Systems and Techniques for Efficient Real-World Graph Analytics

by

Joana M. F. da Trindade

B.S., Universidade Federal do Rio Grande do Sul (2009)
M.S., University of Illinois at Urbana-Champaign (2011)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

Authored by:	Joana M. F. da Trindade Department of Electrical Engineering and Computer Science January 26, 2024
Certified by:	Samuel R. Madden MIT College of Computing Distinguished Professor of Computing Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Systems and Techniques for Efficient Real-World Graph Analytics

by

Joana M. F. da Trindade

Submitted to the Department of Electrical Engineering and Computer Science on January 26, 2024, in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Abstract

Graphs are a natural way to model real-world entities and relationships between them, ranging from social networks and biological datasets to cloud computing infrastructure data lineage graphs. Queries over these large graphs often involve expensive subgraph traversals and complex analytical computations. Furthermore, real-world graphs often encode relationships that evolve through time, which are modeled as a temporal graph, i.e., one in which edges are associated to time attributes (such as start time and duration). These real-world graphs are often substantially more structured than a generic vertex-and-edge model would suggest, but this insight has remained mostly unexplored by existing graph engines for graph query optimization purposes. In addition, most temporal graph processing systems remain inefficient as they rely on distributed processing even for graphs that fit well within a commodity server's available storage. To this end, we present two distinct systems tailored for optimizing large-scale real-world graph processing. The first system, Kaskade, targets the challenges of efficient query evaluation by leveraging structural properties of graphs and queries to infer materialized graph views, speeding up query evaluation by rewriting queries in terms of views it deems beneficial based on input graph and query characteristics. The second system, Kairos, introduces selective indexing, a technique that chooses a subset of target vertices to index based on characteristics of the underlying temporal graphs and input queries. This system further employs a highly-specialized parallel data structure aimed at in-memory storage and fast retrieval of temporal edges. Finally, Kairos is built upon Ligra, the *de facto* benchmark system in shared-memory parallel graph processing, offering similar advantages and a familiar API to application programmers. Both systems offer speedups of up to 50-60x when compared with alternative baselines, and introduce novel classes of query optimization techniques aimed at efficient real-world graph analytics.

Thesis Supervisor: Samuel R. Madden
Title: MIT College of Computing Distinguished Professor of Computing

Acknowledgments

Completing this journey would not have been possible without the support, guidance, and encouragement of many remarkable folks whose contributions have left a significant mark on both my work and personal growth.

First and foremost, I extend my deepest gratitude to my advisor, Prof. Samuel Madden, for his unwavering support and for providing me the freedom to explore the fascinating realm of temporal graphs, as well as other incredibly fun areas of research early on in my PhD. His guidance has provided me constant inspiration and invaluable learning throughout this journey. His research taste is impeccable, and I am particularly draw to his approach, which is often driven by practical problems. This pragmatic perspective, which I hope to have embraced in my own research, resonates deeply with my inclination towards real-world (and often industry-oriented) applications of research.

I am immensely thankful to Prof. Julian Shun, member of my thesis committee and a close collaborator for most of my years at MIT. His endless expertise in virtually every aspect of parallel processing has been invaluable. Furthermore, his mentorship has been instrumental from the beginning, from insightful discussions on various cost models to guidance on maximizing the performance of initially serial implementations. Suffice to say, his comprehensive teachings in parallel graph processing to not just me, but to the broader MIT EECS community, have been fundamental in achieving milestones I once thought unattainable.

My sincere thanks also go to Prof. Michael Cafarella, another esteemed member of my thesis committee. His vibrant presence and guidance in the Data Systems Group (DSG) lab have been nothing short of inspiring, dating all the way back to the early days of the knowledge graphs reading group he initiated. Witnessing his style of collaborative work with other folks in our lab, as well as his shared passion for graphs, has been incredibly motivating.

The time-evolving cohort of graduate students, postdocs, and research scientists at MIT DSG will forever hold a special place in my heart. There are too many names to mention here, but I am particularly grateful to Carlo Curino and Nesime Tatbul for all their advice and helpful research discussions, as well as for introducing me to their network of professional collaborators. I am thankful to Raul Castro Fernandez for all his guidance early on, and to Andy Ilyas for allowing me the privilege to contribute to his first research paper at MIT. A heartfelt thanks also goes to PDOS folks, especially Jon Gjengset and Malte Schwarzkopf for offering me the opportunity to venture into a "20%" project on Noria during my first semester at MIT, and for introducing me to the world of Rust. While I was initially reserved and mindful of my limitations (that Rust learning curve is no joke), getting to see their ego-less approach to learning a new skillset was illuminating, and one which I tried to cultivate later on when trying to learn as much as I could from Julian.

I would also like to extend my most sincere gratitude to my graduate counselor, Prof. Charles Leiserson, for his invaluable advice and for inspiring me to significantly enhance my skills as a presenter: his wisdom and guidance have been instrumental in my development both as a scholar and as a communicator. I am also deeply thankful to Prof. Leslie Kolodziejski, Janet Fischer, and Alicia Duarte from the graduate office for their unwavering support and assistance. Their dedication and hard work behind the scenes have been crucial in navigating The Institute and its many administrative processes. From thesis and minor approvals, to wonderful GW6 events and wholesome celebrations: their bright energy and support have been indispensable throughout the academic journey of many of us womXn at MIT.

To my dear friend Manoela Alencastro de Moraes, your periodic check-ins and our conversations have been rays of light during the more challenging times of this journey. Your friendship means the world to me, and I will always have a place for you in my heart. To MIT Programming Languages folks, especially Sara Achour, thank you for the friendship and including me in the many special events you organized: your thanksgiving dinner parties in East Cambridge were the absolute best. To Anna Zeng and Calvin Li for our fun chats, hiking and cooking adventure(s), and for making my defense celebration so special: I sincerely hope cheese fondue becomes a tradition at Stata G9 lounge. To Oscar Moll for sharing great advice, fun jokes, and for creating a welcoming environment to another fellow latinX at DSG. I also owe a special thanks to Derek Leung for his presence during numerous significant milestones in this past year, ranging from birthday festivities to my thesis defense and celebration. Your emails make Fridays a lot more fun, thank you for carrying the CSAIL GSB tradition forward.

Lastly, my appreciation goes to my partner Gabriela Jacques da Silva and her family, who have embraced me as one of their own over the last few years. Your love and support, especially during these past few weeks, have been incredibly important to me. To our furry four-legged boys, and especially to Christian Grey, the chonky kitty whose absence is deeply felt, you remain the heart of our family.

This journey has immensely enriched me both professionally and personally, pushing me beyond the boundaries of my comfort zone, and blessing me with invaluable human connections. To each one of you I am forever grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs offer a natural way to model real-world entities and relationships between them. From social networks to biological systems, and from transportation networks to data lineage graphs in cloud computing infrastructure, the ability to analyze and extract meaningful insights from graph data is crucial to a host of human endeavors. For example, cloud computing companies extract large graphs from terabytes of production systems logs every day to inform near real-time operational decisions [67]. As a result, systems aimed at answering queries (i.e., declarative questions) over graph-structured data need to do so efficiently.

Furthermore, real-world graphs often encode relationships that evolve through time. These relationships are commonly modeled as a temporal graph, i.e., one in which edges are associated to time attributes (such as start time and duration), denoting their validity during a certain interval in time. Because of their ability to capture not only relationships, but also their evolution over time, temporal graphs are crucial for uncovering meaningful insights and patterns [23, 75, 91]. For instance, being able to track the chronology of friendship formations on a social network or communication events can enhance our understanding of behavioral patterns. Similarly, the ability to observe the evolution of transportation routes or the progression of a biological system over time can provide critical data for predictive modeling and decision-making.

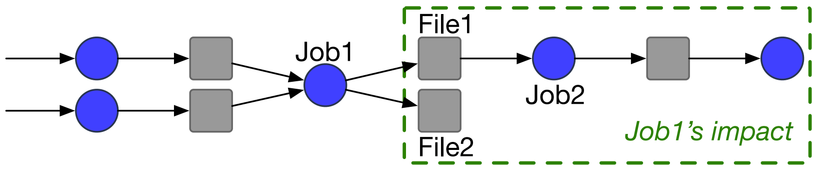


Figure 1-1: Example query over a machine generated data lineage graph extracted from cloud computing infrastructure logs [26, 67]. Blue circles correspond to "jobs" (i.e., logical collection of tasks) running in the cluster. Gray squares correspond to "files". Directed paths in a data lineage graph follow the flow of data being produced or consumed. Cluster administrators often execute the "blast radius" impact query (i.e., a k-hop transitive closure over the data lineage graph) for post-mortem analysis and capacity planning tasks.

1.1 Challenges in Graph Query Processing

The inherent characteristics of graph query workloads as well as data properties from these real-world graphs pose significant challenges for their efficient processing. In this section, we outline the two types of problems that constitute the primary focus of the research we present in this thesis.

1.1.1 Expensive Subgraph Traversals

Queries over large real-world graphs often involve expensive sub-graph traversals and complex analytical computations. A particular characteristic of graph query workloads is that they contain queries that can refer to long chains of dependencies, such as $k-$hop queries. In Figure 1-1 we depict one such example: a "job blast radius query", which is often used for in cloud computing infrastructure to power operational decisions, and which we later detail in §4.1. If executed over distributed relational engines, such path traversal queries can lead to inefficient query plans, especially in the presence of skewed degree distributions (often the case for real-world graphs). Some of the reasons for this inefficiency include the materialization of intermediate computations, as well as the *i.i.d.* assumption that traditional query optimizers often make.

1.1.2 Temporal Dynamics of Real-World Graphs

The dynamic nature of temporal graphs also brings about new challenges. This additional expressiveness comes at a cost, as the temporal dimension often introduces new sources of skew, such as exponentially distributed inter-arrival times and event durations. In addition, existing graph frameworks and query systems frequently encounter difficulties when handling graph processing tasks required in temporal graph analytics applications.

The underlying reasons for these challenges are twofold. First, many of these systems were primarily designed for traditional graph processing, and as a result, they are not well-equipped to handle the unique characteristics and requirements of temporal graphs, in some cases leading to incorrect results [95]. Second, some systems [46, 80, 19, 59] that specifically target temporal graph processing rely on Pregel-like distributed computation models. These models can be highly inefficient due to the message passing overhead across servers in a cluster, especially when the input graph fits comfortably within the memory resources of a single commodity machine. This limitation leads to suboptimal performance and an inability to fully exploit the temporal information available in the data.

1.2 Contributions

To address the challenges outline above, in this thesis we present two distinct systems tailored for optimizing large-scale real-world graph processing.

The first system, Kaskade [26], targets the challenges of efficient query evaluation over very large graphs by leveraging structural properties of graphs and queries to infer materialized graph views. Kaskade formulates the problem of finding valid graph views as a Boolean satisfiability problem. Using this formulation, Kaskade employs a novel *constraint-based view enumeration* technique that mines constraints from query workloads and graph schemas, and injects them during view enumeration to significantly reduce the search space of views to be considered. Materialized view inference is especially useful in this setting because it significantly speeds up query evaluation by rewriting input queries in terms of these views, guided by a cost model that estimates the benefit of such views based on input graph and query characteristics.

The second system, KAIROS [27], introduces *selective indexing*, a technique that chooses a subset of target vertices to index based on characteristics of the underlying temporal graphs and input queries. This system further employs a highly-specialized parallel data structure aimed at in-memory storage and fast retrieval of temporal edges. KAIROS introduces programming primitives and APIs that organize graph edges by time while preserving data locality for vertex neighbors retrieval. In addition, its novel temporal graph index allows for efficiently looking up vertex neighbors in specific time ranges, while taking into account temporal dynamics of vertex degree distributions and information on the input query workload. Finally, KAIROS is built upon Ligra [84, 85], the *de facto* benchmark system in shared-memory parallel graph processing.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the computational model and key concepts we build upon in this work. In Chapter 3 we present a survey of related work. Chapter 4 details the contributions of KASKADE to graph query optimization, including the idea of constrained-based view inference to identify and materialize beneficial graph views. Next, in Chapter 5 we describe KAIROS, the system we built to leverage temporal dynamics present in real-world graph data for efficient graph analytics. Finally, we discuss ideas for future work in Chapter 6, and present our concluding remarks in Chapter 7.

Chapter 2

Background

In this section, we provide key concepts we build upon in this thesis. We define the data models and programming paradigms introduced in prior work, as well as work from which we draw inspiration in concepts that we introduce in later sections. We also categorize different analytics tasks that are commonly performed in some of the contexts that these systems consider. Finally, we conclude with a brief discussion highlighting which concepts are used by which system, as well as in which other contexts these concepts might be applicable.

2.1 Property Graph Data Model

Graph data systems often assume property graphs as their data model [8], in which both vertices and edges are typed and may have properties in the form of key-value pairs. This *schema* captures constraints such as domain and range of edge types. In our data lineage graph example (Figure 1-1), an edge of type "read" only connects vertices of type "job" to vertices of type "file" (and thus never connects two vertices of type "file"). We show a visual example of the schema for such a data lineage graph in Figure 2-1. Most mainstream graph engines provide support for this data model, including the use of schema constraints.

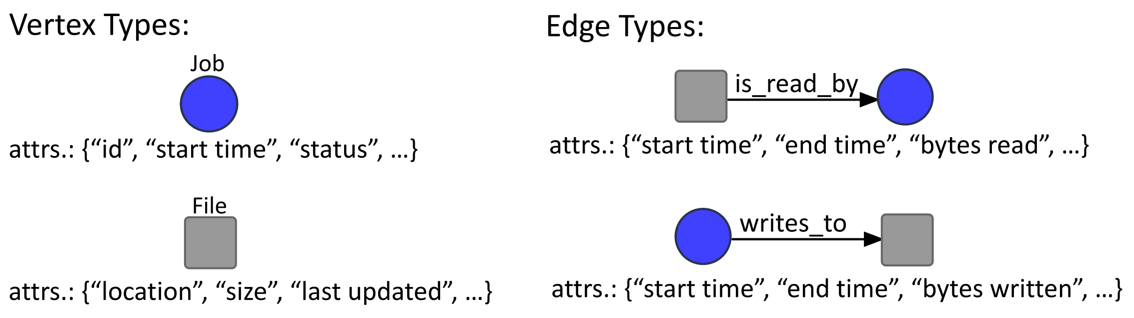

Vertex Types:
Job
attrs.: {"id", "start time", "status", …}
File
attrs.: {"location", "size", "last updated", …}
Edge Types:
is_read_by
attrs.: {"start time", "end time", "bytes read", …}
writes_to
attrs.: {"start time", "end time", "bytes written", …}

Figure 2-1: Example schema for a data lineage graph used in cloud computing data infrastructure to record where data came from in a datacenter.

2.2 Query Language

Kaskade combines regular path queries with relational constructs in its query language. In particular, it leverages the graph pattern specification from Neo4j's Cypher query language [39] and combines it with relational constructs for filters and aggregates. This hybrid query language resembles that of recent industry offerings for graph-structured data analytics, such as those from AgensGraph DB [1], SQL Server [4], and Oracle's PGQL [89].

As an example, consider the job blast radius use case introduced in Chapter 1 (Figure 1-1). Listing 1 illustrates a query that combines OLAP and anchored path constructs to rank jobs in its blast radius based on average CPU consumption.

```
SELECT A.pipelineName, AVG(T_CPU) FROM (
  SELECT A, SUM(B.CPU) AS T_CPU FROM (
    MATCH (q_j1:Job)-[:WRITES_TO]->(q_f1:File)
      (q_f1:File)-[r*0..8]->(q_f2:File)
      (q_f2:File)-[:IS_READ_BY]->(q_j2:Job)
      RETURN q_j1 as A, q_j2 as B
  ) GROUP BY A, B
) GROUP BY A.pipelineName
```

Listing 1: Job blast radius query over raw graph.

Specifically, the query in Listing 1 ranks jobs up to 10 hops away in the downstream of a job (q_j1). In the example query, this is accomplished in Cypher syntax by using a *variable length path* construct of up to 8 hops (`-[r*0..8]->`) between two file vertices (`q_f1` and `q_f2`), where the two files are endpoints of the first and last edge on the complete path, respectively.

2.3 Temporal Graph Data Model

A temporal graph is represented by the tuple $G = (V, E, T, \tau)$:

- V denotes a set of vertices.
- E denotes a set of edges.
- $T = [0, 1, ..., t_{max}] \in \mathbb{N}$ represents a discrete time domain.
- $\tau : V \times V \times T \times T \to \{False, True\}$ is a function that determines for each pair of vertices $u, v \in V$, and each pair of timestamps $t_{start}, t_{end} \in T$ where $t_{start} \leq t_{end}$, whether $(u, v) \in E$, i.e., whether the edge (u, v) exists during the discrete time period from t_{start} to t_{end}.

In other words, each edge in a temporal graph is associated with a discrete time interval indicating its validity. For instance, in interaction networks, this time interval indicates the period during which two vertices have interacted.

A weighted temporal graph is represented by the tuple $G = (V, E, T, \tau, w)$, where w is a function that maps a temporal edge (as defined above) to a real value (its weight). The number of vertices in a temporal graph is $n_v = |V|$, and the number of edges is $n_e = |E|$. Vertices are assumed to be labeled from 0 to $n_v - 1$. For undirected temporal graphs, we use $deg(v)$ to denote the number of edges incident to a vertex $v \in V$. In directed temporal graphs, vertices contain both incoming and outgoing edges. We use $deg^{+}(v)$ to denote the number of outgoing edges, and $deg^{-}(v)$ to denote the number of incoming edges that a vertex v has.

2.4 Allen's Interval Algebra

We draw inspiration from Allen's Interval Algebra [13] to specify the relationships that subsequent edges in the same time-respecting path must have. The following subset of this algebra defines the validity of temporal paths:

- **Succeeds:** For two time intervals A and B, B *succeeds* A if and only if the end time of A is smaller than or equal to the start time of B, i.e., $\textit{end}(A) \leq \textit{start}(B)$.
- **Strictly succeeds:** B *strictly succeeds* A if and only if the end time of A is strictly smaller than the start time of B, i.e., $\textit{end}(A) < \textit{start}(B)$.
- **Overlaps:** B *overlaps* A if and only if the start time of A is less than the start time of B, and the end time of A is less than the end time of B, i.e., $\textit{start}(A) \leq \textit{end}(B)$ and $\textit{end}(A) \leq \textit{start}(B)$.

We refer to this subset as *ordering predicates* in KAIROS, and describe its application in §5.4.1.

2.5 Temporal Graph Analytics Tasks

Here we provide a survey of applications of temporal graph analytics algorithms and queries from the literature, listed in Table 2.1. For each algorithm and query, we present an example application that relies on it as a core primitive for analysis. We focus on a core set of algorithms that address the most common use cases in temporal graph analytics and can serve as primitives for building more advanced analysis tasks. The primary categories for these algorithms are temporal paths, temporal connectivity, and temporal centrality.

Temporal Paths: A temporal path in graph G is a path where every subsequent edge in the path must satisfy certain temporal constraints. Examples of minimal temporal paths include earliest arrival, latest departure, fastest path, and shortest path [95, 94].

Category	Specific instantiation
Temporal Graph Algorithms	
Temporal Minimal Paths	[Provenance] Tracking the origin and flow of information in different systems [29, 91]
	[Indoor Routing] Temporal shortest paths that consider different obstacles for correct navigation [14, 62]
	[Transportation] Route planning that considers real-time traffic conditions in transportation networks [57]
	[Epidemiology] Identifying infection transmission paths in contact tracing [96]
Temporal Connectivity	[Social Networks] Analyzing community evolution and detecting temporal clusters [73, 100, 101, 58]
Temporal Centrality	[Epidemiology] Identifying critical nodes in the spread of infections [23]
Temporal Graph Queries	
Time-constrained reachability	[Social Networks] Analyzing influence propagation and information cascades [76, 70, 102]
Temporal subgraph matching	[Bioinformatics] Detecting conserved patterns in dynamic biological networks [72, 53]

Table 2.1: A Categorization of tasks commonly performed in temporal graph analytics.

Temporal Connectivity: Temporal connectivity deals with the temporal version of connected components. This involves identifying sets of vertices that are connected through time.

Temporal Centrality: Temporal centrality measures the importance of a vertex in a temporal graph. An example of interest is temporal betweenness centrality, which quantifies how frequently a vertex appears on temporal shortest paths between that vertex and other vertices in the graph.

2.6 The Compressed Sparse Row (CSR) Format

In the context of parallel graph processing, a common data structure used to store graph data is the Compressed Sparse Row (CSR) format. This format is largely preferred over other competing data structures as it allows for efficient storage and manipulation of sparse graphs, in which the majority of potential edges are absent [93].

The CSR representation of a graph is characterized by the use of three arrays: an adjacency array, an offset array, and a vertex array. The adjacency array serves as a storage mechanism for destination vertices of all edges in a graph. The edges are placed in a contiguous block of memory, where they are sorted by the source vertex for outgoing edges, and sorted by the destination vertex for incoming edges. This efficient organization supports the fast retrieval of adjacent vertices in graph traversal operations, as all vertices in the same 1-hop neighborhood are located contiguously in memory. The offset array then stores the indices of outgoing / incoming edges into the adjacency array. These indices mark the starting point of the adjacency list for each vertex, enabling quick access to the set of edges associated with any given vertex. Lastly, the vertex array consists of the vertex identifiers, or vertex ids. These ids can be used as a reference to retrieve additional attributes or properties associated with a vertex, or they can represent the vertices themselves in some implementations. In practice, if there is no metadata associated to vertices, then the vertex id is implicit (i.e., offset[i] contains the offset for vertex i), and no separate vertex ids array is needed.

2.7 Shared-memory Graph Processing: Ligra

Ligra is a lightweight graph processing framework designed for shared-memory parallel systems [84]. It enables efficient parallel graph processing by employing a simple and flexible programming model, making it easy for developers to write high-performance algorithms for large-scale graphs.

The programming model of Ligra is centered around two primary operations: *EdgeMap* and *VertexMap*. These operations enable parallel traversal of the graph and are responsible for most of the computation in a Ligra-based algorithm.

EdgeMap is a higher-order function that takes as input a graph G, a subset of vertices V', and an edge function f. It applies the edge function f to all edges (u, v) in the graph, where $u \in V'$ and $v \in V$. The edge function f is responsible for implementing the logic of the specific graph algorithm and can perform various operations, such as updating vertex properties or computing edge weights. EdgeMap efficiently handles parallelism by processing edges in parallel, allowing for scalable performance on shared-memory systems.

VertexMap is another higher-order function that takes as input a graph G, a subset of vertices V', and a vertex function g. It applies the vertex function g to all vertices in V'. Similar to the edge function, the vertex function is responsible for implementing the algorithm-specific logic and can perform operations such as updating vertex properties or aggregating information from neighboring vertices. VertexMap also processes vertices in parallel, ensuring scalable performance.

By using these two core operations, Ligra enables developers to write graph algorithms that can efficiently exploit the parallelism offered by shared-memory systems.

2.8 Discussion

We described above the key concepts that we build upon with our work in KASKADE and KAIROS. Here we briefly discuss what role these ideas from prior work play in each system.

The property data graph model we describe in §2.1 defines what types of vertices can relate to what other types, and how. As we later show in Chapter 4, these schema constraints play an essential role in view enumeration (§4.6).

KAIROS, on the other hand, does not rely on graph schemas, which is a direct result of our decision to build upon Ligra (§2.7), in which vertices and edges have no types. A natural extension here would be to add support for graph schemas, which could be done either directly on Ligra, or by adding a separate key-value store component to store vertex and edge attributes.

Furthermore, although we use the Cypher query language (§2.2) syntax for graph queries in KASKADE, the techniques we introduce in KASKADE can be coupled with any query language, as long as it can express graph pattern matching and schema constructs. KAIROS, on the other hand, does not offer support for declarative queries, and instead offers a an API which we describe in §5.4.

In addition, as we later describe in Chapter 5, KAIROS introduces an extension to the CSR format for the temporal setting. This compact and cache-friendly format allows for efficient computations in a number of graph analytics tasks. In KAIROS, this temporal version of CSR is used in conjunction with a specialized data structure that further improves performance of algorithms over large temporal graphs. Kaskade's contributions, on the other hand, are engine agnostic in the sense that they only rely on fundamental graph transformations, and leverage the target storage engine's API for computing graph views. In other words, the internal data representation that the graph engine uses is transparent to Kaskade: hence, if an engine uses CSR (temporal or not) to store data graphs, Kaskade will automatically use it as well.

We also introduced in §2.7 the Ligra programming model for shared-memory parallel graph processing applications. In Chapter 5 we further detail how we extend it to the temporal setting in our work with Kairos. In our experimental evaluation of Kaskade we store materialized graph views on Neo4j, a graph database that at the time already supported graph schemas. Nevertheless, if one implements support for the property graph data model in Ligra, adding support to schemas, then we could also use Ligra to materialize as well as precompute some of those views. This would probably lead to faster execution runtimes, as Ligra makes better use of computational resources with its support for cache-efficient graph data representations, such as CSR.

Chapter 3

Related Work

The work we present in this thesis is connected to prior literature in two key areas. We start with work that relates to materialization of graph views (§3.1), and conclude with related works in systems for efficient processing of temporal graph data (§3.2).

3.1 Materialized Graph Views for Query Rewriting

3.1.1 Graph Views and Query Language

Materialized views have been widely used in the relational setting to improve query runtime by amortizing and hiding computation costs [41]. This inspires our work in KASKADE, but the topological nature of graph views makes them new and different. The notion of graph views and algorithms for their maintenance was first introduced in[104] by Zhuge and Garcia-Molina in 1998. When we first proposed KASKADE, there had been little attention in this area. With KASKADE, we aim at providing a practical approach that can be used to deal with large real-world graphs. It addresses various problems related to graph views, including view enumeration, selection, as well as view-based query rewriting.

In KAIROS, we focus on extracting views for the graph traversal portion of our queries, since these are the most crucial for performance. An interesting avenue for future work is to address a combined analysis of the relational and graph query fragments, related to what [60] proposes for a more restricted set of query types or [21] does for OLAP on graphs scenarios. The challenge is to identify and overcome the limits of relational query containment and view rewriting techniques for our hybrid query language. A related challenge is to support maintenance of our graph views in the case of updates, which we leave as future work.

Fan et al. [32] provide algorithms to generate views for speeding up fixed-sized subgraph queries, whereas KASKADE targets traversal-type queries that can contain an arbitrary number of vertices/edges. They do not provide a system for selecting the best views to generate based on a budget constraint. Le et al. [54] present algorithms for rewriting queries on SPARQL views, but lack view selection. Katsifodimos et al. [50] present techniques for view selection to improve performance of XML queries, which are limited to trees due to XML's structure.

3.1.2 Property Graph Databases

KASKADE is a graph query optimization framework that proposes a novel *constraint-based view inference* technique for materialized view selection and query rewriting. Although it is not a graph engine itself, in its current design it ultimately acts as one. However, we believe existing graph query execution engines can leverage KASKADE for query optimization. There are many such systems (see [98] for a survey). Our approach is mostly independent of the underlying graph engine, and while we run our experiments on Neo4J, KASKADE is directly applicable to any engine supporting Cypher [39] (or SQL+Cypher).

3.1.3 RDF and XML

The Semantic Web literature has explored the storage and inference retrieval of RDF and OWL data extensively [15]. While most efforts focused on indexing and storage of RDF triples, there has also been work on maintenance algorithms for aggregate queries over RDF data [48]. While relevant, this approach ignores the view selection and rewriting problems we consider here, and it has limited applicability outside

RDF. RDFViewS [40] addresses the problem of view selection in RDF databases. However, the considered query and view language support pattern matching (which are translatable to relational queries) and not arbitrary path traversals, which are crucial in the graph applications we consider. Similar caveats are present in prior work on XML rewriting [71, 30], including lack of a cost model for view selection.

3.1.4 Graph Functional Dependencies

Graph Functional Dependencies (GFDs) work aims at discovering parts of a graph that violate topological constraints [38, 34, 31, 33, 37]. Our focus with graph query optimization is different: materialize smaller graphs to speed up query execution. We see their work as complementary to ours, in that dependencies discovered by their approach may be provided in the form of schema constraints inference rules in KASKADE. A future work direction is to translate instantiations constraint mining rules (§4.6.1) into GFDs.

3.1.5 Graph Summarization and Compression

Summarization finds smaller graphs that are representative of the original graph to speed up graph algorithms or queries, for graph visualization, or to remove noise [63]. Most related is summarization to speed up graph computations for certain queries (lossless or lossy) [36, 69, 97, 86, 20]. As far as we know, prior work in this area has not explored the use of connectors and summarizers as part of a general system to speed up graph queries. Rudolf et al. [79] describe summarization templates in SAP HANA, which can be used to produce what we call graph views. However, their paper lacks a system that determines what views to materialize, nor uses the views to speed up graph queries. There has been significant work on lossless graph compression to reduce space usage or improve performance of graph algorithms (see, e.g., [18, 85, 65]). This is complementary to our work on graph views, and compression could be applied to reduce their memory footprint.

3.2 Efficient Processing of Temporal Graph Data

3.2.1 Temporal Graph Engines

While the authors of [95, 94] have been the first to propose one-pass parallel versions of temporal graph algorithms, as far as we know [46] is the only other shared-memory temporal graph analytics system. Previous systems for temporal graph analytics have primarily relied on distributed message passing, often building upon programming paradigms like Pregel or stateful data stream processing frameworks such as Apache Flink [80, 74, 59, 77]. The large number of messages exchanged when relying on these programming paradigms imposes considerable overhead on graph processing. This becomes particularly noticeable in the case of graphs that fit within available memory of current commodity servers, as shown by the speedups KAIROS gets when compared against these systems (Chapter 5).

3.2.2 Time-evolving Graph Engines

There has been significant work on developing frameworks for processing of graphs that evolve over time [44, 68, 22, 52, 49]. *While these systems allow processing of snapshots, streaming graphs, or dynamic graphs dynamic graphs, for their most part they do not support temporal graphs nor temporal graph algorithms.* Rather, timestamps associated to edges or vertices are treated as graph updates in these systems. For this reason, we consider these orthogonal.

3.2.3 Range Query Data Structures

There has been extensive work on data structures to handle range queries. Traditional range query data structures often used in relational systems include quad-trees, R-trees, kd-trees. Specifically aimed at interval data – such as that present in temporal edges – are interval trees, segment trees, and priority search trees [16]. All three of these data structures have runtime complexity of $O(logn + k)$, where k is the number of results for the range query. Where they differ is on space complexity, which is highest

Data Structure	Runtime	Queries Supported
Interval Tree	$O(\log n + k)$	stabbing
Priority Search Tree	$O(\log n + k)$	3-sided region
kd-Tree	$O(\sqrt{n} + k)$	4-sided region

Table 3.1: Example data structures for range search over time intervals. Asymptotic query runtime complexity is given for n intervals stored, with k matching results for the input query.

for segment trees at $O(nlogn)$, as well as on properties relating to how they can be queried (e.g., stabbing vs 3-sided queries). In Table 3.1 we offer a brief comparison of the three data structures we considered when initially designing Kairos. As we describe later in Chapter 5, we opted for building a highly-optimized, specialized, and re-configurable version of priority search tree as index for storing temporal edges.

3.2.4 Graph Neural Networks

Graph Neural Networks (GNNs) have emerged as a powerful framework for learning representations of graph-structured data, tackling problems in various domains such as social networks, molecular biology, and recommendation systems [90, 103, 64]. GNNs rely on the underlying graph structure and the attributes of nodes and edges to learn complex patterns and make predictions, with recent increased focus on temporal graphs [103, 47]. By employing efficient graph analytics algorithms, these systems can compute a wide range of graph properties, such as centrality measures, graph motifs, or community structures, which can then be used as additional input features for GNNs. In the context of GNNs, most related to our work is the version of Temporal CSR used for sampling temporal edges in [103]. While it does not account for end time edges, as far as we know this is the only other work that extends CSR to the temporal setting.

3.3 Discussion

To further contextualize our work, here we briefly discuss how the techniques and systems we propose in this thesis fit into the overall picture of graph data processing systems. First, KASKADE can be seen as a *graph query optimization framework* and sits atop other graph data systems. As KASKADE is not itself, a graph data processing system, we do not discuss related work in this area. For a comprehensive survey of current challenges in large-graph data processing systems, which also covers overall directions and workload trends, we direct the reader to [81].

Additionally, while KAIROS is a temporal graph processing system, it assumes its input temporal graphs are static. In other words, it does not currently handle data dynamism such as graph edge removals or updates. For systems that do not necessarily target temporal graphs, but that can handle dynamic graphs as well as streaming graphs, we direct the reader to a comprehensive survey by Maciej Besta et al [17].

Chapter 4

Kaskade

As we outlined in Chapter 1, an increasingly relevant type of workload over real-world graphs involves analytics computations that mix traversals and computation, e.g., finding subgraphs with specific connectivity properties or computing various metrics over sub-graphs. This has resulted in several systems being designed to handle complex queries over such graphs [98]. In such scenarios, graph analytics queries require response times on the order of a few seconds to minutes, because they are either exploratory queries run by users (e.g., recommendation or similarity search queries) or they power systems making online operational decisions (e.g., provenance use cases, capacity planning and incident response, as well as detecting privacy violations in the context of GDPR). However, many of these queries involve the enumeration of large subgraphs of the input graph, which can easily take minutes to hours to compute over large graphs on modern graph systems. To achieve our target response times over large graphs, new techniques are needed.

We observe that the graphs in many of these applications have an inherent structure: their vertices and edges have specific types, following well-defined schemas and connectivity properties. For instance, social network data might include users, pages, and events, which can be connected only in specific ways (e.g., a page cannot "like" a user), or workload management systems might involve files and jobs, with all files being created or consumed by some job. As we discuss in §4.1, the data lineage graphs that cloud data infrastructure companies maintain often have similar structural

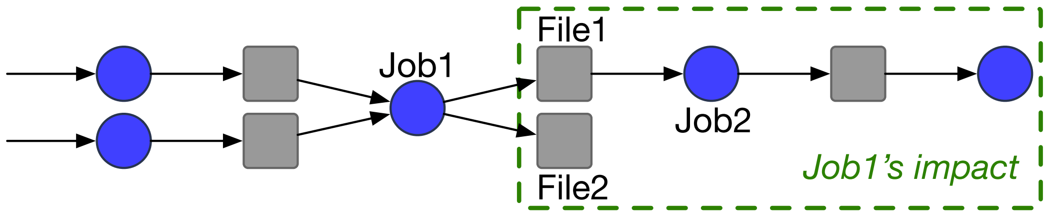


Figure 4-1: Running example of query over a heterogeneous network: the "blast radius" impact for a given job in a data lineage graph (blue circles correspond to jobs; gray squares to files).

constraints. However, most existing graph query engines do not take advantage of this structure to improve query evaluation time.

At the same time, we notice that *similar queries are often run repeatedly* over the same graph. Such queries can be identified and materialized as views to avoid significant computation cost during their evaluation. The aforementioned *structural regularity* of these graphs can be exploited to efficiently and automatically derive these materialized views. Like their relational counterparts, such *graph views* allow us to answer queries by operating on much smaller amounts of data, hiding/amortizing computational costs and ultimately delivering substantial query performance improvements of up to 50X in our experiments on real-world graphs. As we show, the benefits of using graph views are more pronounced in *heterogeneous* graphs[1] that include a large number of vertex and edge types with connectivity constraints between them.

4.1 Motivating example

Cloud computing infrastructure often operate large data lakes, at times storing several exabytes of data and processing them with hundreds of thousands of jobs, spawning billions of tasks daily [24]. Operating such a massive infrastructure requires cloud providers handle data governance and legal compliance (e.g., GDPR), optimize systems based on our query workloads, supporting metadata management and enterprise search, as detailed in [42, 66]. A natural way to represent this data and track datasets and

[1]By heterogeneous, we refer to graphs that have more than one vertex types, as opposed to homogeneous with a single vertex type.

computations at various levels of granularity is to build a *provenance graph* that captures data dependencies among jobs, tasks, files, file blocks, and users in the lake. As discussed above, only specific relationships among vertices are allowed, e.g., a user can submit a job, and a job can read or write files.

To enable these applications over the provenance graph, we need support for a wide range of *structural* queries. Finding files that contain data from a particular user or created by a particular job is an anchored graph traversal that computes the reachability graph from a set of source vertices, whereas detecting overlapping query sub-plans across jobs to avoid unnecessary computations can be achieved by searching for jobs with the same set of input data. Other queries include label propagation (i.e., marking privileged derivative data products), data valuation (i.e., quantifying the value of a dataset in terms of its "centrality" to jobs or users accessing them), copy detection (i.e., finding files that are stored multiple times by following copy jobs that have the same input dataset), and data recommendation (i.e., finding files accessed by other users who have accessed the same set of files that a user has).

We highlight the optimization opportunities in these types of queries through a running example: the *job blast radius*. Consider the following query operating on the provenance graph: *"For every job j, quantify the cost of failing it, in terms of the sum of CPU-hours of (affected) downstream consumers, i.e., jobs that directly or indirectly depend on j's execution."* This query, visualized in Figure 4-1, traverses the graph by following read/write relationships among jobs and files, and computes an aggregate along the traversals. Answering this query is necessary for cluster operators and analysts to quantify the impact of job failures—this may affect scheduling and operational decisions.

By analyzing the query patterns and the graph structure, we can optimize the job blast radius query in the following ways. First, observe that the graph has structural connectivity constraints: jobs produce and consume files, but there are no file-file or job-job edges. Second, not all vertices and edges in the graph are relevant to the query, e.g., it does not use vertices representing tasks. Hence, we can prune large amounts of data by storing as a view only vertices and edges of types that are required by the query. Third, while the query traverses job-file-job dependencies, it only uses metadata from jobs. Thus, a view storing only jobs and their (2-hop) relationships to other jobs further reduces the data we need to operate on and the number of path traversals to perform. A key contribution of our work is that these views of the graph can be used to answer queries, and if materialized, query answers can be computed much more quickly than if computed over the entire graph.

4.2 Contributions

Motivated by the above scenarios, we have built Kaskade, a graph query optimization framework that employs graph views and materialization techniques to efficiently evaluate queries over graphs. Our contributions are as follows:

Query optimization using graph views. We identify a class of graph views that can capture the graph use cases we discussed above. We then provide algorithms to perform view selection (i.e., choose which views to materialize given a query set) and view-based query rewriting (i.e., evaluate a query given a set of already materialized views). As far as we know, this is the first work that employs graph views in graph query optimization.

Constraint-based view enumeration. Efficiently enumerating candidate views is crucial for the performance of view-based query optimization algorithms. To this end, we introduce a novel technique that mines constraints from the graph schema and queries. It then leverages view templates expressed as inference rules to generate candidate views, injecting the mined constraints at runtime to reduce the search space of views to consider. The number of views our technique enumerates is further lowered when using the query constraints.

Cost model for graph views. Given the importance of path traversals, we introduce techniques to estimate the size of views involving such operations and to compute their creation cost. Our cost model is crucial for determining which views are the most beneficial to materialize. Our experiments show that by leveraging graph schema constraints and associated degree distributions, we can estimate the size of various

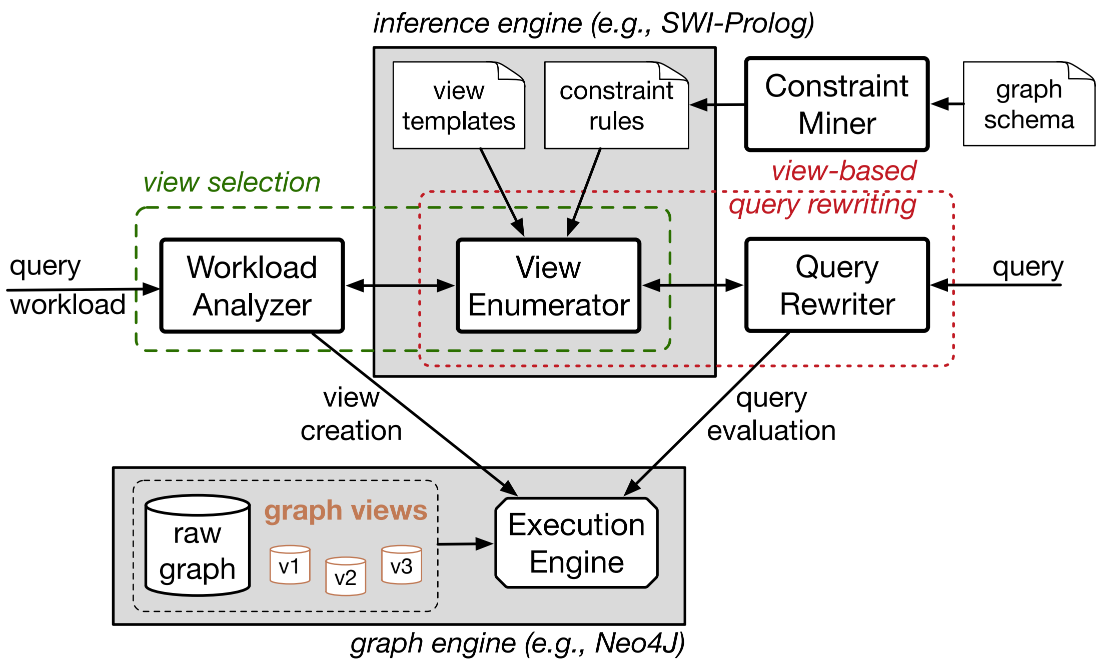


Figure 4-2: Architecture of Kaskade.

path views in a number of real-world graphs reasonably well.

Results over real-world graphs. We have incorporated all of the above techniques in our system, Kaskade, and have evaluated its efficiency using a variety of graph queries over both heterogeneous and homogeneous graphs. Kaskade is capable of choosing views that, when materialized, speed up query evaluation by up to 50X on heterogeneous graphs.

4.3 Overview

Kaskade is a graph query optimization framework that materializes graph views to enable efficient query evaluation. As noted in Chapter 1, it is designed to support complex enumeration queries over large subgraphs, often involving reporting-oriented applications that repeatedly compute filters and aggregates, and apply various analytics over graphs.

Kaskade's architecture is depicted in Figure 4-2. Users submit queries in a language that includes graph pattern constructs expressed in Cypher [39] and relational constructs expressed in SQL. They use the former to express path traversals, and the latter for filtering and aggregation operations. This query language, described in §4.4, is capable of capturing many of the applications described above.

KASKADE supports two main view-based operations: (i) *view selection*, i.e., given a set of queries, identify the most useful views to materialize for speeding up query evaluation, accounting for a space budget and various cost components; and (ii) *view-based query rewriting*, i.e., given a submitted query, determine how it can be rewritten given the currently materialized views in the system to improve the query's execution time by leveraging the views. These operations are detailed in §4.7. The *workload analyzer* drives view selection, whereas the *query rewriter* is responsible for the view-based query rewriting.

An essential component in both of these operations is the *view enumerator*, which takes as input a query and a graph schema, and produces candidate views for that query. A subset of these candidates will be selected for materialization during view selection and for rewriting a query during view-based query rewriting. As we show in §4.6, KASKADE follows a novel constraint-based view enumeration approach. In particular, KASKADE's *constraint miner* extracts (explicit) constraints directly present in the schema and queries, and uses constraint mining rules to derive further (implicit) constraints. KASKADE employs an inference engine (we use Prolog in our implementation) to perform the actual view enumeration, using a set of view templates it expresses as inference rules. Further, it injects the mined constraints during enumeration, leading to a significant reduction in the space of candidate views. Moreover, this approach allows us to easily include new view templates, extending the capabilities of the system, and alleviates the need for writing complicated code to perform the enumeration.

KASKADE uses an execution engine component to create the views that are output by the workload analyzer, and to evaluate the rewritten query output by the query rewriter. In this work, we use Neo4j's execution engine [6] for the storage of materialized views, and to execute graph pattern matching queries. However, our query rewriting techniques can be applied to other graph query execution engines, so long as their query language supports graph pattern matching clauses.

4.4 Preliminaries

4.4.1 Graph Views

We define a *graph view* over a graph G as the graph query Q to be executed against G. This definition is similar to the one first introduced by Zhuge and Garcia-Molina [104], but extended to allow the results of Q to also contain new vertices and edges in addition to those in G. A *materialized graph view* is a physical data object containing the results of executing Q over G.

Kaskade can support a wide range of graph views through the use of *view templates*, which are essentially inference rules, as we show in §4.6. Among all possible views, we identify two classes, namely *connectors* and *summarizers*, which are sufficient to capture most of the use cases that we have discussed so far. Intuitively, connectors result from operations over paths (i.e., path contractions), whereas summarizers are obtained via summarization operations (filters or aggregates) that reduce the size of the original graph.

As an example, Figure 4-3 shows the construction of 2-hop same-vertex-type connector views over a data lineage graph (similar to the graph of Figure 4-1). In Figure 4-3(a), the input graph contains two types of vertices, namely jobs (vertex labels with a j prefix) and files (vertex labels with an f prefix), as well as two types of edges, namely w (a job *writes to* a file) and r (a file *is read by* a job). Figure 4-3(b) shows two different types of connector edges: the first contracts 2-hop paths between pairs of job vertices (depicted as blue dashed edges); the second contracts 2-hop paths between pairs of file vertices (depicted as red dashed edges). Finally, Figure 4-3(c) shows the two resulting connector graph views: one containing only the *job-to-job* connector edges (left), and one with only the *file-to-file* connector edges (right).

Next, we introduce a formal definition and examples of connectors and summarizers. We consider their definition as a means to an end rather than a fundamental contribu-

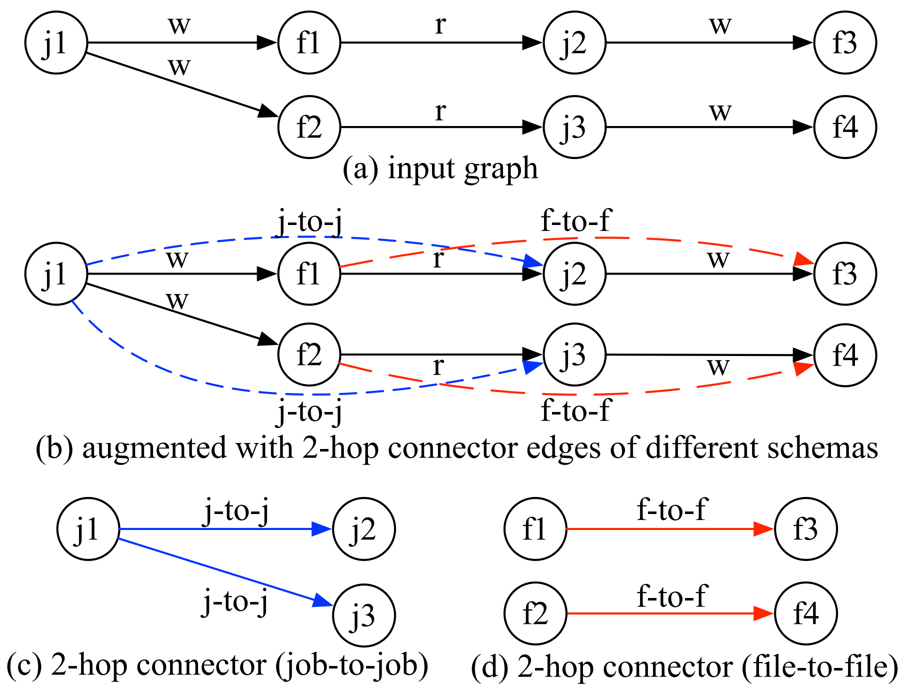

j1
w
f1
r
j2
w
f3
w
f2
r
j3
w
f4
(a) input graph
j-to-j
f-to-f
j-to-j
f-to-f
(b) augmented with 2-hop connector edges of different schemas
j-to-j
j-to-j
f-to-f
f-to-f
(c) 2-hop connector (job-to-job)
(d) 2-hop connector (file-to-file)

Figure 4-3: Construction of different 2-hop connector graph views over a heterogeneous network (namely, a data lineage graph) with two vertex types ($N = 2$) and two edge types ($M = 2$).

tion of this work. Thus, here we only provide sufficient information over graph views for the reader to follow our techniques in Sections 4.6 and 4.7. As path operations tend to be the most expensive operations in graphs, our description will pivot mostly around connectors. Materializing such expensive operations can lead to significant performance improvements at query evaluation time. Moreover, the semantics of connectors is less straightforward than that of summarizers, which resemble their relational counterparts (filters and aggregates). Note, however, that the techniques described here are generic and apply to any graph views expressible as view templates.

4.5 View Definitions and Examples

Next we provide formal definitions of the two main classes of views supported in Kaskade, namely connectors and summarizers, which were briefly described in §4.4.1. Moreover, we give various examples of the views from each category that are currently

Table 4.1: Connectors in Kaskade

Type	Description
Same-vertex-type connector	Target vertices are all pairs of vertices with a specific vertex type.
k-hop connector	Target vertices are all vertex pairs that are connected through k-length paths.
Same-edge-type connector	Target vertices are all pairs of vertices that are connected with a path consisting of edges with a specific edge type
Source-to-sink connector	Target vertices are (source, sink) pairs, where sources are the vertices with no incoming edges and sinks are vertices with no outgoing edges.

present in Kaskade's view template library.

While the examples below are general enough to capture many different types of graph structures, they are by no means an exhaustive list of graph view templates that are possible in Kaskade; as we mention in §4.6, Kaskade's library of view templates and constraint mining rules is readily extensible.

4.5.1 Connectors

A connector of a graph $G = (V, E)$ is a graph G' such that every edge $e' = (u, v) \in E(G')$ is obtained via contraction of a single directed path between two target vertices $u, v \in V(G)$. The vertex set $V(G')$ of the connector view is the union of all target vertices with $V(G') \subseteq V(G)$. Based on this definition, a number of specialized connector views can be defined, each of which differs in the target vertices that it considers. Table 4.1 lists examples currently supported in Kaskade.

Additionally, it is easy to compose the definition of k-hop connectors with the other connector definitions, leading to more connector types. As an example, the k-hop same-vertex-type connector is a same-vertex-type connector with the additional requirement that the target vertices should be connected through k-hop paths. Finally, connectors are useful in contexts where a query's graph pattern contains relatively long paths that can be contracted without loss of generality, or when only the endpoints of the graph pattern are projected in subsequent clauses in the query.

4.5.2 Summarizers

A summarizer of a graph $G = (V, E)$ is a graph G' such that $V(G') \subseteq V(G)$, $E(G') \subseteq E(G)$, and at least one of the following conditions are true: (i) $|V(G')| < |V(G)|$, or (ii) $|E(G')| < |E(G)|$. The summarizer view operations that KASKADE currently provides are filters that specify the type of vertices or edges that we want to preserve (inclusion filters) or remove (exclusion filters) from the original graph.[2] Also, it provides aggregator summarizers that either group a set of vertices into a supervertex, a group of edges into a superedge, or a subgraph into a supervertex. Finally, aggregator summarizers require an aggregate function for each type of property present in the domain of the aggregator operation. We provide examples of summarizer graph views currently supported in Listing 2. KASKADE's library of template views currently does not support aggregation of vertices of different types. The library is readily extensible, however, and aggregations for vertices or edges with different types are expressible using a higher-order aggregate function to resolve conflicts between properties with the same name, or to specify the resulting aggregate type.

Lastly, summarizers can be useful when subsets of data (e.g., entire classes of vertices and edges) can be removed from the graph without incurring any side-effects (in the case of filters), or when queries refer to logical entities that correspond to groupings of one or more entity at a finer granularity (in the case of aggregators).

4.6 Constraint-Based View Enumeration

Having introduced a preliminary set of graph view types in §4.4.1 (see also §4.5 for more examples), we now describe KASKADE's view enumeration process, which generates candidate views for the graph expressions of a query. As discussed in §4.3, view enumeration is central to KASKADE (see Figure 4-2), as it is used in both view selection and query rewriting, which are described in §4.7. Given that the space of view candidates can be large, and that view enumeration is on the critical path of

[2]Summarizer views can also include predicates on vertex/edge properties in their definitions. Using such predicates would further reduce the size of these views, but given they are straightforward, here we focus more on predicates for vertex/edge types.

```
% summarizers: filter vertices and edges by type
summarizerRemoveEdges(X, Y, ETYPE_REMOVE, ETYPE_KEPT) :-
  queryEdge(X, Y), not(queryEdgeType(X, Y, ETYPE_REMOVE)),
  queryEdgeType(X, Y, ETYPE_KEPT).
summarizerRemoveVertices(X, VTYPE_REMOVE, VTYPE_KEPT) :-
  queryVertex(X), not(queryVertexType(X, VTYPE_REMOVE)),
  queryVertexType(X, VTYPE_KEPT).

% Example aggr function for higher-order functions such
% as aggregator graph view templates.
sum(X, Y, R) :- R is X + Y.

% Ego-centric k-hop neighborhood (undirected).
queryVertexKHopNbors(K, X, LIST) :- queryVertex(X),
  findall(SRC, queryKHopPath(SRC, X, K), INLIST),
  findall(DST, queryKHopPath(X, DST, K), OUTLIST),
  append(INLIST, OUTLIST, TMPLIST), sort(TMPLIST, LIST).

% Example aggregator using k-hop neighborhood, e.g.,
% aggregate all 1-hop neighbors as sum of their
% bytes: "kHopNborsAggregator(1, j2, 'bytes', sum, R)."
kHopNborsAggregator(K, X, P, AGGR, RESULT) :-
  queryVertexKHopNbors(K, X, NBORS),
  convlist(property(P), NBORS, OUTLIST),
  foldl(AGGR, OUTLIST, 0, RESULT).
```

Listing 2: Example view template definitions for summarizers.

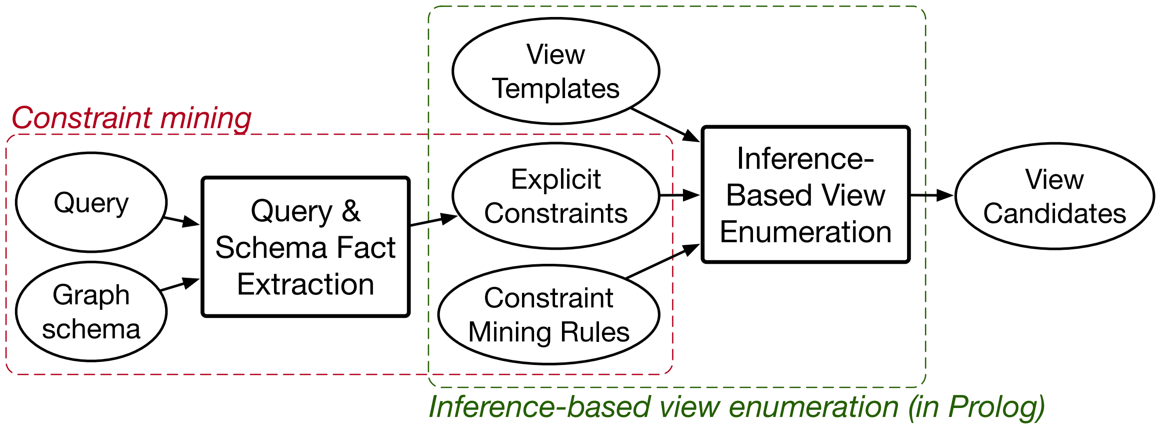

Constraint mining
Query
Graph schema
Query & Schema Fact Extraction
View Templates
Explicit Constraints
Constraint Mining Rules
Inference-Based View Enumeration
View Candidates
Inference-based view enumeration (in Prolog)

Figure 4-4: Constraint-based view enumeration in Kaskade.

view selection and query rewriting, its efficiency is crucial.

In Kaskade we use a novel approach, which we call *constraint-based view enu-*

meration. This approach mines constraints from the query and the graph schema and injects them at view enumeration time to drastically reduce the search space of graph view candidates. Figure 4-4 depicts an overview of our approach. Our view enumeration takes as input a query, a graph schema and a set of declarative view templates, and searches for instantiations of the view templates that apply to the query. KASKADE expresses view templates as inference rules, and employs an inference engine (namely, Prolog)[3] to perform view enumeration through rule evaluation. Importantly, KASKADE generates both *explicit* constraints, extracted directly from the query and schema, and *implicit* ones, generated via constraint mining rules. This constraint mining process identifies structural properties from the schema and query that allow it to significantly prune the search space for view enumeration, discarding infeasible candidates (e.g., job-to-job edges or 3-hop connectors in our provenance example).

Apart from effectively pruning the search space of candidate views, KASKADE's view enumeration provides the added benefit of not requiring the implementation of complex transformations and search algorithms—the search is performed by the inference engine automatically via our view templates and efficiently via the constraints that we mine. Moreover, view templates are readily extensible to modify the supported set of views. In our current implementation, we employ SWI-PL's inference engine [11], which gives us the flexibility to seamlessly add new view templates and constraint mining rules via Prolog rules. On the contrary, existing techniques for view enumeration in the relational setting typically decompose a query through a set of application-specific transformation rules [83] or by using the query optimizer [78], and then implement search strategies to navigate through the candidate views. Compared to our constraint-based view enumeration, these approaches require higher implementation effort and are inflexible when it comes to adding or modifying complex transformation rules.

We detail the two main parts of our constraint-based view enumeration, namely the constraint mining and the inference-based view enumeration, in Sections 4.6.1

[3]While a subset of KASKADE's constraint mining rules are expressible in Datalog, Prolog provides native support (i.e., without extensions) for aggregation and negation, and we use both in summarizer view templates. KASKADE also relies on higher-order predicates (e.g., `setof`, `findall`) in constraint mining rules, which Datalog does not support.

and 4.6.2, respectively.

4.6.1 Mining Structural Graph Constraints

Kaskade exploits information from the graph's schema and the given query in the form of constraints to prune the set of candidate views to be considered during view enumeration. It mines two types of constraints:

- **Explicit constraints** (§4.6.1) are first-order logic statements (facts) extracted from the schema and query (e.g., that files do not write files, only jobs);
- **Implicit constraints** (§4.6.1) are not present in the schema or query, but are derived by combining the explicit constraint facts with constraint mining rules. These constraints are essential in Kaskade, as they capture structural properties that otherwise cannot be inferred by simply looking at the input query or schema properties.

Extracting explicit constraints

The first step in our constraint-based view enumeration is to extract explicit constraints (facts) from the query and schema.

Transforming the query to facts. Our view enumeration algorithm goes over the query's `MATCH` clause, i.e., its graph pattern matching clause, and for each vertex and edge in the graph pattern, Kaskade's constraint miner emits a set of Prolog facts. In our example of the job "blast radius" query (Listing 1), Kaskade extracts the following facts from the query:

```
queryVertex(q_f1). queryVertex(q_f2).
queryVertex(q_j1). queryVertex(q_j2).
queryVertexType(q_f1, 'File').
queryVertexType(q_f2, 'File').
queryVertexType(q_j1, 'Job').
queryVertexType(q_j2, 'Job').
```

```
queryEdge(q_j1, q_f1). queryEdge(q_f2, q_j2).
queryEdgeType(q_j1, q_f1, 'WRITES_TO').
queryEdgeType(q_f2, q_j2, 'IS_READ_BY').
queryVariableLengthPath(q_f1, q_f2, 0, 8).
```

The above set of facts contains all named vertices and edges in the query's graph pattern, along with their types, and any variable-length regular path expression (`queryVariableLengthPath( X,Y,L,U)` corresponds to a path between nodes `X` and `Y` of length between `L` and `U`). In Listing 1, a variable-length regular path of up to 8 hops is specified between query vertices `q_f1` and `q_f2`.

Transforming the schema to facts. Similar to the extraction of query facts, our view enumeration algorithm goes over the provided graph schema and emits the corresponding Prolog rules. For our running example of the data lineage graph, there are two types of vertices (files and jobs) and two types of edges representing the producer-consumer data lineage relationship between them. Hence, the set of facts extracted about this schema is:

```
schemaVertex('Job'). schemaVertex('File').
schemaEdge('Job', 'File', 'WRITES_TO').
schemaEdge('File', 'Job', 'IS_READ_BY').
```

Mining implicit constraints

Although the explicit query and schema constraints that we introduced so far restrict the view enumeration to consider only views with meaningful vertices and edges (i.e., that appear in the query and schema), they still allow many infeasible views to be considered. For example, if we were to enumerate k-hop connectors between two files to match the variable-length path in the query of Listing 1, all values of $k \geq 2$ would have to be considered. However, given the example schema, we know that only even values of k are feasible views. Similarly, since the query specifies an upper limit $u = 8$ in the number hops in the variable-length path, we should not be enumerating k-hop connectors with $k \geq 8$.

To derive such implicit schema and query constraints that will allow us to significantly prune the search space of views to consider, KASKADE uses a library of constraint mining rules[4] for the schema and query. Listing 3 shows an example of such a constraint mining rule for the schema. Rule `schemaKHopPath`, expressed in Prolog in the listing, infers all valid k-hop paths given the input schema. Note that the rule takes advantage of the explicit schema constraint `schemaEdge` to derive this more complex constraint from the schema, and it considers two instances of `schemaKHopPath` different if and only if all values in the resulting unification tuple are different. For example, a `schemaKHopPath(X=‘Job’,Y=‘Job’,K=2)` unification, i.e., a *job-to-job* 2-hop connector is different from a `schemaKHopPath(X=‘File’,Y=‘File’,K=2)`, i.e., a *file-to-file* 2-hop connector. For completeness, we also provide a procedural version of this constraint mining rule in 1. When contrasted with the Prolog rule, the procedural version is not only more complex, but it also explores a larger search space. This is because the procedural function cannot be injected at view enumeration time as an additional rule together with other inference rules that further bound the search space (see §4.6.2).

Similar constraint mining rules can be defined for the query. The extended preprint version at [25] shows examples of such rules, e.g., to bound the length of considered k-hop connectors (in case such limits are present in the query's graph pattern), or to ensure that a node is the source or sink in a connector.

These constraints play a crucial role in limiting the search space for valid views. As we show in §4.6.2, by injecting such rules at view enumeration time, KASKADE can automatically derive implicit knowledge from the schema and query to significantly reduce the search space of considered views. Interestingly, KASKADE can derive this knowledge only on demand, in that the constraint mining rules get fired only when required and do not blindly derive all possible facts from the query and schema.

Importantly, the combination of both schema and query constraints is what makes it possible for our view enumeration approach to reducing the search space of possible

[4]The collection of constraint mining rules KASKADE provides is readily extensible, and users can supply additional constraint mining rules if desired.

Algorithm 1 Procedural version of `schemaKHopPaths` constraint mining Prolog rule

```
// Procedural version of one the declarative constraint mining
// programs that bound the search space for valid candidate
// views (Listing 3 and §4.6.1)
function K_HOP_SCHEMA_PATHS(schema_edges, paths, k, curr_k)
  if curr_k == 0 then return [p for p ∈ paths if len(p) == k]
  if k == curr_k then
    new_paths ← [[e] for e ∈ schema_edges]
    return k_hop_schema_paths(schema_edges, new_paths, k, k-1)
  new_paths ← []
  for {i, path} ∈ paths do
    src, dst ← path[0][0], path[-1][1]
    for {j, edge} ∈ schema_edges do
      // Add edge to the end of the path.
      if dst == edge[0] then new_paths.append(path + [edge])
      // Add edge to the front of the path.
      if src == edge[1] then new_paths.append([edge] + path)
  // Step omitted: duplicate paths removal.
  // Fix-point: only include paths that grew this round.
  paths ← [p for p ∈ new_paths if len(p) == (k-curr_k+1)]
  return K_HOP_SCHEMA_PATHS(schema_edges, paths, k, curr_k-1)
```

```
% Determine whether acyclic directed k-length paths
% between two nodes X and Y are feasible over the input
% graph schema. schemaEdge are explicit constraints
% extracted from the schema.
schemaKHopPath(X,Y,K) :- schemaKHopPath(X,Y,K,[]).
schemaKHopPath(X,Y,1,_) :- schemaEdge(X,Y,_).
schemaKHopPath(X,Y,K,Trail) :-
  schemaEdge(X,Z,_), not(member(Z,Trail)),
  schemaKHopPath(Z,Y,K1,[X|Trail]), K is K1 + 1.
```

Listing 3: Example of constraint mining rule for the graph schema.

rewritings. As an example, consider the process of enumerating valid k-hop connector views. Without any query constraints, the number of such views equals the number of k-length paths over the *schema graph*, which has M *schema edges*. While there exists no closed formula for this combination, when the schema graph has one or more cycles (e.g., a schema edge that connects a schema vertex to itself), at least M^k k-hop schema paths can be enumerated. This space is what the `schemaKHopPath` schema constraint mining rule would search over, were it not used in conjunction with query

```
% k-hop connector between nodes X and Y.
kHopConnector(X, Y, XTYPE, YTYPE, K) :-
  % query constraints
  queryVertexType(X, XTYPE), queryVertexType(Y, YTYPE),
  queryKHopPath(X, Y, K),
  % schema constraints
  schemaKHopPath(XTYPE, TYPE, K).

% k-hop connector where all vertices are of the same type.
kHopConnectorSameVertexType(X, Y, VTYPE, K) :-
  kHopConnector(X, Y, VTYPE, VTYPE, K).

% Variable-length connector where all vertices are of
% the same type.
connectorSameVertexType(X, Y, VTYPE) :-
  % query constraints
  queryVertexType(X, VTYPE), queryVertexType(Y, VTYPE),
  queryPath(X, Y),
  % schema constraints
  schemaPath(X, Y).

% Source-to-sink variable-length connector.
sourceToSinkConnector(X, Y) :-
  % query constraints
  queryVertexSource(X), queryVertexSink(Y), queryPath(X, Y),
  % schema constraints
  schemaPath(X, Y).
```

Listing 4: Example view template definitions for connectors.

constraint mining rules on view template definitions. KASKADE's *constraint-based view inference* algorithm enumerates a significantly smaller number of views for input queries, as the additional constraints enable its inference engine to efficiently search by early-stopping on branches that do not yield feasible rewritings.

4.6.2 Inference-based View Enumeration

As shown in Figure 4-4, KASKADE's view enumeration takes as input (i) a query, (ii) a graph schema, and (iii) a set of view templates. As described in §4.6.1, the query and schema are used to mine explicit and implicit constraints. Both the view templates and the mined constraints are rules that are passed to an inference engine to generate candidate views via rule evaluation. Hence, we call this process *inference-based view enumeration.* The view templates drive the search for candidate views, whereas the constraints restrict the search space of considered views. This process outputs a collection of instantiated candidate view templates, which are later used either by the workload analyzer module (see Figure 4-2) when determining which views to materialize during view selection (§4.7.2), or by the query rewriter to rewrite the query using the materialized view (§4.7.3) that will lead to its most efficient evaluation.

Listing 4 shows examples of template definitions for connector views. Each view template is defined as a Prolog rule and corresponds to a type of connector view. For instance, `kHopConnector(X,Y,XTYPE,YTYPE,K)` corresponds to a connector of length `K` between nodes `X` and `Y`, with types `XTYPE` and `YTYPE`, respectively. An example instantiation of this template is `kHopConnector(X,Y,‘Job’,‘Job’,2)`, which corresponds to a concrete *job-to-job* 2-hop connector view. This view can be translated to an equivalent Cypher query, which will be used either to materialize the view or to rewrite the query using this view, as we explain in §4.7. Other templates in the listing are used to produce connectors between nodes of the same type or source-to-sink connectors. Additional view templates can be defined in a similar fashion, such as the ones for summarizer views that we provide in Listing 2.

Note that the body of the view template is defined using the query and schema constraints, either the explicit ones (e.g., `queryVertexType`) coming directly from the query or schema, or via the constraint mining rules (e.g., `queryKHopPath`, `schema-KHopPath`), as discussed in §4.6.1. For example, `kHopConnector`'s body includes two explicit constraints to check the type of the two nodes participating in the connector, a query constraint mining rule that will check whether there is a valid k-hop path between the two nodes in the query, and a schema constraint mining rule that will do the same check on the schema.

Having gathered all query and schema facts, KASKADE's view enumeration algorithm performs the actual view candidate generation. In particular, it calls the inference engine for every view template. As an example, assuming an upper bound of $k = 10$ — in Listing 1 we have a variable-length path of at most 8 hops between 2 *File* vertices, and each of these two vertices is an endpoint for another edge — the following are valid instantiations of the `kHopConnector(X,Y,XTYPE,YTYPE,K)` view template for query vertices `q_j1` and `q_j2` (the only vertices projected out of the `MATCH` clause):

```
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=2)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=4)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=6)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=8)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=10)
```

Or, in other words, the unification `(X=‘q_j1’, Y=‘q_j2’, XTYPE=‘Job’, XTYPE=‘Job’, K=2)` for the view template `kHopConnector(X,Y,XTYPE,YTYPE,K)` implies that a view where each edge contracts a path of length $k = 2$ between two nodes of type *Job* is feasible for the query in Listing 1.

Similarly, KASKADE generates candidates for the remaining templates of Listings 4 and the additional example templates at [25]. For each candidate it generates, KASKADE's inference engine also outputs a rewriting of the given query that uses this candidate view, which is crucial for the view-based query rewriting, as we show in §4.7.3. Finally, each candidate view incurs different costs (see §4.7.1), and not every view is necessarily materialized or chosen for a rewrite.

4.7 View Operations

In this section, we present the main two operations that KASKADE supports: selecting views for materialization given a set of queries (§4.7.2) and view-based rewriting of a query given a set of pre-materialized views (§4.7.3). To do this, KASKADE uses a cost model to estimate the size and cost of views, which we describe next (§4.7.1).

4.7.1 View Size Estimation and Cost Model

While some techniques for relational cost-based query optimization may at times target filters and aggregates, most efforts in this area have primarily focused on join cardinality estimation [28]. This is in part because joins tend to dominate query costs, but also because estimating join cardinalities is a harder problem than, e.g., estimating the cardinality of filters. Furthermore, KASKADE can leverage existing techniques for cardinality estimation of filters and aggregates in relational query optimization for summarizers.

Therefore, here we detail our cost model contribution as it relates to connector views, which can be seen as the graph counterpart of relational joins. We first describe how we estimate the size of connector views, which we then use to define our cost model.

Graph data properties During initial data loading and subsequent data updates, KASKADE maintains the following graph properties: (i) *vertex cardinality* for each vertex type of the raw graph; and (ii) *coarse-grained out-degree distribution summary statistics*, i.e., the 50th, 90th, and 95th out-degree for each vertex type of the raw graph. KASKADE uses these properties to estimate the size of a view.

View size estimation. Estimating the size of a view is essential for the KASKADE's view operations. First, it allows us to determine if a view will fit in a space budget during view selection. Moreover, our cost components (see below) use view size estimates to assess the benefit of a view both in view selection and view-based query rewriting.

We estimate the size of a view as the number of edges that it has when materialized since the number of edges usually dominates the number of vertices in real-world graphs. Next, we observe that the number of edges in a k-hop connector over a graph G equals the number of k-length simple paths in G. Therefore, for a directed graph G that is homogeneous (i.e., has only one type of vertex, and one type of edge), we define the following estimator for its number of k-length paths:

$$\hat{E}(G, k, \alpha) = n \cdot deg_{\alpha}^{k} \tag{4.1}$$

where $n = |V|$ is the number of vertices in G, and deg_α is the α-th percentile out-degree of vertices in G ($0 < \alpha \leq 100$). For a directed graph G that is heterogeneous (i.e., has more than one type of vertex and/or more than one type of edge), an estimator for its number of k-length paths is as follows:

$$\hat{E}(G, k, \alpha) = \sum_{t \in T_G} n_t \cdot (deg_\alpha(n_i))^k \tag{4.2}$$

where T_G is the set of types of vertices in G that are edge sources (i.e., are the domain of at least one type of edge), n_t is the number of vertices of type $t \in T_G$, and $deg_\alpha(n_t)$ is the maximum out-degree of vertices of type $t \in T_G$.

Observe that if $\alpha = 100$, then deg_α is the maximum out-degree and the estimators above are upper bounds on the number of k-length paths in the graph. This is because there are n possible starting vertices, each vertex in the path has at most deg_{100} (or $deg_{100}(n_i)$) neighbors to choose from for its successor vertex in the path, and such a choice has to be made k times in a k-length path. In practice, we found that $\alpha = 100$ gives a very loose upper bound, whereas $50 \leq \alpha \leq 95$ gives a much more accurate estimate depending on the degree distribution of the graph. We present experiments on the accuracy of our estimator in §5.6.

View creation cost. The creation cost of a graph view refers to any computational and I/O costs incurred when computing and materializing the views' results. Since the primitives required for computing and materializing the results of the graph views that we are interested in are relatively simple, the I/O cost dominates computational costs, and thus the latter is omitted from our cost model. Hence, the view creation cost is directly proportional to $\hat{E}(G, k, \alpha)$.

Query evaluation cost. The cost of evaluating a query q, denoted $EvalCost(q)$, is required both in the view selection and the query rewriting process. KASKADE relies on an existing cost model for graph database queries as a proxy for the cost to compute a given query using the raw graph. In particular, it leverages Neo4j's [6] cost-based optimizer, which establishes a reasonable ordering between all vertex scans without indexes, scans from indexes, and range scans. For future work, we plan to incorporate our findings from graph view size estimation to further improve the query evaluation cost model.

4.7.2 View Selection

Given a query workload, KASKADE's view selection process determines the most effective views to materialize for answering the workload under the space budget that KASKADE allocates for materializing views. This operation is performed by the workload analyzer component, in conjunction with the view enumerator (see §4.6). The goal of KASKADE's view selection algorithm is to select the views that lead to the most significant performance gains relative to their cost, while respecting the space budget.

To this end, we formulate the view selection algorithm as a 0-1 knapsack problem, where the size of the knapsack is the space budget dedicated to view materialization.[5] The items that we want to fit in the knapsack are the candidate views generated by the view enumerator (§4.6). The weight of each item is the view's estimated size (see §4.7.1), while the value of each item is the performance improvement achieved by using that view divided by the view's creation cost (to penalize views that are expensive to materialize). We define the performance improvement of a view v for a query q in the query workload Q as q's evaluation cost divided by the cost of evaluating the rewritten version of q that uses v. The performance improvement of v for Q is the sum of v's improvement for each query in Q (which is zero for the queries for which v is not applicable). Note that we can extend the above formulation by adding weights to the value of each query to reflect its relative importance (e.g., based on the query's frequency to prioritize frequent queries, or on the query's estimated execution time to prioritize expensive queries).

The views that the above process selects for materialization are instantiations of Prolog view templates output by the view enumeration (see §4.6). KASKADE's

[5]The space budget is typically a percentage of the machine's main memory size, given that we are using a main memory execution engine. Our formulation can be easily extended to support multiple levels of memory hierarchy.

workload analyzer translates those views to Cypher and executes them against the graph to perform the actual materialization. As a byproduct of this process, each combination of query q and materialized view v is accompanied by a rewriting of q using v. In other words, a pair of candidate view and query rewriting are both inferred *and* costed together during inference-based view enumeration. This does not imply, however, that all possible candidate views and rewriting pairs are enumerated; rather, only the rewritings that are feasible given the explicit and implicit mined constraints. Further, we also note that iteratively rewriting the query as part of view enumeration is a commonly accepted approach, with significant advantages over query rewriting done only after enumeration [40, 87]. This is crucial in the view-based query rewriting, described next. The rewriter component converts the rewriting in Prolog to Cypher, so that KASKADE can run it on its graph execution engine.

4.7.3 View-Based Query Rewriting

Given a query and a set of materialized views, view-based rewriting is the process of finding the rewriting of the query that leads to the highest reduction of its evaluation cost by using (a subset of) the views. In KASKADE, the query rewriter module (see Figure 4-2) performs this operation.

When a query q arrives in the system, the query rewriter invokes the view enumerator, which generates the possible view candidates for q, pruning those that it has not materialized. Among the views that are output by the enumerator, the query rewriter selects the one that, when used to rewrite q, leads to the smallest evaluation cost for q. As discussed in §4.6, the view enumerator outputs the rewriting of each query based on the candidate views for that query. If this information is saved from the view selection step (which is true in our implementation), we can leverage it to choose the most efficient view-based rewriting of the query without having to invoke the view enumeration again for that query. As we mentioned in §4.6, KASKADE currently supports rewritings that rely on a single view. Combining multiple views in a single rewriting is left as future work.

Listing 5 shows the rewritten version of our example query of Listing 1 that uses a

2-hop connector ("job-to-job") graph view.

```
SELECT A.pipelineName, AVG(T_CPU) FROM (
  SELECT A, SUM(B.CPU) AS T_CPU FROM (
    MATCH (q_j1:Job)-[:2_HOP-JOB_TO_JOB*1..4]->(q_j2:Job)
    RETURN q_j1 as A, q_j2 as B
  ) GROUP BY A, B
) GROUP BY A.pipelineName
```

Listing 5: Job blast radius query rewritten over a 2-hop connector (*job-to-job*) graph view.

4.8 Experimental Evaluation

In this section, we experimentally confirm that by leveraging graph views (e.g., summarizers and connectors) we can: (i) accurately estimate graph view sizes—§4.8.4; (ii) effectively reduce the graph size our queries operate on—§4.8.5; and (iii) improve query performance—§4.8.6. We start by providing details on Kaskade's current implementation, and follow by introducing our datasets and query workload.

4.8.1 Implementation

We have implemented Kaskade's components (see Figure 4-2) as follows. The *view enumerator* component (§4.6) uses SWI-Prolog [11] as its inference engine. We wrote all constraint mining rules, query constraint mining rules, and view templates using SWI-Prolog syntax [12]. We wrote the knapsack problem formulation (§4.7.2) in Python 2.7, using the branch-and-bound knapsack solver from Google OR tools combinatory optimization library [3]. As shown in Figure 4-2, Kaskade uses Neo4J (version 3.2.2) for storage of raw graphs, materialized graph views, and query execution of graph pattern matching queries. All other components were written in Java.

For the experimental results below, Kaskade extracted schema constraints only once for each workload, as these do not change throughout the workload. Furthermore, Kaskade extracted query constraints as part of view inference only the first time we

Table 4.2: Networks used for evaluation: `prov` and `dblp` are heterogeneous, while `roadnet-usa` and `soc-livejournal` are homogeneous and have one edge type.

Short Name	**Type**	$\|V\|$	$\|E\|$
prov (raw)	Data lineage	3.2B	16.4B
prov (summarized)	Data lineage	7M	34M
dblp-net	Publications [5]	5.1M	24.7M
soc-livejournal	Social network [9]	4.8M	68.9M
roadnet-usa	Road network [7]	23.9M	28.8M

entered the query into the system. This process introduces a few milliseconds to the total query runtime and is amortized for multiple runs of the same query.

4.8.2 Datasets

In our evaluation of different aspects of KASKADE, we use a combination of publicly-available heterogeneous and homogeneous networks and a large provenance graph from Microsoft (heterogeneous network). Table 4.2 lists the datasets and their raw sizes.

For our size reduction evaluation in Section 4.8.5, we focus on gains provided by different graph views over the two heterogeneous networks: `dblp-net` and a data lineage graph from Microsoft. The `dblp-net` graph, which is publicly available at [5], contains 5.1M vertices (authors, articles, and venues) with 24.7M with 2.2G on-disk footprint. For the second heterogeneous network, we captured a provenance graph modeling one of Microsoft's large production clusters for a week. This raw graph contains 3.2B vertices modeling jobs, files, machines, and tasks, and 16.4B edges representing relationships among these entities, such as job-read-file, or task-to-task data transfers. The on-disk footprint of this data is in the order of 10+ TBs.

After showing size reduction achieved by summarizers and connectors in Section 4.8.5, for our query runtime experiments (§4.8.6), we consider already summarized versions of our heterogeneous networks, i.e., of `dblp-net` and provenance graph. The summarized graph over `dblp-net` contains only authors and publications ("article", "publication", and "in-proc" vertex types), totaling 3.2M vertices and 19.3M edges, which requires 1.3G on disk. The summarized graph over the raw provenance graph contains only jobs and files and their relationships, which make up its 7M vertices and 34M edges. The on-disk footprint of this data is 4.8 GBs. This allows us to compare runtimes of our queries on the Neo4j 3.2.2 graph engine, running on a 128 GB of RAM and 28 Intel Xeon 2.40GHz cores, 4 x 1TB SSD Ubuntu box. We chose to use Neo4j for storage of materialized views and query execution because it is the most widely used graph database engine as of writing, but our techniques are graph query engine-agnostic.

4.8.3 Queries

We use the following 8 queries in our evaluation of query runtimes. Queries Q1 through Q3 are motivated by telemetry use cases at cloud computing infrastructure, and are defined as follows. The first query retrieves the job blast radius, up to 8 hops away, of all job vertices in the graph, together with their average CPU consumption property. Query Q2 retrieves the ancestors of a job (i.e., all vertices in its *backward data lineage*) up to 4 hops away, for all job vertices in the graph. Conversely, Q3 does the equivalent operation for *forward data lineage* for all vertices in the graph, also capped at 4 hops from the source vertex. Both Q2 and Q3 are also adapted for the other 3 graph datasets: on `dblp`, the source vertex type is "author" instead of "job", and on homogeneous networks `roadnet-usa` and `soc-livejournal` all vertices are included.

Next, queries Q4 through Q7 capture graph operation primitives which are commonly required for tasks in dependency driven analytics [66]. The first, query Q4 ("path lengths"), computes a weighted distance from a source vertex to all other vertices in its forward k-hop neighborhood, limited to 4 hops. For each vertex in the 4-hop neighborhood, it performs an aggregation operation (max) over a data property (edge timestamp) of all edges in the 4-hop path. Queries Q5 and Q6 both measure the overall size of the graph (edge count and vertex count, respectively).

Finally, Q7 ("community detection") and Q8 ("largest community") are representative of common graph analytics tasks. The former runs a 25 passes iterative version

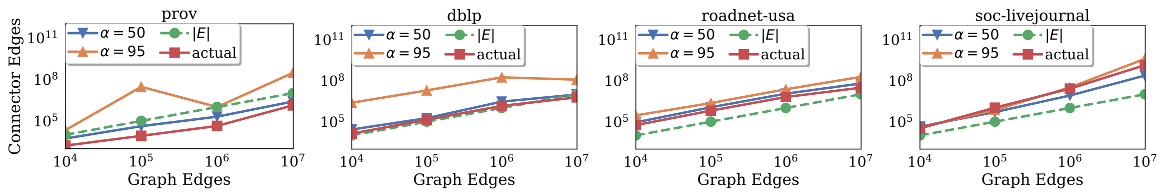


Figure 4-5: Estimated, actual, and original graph sizes for 2-hop connector views over different datasets. Here we show estimates for two upper bound variations derived from summary statistics over the graph's degree distribution detailed in §4.7.1. We also plot the original graph size (x-axis, and dashed $|E|$ series). Plots are in log-log scale.

of community detection algorithm via label-propagation, updating a "community" property on all vertices and edges in the graph, while Q8 uses the community label produced by Q7 to retrieve the largest community in terms of graph size as measured by number of "job" vertices in each community. The label propagation algorithm used by Q7 is part of the APOC collection of graph algorithm UDFs for Neo4j [2].

For query runtimes experiments in §4.8.6, we use the equivalent rewriting of each of these queries over a 2-hop connector. Specifically, queries Q1 through Q4 go over half of the original number of hops, and queries Q7 and Q8 run around half as many iterations of label propagation. These rewritings are equivalent and produce the same results as queries Q1 through Q4 over the original graph, and similar groupings of "job" nodes in the resulting communities. Queries Q5 and Q6 need not be modified, as they only count the number of elements in the dataset (edges and vertices, respectively).

4.8.4 View Size Estimation

In this section, we evaluate the accuracy of KASKADE's view size estimators (§4.7.1). Figure 4-5 shows our results for different heuristics estimators on the size of a 2-hop connector view materialized over the first n edges of each public graph dataset. We focus on size estimates for 2-hop connectors since, similar to cardinality estimation for joins, the larger the k, the less accurate our estimator. We do not report results for view size estimates for summarizers, as for these KASKADE can leverage traditional cardinality estimation based on predicate selectivity for filters, as well as multidimensional histograms for predicate selectivity of group-by operators [28].

We found that k-hop connectors in homogeneous networks are usually larger than the original graph in real-world networks. Further, we find that there is no direct relationship between the size of a graph schema, and the expected size of a k-hop view. This is because k-length paths can exist between any two vertices in a homogeneous graph (smaller schema), as opposed to only between specific types of vertices in heterogeneous networks (larger schema), such as Microsoft's data lineage graph. Note that the $\alpha = 50$ line does a good job of approximating the size of the graph as the number of edges grows. Also, for networks with a degree distribution close to a power-law, such as `soc-livejournal`, the estimator that relies on 95th percentile out-degree ($\alpha = 95$) provides an upper bound, while the one that uses the median out-degree ($\alpha = 50$) of the network provides a lower bound. On other networks that lack a power-law degree distribution, such as `road-net-usa`, the median out-degree estimator better approximates an upper bound on the size of the k-hop connector.

In practice, Kaskade relies on the estimator with $\alpha = 95$ as it provides an upper bound for most real-world graphs that we have observed. Note that the estimator with $\alpha = 95$ for `prov` decreases when the original graph increases in size from $100K$ to $1M$ edges, increasing again at $10M$ edges. This is due to a decrease in the percentage of "job" vertices with a large out-degree, shifting the 95th percentile to a lower value. This percentile remains the same at $10M$ edges, while the 95th out-degree for "file" vertices increases at both $1M$ and $10M$ edges.

4.8.5 Size Reduction

This experiment shows how by applying summarizers and connectors over heterogeneous graphs we can reduce the effective graph size for one or more queries. Figure 4-6 shows that for co-authorship queries over the `dblp`, and query Q1 over the provenance graph, the schema-level summarizer yields up to three orders of magnitude reduction. The connector yields another two orders of magnitude data reduction by summarizing the job-file-job paths in the provenance graph, and one order of magnitude by summarizing the author-publication-author paths in the `dblp` graph.

Besides the expected performance advantage since queries operate on less data, such a drastic data reduction allows us to benefit from single-machine in-memory technologies (such as Neo4j) instead of slower distributed on-disk alternatives for query evaluation, in the case of the provenance graph. While this is practically very important, we do not claim this as part of our performance advantage, and all experiments are shown as relative speed-ups against a baseline on this reduced provenance graph on the same underlying graph engine.

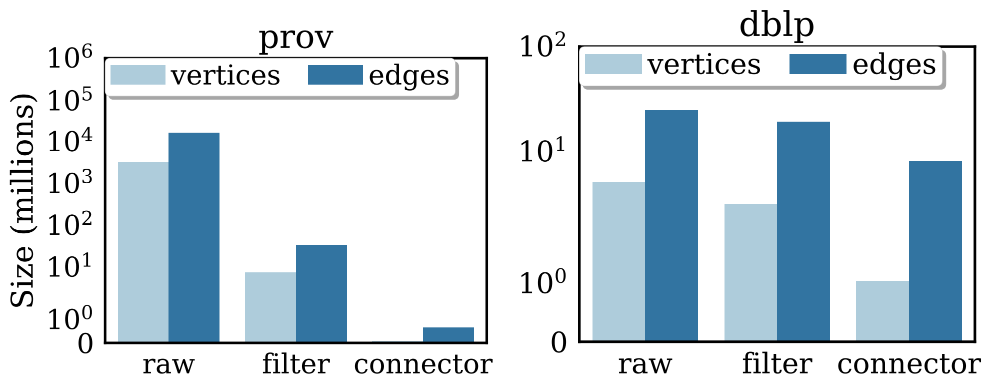


Figure 4-6: Effective graph size reduction when using summarizer and 2-hop connector views over `prov` and `dblp` heterogeneous networks (y-axis is in log scale).

4.8.6 Query Runtimes

This experiment measures the difference in total query runtime for queries when executed from scratch over the filtered graph versus over an equivalent connector view on the filtered graph. Figure 4-7 shows our results, with runtimes averaged over 10 runs, plotted in log scale on the y-axis. Because the amount of data required to answer the rewritten query is up to orders of magnitude smaller (§4.8.5) in the heterogeneous networks, virtually every query over the `prov` and `dblp` graphs benefit from executing over the connector view. Specifically, Q2 and Q3 have the least performance gains (less than 2 times faster), while Q4 and Q8 obtain the largest runtime speedups (13 and 50 times faster, respectively). This is expected: Q2 ("ancestors") and Q3 ("descendants") explore a k-hop ego-centric neighborhood in both the filtered graph and in the connector view that is of the same order of magnitude. Q4 ("path lengths")

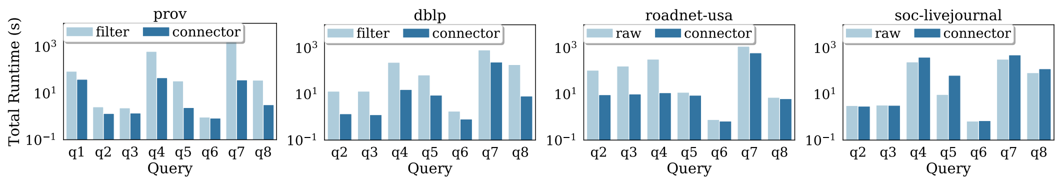


Figure 4-7: Total query execution runtimes over the graph after applying a summarizer view ("filter"), and rewritten over a 2-hop connector view. Connectors are *job-to-job* (`prov`), *author-to-author* (`dblp`), and *vertex-to-vertex* for homogeneous networks `roadnet-usa` and `soc-livejournal` (y-axis is in log scale).

and Q8 ("community detection"), on the other hand, are queries that directly benefit from the significant size reduction of the input graph. In particular, the maximum number of paths in a graph can be exponential on the number of vertices and edges, which affects both the count of path lengths that Q4 performs, as well as the label-propagation updates that Q8 requires. Lastly, KASKADE creates views through graph transformations that are engine-agnostic, hence these gains should be available in other systems as well.

For the homogeneous networks, we look at the total query runtimes over the raw graph and over a *vertex-to-vertex* materialized 2-hop connector, which may be up to orders of magnitude larger than the raw graph (§4.8.4), as these networks have only one type of edge. Despite these differences in total sizes, a non-intuitive result is that the total query runtime is still linearly correlated with the order of magnitude increase in size for the network with a power-law degree distribution (`soc-livejournal`), while it decreases for path-related queries in the case of `roadnet-usa`. This is due to a combination of factors, including the larger fraction of long paths in `roadnet-usa`. Lastly, while the decision on which views to materialize heavily depends on the estimated view size (e.g., size budget constraints, and proxy for view creation cost), these 2-hop connector views are unlikely to be materialized for the two homogeneous networks, due to the large view sizes predicted by our cost model (Figure 4-5).

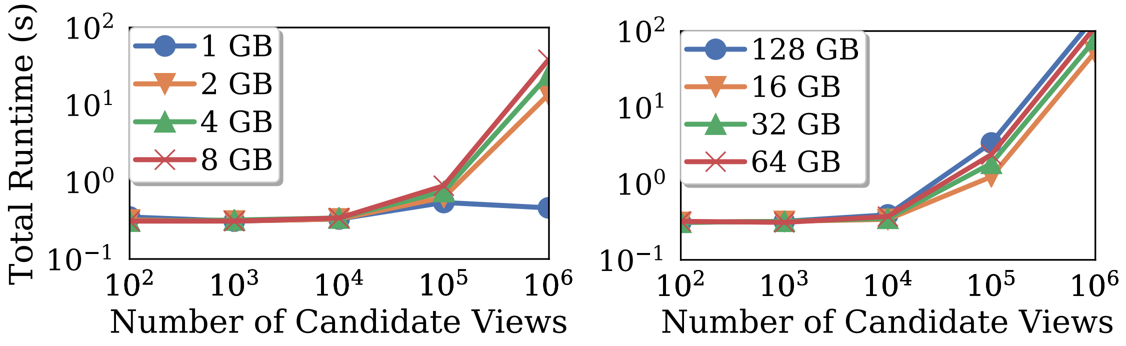


Figure 4-8: View selection runtime vs. number of views (x-axis). Here we show runtimes for eight different space budgets using the knapsack formulation detailed in 4.7.2. Plots are in log-log scale.

4.8.7 View Selection Runtimes

This experiment measures the runtime of our view selection algorithm, which identifies the most useful views to materialize subject to a space budget. As described in 4.7.2, our algorithm is formulated as a $0-1$ knapsack problem. Figure 4-8 shows our results for different sets of uniformly distributed view weights and values, with runtimes averaged over 10 runs, plotted in log-log scale. Aside from the 1 GB budget case (where the solver finishes more quickly due to more views having weight larger than the budget) all runtimes scale superlinearly on number of views. We believe this is largely due to a core component in our selection algorithm being currently single-threaded [3]. Nonetheless, our results with the single-threaded implementation show that even for a large set of candidate views (e.g., 10^5), KASKADE can select the most useful views for materialization in less than a few seconds. It is possible to further reduce the runtime by splitting the knapsack problem into multiple parallel subproblems, or to parallelize the solver implementation, which we leave as future work.

Chapter 5

Kairos

5.1 Introduction

The growing demand for temporal graph applications has given rise to new challenges in temporal graph analytics. As an increasing number of real-world systems and processes can be modeled as temporal graphs, the need for effective analysis tools and techniques has become more pressing. These applications range from social networks and communication systems to transportation networks and biological systems, where understanding the temporal dynamics is crucial for uncovering meaningful insights and patterns [45, 96, 53, 47].

Temporal graphs offer a unique perspective as they can capture the dynamics of interactions and relationships over time, which non-temporal graphs are unable to provide. Furthermore, temporal graphs lend themselves to the exploration of time-ordered events and the impact of such sequences in the network, creating an opportunity for more nuanced insights. For instance, being able to track the chronology of friendship formations on a social network or communication events can enhance our understanding of behavioral patterns. Similarly, the ability to observe the evolution of transportation routes or the progression of a biological system over time can provide critical data for predictive modeling and decision-making.

However, the temporal dynamics inherent in these graphs also brings about new challenges. Existing graph frameworks and query systems frequently encounter diffi-

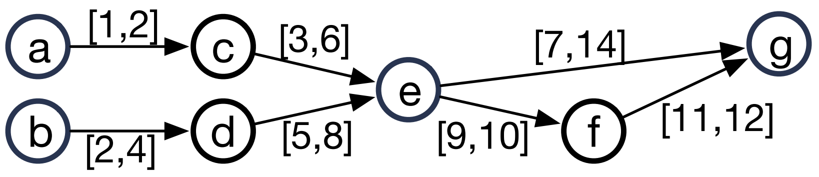


Figure 5-1: An example temporal graph representing vertices $\{a, b, c, d, e, f, g\}$. Each edge is associated with a time interval (start, end) denoting its validity.

culties when handling graph processing tasks required in temporal graph analytics applications. The underlying reasons for these challenges are twofold. First, many of these systems were primarily designed for traditional graph processing, and as a result, they are not well-equipped to handle the unique characteristics and requirements of temporal graphs. This shortcoming leads to suboptimal performance and a limited ability to fully leverage the available temporal information in the data. Second, some systems [80, 59, 77] that specifically target temporal graph processing rely on Pregel-like distributed computation models. These models can be highly inefficient due to the message passing overhead across servers in a cluster, especially when the input graph fits comfortably within the memory resources of a single commodity machine. This limitation leads to suboptimal performance and an inability to fully exploit the temporal information available in the data. Finally, existing shared-memory temporal graph processing systems [46] do not have these limitations, but still rely on expensive pre-processing of the graph that increases the original size of the dataset – a step which is not necessary for correctness of the target algorithms.

Temporal graph applications typically require querying small time slices of data. While existing systems can represent time as an attribute of nodes and edges, filtering by time necessitates either an expensive scan or the use of a range index, resulting in poor performance. Furthermore, traditional graph processing frameworks lack support for temporal algorithms, leading to complex and costly implementations as they require additional programming effort.

Temporal graphs also exhibit unique properties that are not necessarily considered in existing systems. For example, they tend to be large due to the increased number of attributes and edges, resulting from the preservation of interactions between vertices over time. Additionally, temporal graphs exhibit skewed data distributions, both in terms of degree distributions and their evolution over time [56, 45]. To address these challenges, temporal graph analytics systems must provide interactive response times for various use cases, such as operational decisions, contact tracing, and routing.

Our Approach and Contributions. In response to the challenges described above, we have developed KAIROS, an in-memory temporal graph analytics system that leverages a highly-optimized parallel data structure for to efficiently execute temporal algorithms over temporal graphs on multi-core machines. KAIROS is designed to address the unique properties and requirements of temporal graphs, providing an efficient solution for processing large-scale temporal graphs. Specifically, it introduces programming primitives and APIs that organize temporal graphs by time, as well as a novel Temporal Graph Index designed for efficiently looking up data in specific time ranges. This approach enables efficient temporal graph algorithm implementations, treating temporal edge information as first-class citizens in the model. The system also incorporates selective indexing, a query optimization technique that relies on a novel cost model to speed up query execution by choosing the best access method for neighbors of a given vertex at runtime.

We have implemented a number of parallel temporal graph algorithms for various application classes, including single-source shortest paths (earliest arrival, latest departure, fastest, and shortest duration), connectivity (temporal connected components), and centrality (temporal betweenness centrality). Our system provides significant performance improvements compared to existing state-of-the-art temporal graph processing systems. We make the following contributions:

I. We present TGER, a novel "time-first" data structure that acts as an index to enable efficient processing of temporal graph queries and algorithms.

II. We introduce *selective indexing*, a technique that speeds up query execution by choosing the best access method for retrieving neighbors of a given vertex at runtime.

III. We present efficient shared-memory parallel algorithm implementations for various temporal graph application classes, offering significant performance improvements compared to existing state-of-the-art temporal graph processing systems.

IV. Our results show substantial speedup compared to existing systems and provide insights into the performance characteristics and guarantees of our system.

5.2 Problem Formulation

The primary challenge in designing a temporal graph analytics system is to efficiently process and analyze temporal graphs while taking into account the unique characteristics of such graphs, such as time-varying edges and vertices. The problem can be stated as follows: given a temporal graph $G = (V, E, T, \tau)$, our goal is to efficiently support the execution of temporal graph analytics tasks, including temporal paths, temporal connectivity, and temporal centrality.

The example queries and algorithms discussed above require different strategies for selecting, at runtime, the appropriate set of vertices and temporal edges to be explored in the frontier being traversed by the graph processing engine. The efficiency of these strategies is crucial for the performance of temporal graph analytics algorithms. In this paper, we aim to address this challenge by introducing selective indexing and other design decisions that help significantly improve the performance of executing different tasks over an input temporal graph. To address this problem, we must fulfill the following desiderata:

1. Efficient storage and indexing of temporal graph data, especially as it pertains to temporal edges.

2. Design and implementation of efficient parallel algorithms for temporal graph analytics tasks, such as temporal paths, temporal connectivity, and temporal centrality.

3. Effective runtime selection of the edges to be explored in the frontier to minimize the time and space complexity of the operations involved in the analytics tasks.

4. Provide a flexible and easy-to-use programming model to support the implementation of various temporal graph analytics tasks and queries.

By addressing these aspects, our goal with KAIROS is to provide an efficient temporal graph analytics system that can handle large-scale temporal graphs and deliver appropriate programming primitives for writing temporal graph applications.

5.3 Architectural Overview and Key Ideas

KAIROS provides an API that is compatible with that of a state-of-the-art high-performance graph processing system ([84]), and extends it to the temporal setting. The API provides methods to load graphs, perform operations such as traversal, filtering, and allows computations to be expressed in terms of an input temporal query.

In the subsequent sections, we will delve deeper into the key ideas behind our approach, as outlined in KAIROS's architecture diagram (Figure 5-2). We introduce KAIROS's core data structures, the need to choose between different access methods, as well as the types of information needed to make this decision. First, we describe a high-specialized parallel data structure for indexing temporal edges (§5.3.1). Second, we introduce novel approach to *selectively* decide which subset of vertices are benefitial to index, as well as how to access each vertex at runtime (§5.3.2) and associated cost model.

5.3.1 Temporal Graph Edge Registry (Key Idea A)

Temporal graphs introduce an additional layer of complexity in graph computation due to the addition of time as a variable. To process queries and algorithms over temporal graphs, one could consider a naive approach based on CSR (§2.6). With this approach, the entire list of edges associated to a given vertex is stored in the corresponding offset of the CSR, and then a parallel filter is applied to retrieve neighbors satisfying an input temporal predicate. This can become expensive, however, particularly in graphs with high-degree vertices, or when the output of this filtering is much smaller than the

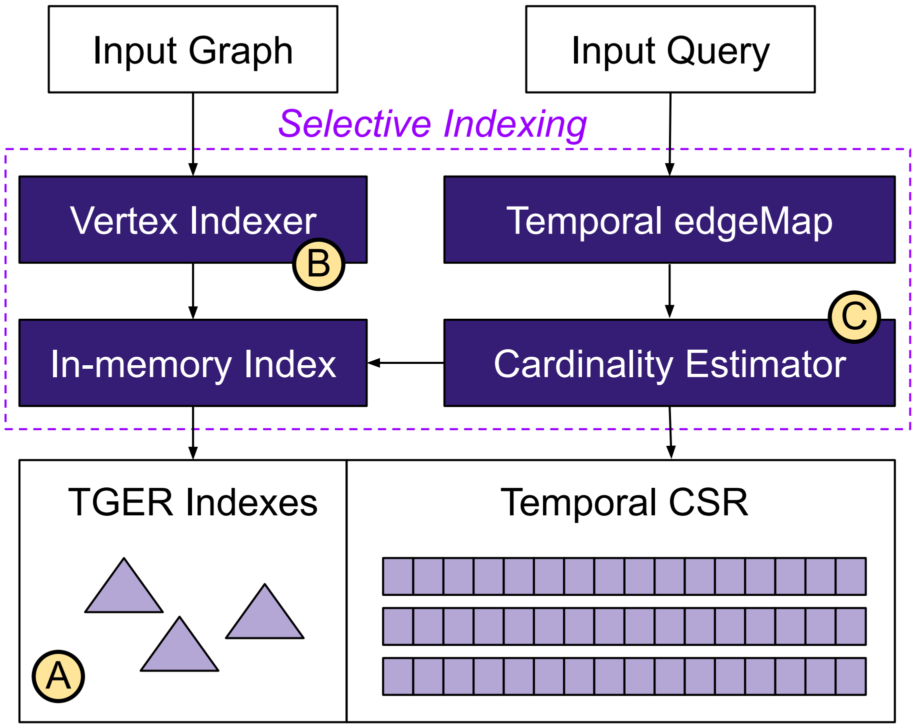
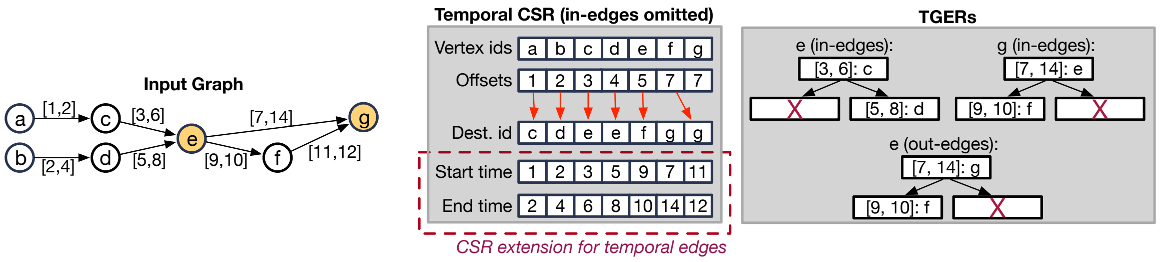


Figure 5-2: Architectural overview and key ideas.



Vertex ids	a	b	c	d	e	f	g
Offsets	1	2	3	4	5	7	7
Dest. id	c	d	e	e	f	g	g
Start time	1	2	3	5	9	7	11
End time	2	4	6	8	10	14	12

Figure 5-3: Data layout for example temporal graph in Figure 5-1. Temporal CSR is available for all vertices in the graph (incoming edges omitted for sake of clarity). In this example, TGER is used to index only the vertices with out or in-degree of at least two edges. In practice, KAIROS only indexes vertices with much larger degrees.

degree of the vertex. This can decrease performance, as a large number of edges that are not relevant to the specific input temporal predicate might need to be processed.

Therefore, the efficient processing of queries and temporal algorithms over temporal graphs demands specialized data structures for storing and retrieving vertex neighbors. To this end, we have designed and implemented TGER (Temporal Graph Edges

Registry), a parallel data structure based on priority search trees [16] for effectively processing queries and temporal algorithms over temporal graphs. TGER lets Kairos treat temporal edge information as first-class citizens in the data model, enabling efficient temporal graph algorithm implementations. Moreover, TGER is storage-efficient (i.e., $O(m)$ space, where m is the number of temporal edges stored in it) and can answer interval containment queries efficiently ($O(\log m + k)$ work, where k is the number of results [16]). Finally, TGER makes efficient use of computational resources, employing fork-join parallelism in its implementation of both construction and query operations. As we show in our experiments, TGER can provide a substantial advantage over the naive CSR-based approach.

5.3.2 Selective Indexing (Key Ideas B and C)

The efficient retrieval of vertices and edges of interest is crucial for the performance of algorithms and queries in large graphs. This is particularly true for large-scale real-world graphs, where data is often skewed, and a small number of vertices can account for most edges. The skewness is intensified in real-world temporal graphs, where data can also be unevenly distributed over time (e.g., seasonal data patterns, or growth in popularity in the case of social networks). Furthermore, the skew present in real-world temporal graphs is a well studied phenomena, and which can be attributed to inter-contact time distributions, burstiness, and even circadian or otherwise weekly rhythms that are inherent in human activity [45]. For this reason, different approaches have been proposed to generate temporal graphs that are closer to real-world skewed distributions [35, 92], and more recently data imputation [43] that can handle skew.

To address this challenge, we introduce a new class of problems in query optimization for graph queries, as well as a unique technique for *selectively* indexing different parts of a graph relevant to query answering. As part of our approach, we present a novel cost model and related algorithms for determining the most suitable access method (e.g., index-based vs. scan-based) for graph traversal. Our algorithms take into account the characteristics of the underlying data (e.g., data skew) and workload (e.g., selectivity of temporal predicates present in the queries) to choose the most efficient access method (index vs. scan) for each vertex at runtime. §5.5 (Selective Indexing) provides a description of our technique.

Vertex Indexer. This component is responsible for deciding which vertices warrant a TGER index. First, a vertex size is defined in terms of the size of their out-degree or in-degree neighborhood (i.e., how many outgoing or incoming edges it has). Based on a predefined vertex size threshold (heuristically obtained via experimental analysis), the indexer builds a TGER only for those vertices whose size exceeds this threshold. To quickly identify which vertices have a TGER, the Vertex Indexer maintains an in-memory sparse associative array where each entry maps a vertex id to the in-memory location of its corresponding TGER. At runtime, KAIROS employs a cost model (§5.5) to evaluate whether it should access the corresponding TGER for that vertex, or opt for a linear scan using Temporal CSR (T-CSR, §5.4.2).

Cardinality Estimator. We introduce a cost model for deciding at runtime which access method to use for retrieving a vertex's neighborhood. As part of this cost model, a cardinality estimator plays a key role in determining whether a query is selective enough to warrant index-based retrieval of outgoing/incoming edges via TGER. We further describe it and its associated algorithms in §5.5.1 and §5.5.2.

5.4 KAIROS Framework

This section describes the interface and implementation of KAIROS framework, with focus on how it extends Ligra [84] to the temporal setting. Table 5.1 summarizes its interface.

5.4.1 Interface

KAIROS contains two key data structures: ***VertexSet*** and ***TemporalEdgeSet***, which are used respectively to represent subset of vertices and subset of temporal edges. ***OrderingPredicate*** takes as input a pair of temporal edges, an ***OrderingPredicateType***, and evaluates whether the given temporal edges conform to that predicate (§5.4.3). Using these as input, ***TemporalGraph*** constructs the temporal graph. ***VertexMap***

Interface	**Description**
VertexSet TemporalEdgeSet	Represents a subset of vertices $V' \subseteq V$. Represents a subset of temporaledges $E' \subseteq E$.
ORDERINGPREDICATE(A: *temporaledge*, B: *temporaledge*, T: OrderingPredicateType): *bool*	Evaluates the order between two temporal edges based on one of the three ordering predicate types: Suceeds, StrictlySucceeds, or Overlaps (§5.4.1).
TEMPORALGRAPH(V: VertexSet, E: TemporalEdgeSet, P: OrderingPredicate)	Constructs a temporal graph using given input vertices, temporaledges, and ordering predicate.
VERTEXMAP(U: VertexSet, G: TemporalGraph, F: *vertex* $\rightarrow$ *bool*): VertexSet	Applies $F(u)$ for each $u \in U$; returns a VertexSet $\{u \in U \mid F(u) = true\}$.
TEMPORALEDGEMAP(U: TemporalEdgeSet, G: TemporalGraph, F: *temporaledge* $\rightarrow$ *bool*): TemporalEdgeSet	Applies $F(u)$ for each $u \in U$; returns a TemporalEdgeSet $\{u \in U \mid F(u) = true\}$.

Table 5.1: Core primitives from KAIROS framework programming model interface.

is equivalent to Ligra's (§2.7), with the only difference that it takes a ***TemporalGraph*** as input. Conversely, ***TemporalEdgeMap*** extends Ligra's ***EdgeMap*** to the temporal setting, and is described in more details in §5.4.4).

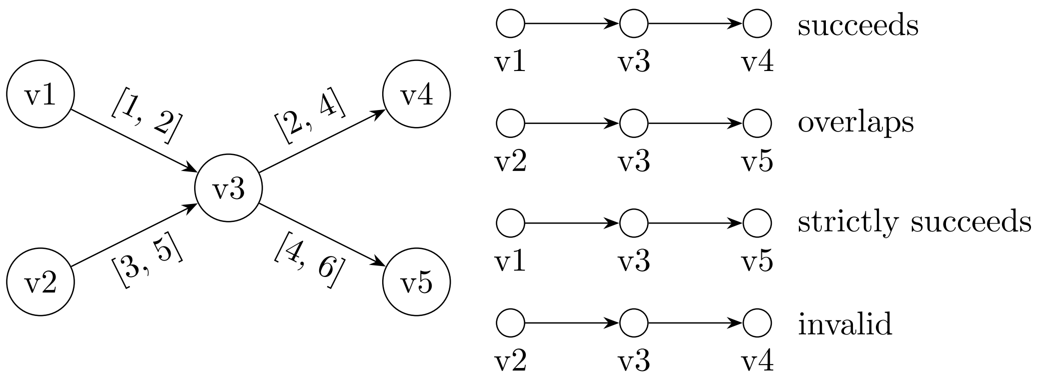


Figure 5-4: Ordering predicates for each of the four possible two-hop directed paths on an example graph.

5.4.2 T-CSR: Temporal CSR

The Temporal CSR (T-CSR) data structure is an extension of the traditional CSR representation (§2.6) designed to accommodate temporal graph data. It retains the core concepts of CSR, while incorporating additional arrays to store the start and end times associated with each temporal edge, as shown in Figure 5-3. Specifically, it extends the standard CSR representation by adding two more arrays, the start time array, and the end time array. As their name suggests, these arrays store the start and end times of each temporal edge in the graph, respectively. They are organized in the same order as the adjacency array, such that the start and end times of the temporal edge at position i in the adjacency array can be found at position i in the start / end time arrays.

With this extended representation, the T-CSR can store temporal graphs and support various temporal graph processing tasks. The additional start time and end time arrays enable the system to take into account the temporal information of the graph when executing queries and algorithms when used in conjunction with ***TemporalEdgeMap*** (§5.4.4). The T-CSR representation retains the advantages of the traditional CSR, such as cache efficiency and fast access to adjacency lists. Furthermore, it can be easily incorporated into existing CSR-based graph processing systems with minimal modifications, such as is the case for our extension of Ligra to the temporal setting. Overall, the Temporal CSR offers an effective and efficient solution for storing and processing temporal graph data. Because of that, it is the default representation used to store the neighbors of most vertices in KAIROS. A similar representation is also used in the Temporal GNN literature [103] for sampling temporal edges.

5.4.3 TGER: Temporal Graph Edges Registry

While T-CSR offers valuable improvements over traditional CSR for temporal graph data handling, its design is still geared towards adjacency-oriented queries and operations. For example, T-CSR does an excellent job when dealing with questions of "who is connected to whom". However, it is less efficient when it comes to handling range-based temporal queries such as "who was connected to whom within a specific time window". This is because in T-CSR, the temporal information is appended to the existing structure, which is space-optimized for traditional graphs (i.e., where edges do not have additional temporal information). As a result, each temporal query needs to scan over the entire adjacency list of a vertex and filter out the relevant edges based on their timestamps. While this operation can be performed in parallel, it can still be inefficient for large graphs, or for queries with small time windows (i.e., queries that are highly selective) relative to the total neighbors a vertex has.

To address this challenge, we propose TGER, an index data structure which accommodates temporal information (in the form of time intervals associated with edges) as a first-class citizen. TGER is a dual index, meaning it indexes both the start and end time attributes. Specifically, it combines a priority queue (by defaullt over the start time attribute) and a Binary Search tree (BST) (by default over the end time attribute). This allows queries with predicates on either start or end time to be answered efficiently. It is heavily inspired by Priority Search Trees (PST) [16], a data structure traditionally used for storing intervals or 2D points in the context of computational geometry. In TGER, all queries are 3-sided (Figure 5-5), meaning that only one bound is specified for one of the dimensions. This allows temporal edges to be queried in $O(\log m + k)$, where m is the number of temporal edges stored in it and k is the number of results from the query. Figure 5-3 shows a visual representation of TGER's data layout, contrasting it with T-CSR, and 2 shows the pseudocode for TGER's parallel build operation.

Ordering Predicates and 3-sided queries. A key aspect of TGER is that the application can specify which of the two dimensions of the intervals should be mapped to the priority queue (or "heap") axis, and which dimension should be mapped to the "BST" axis. In addition, TGER can also be configured as either a min-heap, or a max-heap. Given this flexibility, both Succeeds and StrictlySucceeds ordering predicates can be translated to an equivalent 3-sided query by flipping the axis and/or using a max-heap. The Overlaps ordering predicate, however, requires both ends of the interval of two subsequent edges to be checked against each other, as can be seen in Figure 5-4. In that case, ***TemporalEdgeMap*** (§5.4.4) performs one additional query to match in-neighbors with corresponding out-neighbors.

5.4.4 Temporal Edge Map

We introduce ***TemporalEdgeMap***, a programming primitive inspired by Ligra's ***EdgeMap*** (§2.7), and designed to efficiently handle temporal information during

Algorithm 2 Pseudocode for TGER index parallel build.

Input: A temporal graph $G = (V, E)$.
Output: If applicable, the TGER for each vertex in G.

```
1: procedure IndexVertices($G$)
2:     parallel for each $v \in V$ do
3:         $d$ = out-degree of $v$ // Omitted: in-neighbors processing.
4:         if $d \geq$ min cutoff then // (Section 5)
5:             out-index[$v$] = BuildIndex($e \in$ out-edges of $v$)
6: procedure BuildIndex($E$)
7:     sorted = parallel-sort(E, ByStartTime())
8:     return BuildIndexRecurse(sorted, 0, sorted.size())
9: procedure BuildIndexRecurse(sorted, start, end)
10:    size = end - start
11:    if size == 0 then return nullptr
12:    if size == 1 then return new TGERNode(sorted[start])
13:    TGERNode* root = new TGERNode(sorted[start])
14:    start = start + 1
15:    mid-idx = index of point with median "end time" in sorted
16:    root->BST-mid = sorted[mid-idx].end-time
17:    root->left = spawn BuildIndexRecurse(sorted, start, mid-idx)
18:    root->right = BuildIndexRecurse(sorted, mid-idx, end)
19:    sync
```

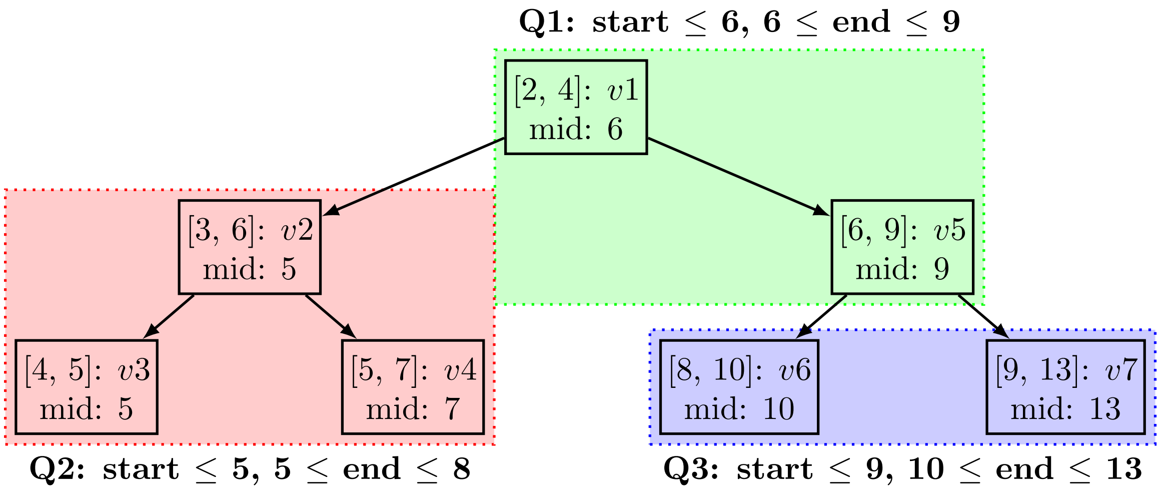


Figure 5-5: Queries in TGER are 3-sided. By default, an upper bound (*min-heap mode*) is specified for start time, and a lower plus upper bound is specified for end time.

parallel processing of temporal graphs. The TemporalEdgeMap extends Ligra's

Algorithm 3 Pseudocode for TGER index parallel query.

Input: An input 3-sided query with $(-\infty, \text{max-heap}] \times [\text{min-BST}, \text{max-BST})$. By default, TGER's heap dimension stores *start time*, and BST dimension stores *end time.* Alternatively representations are also supported, such as min-heap, or mapping *end time* to heap dimension and *start time* to BST dimension.

Output: Matching results from TGER index.

```
 1: procedure QUERY(max-heap, min-BST, max-BST)
 2:     n-parts = get_nworkers() // Number of cores available.
 3:     results = malloc and initialize vector with "n-parts" vectors
 4:     // Workers in parallel store partial results onto results.
 5:     QUERYRECURSE(root, results, max-heap, min-BST, max-BST)
 6:     offsets = malloc and initialize vector with n-parts integers
 7:     parallel for each $i \in [0, ..., \text{max-worker-id}]$ do
 8:         offsets[i] = results[i].size()
 9:     // Omitted: Cilk-specific grain size and related optimizations for
10:     // parallel execution.
11:     n-results = parallel plus-reduce(offsets, n-parts)
12:     offsets = parallel prefix-sum(offsets, n-parts)
13:     merged = merge individual worker results using prefix-sum
14:     return merged
15: procedure QUERYRECURSE(root, results, max-heap, min-BST, max-BST)
16:     if root == nullptr then return
17:
18:     interval = root->value
19:     if heap-dimension(interval) > max-heap then return
20:     if BST-dimension(interval) >= min-BST and interval.BST <= max-BST then
21:         results[get_worker_number()]->push_back(interval)
22:     // Omitted: optimizations for querying multiple leaves per node
23:     // and Cilk-specific grain size for parallel execution.
24:     if root->BST-mid >= min-BST then
25:         spawn QUERYRECURSE(root->left, results, max-heap, min-BST, max-BST)
26:     if root->BST-mid < max-BST then
27:         QUERYRECURSE(root->right, results, max-heap, min-BST, max-BST)
28:     sync // Parallel synchronization barrier
```

EdgeMap to the temporal setting by incorporating selective indexing (see§5.5.1) to dynamically switch between different scanning strategies (Temporal CSR scan vs. TGER) depending on the query. Moreover, the primary distinction between ***TemporalEdgeMap***'s programming interface and that of Ligra's ***EdgeMap*** is that it allows for *mapping over temporal edges while specifying an **ordering predicate** (§5.4.1)* as input. Combined with TGER, the ***TemporalEdgeMap*** acts as a parallel mapping function that enables temporal graph algorithms to efficiently retrieve only the

Algorithm 4 Pseudocode for Earliest Arrival in Kairos.

```
Input: A temporal graph $G = (V, E)$, a target vertex $x \in V$, and a time interval $[t_a, t_b]$.
Output: $t[V]$: The earliest-arrival time from $x$ to every vertex $v \in V$ within query time
    interval $[t_a, t_b]$.
 1: procedure UPDATE($s, d, [t_s, t_e]$)
 2:   if $t_s >= t_a$ or $t_e > t_b$ then return 0
 3:   if $t_s < t[s]$ or $t_e >= t[d]$ then return 0
 4:   return writeMin($t[d], t_e$) and CAS(Visited[$d$], 0, 1)
 5: procedure COND($i$)
 6:   return (Visited[$i$] == 0)
 7: procedure EARLIESTARRIVAL($G, x, [t_a, t_b]$)
 8:   $t[x] = t_b$, $t[v] = \infty$ for all $v \in V \setminus \{x\}$
 9:   Visited[$v$] = 0 for all $v \in V$
10:   Frontier = $\{x\}$
11:   while SIZE(Frontier) $\neq 0$ do
12:     Frontier = TEMPORALEDGEMAP($G, [t_a, t_b]$, Frontier, UPDATE, COND, ORD-
    PRED.STRICTLYSUCCEEDS)
```

edges that satisfy temporal predicates of interest, while still respecting user-defined constraints regarding the validity of temporal paths in their applications. In 4, we demonstrate the use of ***TemporalEdgeMap*** to update vertex frontiers for a parallel implementation of the earliest-arrival temporal path algorithm [95]. This example illustrates how the ***TemporalEdgeMap*** can be used to define complex suitable temporal graph processing logic in the context of parallel graph processing applications.

5.5 Selective Indexing Optimization

In this section, we introduce the concept of *selective indexing.* As we described §5.3.2, the Vertex Indexer builds a TGER only for those vertices whose size (as out/in-degree) meets an experimentally obtained predefined vertex size threshold (as of writing, currently set to 2k edges). With selective indexing, we propose a novel cost model and related algorithms to *determine the most appropriate access method for each vertex in a temporal graph during traversal.* The unique aspect of the cost estimation problem we address is that it selectively assigns different access methods (e.g., index vs full scan) *for the same query* at runtime. As an example, while a conventional relational query optimizer may select an in-memory hash index for all primary keys

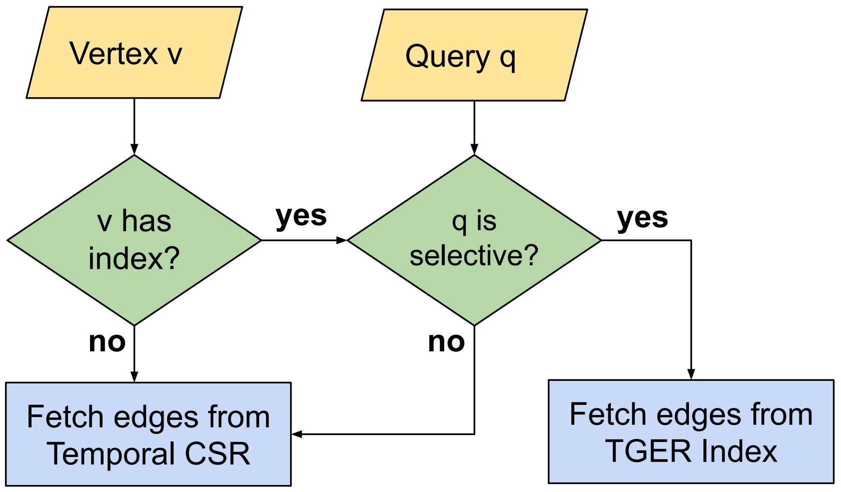


Figure 5-6: Selective indexing: decision tree for accessing a vertex's neighboring edges given an input query.

satisfying a PrimaryKey-ForeignKey join, our selective indexing approach considers the estimated selectivity for the query based on the *value* of each primary key, as well as the anticipated number of matches it has in the corresponding foreign key table.

Figure 5-6 presents the decision tree used by KAIROS's Vertex Indexer (Figure 5-2) to determine the optimal candidate data structure for accessing a vertex's neighboring edges, given an input temporal query. The following sections provide a description of our selective indexing approach's cost model and associated algorithms.

5.5.1 Cost Model

To define a cost model for deciding when it is beneficial to use TGER compared to a parallel scan over the T-CSR, we need to consider the following factors:

Vertex degree distribution. The distribution of the number of outgoing edges for each vertex is important because TGER is only built (and potentially accessed) for vertices with more than a certain number of outgoing (or incoming) edges. If a large portion of the vertices have a high degree, the custom index will be more frequently accessed and contribute more to the overall query performance. Furthermore, more memory will also be used to store index data.

Temporal edges distribution. This refers to the distribution of start times and end times present in these edges for a given vertex. These distributions indicate the expected number of matches for a given temporal predicate query interval. . Even though TGER is balanced, skew in the distribution of start times can lead to more levels being traversed to retrieve all edges that satisfy a given input predicate.

Query workload. For a given input temporal predicate, KAIROS needs to estimate the expected number of results for a specific vertex corresponding to that query.

In summary, the distribution of the start and end times of the temporal edges affects the selectivity of the queries. If the distribution is such that the queries have a high degree of selectivity (i.e., they filter out a significant portion of the data), using the TGER will be more efficient. Conversely, if the distribution is such that the queries have low selectivity (i.e., they return a large portion of the data), TGER's performance advantage over the T-CSR access method is reduced.

Taking these factors into account, we define a cost model that estimates the time required for each method (index-based using TGER and T-CSR-based) to execute a given query. The model acts as a proxy for the estimated query processing time given what we know about the graph (e.g., cardinality of a vertex's neighborhood), the query workload (e.g., estimated selectivity of input query), and characteristics of the data structures used for access (e.g., parallelism potential and asymptotic complexity). For the TGER-based access method, the cost model relies on the time complexity for PSTs, which is $O(\log m + k)$, where m is the number of elements stored in the data structure, and k is the number of results matching the input query. A TGER is created for each vertex (Figure 5-6), so $m = deg(v)$, where $v \in V$, which yields

$$T_v = c \cdot [\log(deg(v)) + k] \tag{5.1}$$

where c acts as a constant factor representing the average cost of performing a single operation using TGER. The value for c captures the parallelism potential when using TGER, and is derived experimentally.

For the T-CSR access method, KAIROS needs to perform a parallel scan to filter out edges that satisfy the input temporal predicate. Despite this operation being highly parallelizable and cache-friendly, the asymptotic time complexity for this scan is still $deg(v)$, as all edges for v need to be scanned, with

$$S_v = c' \cdot deg(v) \tag{5.2}$$

where c' is a constant factor representing the average cost of performing a single operation using the T-CSR. It performs a role similar to that of c for TGER, and like c, it too is derived experimentally. To decide which method is more beneficial for a given query, we compare the estimated time costs for both methods using the cost model. If the estimated time cost for TGER method (T_v) is lower than the estimated time cost for the T-CSR array method (S_v), then it is more beneficial to use TGER. Our cost model parameterizes the cardinality estimator based on the factors described above, with

$$C_v = \begin{cases} T_v & \text{if } \beta \leq \theta_{\text{sel}} \\ S_v & \text{otherwise} \end{cases} \tag{5.3}$$

C_v = the estimated cost of accessing a vertex's neighbors
T_v = the cost of querying for vertex v's neighbors in TGER
S_v = the cost of scanning vertex v's neighbors in T-CSR that satisfy the input temporal predicate
β = the selectivity of the input temporal predicate, defined as k/m
θ_{sel} = the selectivity threshold

For example, assuming a selectivity threshold $\theta_{\text{sel}} = 0.3$ (i.e., queries retrieving 30% of the neighboring edges of a vertex v), if the estimated selectivity β is less than 0.3 (e.g., 0.2), then KAIROS chooses TGER as the access method for vertex v given input query q.

In theory, the choice of threshold primarily depends on the relationship between k and $deg(v)$. As k approaches $dev(v)$, the cost of using TGER converges to $O(deg(v))$, rather than $O(\log(deg(v)))$. In this case, it becomes more beneficial to use T-CSR due to its higher parallelism potential. Parallelism is higher for queries over T-CSR because they are implemented as highly parallelizable scans over a parallel array. On the other hand, TGER queries rely on divide-and-conquer recursive parallel operations over a data structure that resembles a BST in one dimension and a heap in another, leading to reduced parallelism behavior as k nears $deg(v)$. In practice, however, we determine which threshold to use from experimental results, and find that 2k edges strikes a good performance vs estimator accuracy balance. By selecting an appropriate threshold, Kairos can decide at runtime whether to use the TGER index or T-CSR for accessing a vertex's neighbors, thereby improving query performance compared to a baseline that only uses T-CSR.

5.5.2 Cardinality Estimation Algorithm

Our cardinality estimation algorithm aims to help determine the best access method for each vertex while taking into account the temporal predicates present in the query workload and the characteristics of the temporal graph being queried.

During the index construction phase (i.e., when building the TGER indices), Kairos creates a 2D density histogram for each vertex that meets the threshold for TGER index construction. The dimensions of the histogram are the start time and duration (end time − start time) of the edges equally divided into 100 buckets per dimension, for a total of 10000 buckets, capturing the temporal distribution of a vertex's edges. At runtime, Kairos uses the histogram to estimate the density of a vertex's edges that would satisfy the query's temporal predicates. Depending on the estimated density and a selectivity threshold, Kairos selects the most efficient access method. If the estimated density is above the threshold, the query execution for that vertex employs the associated TGER index; otherwise, a parallel scan on the T-CSR is performed. This enables Kairos to adapt the access method according to the unique features of the temporal graph and the specific temporal predicates in the query, leading to improved query performance, as demonstrated in §5.6.

5.6 Experimental Results

In this section, we present the results of an experimental evaluation of KAIROS's query performance and scalability.

Setup. We use a 2nd Generation Intel Xeon Scalable Processor (Cascade Lake) system with 24 physical (48 virtual) cores on each of its two NUMA nodes. The socket has a total of 192GiB of DDR4 2666 MhZ RAM. Our programs use Cilk Plus [55] and are compiled with OpenCilk's [82] clang version 14 and -O3 flag. All of the parallel speedup numbers that we report are based on the running time on 24-cores (single socket) without hyper-threading compared to the running time on a single thread.

Datasets. Table 5.2 shows the number of vertices ($|V|$), number of edges ($|E|$), maximum out-degree ($\max_{v \in V} deg^{+}(v)$), maximum in-degree ($\max_{v \in V} deg^{-}(v)$), and average degree ($avg_{v \in V} deg(v)$) for each of the datasets under use. The SYNTHETIC data set comprises a temporal graph in which vertices are log-normally distributed, the inter-arrival times of start times follow a Poisson distribution, and the edge durations follow an uniform distribution. If the temporal edges in a dataset only have start times, then end time is sampled from a uniform distribution, similar to what is done in [95, 94].

Name	$\|V\|$	$\|E\|$	$\max_{v \in V} deg^{+}(v)$	$\max_{v \in V} deg^{-}(v)$	$avg_{v \in V} deg(v)$
bitcoin [61]	4.80×10^7	1.13×10^8	2.66×10^6	2.53×10^6	4
netflow [88]	3.72×10^8	1.20×10^9	5.78×10^5	1.82×10^8	12
reddit-reply [61]	1.17×10^7	6.46×10^8	3.92×10^5	9.94×10^5	110
stackoverflow [10]	6.02×10^6	6.34×10^7	1.01×10^5	93143	20
transportation [51]	41794	7.93×10^7	50881	50625	3798
twitter-cache [99]	94226	8.55×10^8	5.21×10^6	2.31×10^5	36328
synthetic	10^7	10^9	5.65×10^7	3.08×10^7	99

Table 5.2: Temporal graphs used for evaluation

5.6.1 Scalability

Table 5.3 shows the sequential and parallel running times of our algorithms, as well as their parallel speedup. Runtimes are reported as an average of ten runs. Except

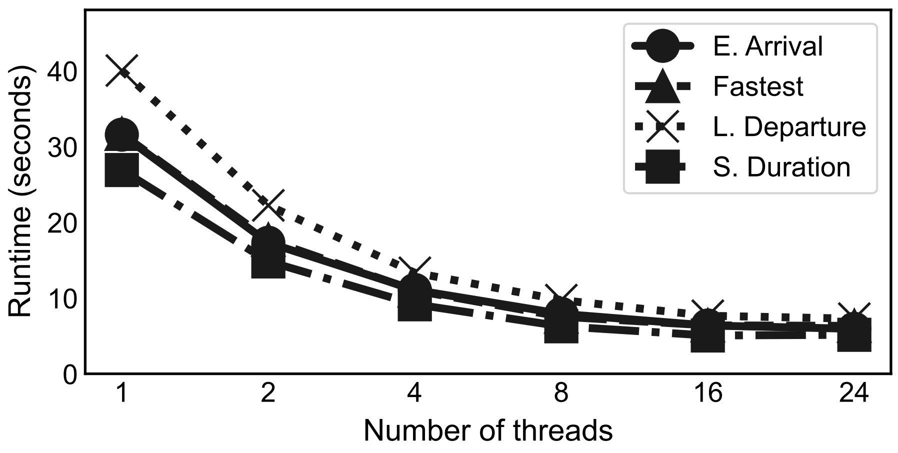


Figure 5-7: Running time vs number of threads on a smaller synthetic dataset (500M edges).

for T. BC, T. CC, k-core, and PageRank, all algorithm runtimes use a single source vertex. For algorithms requiring a single source, we selected the top 100 vertices based on their out-degree, leading to 100 runs in a single execution. For PageRank, the reported runtimes cover 100 iterations. T. BC uses the number of temporal S. Duration paths when calculating centrality for each vertex. The algorithms E. Arrival, L. Departure, Fastest, and S. Duration inherently expect a start and end time in their original definitions. For BC, BFS, CC, k-core, and PageRank, we have adapted the original algorithms to accept a start and end time as input. For every algorithm, we set the start time to align with the 95th percentile of the latest start times in the dataset. The end time is set to the maximum value, representing the 100th percentile.

The algorithms overall get good parallel speedup, with a maximum of 22.6x and mean speedup 8.7x. The lower speedups are in general observed in the smaller datasets, as they are too small to benefit from parallel processing. However, the larger speedups are not necessarily always observed in the largest dataset. Rather, they seem to be influenced by a number of factors, including how skewed the vertex degree distribution is, as well as fraction of edges matching the input temporal predicate. In other words, while we use the same query interval size for all experiments, the edges matching that input query predicate (i.e., its selectivity) may not be evenly distributed over all vertices, and thus do not offer the same parallelism potential. Furthermore, some co-routines in KAIROS are only parallelized if the minimum number of interations meets an heuristic threshold (e.g., at least 1000 iterations in parallel CILK_FOR loops), which again may be influenced by skew in the underlying vertex-degree and inter-arrival times distribution for a given dataset.

Figure 5-7 shows the running time vs. number of threads for all of the minimal temporal paths over a smaller synthetically generated dataset (500M edges). We see good parallel scalability for all of these algorithms, with speedups ranging from 5x to 8x, though see little benefit from 16 to 24 cores.

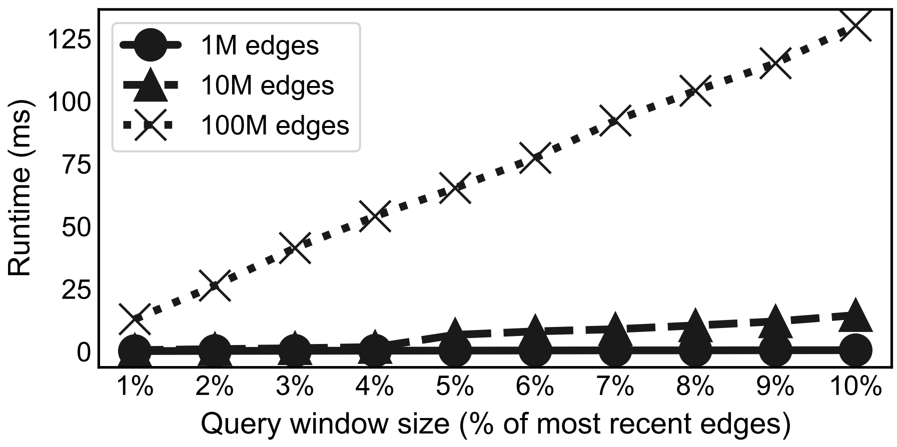


Figure 5-8: Runtimes vs. query window size (fraction of most recent edges) for different TGER sizes.

5.6.2 Impact of Selective Indexing

Figure 5-9 shows a comparison against a Temporal Ligra baseline, i.e., T-CSR is used for all vertices and no selective indexing is present, as described in §5.4.2. As far as we know, this is the fastest available shared-memory implementation of graph processing algorithms (temporal and traditional) over temporal graphs. The algorithms are configured in the same manner as in §5.6.1, and here we show the running times for a subset comprised mostly of temporal minimal paths, and over one small (reddit-reply), and two large datasets (twitter-cache and synthetic). Our results show that the selective indexing approach does reasonably well, with up to 8x improvement in some cases. As expected, the highly selective queries are the ones with the most

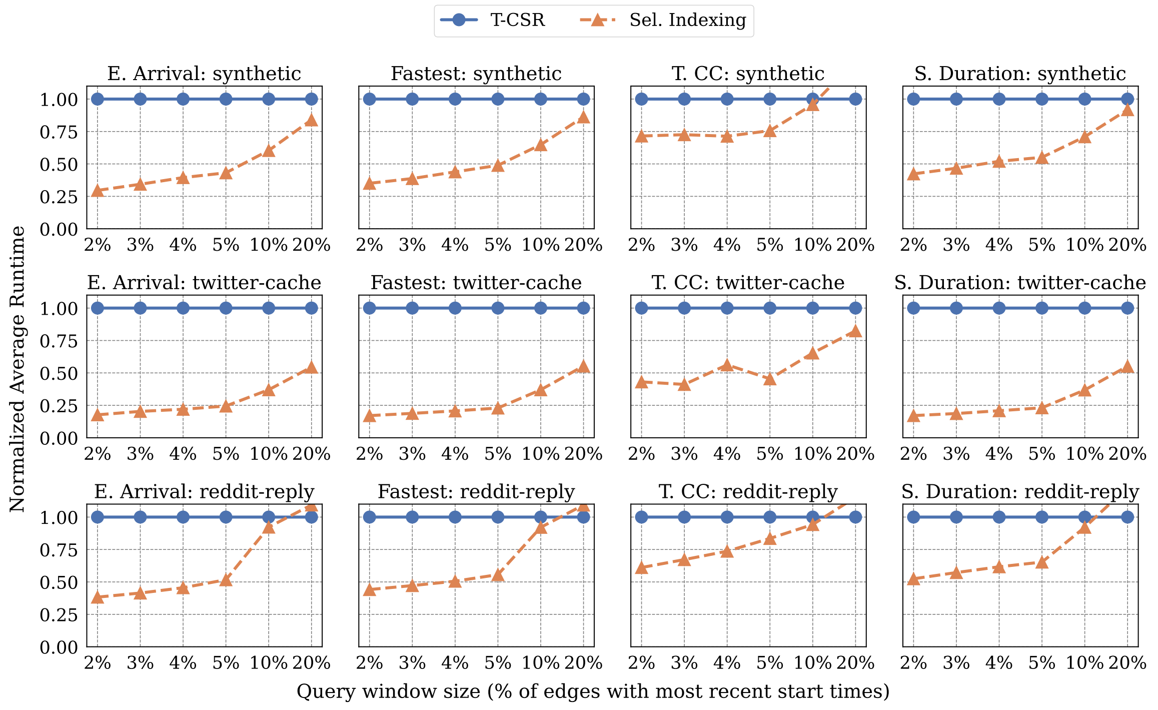


Figure 5-9: Normalized average runtime vs input query window size. Boundaries of the input query are varied so as to match exactly against a fixed percentage of most recent edges present in the corresponding temporal graph dataset. Runtimes are normalized against T-CSR as baseline.

improvement. Furthermore, between 10% and 20% selectivity, the T-CSR approach starts being more advantageous.

5.6.3 Microbenchmark: TGER query runtimes

Figure 5-8 shows the time it takes to run a query over a single TGER of various sizes (synthetically generated 1M, 10M and 100M edges). We note that each TGER in Kairos is associated to a single vertex. In other words, the running times reported in this experiment should be interpreted as the amount it takes to retrieve edges of interest from a *single vertex* that has been indexed with TGER. Similar to the results reported in experiments above, input queries are sized to match a portion of the most recent edges (by start time) in the input data (in this case, a vertex's neighboring edges). Results here show that it takes less than 125 milliseconds to retrieve roughly 10% of a TGER with 100M edges.

5.6.4 Comparison to Alternatives

Distributed memory. Although a direct comparison with Tink, ICM is challenging due to their distributed memory design, we offer a comparison based on the runtimes presented in their papers [59, 80]. Using 8 cores, Tink processes the E. Arrival algorithm for just one source vertex in 37s on a synthetic graph of 25 million edges. In contrast, KAIROS, with the same number of cores, processes the same algorithm for the 100 source vertices (equivalent to 100 runs) on a synthetic dataset of 500M edges, as shown in Figure 5-7 in less than 10s. *Put simply, for E. Arrival – the sole algorithm with reported runtimes by Tink – KAIROS handles data that's 20 times larger, addresses 100 times more source vertices, and completes the task in a third of the time.* We could not locate total runtimes for ICM in its publication. However, in its "weak scaling" analysis, it reports nearly 500 seconds to run E. Arrival on a single source vertex for a graph segment with 100M edges, using 8 cores. KAIROS, in contrast, requires under 10 seconds for 100 vertices on 500 million edge dataset (Figure 5-7). *In essence, when compared with ICM, KAIROS operates 50 times faster, handles 100 times more algorithm executions, processing a dataset that is 50 times larger.*

Shared memory. At the time of writing, the only other temporal graph analytics system in this category is TeGraph [46]. We have contacted the authors, who shared an implementation of shortest paths with us, as well DELICIOUS, one of the datasets they used for evaluation. This dataset has around 300M edges, and 34M vertices. Unfortunately, we were not able to reproduce the results in [46], which report around one second as the total runtime for executing shortest path on the aforementioned dataset using 100 source vertices. Instead, we find that their implementation of shortest path takes around three seconds to process a single vertex, regardless of which vertex is provided as input. Extrapolating from this result, we get around 300 seconds to process 100 source vertices – as opposed to the one second reported in their paper. Furthermore, the results for their "OnePass" baseline (which is closer to our approach) are significantly slower than KAIROS. Specifically, for the same DELICIOUS dataset,

	bitcoin			netflow			reddit-reply			stackoverflow			transportation			twitter-cache			synthetic		
Application	T_1	T_{24}	SU	T_1	T_{24}	SU	T_1	T_{24}	SU	T_1	T_{24}	SU	T_1	T_{24}	SU	T_1	T_{24}	SU	T_1	T_{24}	SU
E. Arrival	74.2	7.87	9.4	124.8	21.6	5.8	3.89	.76	5.1	17.1	1.99	8.6	58.8	6.49	9.1	246.9	16.11	15.3	440	31.7	13.9
L. Departure	56.4	6.63	8.5	125.2	22	5.7	5.1	.76	3.9	11.2	1.53	7.3	45.4	5.22	8.7	214.2	15.15	14.1	391	30.6	12.8
Fastest	56.3	7.73	7.3	185.6	31.6	5.9	5.81	1.11	5.2	9.16	1.32	6.9	38.7	4.73	8.2	222.6	12.84	17.4	311	23.1	13.5
S. Duration	56.6	7.76	7.3	187	32.2	5.8	5.84	1.07	5.5	9.22	1.33	6.9	2.7	19.7	7.3	12.84	223.5	17.4	285	21.6	13.2
T. BFS	8.5	1.5	5.7	97.2	16.5	5.9	1.9	.4	4.8	.8	.2	4	.009	.002	4.5	9.8	1.4	7	19.6	3.2	6.2
T. CC	5.65	.59	9.6	24.4	2.2	11.1	3.99	.2	19.9	1.85	.11	16.8	6.83	.49	13.9	10.2	.71	14.4	5.41	4.57	1.2
T. k-core	13.7	1.22	11.4	38.9	3.6	10.8	32.4	4.2	7.7	2.2	.4	5.5	.02	.01	2	2.7	.9	3	2	.5	4
T. BC	10.8	.57	18.95	8.04	1.26	6.38	.032	.168	5.25	.085	.014	6.07	.005	.003	1.67	2.84	.692	4.1	4.57	3.76	1.22
T. PageRank	65.3	5.23	12.5	35.8	2.6	13.8	11.3	.50	22.6	8.51	.42	20.3	2.6	.21	12.4	17.22	1.2	14.4	11	4.62	2.4

Table 5.3: Running times (in seconds) of single-threaded (T_1), 24-core no hyper-threading (T_{24}), and parallel speedup as single-thread time divided by 24-core time (SU).

their "OnePass" baseline takes close to 90 seconds for 100 vertices using 16 threads on a 8-core machine. *KAIROS, on the other hand, takes around five seconds using 8 cores, which is a 18x speedup compared to their "OnePass" baseline, and 60x compared to our local results from their implementation.*

5.6.5 Selective Indexing: Estimator Accuracy

In §5.5.1, we proposed a novel approach for selectively indexing a subset of vertices in a temporal graph, as well as cost model for deciding which access method to use for each vertex at runtime. In this section, we present an experiment assessing the accuracy of the *cardinality estimator*, a key component of our proposed cost model. For this evaluation, we vary the size of an input temporal predicate, and use a selectivity threshold of 20% (the "q is selective?" decision in Figure 5-6). As in the experiments we describe above, here we also vary the size of an input query interval to purposefully match against a percentage of the most recent edges (by start time) in the dataset. To measure accuracy, we define true positives as "should use TGER, and did" and true negatives as "should not use TGER, and did not". Specifically, "should" here takes into account the estimated selectivity compared to an oracle with the actual selectivity of the query. Furthermore, we only evaluate this decision for vertices that have been large enough to be indexed with a TGER (otherwise, no decision needs to be made, as T-CSR is used a 100% of the time), and vary the minimum cutoff for indexing from 1k to 8k edges (multiples of 2). We find that for all datasets, the accuracy of this decision stays consistently above 90% for input query intervals sized under 1%, and above 95% for all other input query interval sizes we assessed (2 through 5%, 10%, and 20%). Furthermore, the accuracy also increases as a function the cutoff size. As expected, this increase is largely due to the cardinality estimator's 2D density histogram (§5.5.2) having more samples in this case.

Chapter 6

Discussion and Future Work

We have presented Kaskade and Kairos, two systems tailored for query optimization for efficient graph analytics on real-world graphs. Kaskade exploits structural patterns and otherwise regularity present in graphs to infer materialized views that are beneficial for query processing. Kairos targets temporal graph processing and introduces *selective indexing*, a novel technique for choosing subset of vertices to index. Furthermore, it leverages state-of-the-art parallel processing techniques, bringing them to the temporal setting, and employs a highly-optimized parallel index for large vertices.

This chapter delves into potential improvements and extensions for Kaskade and Kairos, aiming to address their current limitations and suggest paths for further research in real-world graph analytics. Finally, we conclude by suggesting ways in which both systems could be integrated.

6.1 Kaskade

1. **Approximate View Inference.** Kaskade's formulates query rewriting in terms of materialized views as a satisfiability problem, leveraging an logical programming engine (Prolog) to infer candidate views. This problem formulation has the advantageous side-effect that views inferred in this manner are "correct by construction". Nevertheless, if the correctness requirement can be relaxed, then

it should be possible to instead leverage approximate processing techniques, such as those present in machine learning. In particular, investigating graph learning techniques (e.g., GNNs) to "predict" the most beneficial views for given graphs and queries could lead to more dynamic optimization in KASKADE.

2. **Expand View Types.** KASKADE is engine-agnostic, as it only relies on fundamental transformations (e.g., path contractions and filtering operations) of the underlying graph and input queries. This presents a limitation, however, as the only immediate class of views that we considered were natural combinations of these primitives, such as $k - hop$ connector views. A more detailed exploration of other graph transformation primitives (e.g., aggregating ego-centric neighborhoods into a super-vertex) might yield additional classes of graph views of interest – if not more powerful than those we proposed in KASKADE.

3. **Incremental View Maintenance.** The worst-case complexity of incrementally maintaining graph views is a direct function of the complexity of each graph transformation used to assemble that view. For example, a loose upper bound for incrementally maintaining a $2-$hop connector view is $max(\deg(v))^2$. In our experiments, we find that summary statistics such as lower percentiles of the vertex degree distribution provide more useful approximations than the maximum vertex degree. This is an encouraging result, as it hints at the possibility of efficient incremental maintenance for a subset of $k-$hop graph views. Furthermore, if new types of graph views are introduced in which the building blocks have amenable asymptotic complexity, it should be possible to devise incremental maintenance algorithms which are potentially efficient. If used in conjunction with dynamic data structures, this is potentially a fruitful avenue for future work. In other words, developing methods for KASKADE to adapt materialized views in real-time as the underlying graph data changes could greatly enhance its suitability for processing of dynamic graphs.

4. **Automated View Management.** While some of the contributions in KASKADE were deployed in production at Microsoft data infrastructure, this work did not

directly target the full life cycle of graph data analytics, which often requires automated management. Implementing automated view management that can handle view creation, maintenance, and retirement based on usage patterns and storage constraints would be especially important in this context. Some topics for future research include intelligent policies for retiring views, or predicting when a view would be needed in the future. Just like in **Approximate View Inference**, this too is a research topic which can greatly benefit from machine learning techniques.

5. **Integration with Streaming Data Sources.** Finally, in Kaskade we assumed a scenario that was reflective of batch pipelines which daily added new edges to the data lineage graph at Microsoft. Therefore, we did not consider real-time graph analytics, despite the fact that some of the optimizations our work enables directly facilitate that. Hence, extending Kaskade to handle streaming graph data, enabling real-time analytics on continuously evolving graphs, would be especially beneficial. While such tasks are often considered purely engineering efforts (e.g., integrating open source software projects such Kafka queues to consume events in real-time that get added to the graph), depending on the sizes and frequency of data ingestion requirements, they can yield interesting research challenges.

6.2 Kairos

1. **Dynamic Index Updating.** One of Kairos's key contributions is TGER, a highly-optimized parallel data structure to index temporal edges. Since in this work we only considered *static* temporal graphs, our data structure did not address updates to the underlying graph. Therefore, application programmers need to rebuild the graph from scratch – including any TGER indexes – if they require new edges to be considered for processing. Implementing support for dynamism in TGER would better support such needs. This could be conbined with PackedCSR [93] to support updates in the underlying graph, and would improve Kairos's efficiency in environments with frequently changing data.

2. **Distributed Processing Integration.** KAIROS intentionally targets shared-memory environments in which a single machine is used for temporal graph processing. Nevertheless, nothing prevents application programmers from deploying KAIROS in a distributed setting. This would, however, require careful partitioning of temporal edges so that they can be dispatched to the appropriate server for processing. A number of works have been proposed in the distributed graph processing literature which propose partitioning schemes, including more recently the use of machine learning for this task. This hybrid integration would yield a system that has both scale-out and scale-up properties, which would be an attractive "best of both worlds" proposition.

3. **Enhanced Query Parallelization.** A current bottleneck of KAIROS is on TGER's parallel processing capabilities. Specifically, due to it being a tree-based data structure, it naturally suffers from underutilization of parallel tasks, as divide-and-conquer is not as efficient as a parallel array scan. Thus, developing more sophisticated techniques for parallel query processing in KAIROS as it relates to TGER to fully exploit multicore architectures would beneficial.

4. **Index Size Optimization.** Unlike Ligra+ [85], TGER currently makes no use of compression. Researching methods to reduce the memory footprint of the TGER index without compromising its query performance would be of interest here.

5. **Adaptive Indexing Strategies.** The cost model used in KAIROS uses a pretty straightforward 2D histogram to capture correlations in the join-distribution of start and end times in the underlying temporal graph. A more sophisticated cost model could be considered, instead. Furthermore, KAIROS uses a binary decision diagram to either fully index all temporal edges of a vertex, or not index it at all. Instead, one could consider adaptive indexing strategies that can adjust the level of indexing based on query patterns and data distribution. Eviction policies could also be considered in tandem here.

6.3 Integrating Both Systems

In Kaskade and Kairos we have proposed query optimization systems and techniques that specifically target real-world graph analytics. Their continuous evolution, through the improvements we propose above, can lead to more dynamic, efficient, and powerful tools that are necessary for handling the complexities of real-world graph data. In addition, the problems each of these two systems address are complementary. As such, we consider below potential ways in which both systems could work together in a single unified solution.

1. **Temporal Views.** In §5.4.3 we introduce the notion of Ordering Predicates as a programming primitive to specify the semantics of valid time-respecting paths. Specifically, we show at least three different orderings of interest: *overlaps*, *succeeds*, and *strictly succeeds*. As such, it follows that *temporal graph views* would need to provide similar flexibility for application programmers to be able to correctly define views of interest. This could be done via template definitions that can take as input an ordering predicate. Alternatively, view template definitions could either default to one of the three ordering predicate options (e.g., succeeds, as is the case for Kairos), or enumerate all three options. The latter could be detrimental, however, as it could lead to an increase in the number of possible views to be inferred, depending on the underlying graph structure.

2. **Selective Indexing.** The selective indexing approach we introduced in Chapter 5 naturally extends to non-temporal settings. In this case, vertices with large-degrees would be chosen for indexing. However, selective indexing alone does not prescribe what type of index data structure should be used for each vertex. In the case of Kairos, we want to be able to filter temporal edges of interest based on the temporal filter predicate contained in the input query, so the selective indexing cost model takes into account properties that are specific to temporal data correlations. Depending on the target query workload, however, there might be other index data structures of interest which might be better suited for non-temporal settings.

Chapter 7

Conclusion

We presented two systems aimed at efficiently executing graph analytics tasks (queries and algorithms) over large scale real-world graph data.

The first system we introducted, Kaskade, is a graph query optimization framework that employs materialization to efficiently evaluate queries over graphs. Many application repeatedly run similar queries over the same graph, and many production graphs have structural properties that restrict the types of vertices and edges in them. These facts motivate Kaskade to automatically derive graph views using a new constraint-based view enumeration technique, and a novel cost model that accurately estimates the size of graph views. We show that queries rewritten over some of these views can provide up to 50 times faster response times. Finally, the rewriting techniques we have proposed are engine-agnostic (i.e., only use fundamental graph transformations that typically yield smaller graphs), thus applicable to other graph systems.

In addition, we introduced Kairos, a temporal graph analytics system that provides application developers a framework for efficiently executing temporal algorithms over temporal graphs. Specifically, it employs TGER, a highly-optimized parallel data structure, as efficient index for temporal graph processing. Kairos is built atop Ligra [84], a state-of-the-art parallel graph processing system. With TGER, Ligra's vertex-centric computational model takes advantage of locality that is naturally occurring on temporal graphs and queries over such graphs. We show in our experiments that using Kairos, a number of minimal temporal path algorithms are up to 8x times faster than an already competitive baseline, which we also provide. When compared with alternative shared-memory system, it achieves up to 60x speedups.

Through the query optimization systems and techniques we introduced in this thesis, we aim to advance the field of query optimization for graph analytics, enhancing both its efficiency and its accessibility to practitioners. Our work highlights the benefits of adapting to underlying characteristics that are inherent to real-world data, such as high levels of structure and skew in multiple dimensions. Furthermore, the research directions we identify in our work with Kaskade and Kairos can serve as foundation upon which to improve our collective understanding of efficient techniques for processing real-world graphs.

Bibliography

[1] AgensGraph Hybrid Graph Database. http://www.bitnine.net.

[2] APOC Neo4j. https://neo4j-contrib.github.io/neo4j-apoc-procedures/#algorithms.

[3] Google OR Tools. https://developers.google.com/optimization/bin/knapsack.

[4] Graph processing with SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview.

[5] GraphDBLP. https://github.com/fabiomercorio/GraphDBLP.

[6] Neo4j Graph Database. http://neo4j.org.

[7] Network Repository. http://networkrepository.com.

[8] Property Graphs. www.w3.org/2013/socialweb/papers/Property_Graphs.pdf.

[9] SNAP LiveJournal dataset. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[10] SNAP StackOverflow dataset. https://snap.stanford.edu/data/sx-stackoverflow.html.

[11] SWI-Prolog Engine. http://www.swi-prolog.org.

[12] SWI-Prolog Syntax. http://www.swi-prolog.org/man/syntax.html.

[13] James F Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[14] Chrysovalantis Anastasiou, Constantinos Costa, Panos K Chrysanthis, Cyrus Shahabi, and Demetrios Zeinalipour-Yazti. ASTRO: Reducing COVID-19 exposure through contact prediction and avoidance. *ACM Trans. Spatial Algorithms Syst.*, 8(2):1–31, 2022.

[15] Grigoris Antoniou and Frank Van Harmelen. *A semantic web primer*. MIT press, 2004.

[16] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 3rd ed. edition, 2008.

[17] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems*, 34(6):1860–1876, 2023.

[18] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. *WWW*, pages 595–602, 2004.

[19] J. Byun, S. Woo, and D. Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *TKDE*, 32(3):424–437, 2020.

[20] Chen Chen, Cindy X. Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. Mining graph patterns efficiently via randomized summaries. *PVLDB*, pages 742–753, 2009.

[21] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and S Yu Philip. Graph OLAP: Towards online analytical processing on graphs. In *ICDM 2008*, pages 103–112.

[22] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. *EuroSys*, pages 85–98, 2012.

[23] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. Relevance of temporal cores for epidemic spread in temporal networks. *Nature Scientific reports*, 10(1):12529, 2020.

[24] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, 2019.

[25] Joana M F da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. Kaskade: Graph Views for Efficient Graph Analytics. *CoRR 2019*, abs/1906.05162. Preprint, `http://arxiv.org/abs/1906.05162`.

[26] Joana M F da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. Kaskade: Graph views for efficient graph analytics. In *ICDE*, pages 193–204, 2020.

[27] Joana M F da Trindade, Julian Shun, Samuel Madden, and Tatbul Nesime. c. In *NEDB (poster)*, 2023.

[28] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, pages 1–140, 2007.

[29] Benjamin Erb, Dominik Meißner, Jakob Pietron, and Frank Kargl. Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs. In *DEBS*, pages 78–87, 2017.

[30] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting Regular XPath Queries on XML Views. In *ICDE 2007*, pages 666–675.

[31] W. Fan, X. Liu, and Y. Cao. Parallel reasoning of graph functional dependencies. In *ICDE*, pages 593–604, apr 2018.

[32] W. Fan, X. Wang, and Y. Wu. Answering Pattern Queries Using Views. *IEEE Transactions on Knowledge and Data Engineering*, 28(2):326–341, Feb 2016.

[33] Wenfei Fan. Dependencies for graphs: Challenges and opportunities. *J. Data and Information Quality*, 11(2):5:1–5:12, February 2019.

[34] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. Discovering Graph Functional Dependencies. In *SIGMOD 2018*, pages 427–439.

[35] Wenfei Fan, Ruochun Jin, Ping Lu, Chao Tian, and Ruiqi Xu. Towards event prediction in temporal graphs. *PVLDB*, pages 1861—-1874, 2022.

[36] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[37] Wenfei Fan and Ping Lu. Dependencies for graphs. *ACM Trans. Database Syst.*, 44(2):5:1–5:40, February 2019.

[38] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional Dependencies for Graphs. In *SIGMOD 2016*, pages 1843–1857.

[39] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD 2018*, pages 1433–1445.

[40] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View Selection in Semantic Web Databases. In *VLDB 2011*, pages 97–108.

[41] Ashish Gupta and Inderpal Singh Mumick. In *Materialized Views*, chapter Maintenance of Materialized Views: Problems, Techniques, and Applications, pages 145–157. 1999.

[42] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing google's datasets. In *SIGMOD*, 2016.

[43] Bin Han and Bill Howe. Adapting to skew: Imputing spatiotemporal urban data with 3D partial convolutions and biased masking. January 2023.

[44] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *EuroSys*, pages 1–14, 2014.

[45] Petter Holme and Jari Saramäki. Temporal networks. *Physics Reports*, abs/1108.1780:97–125, 2012. Preprint, `http://arxiv.org/abs/1108.1780`.

[46] Chengying Huan, Hang Liu, Mengxing Liu, Yongchao Liu, Changhua He, Kang Chen, Jinlei Jiang, Yongwei Wu, and Shuaiwen Leon Song. TeGraph: A novel General-Purpose temporal graph computing engine. *ICDE*, pages 578–592, 2022.

[47] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. Temporal graph benchmark for machine learning on temporal graphs. *CoRR 2023*, abs/2307.01026, 2023. Preprint, `http://arxiv.org/abs/2307.01026`.

[48] Edward Hung, Yu Deng, and Venkatramanan S Subrahmanian. RDF aggregate queries and views. In *ICDE 2005*.

[49] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *GRADES*, pages 5:1–5:6, 2016.

[50] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD 2012*, pages 565–576.

[51] Rainer Kujala, Christoffer Weckström, Richard K Darst, Miloš N Mladenović, and Jari Saramäki. A collection of public transport network data sets for 25 cities. *Nature Sci. Data*, 5, 2018.

[52] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[53] Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes. From networks to optimal higher-order models of complex systems. *Nature Phys.*, 15(4):313–320, 2019.

[54] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. Rewriting queries on SPARQL views. In *WWW 2011*, pages 655–664.

[55] Charles E Leiserson. The cilk++ concurrency platform. In *DAC*, pages 522–527, 2009.

[56] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, KDD, pages 177–187, 2005.

[57] Lei Li, Sibo Wang, and Xiaofang Zhou. Time-dependent hop labeling on road network. In *ICDE*, pages 902–913, 2019.

[58] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. Persistent community search in temporal networks. In *ICDE*, pages 797–808, 2018.

[59] Wouter Lightenberg, Yulong Pei, George Fletcher, and Mykola Pechenizkiy. Tink: A temporal graph analytics library for apache flink. *WWW*, pages 71–72, 2018.

[60] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on typed graphs. In *PVLDB*, pages 265–276, 2016.

[61] Paul Liu, Austin R. Benson, and Moses Charikar. Sampling methods for counting temporal motifs. In *WSDM*, 2019.

[62] Tiantian Liu, Huan Li, Hua Lu, Muhammad Aamir Cheema, and Lidan Shou. Towards crowd-aware indoor path planning. *PVLDB*, page 1365–1377, 2021.

[63] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3):62:1–62:34, June 2018.

[64] Antonio Longa, Veronica Lachi, Gabriele Santin, Monica Bianchini, Bruno Lepri, Pietro Lio, franco scarselli, and Andrea Passerini. Graph neural networks for temporal graphs: State of the art, open challenges, and opportunities. *Transactions on Machine Learning Research*, 2023.

[65] Antonio Maccioni and Daniel J. Abadi. Scalable pattern matching over compressed graphs via dedensification. In *KDD*, pages 1755–1764, 2016.

[66] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudré-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *CIDR 2017*.

[67] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudre-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *NDSS*, 2017.

[68] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *TOS*, 11(3):14:1–14:34, 2015.

[69] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[70] Daniele Notarmuzi, Claudio Castellano, Alessandro Flammini, Dario Mazzilli, and Filippo Radicchi. Universality, criticality and complexity of information propagation in social media. *Nature communications*, 13(1):1308, 2022.

[71] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. Rewriting Nested XML Queries Using Nested Views. In *SIGMOD 2006*, pages 443–454.

[72] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *WSDM*, pages 601–610, 2017.

[73] Hongchao Qin, Rong-Hua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. Mining bursting core in large temporal graphs. *PVLDB*, pages 3911–3923, 2022.

[74] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. A distributed path query engine for temporal property graphs. *CoRR 2020*, abs/2002.03274. Preprint, `https://arxiv.org/abs/2002.03274`.

[75] Manuel Gomez Rodriguez, Jure Leskovec, David Balduzzi, and Bernhard Schölkopf. Uncovering the structure and temporal dynamics of information propagation. *Network Science*, 2(1):26–65, 2014.

[76] Manuel Gomez Rodriguez, Jure Leskovec, David Balduzzi, and Bernhard Schölkopf. Uncovering the structure and temporal dynamics of information propagation. *Network Science*, 2(1):26 – 65, 2014.

[77] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. Distributed temporal graph analytics with GRADOOP. *PVDLB*, pages 375–401, 2022.

[78] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD 2000*, pages 249–260.

[79] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. Synopsys: Large graph analytics in the sap hana database through summarization. In *GRADES*, pages 16:1–16:6, 2013.

[80] S. Gandhi and Y. Simmhan. An interval-centric model for distributed computing over temporal graphs. In *ICDE*, pages 1129–1140, 2020.

[81] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian

Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021.

[82] Tao B Schardl and I-Ting Angelina Lee. OpenCilk: A modular and extensible software infrastructure for fast Task-Parallel code. In *PPoPP*, pages 189–203, 2023.

[83] Timos K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[84] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[85] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, pages 403–412, 2015.

[86] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *SIGMOD*, pages 1481–1496, 2016.

[87] Dimitri Theodoratos, Spyros Ligoudistianos, and Timos Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3):219–240, 2001.

[88] Melissa J. M. Turcotte, Alexander D. Kent, and Curtis Hash. *Unified Host and Network Data Set*, pages 1–22. 2018.

[89] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *GRADES*, pages 7:1–7:6, 2016.

[90] Minjie Yu Wang. Deep graph library: towards efficient and scalable deep learning on graphs. *ICLR*, 2019.

[91] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*, 2020.

[92] Zhihao Wen and Yuan Fang. TREND: temporal event and node dynamics for graph representation learning. In *WWW*, pages 1159–1169, 2022.

[93] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *HPEC*, pages 1–7, 2018.

[94] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu. Efficient algorithms for temporal path computation. *TKDE*, 28(11):2927–2942, 2016.

[95] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *PVLDB*, pages 721–732, 2014.

[96] Mincheng Wu, Chao Li, Zhangchong Shen, Shibo He, Lingling Tang, Jie Zheng, Yi Fang, Kehan Li, Yanggang Cheng, Zhiguo Shi, Guoping Sheng, Yu Liu, Jinxing Zhu, Xinjiang Ye, Jinlai Chen, Wenrong Chen, Lanjuan Li, Youxian Sun, and Jiming Chen. Use of temporal contact graphs to understand the evolution of COVID-19 through contact tracing data. *Nature Commun. Phys*, 5, 2022.

[97] Konstantinos Xirogiannopoulos and Amol Deshpande. Extracting and analyzing hidden graphs from relational databases. In *SIGMOD*, pages 897–912, 2017.

[98] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[99] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI*, pages 191–208, 2020.

[100] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffery Xu Yu. Scalable time-range k-core query on temporal graphs (full version). *PVLDB*, pages 1168–1180, 2023.

[101] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. On querying historical k-cores. pages 2033–2045, 2021.

[102] Tianming Zhang, Yunjun Gao, Lu Chen, Wei Guo, Shiliang Pu, Baihua Zheng, and Christian S Jensen. Efficient distributed reachability querying of massive temporal graphs. pages 871–896, 2019.

[103] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A general framework for temporal gnn training on billion-scale graphs. *PVLDB*, page 1572–1580, 2022.

[104] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE 1998*, pages 116–125.