$$

The estimated cardinality of joining the sketches for X and Y, i.e., estimate of I_{XY} is 9.6. We then move to the join at the root of the query plan: $I_{XY} \bowtie_{I_{XY}.trg=\text{Z}.src}$ Z. For these the following computations are done:

For I_{XY}:

$$M_{XY} = \begin{bmatrix} 9.6 \end{bmatrix}$$

Distinct Source Node Counts: [5]
Distinct Target Node Counts: [4]
Total Estimated Edge Count: 9.6

Edge Label Z:

$$M_Z = \begin{bmatrix} 7 \end{bmatrix}$$

Distinct Source Node Counts: [7]
Distinct Target Node Counts: [5]
Total Edge Count: 7

And finally:

$$leftES[0][0] \bowtie rightES[0][0] \Rightarrow EdgeSketch(I_{X,Y})[0][0] = \left(\frac{9.6}{4}\right) \times \left(\frac{7}{7}\right) \times 4 = 9.6$$

This computation confirms that the cardinality estimate for the joined graph remains consistent at 9.6, thereby establishing the equivalency of GraphSketch-1 with the System R method.

Chapter 4

Implementation and Evaluation

4.1 Implementation

GraphSketch cardinality estimation technique has been implemented on top of the S-Graffito Streaming Graph Management System[1]. S-Graffito query processor encompasses multiple components. In this project, we enhanced the S-Graffito system by integrating parsers and implementing a novel cardinality estimation technique, as detailed in Chapter 3, to facilitate a more streamlined and efficient query optimization and processing pipeline.

The system includes a parser, a query optimizer, and a query execution engine. Although the S-Graffito system implements a powerful streaming graph query (SGQ) model [49], our focus is limited to cardinality estimation of simple paths, stars and cycles. These queries can easily be formulated by standard SQL and the workload generator produces SQL queries that are input to the parser that generates logical query plans as output. These query plans are subsequently input into the query optimizer, where they undergo optimization in accordance with Streaming Graph Algebra operators [49], resulting in optimized plans as output. Finally, the query execution engine takes the optimized query plans as input, executes them, and measures their latencies. We give more details about the query optimizer component as it's the main focus of this thesis. It should be noted that each of these components undergoes slight implementation modifications to accommodate the experiments we conduct. Our query processing pipeline is visualized in Figure 4.1.

[1] https://dsg-uwaterloo.github.io/s-graffito/

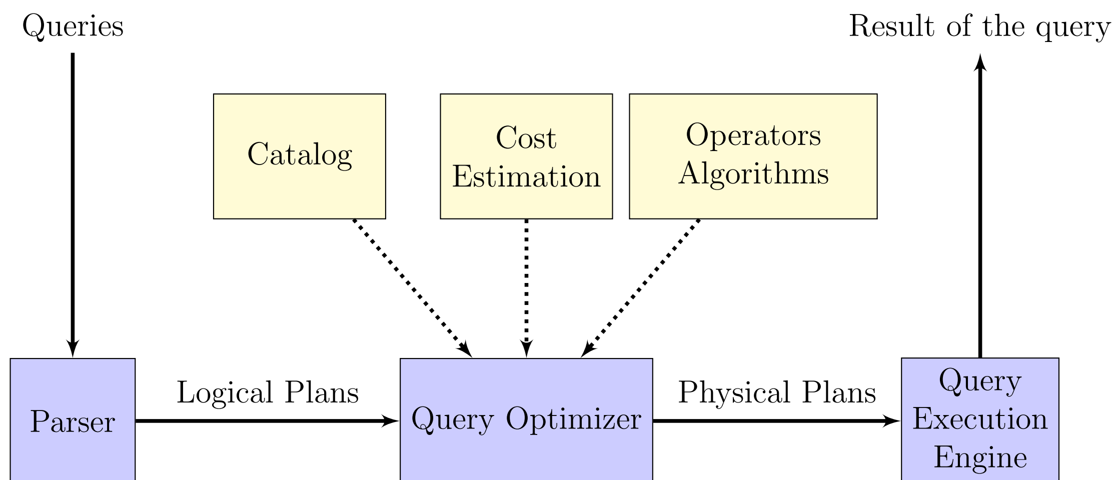


Figure 4.1: SGrafitto's Query Processing Pipeline.

4.2 Source Code

Each component of our system is implemented in a different programming language because of practical reasons. The query parser implementation is in Python, utilizing the SQL Glot library. The query optimizer extends Apache Calcite's in-memory implementation, utilizing Java 13. Lastly, the query execution engine is written in Rust, which leverages the Timely Dataflow [46] framework for managing and executing data-parallel dataflow computations. Consequently, we have three separate repositories for the source code. Our source code for this project is available at the following links:

- https://github.com/keremakillioglu/sgraffito-ground-truth-generator
- https://github.com/keremakillioglu/sgq-cardinality-estimation
- https://github.com/keremakillioglu/sgraffito-query-execution.

Instructions on how to run the code are provided in the repository readme.

4.3 Experimental Platform

Experiments are run on a Linux server of Xeon(R) Platinum 8380 CPU with 160 cores and 2 threads per core, resulting in a total of 320 logical processing units and 1 Terabyte of

DDR4 RAM. The memory architecture was complemented by a multi-level cache hierarchy.

4.4 Evaluation

Our evaluation of the cardinality estimation method encompasses three key aspects: accuracy, effectiveness, and efficiency. In our accuracy evaluation, we check how close our estimations are to the true cardinalities of queries. In the effectiveness evaluation, we examine the reduction in query latency attributed to our accurate estimations. Finally, the efficiency evaluation concentrates on the promptness of our method in a streaming environment, highlighting the significance of our implementation's incremental maintenance.

It should be noted that the accuracy experiments are conducted in a static environment, recognizing that streaming constraints do not affect accuracy. Implementing sliding windows at this stage would necessitate processing each query with every window slide, a method that is impractical and does not influence the accuracy of results. For the effectiveness experiments, we measure the latency of the queries optimized in the accuracy experiments. Finally, we demonstrate our method's effectiveness in the streaming experiments.

4.5 Dataset

In our experiments, we utilize gMark graph data generator [6] to create a synthetic dataset that mimics the first three months of the StackOverflow graph [53]. StackOverflow dataset is a temporal graph, and its first three months capture 99,689 interactions (edges) among 9,872 users (vertices), with the interactions categorized under 3 labels. Each directed edge (u, v) with timestamp t denotes an interaction between two users: (i) user u answered user v's questions at time t, (ii) user u commented on user v's question, or (iii) comment at time t. For the StackOverflow dataset's first three months, the source vertex degree distribution, target vertex degree distribution, and the edge label distribution are respectively illustrated in Figures 4.4, 4.6 and 4.2.

Transitioning to the objectives behind our synthetic dataset creation, we intended not only to reflect the StackOverflow data's real-world characteristics—such as edge count, distinct vertex count, and vertex degree distribution—but also to incorporate a wider range of labels. This approach allows us to assess the robustness of our method in a context that is reflective of the StackOverflow dataset's complexity with an added dimension of label diversity.

Our synthetic dataset mirrors the characteristics mentioned above while featuring 9,553 unique vertices and maintaining the same edge count; but expands the number of edge labels to 10. The corresponding source vertex degree distribution, target vertex degree distribution and edge label distribution for our dataset are presented in Figures 4.5, 4.7 and 4.3, respectively. Since we maintain the same edge count as the dataset we modeled after, we assigned identical timestamps for the corresponding edges.

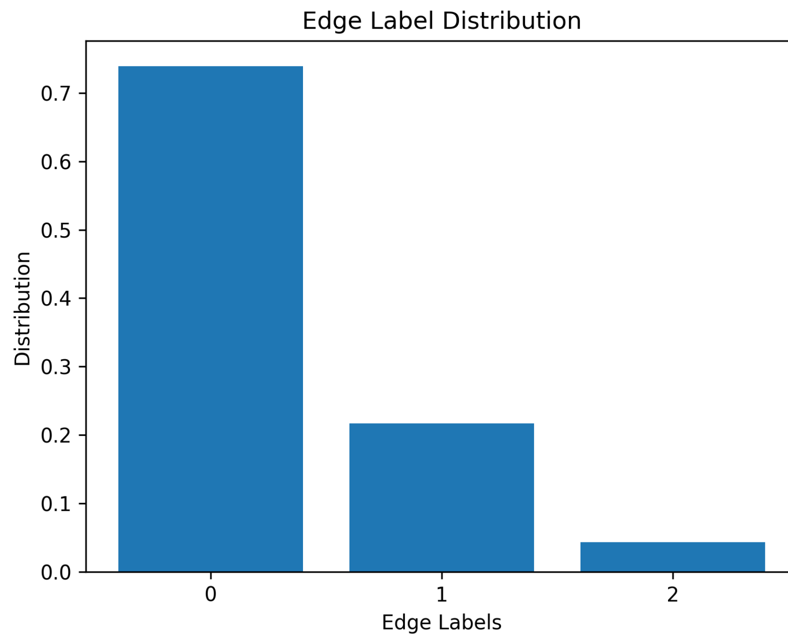


Figure 4.2: Edge Label Distribution: First Three Months of StackOverflow Graph

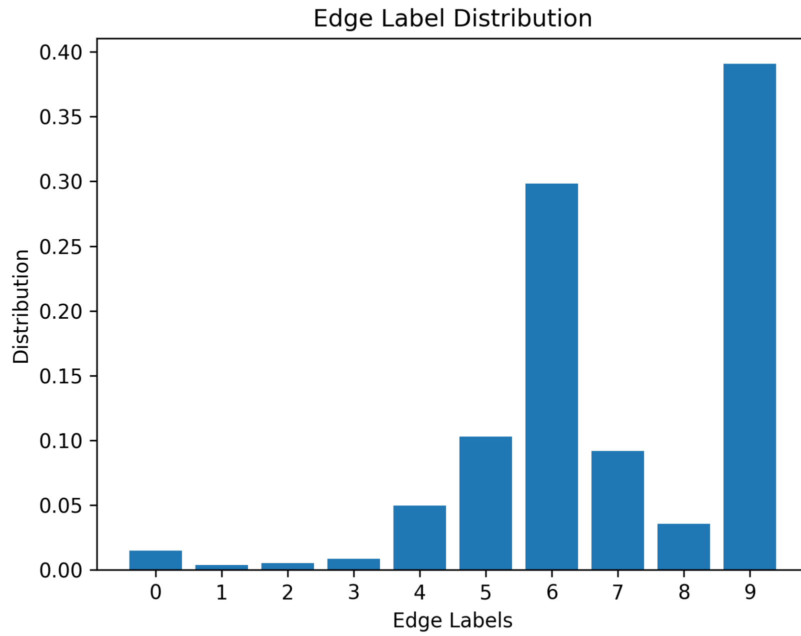


Figure 4.3: Edge Label Distribution: Synthetic Graph

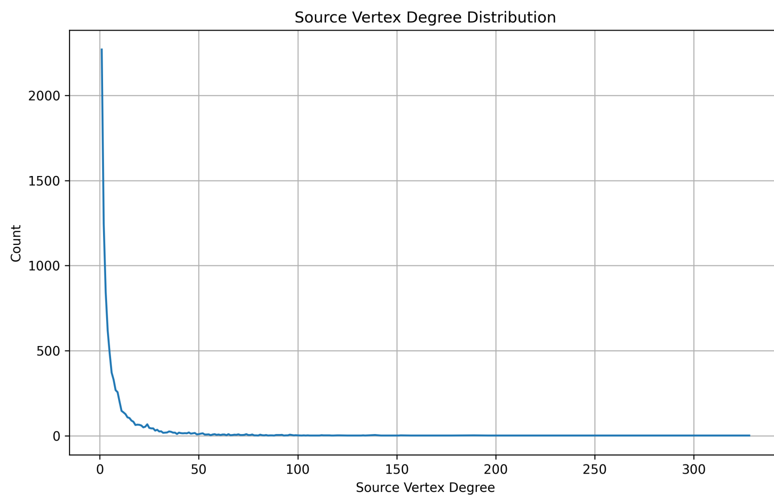


Figure 4.4: Source Vertex Distribution: First Three Months of StackOverflow Graph

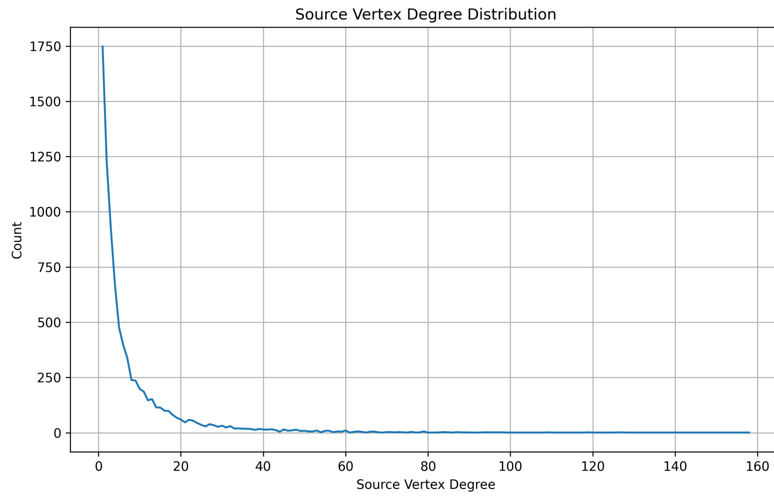


Figure 4.5: Source Vertex Distribution: Synthetic Graph

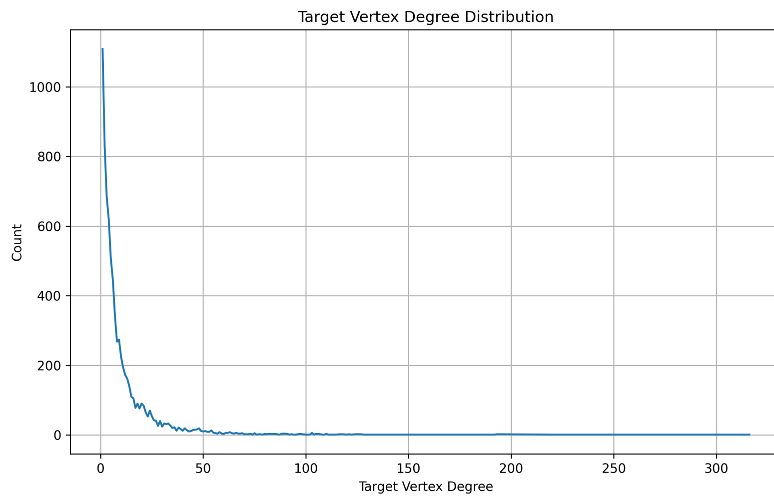


Figure 4.6: Target Vertex Distribution: First Three Months of StackOverflow Graph

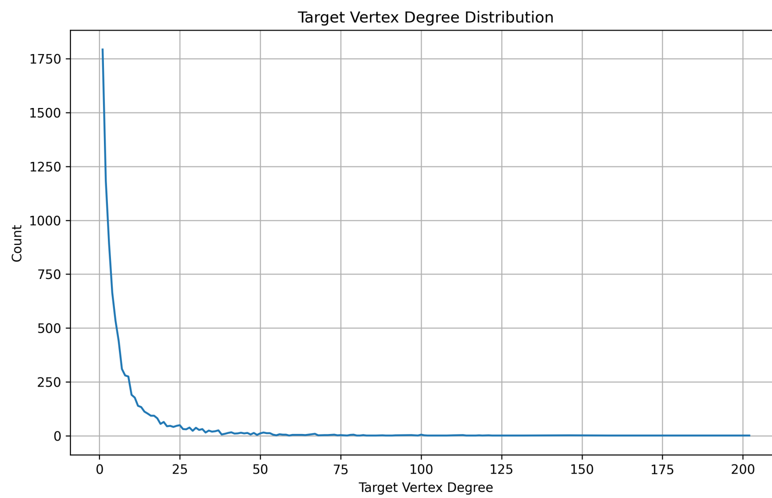


Figure 4.7: Target Vertex Distribution: Synthetic Graph

4.6 Workloads

To comprehensively evaluate our method, we generate a query template encompassing diverse types and lengths, including chains, cycles, and stars with 4 to 8 edges. This range of queries allow us to address various complexity levels effectively. For clarity, each query is identified by its type and edge count. For example, a star-shaped query with 6 edges is denoted as star-6. We visually illustrate chain-6, cycle-6, and star-6 queries in Figures 4.8, 4.9, and 4.10 respectively, to exemplify our query set. These figures, however, represent only a portion of our query workload, specifically queries with 6 edges, and are not comprehensive of the entire spectrum of queries utilized in our evaluation.

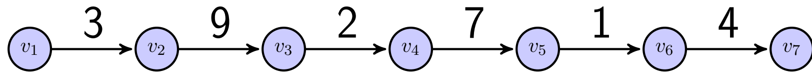


Figure 4.8: A chain query with six edges.

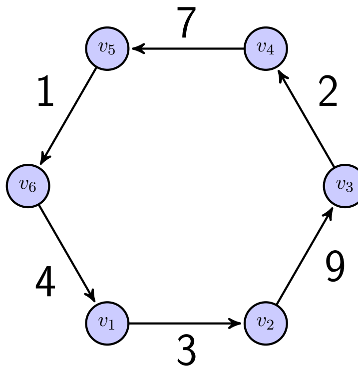


Figure 4.9: A cycle graph query with six edges.

In every experimental iteration, we randomly select edges from the label pool, numbered 0 to 9, to create a uniform permutation order for the entire query set. This technique guarantees consistency across different queries; specifically, star-4, cycle-4, and chain-4 queries each employ an identical sequence of 4 labels. Additionally, queries with fewer edges are subsets of those with more edges, implying that the labels used in a star-4 query are a subset of those in a star-8 query. We generate 100 permutations and apply each to the same query template. We ensure a random but uniform label assignment across our queries. Each template, encompassing three query types (chain, cycle, star) in five edge

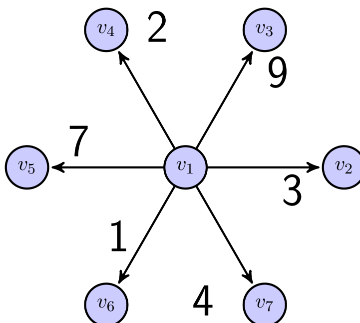


Figure 4.10: A star query with six edges.

length variations (4 to 8), results in 15 unique queries. As we generate 100 permutations, our comprehensive workload contains a total of 1,500 queries.

4.7 Accuracy Experiments

The accuracy experiments are designed to measure the precision of the cardinality estimations against the actual cardinalities of the queries. The objectives of these experiments are to demonstrate three critical aspects: (i) that GS-1 has the same accuracy as System R (validation of the correctness of *GraphSketch*, (ii) that the accuracy of the *GraphSketch* estimations improve with increasing bucket count, and (iii) that improvements in estimation accuracy lead to more changes in the query plan.

The accuracy experiments follow a systematic procedure, graphically depicted in Figure 4.11, enabling a thorough assessment of our method's precision. We employ a Ground Truth Generator for workload generation, creating star, cycle, and chain graph queries in SQL, executing these in PostgreSQL, and capturing the ground truth cardinalities. This process encompasses 100 query sets totaling 1,500 queries, with each set's results compiled into an individual output file. These files then serve as input for our Query Optimizer to estimate cardinalities, comparing both the System R cardinality estimations and *GraphSketch* estimations.

For evaluation, we adopt the widely used Q-error metric [45] to assess the accuracy of our estimations, reflecting the ratio between the true cardinality and predicted cardinality of a query, and is computed as:

$$\text{Q-error} = \max\left(\frac{\text{true}}{\text{predicted}}, \frac{\text{predicted}}{\text{true}}\right)$$

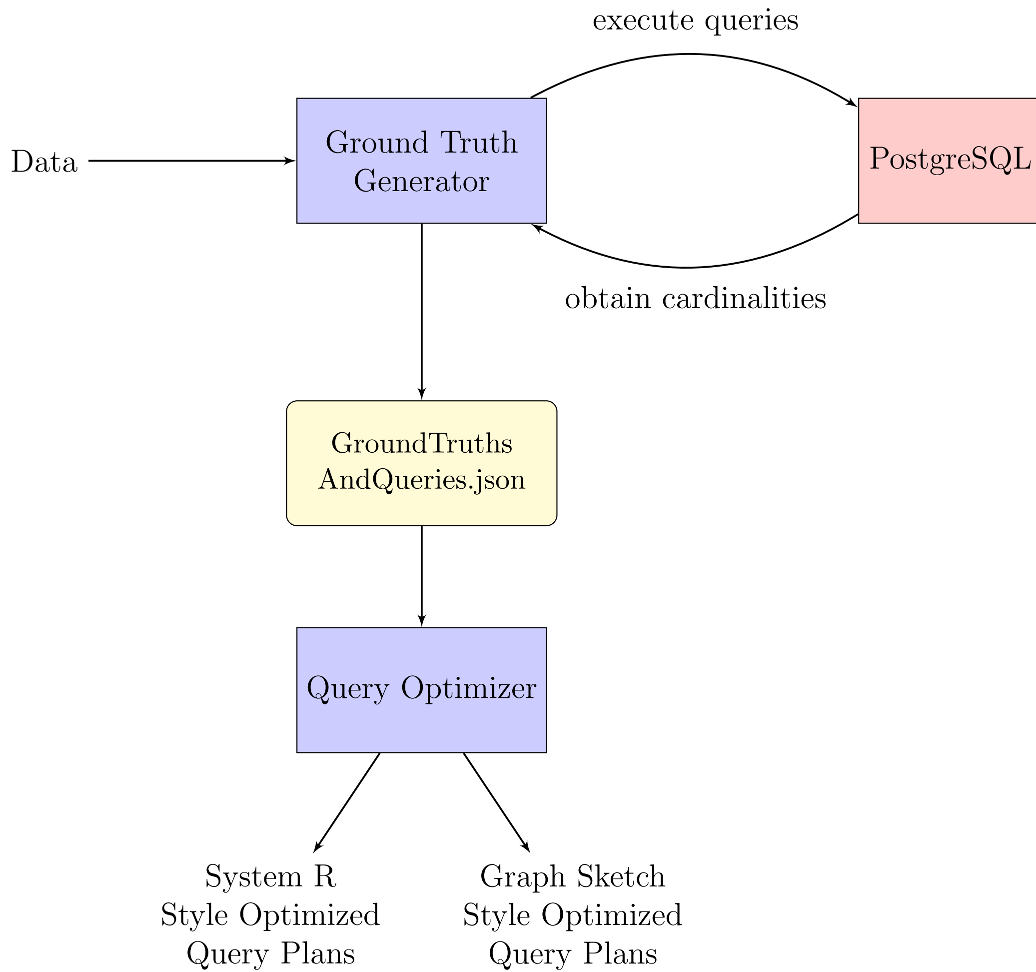


Figure 4.11: Evaluation Workflow for Accuracy Experiments.

A lower Q-error is better, with 1 being a perfect solution. In practice, to avoid division by zero, we replace true = $\max(\text{true}, 1)$ and predicted = $\max(\text{predicted}, 1)$. It should be noted that Q-error serves as an approximate measure for quantifying accuracy. Furthermore, lower Q-errors do not necessarily translate to shorter execution times for query plans [26]. The cardinality estimation function is applied to multiple sub-plan queries to determine the optimal query plan. The estimation errors for these various sub-plan queries affect the overall performance of the final query plan in different ways [26]. The Q-error metric, however, fails to differentiate between these impacts and treats all query estimation errors uniformly. Consequently, this can lead to situations where a more accurate estimation as indicated by Q-error might result in a less efficient query execution plan in practice.

4.7.1 Limitations

During implementation, we faced limitations in various aspects. On the PostgreSQL side, we encountered memory limitations and scalability problems when executing larger queries on the dataset used in our evaluation. A notable example is a single 9-edge query failing to complete after three days. This issue compounded as the dataset size increased, making the execution of even smaller queries difficult. Additionally, increased density in the input graph, defined by the ratio of edges to vertices, led to significant bottlenecks in processing intermediate results. On the query optimizer side, scalability issues also emerged in the query optimizer when attempting to use a bucket count larger than 900, restricting our ability to expand this parameter.

Transitioning to the physical implementation of Streaming Graph Algebra, the JOIN operation in relational algebra is translated as a PATTERN formed from two tuples. This method constrains the simultaneous construction of concurrent patterns necessary for modeling dual joins, influencing our strategy for cycle query construction. To form a cycle, we initially build a chain query and conclude it with a FILTER operator. However, we leave developing a custom FILTER for *GraphSketch* as future work due to the implementation overhead it requires. Currently, we employ the same FILTER operator used in System R.

Although structurally different, this approach does not affect the results, maintaining logical equivalence to direct cycle formation. This method ensures that the source vertex at the beginning of the chain aligns with the target vertex at its end. As an example, consider the chain query in Figure 4.8. To make this a cycle, a FILTER operator is added to ensure the source of edge 3, v_1, is the same as the target of edge 4, v_7. This technique preserves the cycle's integrity within the constraints of our query construction approach.

4.7.2 Accuracy Equivalence of GraphSketch and System R

This section provides experimental validation for the concepts discussed in Section 3.4. The experiments test the hypothesis that *GraphSketch* using one bucket should provide the same accuracy in cardinality estimation as System R.

The results in Table 4.1 validate the hypothesis.

Table 4.1: Comparative Analysis of Average Q-error: GS-1 vs. System R

Query	Avg Q-error GS-1	Avg Q-error System R
chain-4	24.91	24.91
cycle-4	11.36	11.3
star-4	27.21	27.21
chain-5	234.31	234.31
cycle-5	35.84	35.85
star-5	210.79	210.79
chain-6	1157.79	1157.79
cycle-6	139.6	139.1
star-6	2349.58	2349.58
chain-7	24195.46	24195.46
cycle-7	473.77	473.01
star-7	27052.62	27052.62
chain-8	200516.72	200516.72
cycle-8	1642.55	1642.51
star-8	315186.02	315186.0

Our experiments validate our hypothesis. Moreover, we observed that in each case, the cardinality estimates resulted in identical query plans.

4.7.3 Impact of Bucket Count on GraphSketch Accuracy

In this part of our study, we explore how varying bucket counts in *GraphSketch* affect its accuracy. The hypothesis that is tested is that the accuracy of *GraphSketch* should improve as the number of buckets increase as it is possible to do finer granularity estimations. We compare the performance across three distinct bucket counts: 1, 300, and 900. By examining these different settings, we aim to understand the optimal balance between accuracy and computational efficiency in *GraphSketch*'s implementation. The results of

this comparison are presented in the Table 4.2. Additionally, we present bar plots featuring confidence intervals for chain, cycle, and star queries in Figures 4.12, 4.13, and 4.14, respectively.

Table 4.2: Average Q-error Across Different Bucket Counts

Query	Avg Q-error GS-1	Avg Q-error GS-300	Avg Q-error GS-900
chain-4	24.91	17.59	13.65
cycle-4	11.36	9.67	12.84
star-4	27.21	12.54	3.41
chain-5	234.31	165.03	159.29
cycle-5	35.84	33.92	46.83
star-5	210.79	48.01	6.22
chain-6	1157.79	919.37	882.59
cycle-6	139.6	91.31	100.43
star-6	2349.58	208.19	12.61
chain-7	24195.46	20262.44	17212.87
cycle-7	473.77	306.61	470.98
star-7	27052.62	972.59	27.6
chain-8	200516.72	225776.7	153930.63
cycle-8	1642.55	1116.2	1063.73
star-8	315186.02	3821.64	58.84

In our evaluation, significant insights were drawn regarding Q-error reductions:

1. **Overall Q-error reduction:** The reduction in Q-error from GS-1 to GS-300 is significant at 55.7%, demonstrating the impact of increasing bucket count on estimation accuracy. This improvement is further amplified when moving from GS-1 to GS-900, where the Q-error reduction reaches 69.6%, showcasing the method's robustness in finer bucket configurations.

2. **Star queries:** The performance in star queries is particularly remarkable. The Q-error plummets by 98.3% when transitioning from 1 to 300 buckets, indicating a substantial enhancement in estimation precision. An even more dramatic decrease of 99.97% is observed when shifting from 1 to 900 buckets, underscoring the exceptional accuracy attainable in high-bucket scenarios for star queries. Furthermore, the tight confidence intervals depicted in the plots in Figure 4.14 reinforce the effectiveness of this method.

3. **Chain queries:** In chain queries, the reduction of Q-error by 23.9% as bucket count increases from 300 to 900 suggests a notable improvement in accuracy, albeit less pronounced than in star queries. This reduction indicates that higher bucket counts can refine the cardinality estimation in chain queries, although the magnitude of improvement varies with query type. We acknowledge that the confidence intervals shown in the plots in Figure 4.12 make the Q-error differences statistically insignificant. Therefore, we conclude that it would be a better strategy to evaluate this section with the number of query plan changes in Section 4.7.4.

4. **Cycle queries:** The Q-error reductions in cycle queries are inconsistent, likely influenced by our methodology, which excluded specific filters and did not eliminate queries yielding zero results as true cardinalities. This limitation in our experimental setup affects the calculated Q-error reduction across different query length, suggesting that cycle queries may require a more tailored approach for accurate cardinality estimation.

5. **Query length analysis:** The analysis of Q-error reductions across different query lengths with varying bucket counts reveals the following:

   (a) **Size 4 queries:** The Q-error reduction is 37.3% from 1 to 300 buckets, and 52.9% from 1 to 900 buckets. From 300 to 900 buckets, the reduction is 25.0%.

   (b) **Size 5 queries:** Q-error decreases by 48.5% from 1 to 300 buckets, 56.2% from 1 to 900 buckets, and 15.0% from 300 to 900 buckets.

   (c) **Size 6 queries:** There is a 71.9% reduction in Q-error from 1 to 300 buckets, 72.7% from 1 to 900 buckets, and a modest 2.9% from 300 to 900 buckets.

   (d) **Size 7 queries:** Q-error initially increases from 1 to 300 buckets, then decreases by 65.8% from 1 to 900 buckets, and 92.8% from 300 to 900 buckets, indicating a significant improvement.

   (e) **Size 8 queries:** The reduction in Q-error is 55.4% from 1 to 300 buckets, 70.0% from 1 to 900 buckets, and 32.8% from 300 to 900 buckets.

   Our hypothesis posits that the accuracy of cardinality estimators, notably System R and GraphSketch, tends to decrease as the length of queries increases. This decrease in accuracy is attributed to the underlying assumptions of uniformity, independence, and inclusion that these estimators rely on. In larger queries, these assumptions become increasingly unreliable, leading to error accumulation at each step. The uniformity assumption becomes more erroneous in diverse data distributions, independence assumptions might not hold in interconnected datasets, and the inclusion

principle may be challenged by complex query structures. Consequently, errors inherent in the initial stages of a query are not just carried forward but often magnified in subsequent steps, leading to a compounding effect.

This hypothesis is supported by empirical observations in the performance of GraphSketch. The data indicates a pronounced reduction in Q-error for longer queries, particularly those spanning 7 and 8 edges, when adjusting the bucket count in GraphSketch. This suggests that while increased query length typically exacerbates error propagation due to the initial assumptions, GraphSketch's adaptability through bucket count adjustment counters this trend effectively.

These results not only affirm the hypothesis regarding the impact of query length on estimator accuracy but also underscore the capability of GraphSketch to adapt and mitigate the challenges posed by longer queries. This adaptability is a crucial factor in maintaining accuracy in cardinality estimation across varying query lengths and complexities.

These findings underscore the importance of bucket count in improving estimation accuracy, with star queries showing remarkable improvements and cycle queries indicating areas for further optimization.

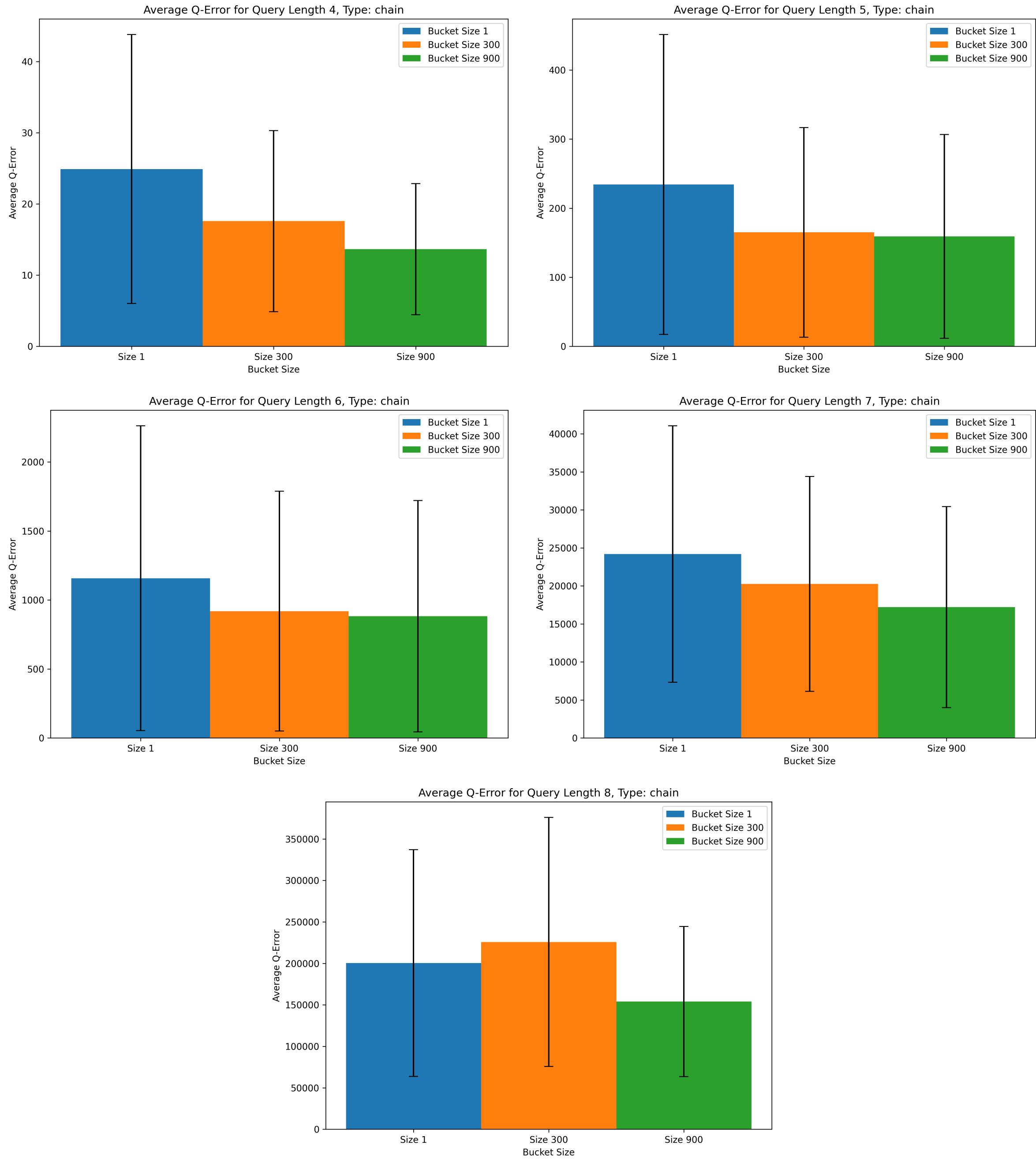


Figure 4.12: Accuracy Plots of Chain Queries

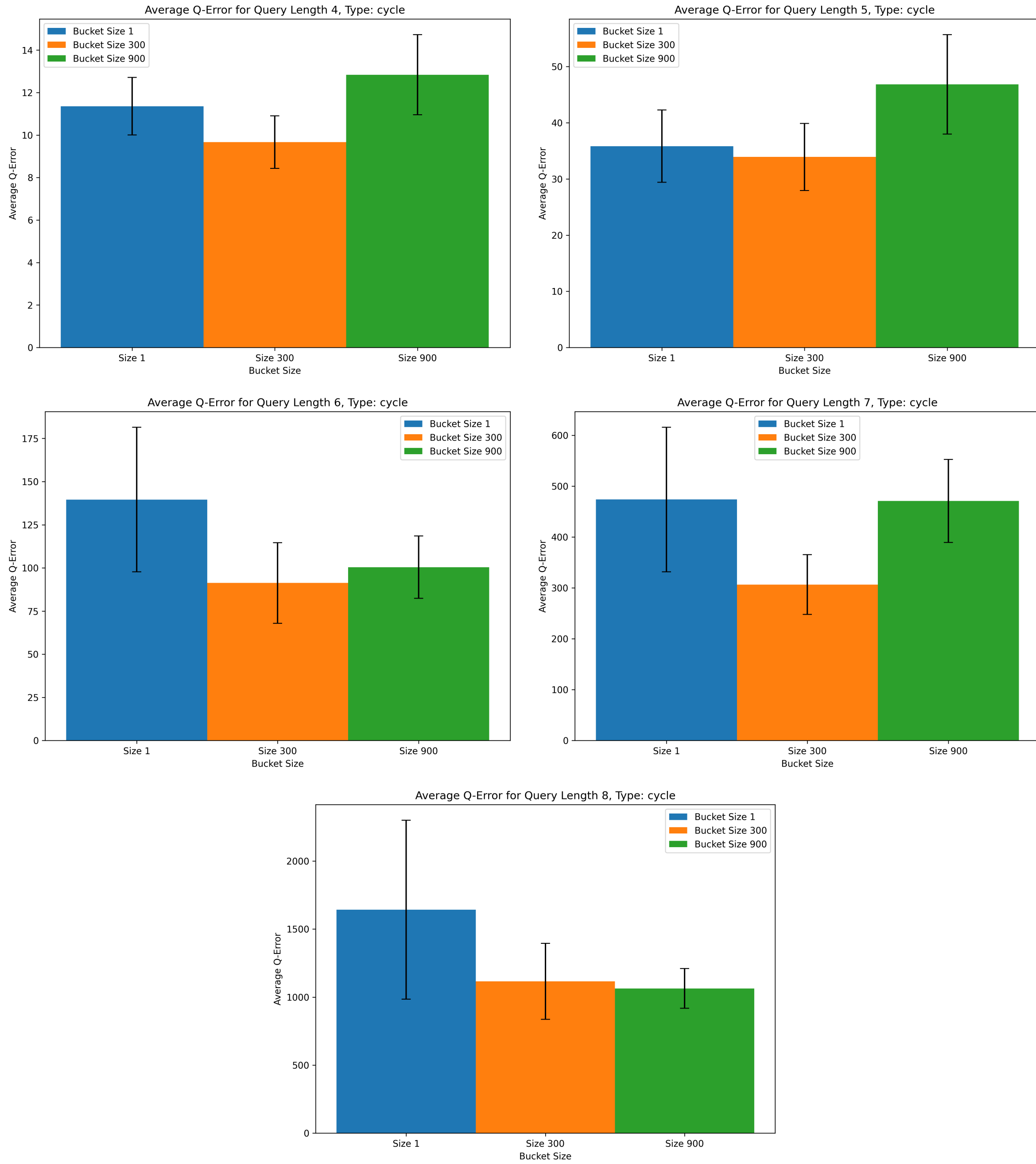


Figure 4.13: Accuracy Plots of Cycle Queries

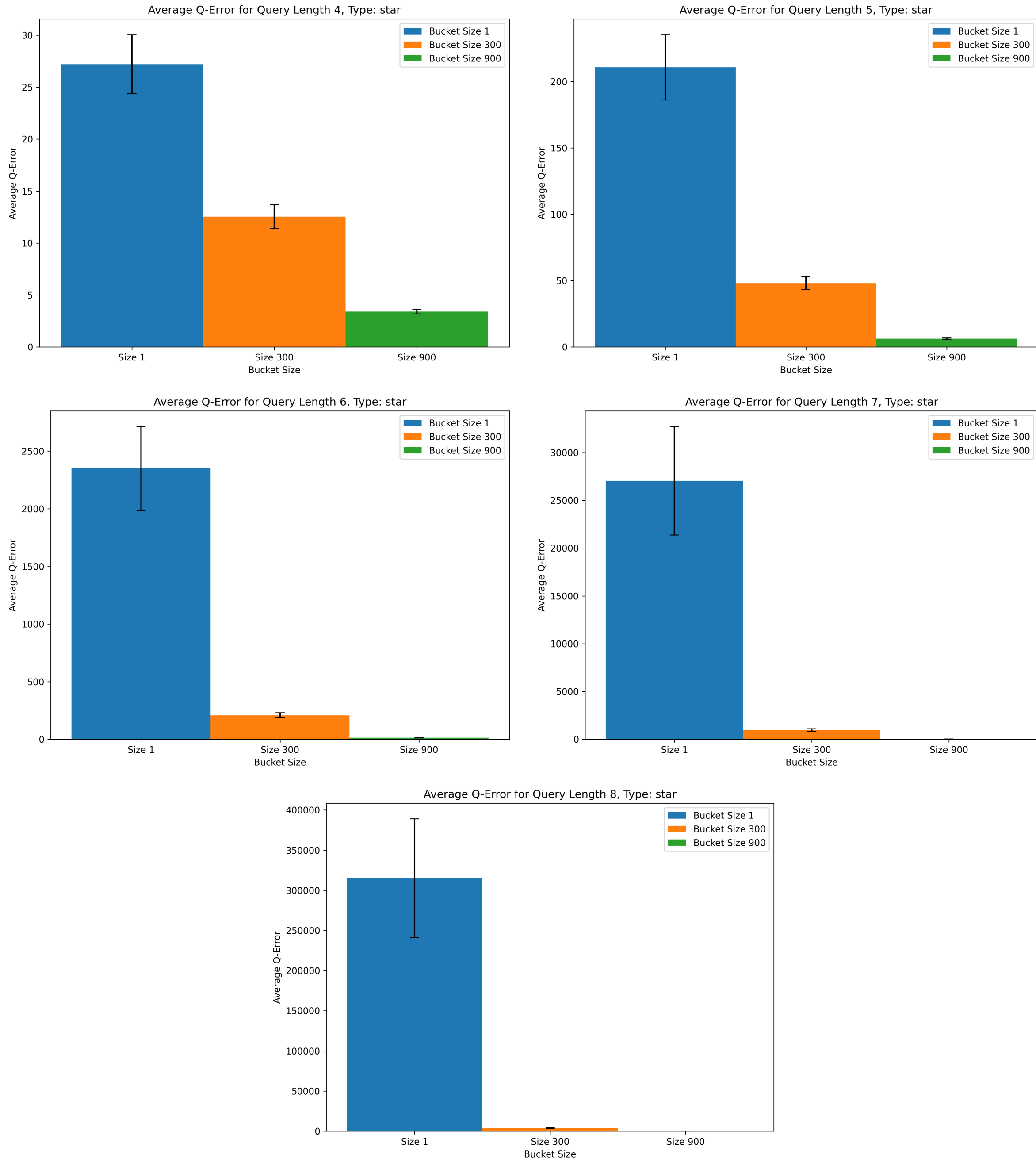


Figure 4.14: Accuracy Plots of Star Queries

4.7.4 Impact of GraphSketch's Bucket Count on Query Plans

This section further investigates the relationship between the bucket count in *GraphSketch* and its subsequent impact on query plan formulation. We investigate whether varying bucket counts, specifically the reduction of error rates, can lead to significant changes in the optimization of query plans. This analysis is crucial in understanding the impact of cardinality estimations, influenced by bucket count adjustments, play a pivotal role in the efficacy and optimization strategies of query planning. The insights from this study are aimed at highlighting the critical balance between accurate cardinality estimation and its practical impact on optimizing database query plans.

We investigated the number of query plan changes for each type of query as the bucket counts are increased. The results are given in Table 4.3.

Table 4.3: Changes in Query Plans Across Different Bucket Counts

Query	Changes 1 → 300	Changes 300 → 900	Changes 1 → 900
chain-4	14	6	17
cycle-4	14	6	17
star-4	11	10	15
chain-5	25	11	26
cycle-5	25	11	26
star-5	21	23	34
chain-6	26	22	36
cycle-6	26	22	36
star-6	30	33	47
chain-7	33	24	43
cycle-7	33	24	43
star-7	52	46	66
chain-8	46	33	51
cycle-8	46	33	51
star-8	65	63	80

This analysis reveals the following:

1. With 100 permutations in our workload, the average query plan changes are as follows:

    - From 1 to 300 buckets: 31.3 changes.

- From 300 to 900 buckets: 24.6 changes.
- From 1 to 900 buckets directly: 39.2 changes.

2. The nature of star queries, characterized by a high-degree central node linked to numerous lower-degree nodes, presents a unique challenge for accurate cardinality estimation. Such queries are expected to be less efficiently handled by the System R technique due to its potential limitations in addressing the non-uniform distribution of vertices that is typical in star structures. This complexity primarily arises from the need to precisely gauge the number of unique paths emanating from a single, highly connected node. System R may struggle with adequately considering the probability distribution and connectivity patterns stemming from such a central node, potentially leading to initial errors in estimation, particularly in overestimation or underestimation.

   On the other hand, *GraphSketch* (GS), with its strategic approach of bucketing, is hypothesized to be more adept at managing the non-uniformity inherent in star queries. By distributing this non-uniformity across different buckets, GS is likely to enhance accuracy in cardinality estimation. The effectiveness of *GraphSketch* in this context becomes more pronounced, especially in scenarios dominated by a single, highly connected central node. This is reflected in empirical observations where star queries exhibit a higher average of 48.4 changes, surpassing the average of 34.6 changes observed for chain and cycle queries. Such a contrast underscores the superior efficiency of *GraphSketch* in handling star queries, where the central node's high connectivity significantly influences the query's complexity and the accuracy of cardinality estimations.

3. The impact of query length on the effectiveness of estimation is evident:

   - For length 8 queries: 60 plan changes on average.
   - For length 7 queries: 50.6 plan changes.
   - For length 6 queries: 39.6 plan changes.
   - For length 5 queries: 28.6 plan changes.
   - For length 4 queries: 16.3 plan changes.

   This trend demonstrates the improved estimation impact of *GraphSketch* longer query lengths.

4. The identical changes in query plans for chain and cycle queries can be attributed to their similar construction patterns, as discussed in the section 4.7.1. The current

implementation strategy in SGA results in chain and cycle queries having similar patterns, thus leading to identical query plan changes.

4.8 Query Latency Experiments

In this section, we utilize the queries from the accuracy experiments (Section 4.7) to measure the latencies of query plans generated by GS-1 and GS-900 cardinality estimations. Notably, cycle query plans were excluded due to their additional implementation requirements. For each query in our set, we calculate the ratio of the latency of the GS-900 plan to the GS-1 plan. This ratio serves as the primary metric for comparison. In cases where GS-900 does not generate a different plan from GS-1, the query is excluded from the ratio calculation but noted for reference.

The scatter plot in Figure 4.15 visualizes the comparative analysis, where the x-axis represents the latencies of GS-1 and the y-axis the latencies of GS-900 query plans in microseconds. A 45-degree linear line on the plot indicates a point of equal latencies between the two estimators. Points on this line suggest that both GS-1 and GS-900 generated query plans with the same execution times for the respective queries. Since a point on this plot represents the latency ratio of GS-900 to GS-1, points above the line indicate instances where GS-900's query plans were less efficient, resulting in higher latencies compared to GS-1. Conversely, points below the line signify queries where GS-900 outperformed GS-1 in terms of execution speed. This scatter plot methodologically represents the efficiency of each estimator in generating optimal query plans. Additionally, to delve deeper into specific query types, in Figure 4.16 we exclusively concentrate on the analysis of star queries, and in Figure 4.17 we only focus on the analysis of chain queries.

Among the changed query plans, there was a notable improvement in performance: 58% of the altered chain query plans and 64% of the star query plans exhibited enhanced speed. Overall, considering the entire workload, 61% of GS-900's query plans demonstrated a marked increase in efficiency. For queries where the GS-900 optimization results in slower performance, the discrepancy in speed compared to GS-1 is relatively marginal. However, in cases where the GS-1's optimized query plans are slower, there is often a substantially wider disparity in performance, with the GS-1 yielding markedly slower results in comparison.

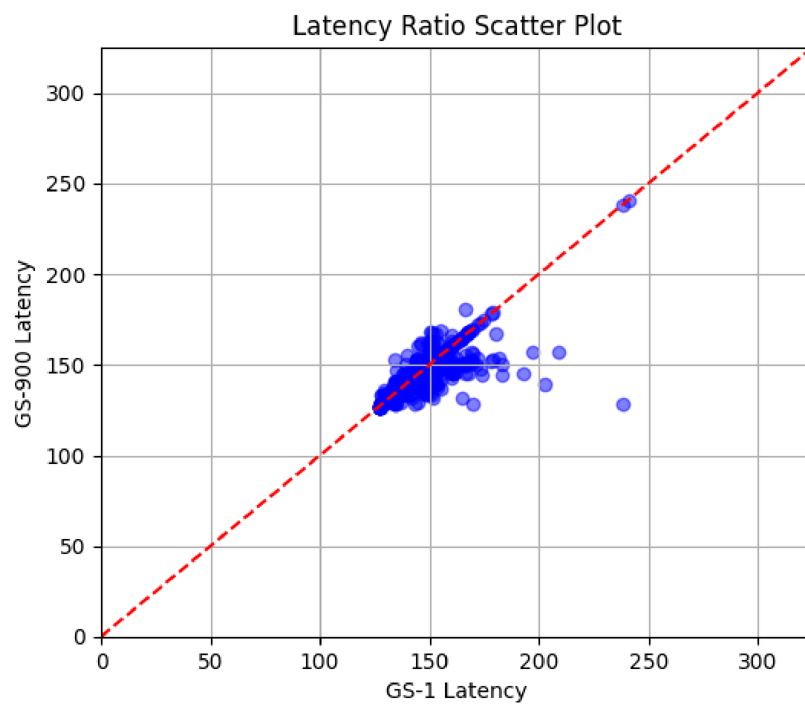


Figure 4.15: Comparative Analysis of Chain and Star Query Latency Between GS-900 and GS-1 Plans

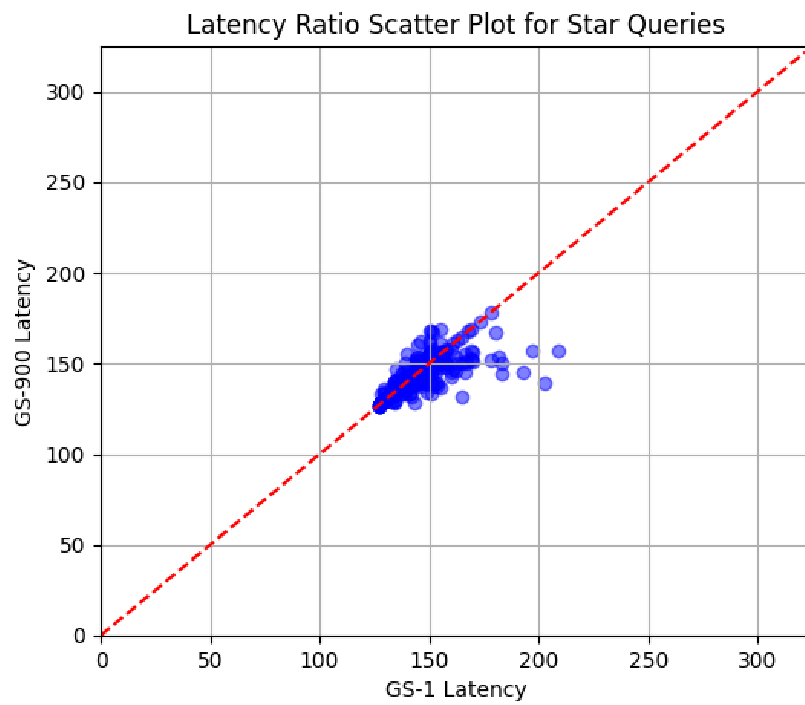


Figure 4.16: Comparative Analysis of Star Query Latency Between GS-900 and GS-1 Plans

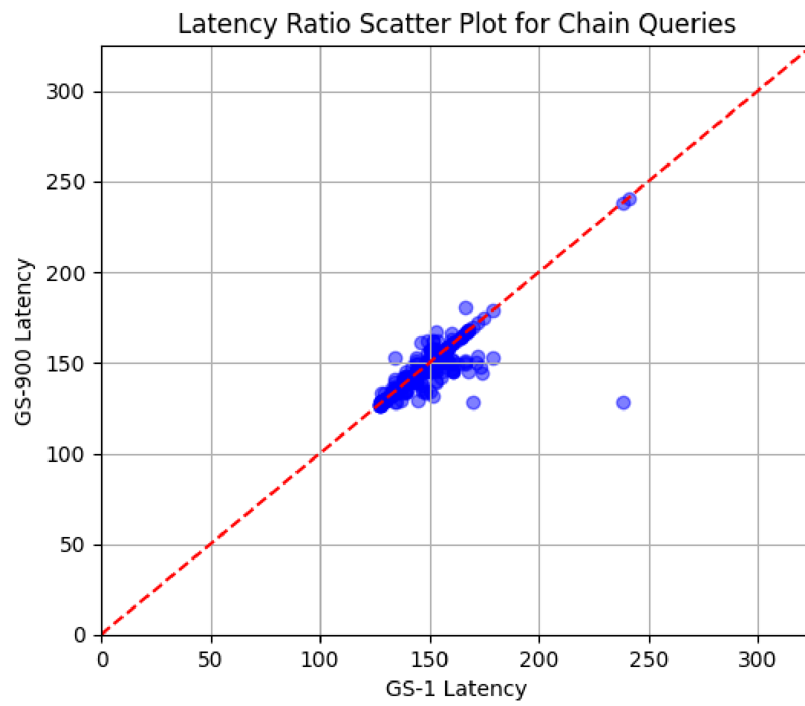


Figure 4.17: Comparative Analysis of Chain Query Latency Between GS-900 and GS-1 Plans

4.9 Efficiency Experiments

In this section, we present the results of our efficiency experiments, with a primary focus on demonstrating the significant boost in efficiency achieved through incremental maintenance of *GraphSketch*es as windows shift. This approach is pivotal in showcasing how incrementally updating *GraphSketch*es as windows slide can substantially enhance computational efficiency which is critical for keeping up with the velocity of the stream arrivals.

For these experiments, we set the bucket count to 900; and configure the streaming variables as follows: the window size is varied from 2 weeks to 3 weeks and 4 weeks, with a sliding interval of 1 day. Given the extensive number of windows these settings yield, presenting the entire result set is impractical. Instead, we focus on providing key metrics: the average sketch construction time, the minimum and maximum construction times, and the comparative efficiency of maintenance. These metrics are further discussed with the aid of frequency histograms in Figure 4.18, offering a comprehensive view of our findings. We present results in Table 4.4.

Table 4.4: Comparison of Graph Sketch Construction Times

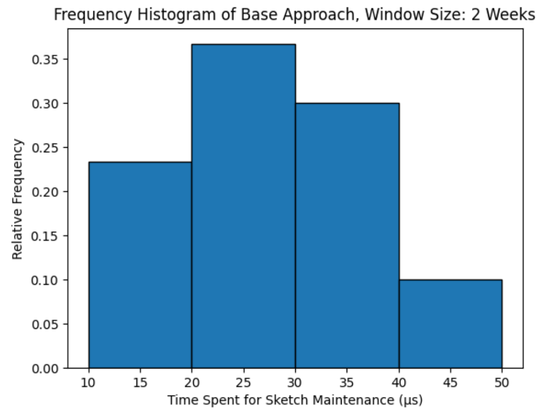
Window Size	Baseline Method			Incremental Maintenance		
	Min (μs)	Max (μs)	Avg (μs)	Min (μs)	Max (μs)	Avg (μs)
2 Weeks	3.88	49.2	19.56	2.02	19.43	4.55
3 Weeks	4.52	83.6	27.28	3.07	21.41	7.69
4 Weeks	8.40	89.7	30.14	6.57	33.53	14.24

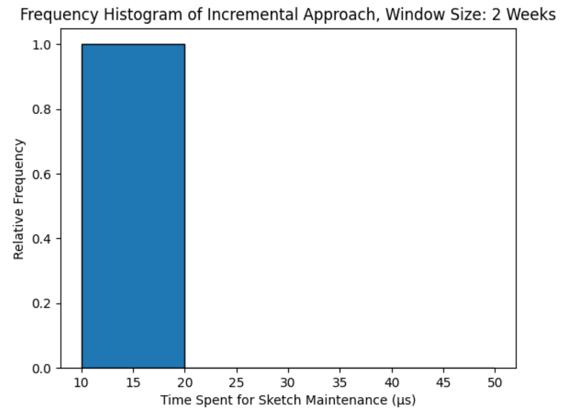
The results from both the baseline method (reconstructing *GraphSketch* from scratch at each window slide) and the incremental maintenance approach reveal significant insights into the efficiency of *GraphSketch* construction. Under the baseline method, a consistent increase in construction time was observed as the window size expanded from 2 weeks to 4 weeks. Specifically, the average construction time rose from 19.56 microseconds for a 2-week window to 25.71microseconds for a 4-week window. This trend underscores the increased computational demands associated with larger window sizes.

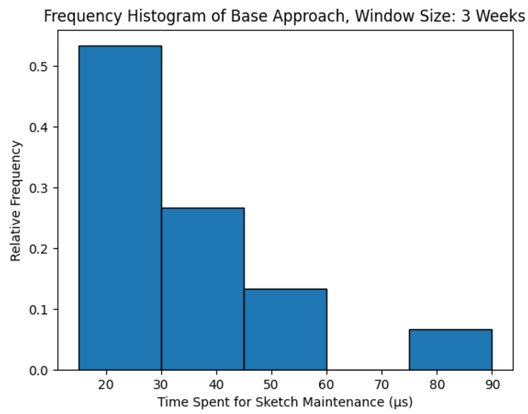
However, the implementation of incremental maintenance brought about a marked improvement in all metrics. Notably, for a 2-week window, the average construction time was nearly halved, decreasing to 4.55 microseconds. This improvement was consistent across larger window sizes, with the average time for a 4-week window being reduced to 11.98 microseconds. This demonstrates the effectiveness of incremental maintenance in reducing the time overhead, particularly in scenarios with larger window sizes.

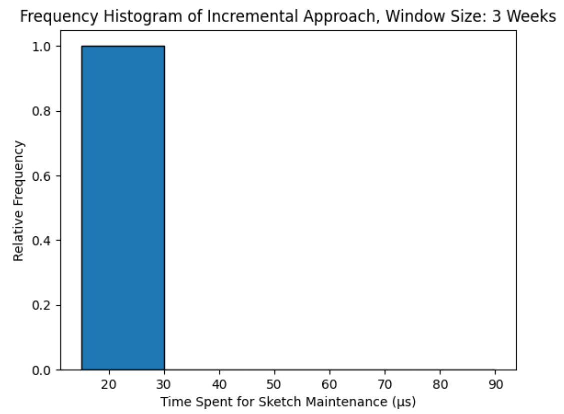
The reduction in maximum construction time is also noteworthy. For instance, in the 3-week window scenario, the maximum time decreased from 83.6 microseconds under the baseline method to 21.41 microseconds with incremental maintenance. This significant reduction highlights the method's robustness, especially under more demanding or complex scenarios.

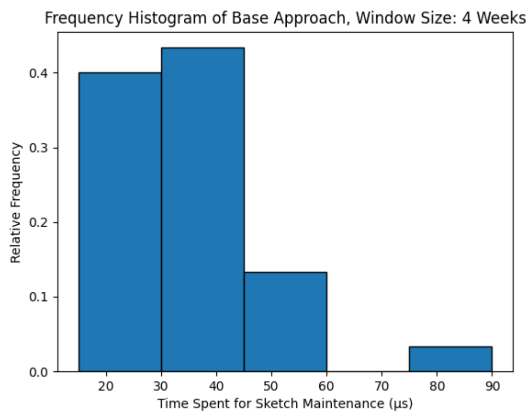
These results clearly demonstrate the benefits of incremental maintenance in handling *GraphSketch*, particularly in terms of scalability and efficiency. The approach not only reduces average construction times across all window sizes but also minimizes the maximum time required under the most strenuous conditions. Therefore, it can be concluded that incremental maintenance is a highly effective strategy for managing *GraphSketch* in dynamic environments, especially when dealing with large datasets or extended time windows.











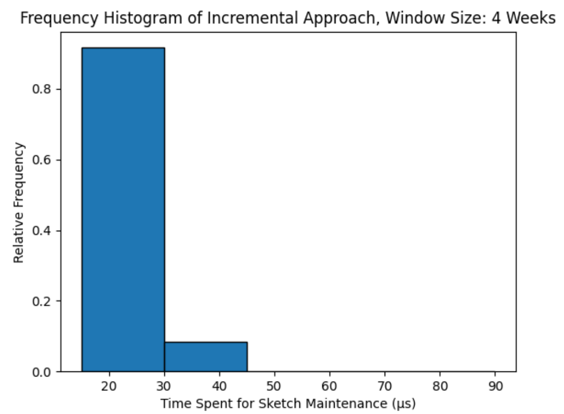


Figure 4.18: Frequency Histograms for Latency Experiments

Chapter 5

Conclusion

In the evolving landscape of data management, efficient query processing is crucial for the effective handling and analysis of the continuous stream of unbounded incoming data. Traditional query optimization methods, when applied to streaming graph environments, do not have the required accuracy and efficiency for processing streaming graph queries. This is primarily due to the lack of specialized techniques for cardinality estimation in streaming graphs, which is critical for optimizing query performance and system throughput. Existing approaches, while functional in static or less dynamic environments, struggle to adapt to the unique demands of streaming graph data, leading to suboptimal performance and increased computational overhead.

This thesis presented *GraphSketch*, a novel cardinality estimation technique designed for streaming graph database management systems. Addressing a critical gap in the realm of streaming data management, *GraphSketch* offers a specialized, sketch-based framework that effectively summarizes streaming graphs, enabling effective and efficient cardinality estimations crucial for query optimization in real-time environments.

Key Contributions of GraphSketch

GraphSketch represents a significant advancement in streaming GDBMSs, with several key contributions:

- **Novel Sketch-Based Framework:** The cornerstone of *GraphSketch* is its ability to provide accurate cardinality estimations through a concise summary of stream-

ing graphs. This framework is pivotal for the efficient processing and analysis of streaming graph data in the context of cardinality estimation.

- **Incremental Update Capability:** *GraphSketch*'s design allows for incremental updates, a critical feature for maintaining efficiency in the dynamic and continually evolving landscape of streaming data.

- **Empirical Evaluations and Benchmarking:** Through extensive empirical studies, *GraphSketch* has been benchmarked against System R's traditional cardinality method that is adapted specifically for streaming graphs. These evaluations have demonstrated the superior performance of *GraphSketch* in terms of accuracy, efficiency, and reduced query latency.

5.1 Future Research Directions

While the cardinality estimation framework discussed in this thesis has demonstrated satisfactory performance for the presented workload, there remains considerable room for enhancement. One immediate consideration is the extension of the method's evaluation to encompass larger datasets. During our analysis, we encountered scalability limitations on the PostgreSQL side when attempting to compute ground truth values. To address this challenge, we intend to leverage a fully analytical database, which will facilitate the computation of cardinalities for intricate queries on more extensive datasets. This strategic approach will enable us to work effectively with larger data.

Our roadmap also includes an expansion of our cardinality estimation framework beyond its current application solely to the PATTERN operator in SG. We aim to develop specialized algorithms tailored for a wider array of operators, including but not limited to the FILTER operator. This broader scope ensures that our framework remains versatile and adaptable to a range of query types, ultimately optimizing query execution efficiency and accuracy across various scenarios.

In addition, we also consider the integration of a stream detection algorithm into our cardinality estimation framework. This addition allows us to analyze the data characteristics of each graph stream, enabling us to take specific actions, such as adjusting the number of buckets within the corresponding *GraphSketch*. This approach introduces adaptiveness to our framework, as it dynamically alters the bucket count per graph stream to strike a more precise balance between accuracy and efficiency, tailored to the users' requirements. Effective implementation of this approach requires modifications to our *GraphSketch*, enabling it to conduct cardinality estimations for various streams, each with varying bucket

counts. This ensures a comprehensive adaptation to the specific requirements of different data streams and their associated characteristics.

Another significant future direction involves the utilization of *GraphSketch* with machine learning techniques. This combination can lead to the cardinality estimation framework for streaming graph databases that do not have the burden of developing specialized algorithms for each operator. In this relatively unexplored field, the absence of prior research indicates the potential for transformative advancements. The combination of *GraphSketch*es' efficiency and machine learning's predictive power could facilitate real-time adaptability and scalability for larger datasets. Additionally, this proposed approach offers opportunities for leveraging incremental sketch updates effectively in a streaming context, promising advancements in data management within streaming graph databases.

References

[1] Soraya Abad-Mota. Approximate query processing with summary tables in statistical databases. In *Advances in Database Technology, Proc. 3rd Int. Conf. on Extending Database Technology*, page 499–515, Berlin, Heidelberg, 1992. Springer-Verlag.

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Ȅetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 277–289, 2005.

[3] Christopher R Aberger, Stephen 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] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-core: A core for future graph query languages. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1421–1432, 2018.

[5] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[6] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. gmark: schema-driven generation of graphs and queries. *IEEE Trans. Knowl. and Data Eng.*, 29(4):856–869, 2016.

[7] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. on Comput.*, 45(2):548–574, 2016.

[8] Till Blume, David Richerby, and Ansgar Scherp. Incremental and parallel computation of structural graph summaries for evolving graphs. In *Proc. 29th ACM Int. Conf. on Information and Knowledge Management*, pages 75–84, 2020.

[9] 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.

[10] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *VLDB J.*, 10(2–3):199–223, 2001.

[11] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proc. VLDB Endowment*, 15(8):1533–1545, 2022.

[12] Chen, Jeremy Yujui. Join cardinality estimation graphs: Analyzing pessimistic and optimistic estimators through a common lens. Master's thesis, University of Waterloo, 2020.

[13] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. on Comput.*, 32(5):1338, 2003.

[14] Edith Cohen and Haim Kaplan. Tighter estimation using bottom k sketches. *Proc. VLDB Endowment*, 1:213–224, 2008.

[15] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[16] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *Proc. 6th Latin American Theor. Informatics Symp.*, pages 29–38, 2004.

[17] Alfredo Cuzzocrea. *Histogram-Based Compression of Databases and Data Cubes*, pages 165–178. IGI Global, 2009.

[18] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. Tigergraph: A native mpp graph database. arXiv 1901.08248, 2019.

[19] George H L Fletcher, J Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amelie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proc. 19th Int. Conf. on Extending Database Technology*, pages 636–639, 2016.

[20] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM SIGMOD Rec.*, 32(2):5–14, 2003.

[21] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th USENIX Symp. on Operating System Design and Implementation*, pages 17–30, 2012.

[22] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: graph processing in a distributed dataflow framework. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 599–613, 2014.

[23] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[24] Andrey Gubichev, Srikanta J Bedathur, and Stephan Seufert. Sparqling kleene: fast property paths in rdf-3x. In *Proc. 1st Int. Workshop on Graph Data Management Experiences and Systems*, 2013.

[25] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21th Int. Conf. on Very Large Data Bases*, page 358–369, 1995.

[26] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Cardinality estimation in dbms: A comprehensive benchmark evaluation. *Proc. VLDB Endowment*, 15(4):752–765, 2021.

[27] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proc. 16th Int. Conf. on Extending Database Technology*, 2013.

[28] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[29] Yannis Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. 25th Int. Conf. on Very Large Data Bases*, pages 174–185, 1999.

[30] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. On the optimization of recursive relational queries: Application to graph queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 681–697, 2020.

[31] Chris Jermaine, Sanjay Arumugam, Alin Pol, and Alin Dobra. Scalable approximate query processing with the DBO engine. *ACM Trans. Database Syst.*, 33(4):1–54, 2008.

[32] Jaewoo Kang, Jeffrey Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *Proc. 19th IEEE Int. Conf. on Data Engineering*, pages 341–352, 2003.

[33] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. *Proc. 19th IEEE Int. Conf. on Data Engineering*, pages 341–352, 2003.

[34] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. *Proc. 24th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 1131–1142, 2013.

[35] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[36] Jakub Lacki. Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *Proc. 22nd Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 1438–1445, 2011.

[37] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[38] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB J.*, 27:643 – 668, 2017.

[39] Youhuan Li, Lei Zou, M. Tamer Özsu, and Dongyan Zhao. Time constrained continuous subgraph search over streaming graphs. In *Proc. 35th IEEE Int. Conf. on Data Engineering*, pages 1082–1093, 2019.

[40] Guy Lohman. Is query optimization a “solved” problem? ACM SIGMOD Blog, July 2014.

[41] 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.

[42] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 346–357. Elsevier, 2002.

[43] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *Advances in Database Technology, Proc. 11th Int. Conf. on Extending Database Technology*, pages 618–629, 2008.

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

[45] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endowment*, 2(1):982–993, 2022.

[46] 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. Association for Computing Machinery, 2013.

[47] Van Quoc Nguyen, Quoc-Tuan Huynh, and Kyumin Kim. Estimating searching cost of regular path queries on large graphs by exploiting unit-subqueries. *J. Heuristics*, pages 1–21, 2018.

[48] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[49] Anil Pacaci. *Models and Algorithms for Persistent Queries over Streaming Graphs*. PhD thesis, University of Waterloo, 2022.

[50] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. Evaluating complex queries on streaming graphs. In *Proc. 38th IEEE Int. Conf. on Data Engineering*, pages 272–285, 2022. **Best paper award**.

[51] 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.

[52] Anil Pacaci, Alice Zhou, Jimmy Lin, and M Tamer Özsu. Do we need specialized graph databases?: Benchmarking real-time social networking applications. *Proc. 5th Int. Workshop on Graph Data Management Experiences & Systems*, page 12, 2017.

[53] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proc. 10th ACM Int. Conf. Web Search and Data Mining*, pages 601–610, 2017.

[54] Jason B. Peltzer, Ankur M. Teredesai, and Garrett Reinard. AQUAGP: Approximate query answers using genetic programming. In *Genetic Programming – Proc. European Conf. on Genetic Programming*, pages 49–60, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[55] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endowment*, 11(12):1876–1888, 2018.

[56] Pavan Ravindra, Hyunjung Kim, and Kemafor Anyanwu. Optimization of complex sparql analytical queries. In *Proc. 19th Int. Conf. on Extending Database Technology*, pages 257–268, 2016.

[57] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM J. on Comput.*, 45(3):712–733, 2016.

[58] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endowment*, 11(4):420–431, 2018.

[59] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 23–34, 1979.

[60] Stephan Seufert, Arijit Anand, Srikanta Bedathur, and Gerhard Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proc. 29th IEEE Int. Conf. on Data Engineering*, pages 1009–1020, 2013.

[61] Jennifer Spiegel and Neoklis Polyzotis. Graph-based synopses for relational selectivity estimation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 205–216, 2006.

[62] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. Reachability querying: can it be even faster? *IEEE Trans. Knowl. and Data Eng.*, 29(3):683–697, 2016.

[63] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1481–1496, 2016.

[64] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 1481–1496, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Francesco Tria, Ettore Lefons, and Filippo Tangorra. System architecture for approximate query processing. *Comp. & Inf. Sci.*, 9(2):156–171, 2016.

[66] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. Efficiently answering regular simple path queries on large labeled networks. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1463–1480, 2019.

[67] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Query planning for evaluating sparql property paths. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1875–1889, 2016.

[68] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: scalable reachability index for large graphs. *Proc. VLDB Endowment*, 3(1):276–284, 2010.

[69] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: On query estimation in graph streams. *Proc. VLDB Endowment*, 5(3):193–204, 2011.