Graph Database Management Systems

Storage, Management and Query Processing

A thesis submitted in fulfilment of the requirements for
the degree of Doctor of Philosophy

Oshini Goonetilleke

School of Science

College of Science, Engineering, and Health

RMIT University

December, 2017

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Oshini Goonetilleke
School of Science
RMIT University
20th December, 2017

Acknowledgements

First, I would like to thank my supervisor, Timos Sellis, for all his advice, support and patience throughout my PhD journey. Over the past four years while working with Timos, I have learned many things about research and many important lessons in life. I feel privileged to have worked under his guidance. I would also like to thank my other supervisor, Jenny Zhang, for her advice and support during my PhD.

I had an amazing group of collaborators, each contributing and supporting me in different ways. I would like to thank each of them: Saket Sathe, Danai Koutra, David Meibusch, Ben Barham and Kewen Liao. Saket for his feedback and advice during the early years of my PhD and his humorous insights; and Danai for long conversations about research problems at odd hours in the US, and suggesting interesting directions for my work. Working with all these collaborators was a great learning experience for me.

I spent a great summer at Oracle Labs during my internship with them. I enjoyed working with a small team having many insightful discussions and learned different approaches to problem solving. I believe this time was truly a turning point in my PhD. I'm particularly thankful to David Meibusch who has been a great mentor.

My PhD experience was made so much easier because of my friends scattered throughout the world; especially Pathi for our long chats about work, life and everything else in between. I would specially like to thank Ammi (mother) and Nangi (sister) who have missed me so much back home during this period but have given me emotional and moral support all the way. Thank you for always believing in me. I would also like to thank my late father who has been the silent inspiration behind everything I do. Finally I thank my husband, Dineth, for his love, patience and encouragement; he has been very much my biggest pillar of strength.

Contents

List of Figures

List of Tables

Abstract

The proliferation of graph data, generated from diverse sources, have given rise to many research efforts concerning graph analysis. Interactions in social networks, publication networks, protein networks, software code dependencies and transportation systems are all examples of graph-structured data originating from a variety of application domains and demonstrating different characteristics. In recent years, graph database management systems (GDBMS) have been introduced for the management and analysis of graph data. Motivated by the growing number of real-life applications making use of graph database systems, this thesis focuses on the effectiveness and efficiency aspects of such systems. Specifically, we study the following topics relevant to graph database systems: (i) modeling large scale applications in GDBMS; (ii) storage and indexing issues in GDBMS, and (iii) efficient query processing in GDBMS.

In this thesis, we adopt two different application scenarios to examine how graph database systems can model complex features and perform relevant queries on each of them. Motivated by the popular application of social network analytics, we selected Twitter, a microblogging platform, to conduct our detailed analysis. Addressing limitations of existing models, we propose a data model for the Twittersphere that proactively captures Twitter-specific interactions. We examine the feasibility of running analytical queries on GDBMS and offer empirical analysis of the performance of the proposed approach. Next, we consider a use case of modeling software code dependencies in a graph database system, and investigate how these systems can support capturing the evolution of a codebase overtime. We study a code comprehension tool that extracts software dependencies and stores them in a graph database. On a versioned graph built using a very large codebase, we demonstrate how existing code comprehension queries can be efficiently processed and also show the benefit of running queries across multiple versions.

Another important aspect of this thesis is the study of storage aspects of graph systems. Throughput of many graph queries can be significantly affected by disk I/O performance, therefore graph database systems need to focus on effective graph storage for optimising disk operations. We observe that the locality of edges plays an important role and we address the edge-labeling problem which aims to label both incoming and outgoing edges of a graph maximizing the 'edge-consecutiveness' metric. By achieving a better layout and locality of edges on disk, we show that our proposed algorithms result in significantly improved disk I/O performance leading to faster execution of neighbourhood queries.

Some applications require the integrated processing of queries from graph and the textual domains within a graph database system. Aggregation of these dimensions facilitate gaining key insights in several application scenarios. For example, in a social network setting, one may want to find the closest k users in the network (graph traversal) who talk about a particular topic A (textual search). Motivated by such practical use cases, in this thesis we study the top-k social-textual ranking query that essentially requires efficient combination of a keyword search query with a graph traversal. We propose algorithms that leverage graph partitioning techniques, based on the premise that socially close users will be placed within the same partition, allowing more localised computations. We show that our proposed approaches are able to achieve significantly better results compared to standard baselines and demonstrating robust behaviour under changing parameters.

Keywords. Graph Database Systems, Graph data models, Query processing, Property graphs, Graph storage, Edge Labeling, Versioned codebase, Twitter Data management, Graph keyword search.

Chapter 1

Introduction

Graph theory has a long standing history dating back to the notable mathematical problem of the Seven Bridges of Königsberg in the Eighteenth century [19]. Leonhard Euler, laying the foundations in graph theory in 1736, introduced this problem where the objective was to generate a walk through the city of Königsberg that would cross each bridge once and only once. In its simplest form, a graph is a mathematical structure to model pairwise relations between objects. In the Königsberg problem, the nodes in the graph represented the land mass and the edges represented the bridges that connected them.

Researchers have since modeled both natural and man-made structures as graphs in biological, transport, software, physical and social systems. The types of analyses conducted on these models are diverse, giving rise to interesting research questions with domain-specific challenges. For example, the pertinent use cases in social networks may involve both content and user recommendations; protein interaction networks are modeled to understand complex biology of diseases by analysing substructures that are similar to a given subgraph; and in software systems, dependencies may be modeled for advanced source code comprehension and analysis.

1.1 Graph Data Management Systems

For years researchers have employed ad-hoc tools and techniques to manage and analyse graph data. However, graph data generated from heterogeneous sources are becoming more complex and are growing rapidly in size. Social networks such as Facebook[1] and Twitter[2] have been

[1]https://www.facebook.com/
[2]https://twitter.com/

experiencing exponential growth in their user base since their inception, just over a decade ago. The advent of the scale and complexity of graph data in the recent years have demanded more sophisticated, scalable data management systems that can store, manage and query graph data. This allows researchers to focus on the analysis task with the help of a coherent tool set that takes care of the management aspect of graph data. At the forefront of such tools are Graph Database Management Systems (GDBMS) (or simply *graph database systems*). Design goals are similar to that of Relational Database Management Systems developed for the management of relational data. Considering that modern graph database systems are relatively new, they require in-depth investigation of effectiveness and efficiency in managing graph data. This thesis focuses on the following related topics concerning graph database systems.

- **Data modeling:** Unlike in relational systems, graph data do not conform to a strict schema. Types of nodes, edges and properties on them can be user-defined, with no strict rules on what can and cannot be modeled. The abstraction of the data model is domain- or application- specific, formulated primarily based on the types of queries we want to efficiently run on them.
- **Query Processing:** Graph database systems do not share a universal query language to query its graph data. Some graph systems expose declarative query languages, while others feature versatile APIs to interact with the database. We examine how effective these different query methods are in expressing the information needs of different use cases and how efficiently the queries can be processed.
- **Storage and Indexing:** Although storage and indexing specifications and mechanisms are nearly ubiquitous among many relational databases, in graph databases, they differ significantly. As such there are many opportunities to investigate storage and indexing aspects, with the goal of improved disk I/O and query performance.

Focused on the above aspects, we next outline the motivations for investigating several applications and detail our contributions in each.

1.2 Motivation and Contributions

The focus of this thesis is to explore mechanisms, problems, and challenges associated with graph modeling, storage and query processing aspects of graph database systems. In this section we introduce our motivations and contributions pertaining to these related topics.

1.2.1 Modeling large scale applications in a GDBMS

Modeling large scale and complex applications is an important aspect to realizing the effectiveness of graph database systems. This has also enabled us to understand potential gaps in these systems when modeling such varied structures and use cases. Modeling generally involves preparing the graph data itself in a suitable schema and performing appropriate queries relevant to each application. We adopt two important applications to observe how GDBMS can support complex features on each of them. First, we choose a popular application of GDBMS that involves modeling social networks. The second application requires a more complex analysis; we consider modeling software code dependencies to capture the temporal evolution of a codebase.

Modeling Social Networks. We select Twitter as a representative microblogging platform for social network analytics and review prior work on data collection [22, 24, 28], data management frameworks [22, 14, 20] and query systems [168, 54, 127] to understand the existing models and analytics space. With our observations of the extensive survey, we highlight the need to assimilate the individual work-flows in an integrated solution addressing the limitations of existing systems. Different from traditional RDF and relational data models, we observe potential in a graph-view of Twitter, enabling users to ask interesting graph-based queries on these new models. We propose a data model for the Twittersphere that proactively captures Twitter specific interactions and properties in a graph schema. On this model, we introduce a diverse set of microblogging queries, conduct experiments on a large Twitter dataset and investigate the feasibility of running these queries on existing GDBMS. We share our introspection on working with these graph database systems and discuss open problems and opportunities for future research. We detail this work in Chapter 3.

Modeling Evolving Code Dependencies. Next we consider how GDBMS can be used to model software code dependencies capturing the temporal evolution of a codebase. Code dependencies can be naturally modeled as a dependency graph representing call graphs, type graphs and inheritance hierarchies. Frappe [77] is a source code-querying tool developed by Oracle Labs that supports code comprehension tasks for large C/C++ codebases. Current graph database systems do not have in-built support for efficient management of versioned graphs [164, 177]. We seek an efficient and scalable representation of the dependency graph to model, store and query multiple revisions of a codebase. We evaluate the versioned model with dependency graphs generated from a large codebase consisting of around 13 million lines of code. On this large graph, we demonstrate that the model we propose on the graph database is scalable and performant and is able to seamlessly integrate current comprehension workloads in addition to enabling queries across multiple versions. We detail this work in Chapter 4.

1.2.2 Storage and indexing issues in GDBMS

Many graph database systems (Neo4j [140], Sparksee [131], etc.) take different approaches to define memory hierarchies for efficient query processing; we investigate how graph data storage can be improved at a physical level for efficient processing of neighbourhood queries. Modern graph database systems require further investigation into topics such as physical data management to progress towards the level of maturity of relational database systems. We observe that the locality of edges on disk play an important role in graph systems and our motivation stems from optimizing storage of such systems.

Our goal in this study is to optimally assign edge labels to achieve improved disk locality for efficiently answering typical graph queries, without modification to the storage internals of the graph system at hand. We study algorithms to label edges in a way that maximizes the total 'edge consecutiveness' of graph, i.e., maximize the number of sequentially labeled edges to enable sequential storage, thereby increasing the locality of disk accesses. We conduct extensive experiments on real graphs, and show significant benefits of our approaches over baselines in disk I/Os and query times. We also demonstrate a case study of our methods applied in a streaming graph partitioning scenario. We give details of our investigations in Chapter 5.

1.2.3 Query processing in GDBMS

Query processing in GDBMS may be done as part of data modeling itself to demonstrate the feasibility and efficiency of the proposed schema. In the microblogging setting in Chapter 3, we have explored a diverse set of queries facilitated by the proposed graph model. A set of typical microblogging workloads are translated into both declarative and imperative languages supported by different graph database systems. These queries cover use cases such as providing friend recommendations, analysing user influence, and finding co-occurrences and shortest paths between graph nodes. Similarly in the evolving graph model proposed in Chapter 4, we first demonstrate how existing code comprehension can be efficiently performed on the model followed by a discussion of processing queries spanning multiple versions. Edge labeling schemes introduced in Chapter 5 have been tested on a variety of neighbourhood and shortest path queries.

In many real-world graphs, apart from the general attributes attached to the nodes, the nodes may also contain free text. For example, when tweets are modeled as nodes, tweet text may be associated to the node; and for a LinkedIn user, his/her interests or skills can be thought of as free text. In the social network analytics setting above, we note the lack of support for the combination of graph and text-based queries. The motivation for this is to answer queries of the form – who are the users in my network (graph traversal) and who talk about topic A (keyword search)? In Chapter 6 we study how GDBMS can efficiently support the combination of graph and keyword search queries.

This query must take into account the social connectivity as well as the textual similarity of users' topics to the query. Query processing on each dimension has a long-standing history on its own; queries over graph data has been extensively studied and many solutions, have been proposed to speed up various categories of graph queries. On the other hand, from the field of Information Retrieval (IR), searching through a large text corpus has well-defined query processing strategies with indexing schemes. Our focus is on studying the query with a graph database back-end and support seamless integration of keyword search into graph traversals.

Table 1.1: Structure of the thesis along with the focus, with reference to chapters

	Data modeling	Storage and Indexing	Query Processing	Thesis Chapter
Social network analytics	x		x	Chapter 3
Evolving code dependencies	x		x	Chapter 4
Edge labeling		x	x	Chapter 5
Graph-keyword search			x	Chapter 6

Table 1.1 shows the focus of each of the chapters in this thesis in terms of modeling, storage and query processing of graph database systems.

1.3 Publications

Below is a list of publications in chronological order that resulted from this PhD study.

Paper 1. Oshini Goonetilleke, Timos Sellis, Xiuzhen Zhang, and Saket Sathe. "Twitter analytics: A big data management perspective". SIGKDD Explorations, 16(1), pages 11–20, 2014. The content of this paper is included in Chapter 3.

Paper 2. Oshini Goonetilleke, Saket Sathe, Timos Sellis, and Xiuzhen Zhang. "Microblogging queries on graph databases: An introspection". In Proc. of ACM SIGMOD Workshop on Graph Data Management Experiences and System (GRADES), pages 5:1–5:6, 2015. The content of this paper is included in Chapter 3.

Paper 3. Oshini Goonetilleke, Danai Koutra, Timos Sellis, and Kewen Liao. "Edge labeling schemes for graph data". In Proc. of the 29th International Conference on Scientific and Statistical Database Management (SSDBM), pages 12:1–12:12, 2016. The content of this paper is included in Chapter 5.

Paper 4. Oshini Goonetilleke, David Meibusch, and Ben Barham. "Graph data management of evolving dependency graphs for multi-versioned codebases". In Proc. of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 574–583, 2017. The content of this paper is included in Chapter 4.

Paper 5. Oshini Goonetilleke, Danai Koutra, Kewen Liao and Timos Sellis. "Efficient edge labeling schemes for large scale graph data". The content of this article is included in Chapter 5 and is under review in a journal.

Paper 6. Oshini Goonetilleke, Timos Sellis, Xiuzhen Zhang. "Social-Textual query processing on graph database systems". The content of this paper is included in Chapter 6 and is under review in a conference.

1.4 Thesis Overview

The rest of the thesis is organised as follows. In Chapter 2 we begin by introducing the preliminary graph concepts and definitions used throughout this thesis. In the same chapter we describe graph applications and offer an in-depth investigation into graph database management systems. As shown in Table 1.1, our contributions in Social Network analytics, Evolving code dependencies, Edge labeling and Graph Keyword Search are detailed in Chapters 3, 4, 5 and 6 respectively. Finally, Chapter 7 summarises the contributions of this thesis and provides a discussion on future research directions arising from this thesis.

Chapter 2

Background

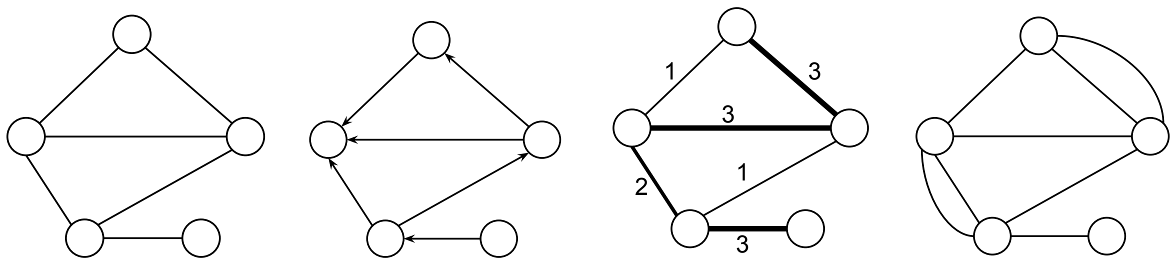
In this thesis we focus on modeling, storage and query processing aspects of graph database management systems that are gaining their momentum in the graph research community. In this chapter we provide the background to the common themes presented in the rest of the thesis. Graphs are a fundamental data structure in computer science; we begin this chapter by a discussion on the theoretical aspects of graph types, properties, and typical graph representations. All the graph database systems we have studied employ a variation of these representations, and provide support for different graph types. We then present an overview of different graphs originating from the real-world and describe interesting applications on them. The GDBMS are then introduced, with the 'property graph' model they support and we compare several options for modeling graph data. Finally, we review some of the popular graph database systems and outline their approaches to graph storage and query processing features.

2.1 Preliminaries

In graph theory, networked data are modeled as *graphs.* A graph G is composed of a set of vertices V (also called as nodes) and a set of edges E (also called as arcs, links) connecting these nodes. The number of nodes $|V|$ and edges $|E|$ is denoted by n and m respectively. Some of the different types of graphs are illustrated in Figure 2.1 and typically include directed, weighted, bipartite and multi-graphs.

Directed and Undirected graphs. A graph is undirected if the edges have unordered pairs of nodes where $(i, j) \in E \Leftrightarrow (j, i) \in E$. If the edges have direction (or the nodes are ordered) then the graph is directed. A directed graph is also known as a digraph.

Weighted graph. In a weighted graph, a numerical *weight* is assigned to each edge. The weight can be a positive or a negative value. A graph that has no weights assigned to the edges is known as an unweighted graph.



(a) Simple graph **(b) Directed graph** **(c) Weighted graph** **(d) Multi-graph**

Figure 2.1: Basic Graph types.

Multi-graph. A graph that has multiple edges between two ordered or unordered pairs of nodes is known as a multi-graph.

Simple graph. An undirected, unweighted graph with no self loops or multi-edges is known as a simple graph.

Attributed graph. Nodes and edges in a graph may have a set of key value pairs attached to them describing non-graph meta data of these entities. The weight of the edge above is considered one of the attributes on the edge.

Labeled graph. A specialized kind of an attribute known as a *label* may also describe a node and/or edge to denote its type or kind. For example, a collaboration network may have authors, venues and publications as node types, while cites, co-authors, published-in can be some of the edge types. Such different types of nodes and edges are identified by a 'label'.

Property graph. A graph that combines the features of a directed, attributed, labeled multi-graph is known as a property graph. We give a formal definition of the property graph in Section 2.4.1.

Bipartite graph. If a vertex set of a graph can be divided into two disjoint sets V_1 and V_2, such that every edge connects a vertex in V_1 to one in V_2. In other words, there cannot be edges connecting the nodes in the same set.

Complete graph. A complete graph is a graph in which each pair of graph vertices is connected by an edge. The number of edges in this graph, m is $n(n-1)/2$.

Subgraph. Given two graphs, $G = (V, E)$ and $G' = (V', E')$, G' is a subgraph of G, (denoted as $G' \subseteq G$) if $V' \subseteq V$ and $E' \subseteq E$.

Next, we describe commonly-used properties of graphs.

Neighbourhood. An *adjacent* vertex of a given vertex v in a graph is a vertex that is connected to v by an edge. The neighbourhood N of a vertex v consists of all vertices adjacent to v. In a directed graph, the distinction between the outgoing (N_{out}) and incoming (N_{in}) neighbourhood can be made.

Degree. The degree of a vertex is the number of edges incident to the vertex and is denoted as $d(v)$. On a directed graph, outgoing and incoming degrees can be distinguished by the number of outgoing neighbuors $d_{out}(v)$ and incoming neighbours $d_{in}(v)$ respectively. A vertex with degree 0 is said to be *isolated* while a vertex with degree 1 is known as a leaf or end node.

Connected Components. For an undirected graph, a connected component is the maximal set of nodes where for every pair of nodes u and v in the subgraph, a path connects them. A graph may have several such subgraphs of connected components, and the one with the highest number of nodes is known as the *largest connected component.*

Strongly, weakly connected Components. A directed graph is said to be strongly connected if there is a directed path connecting any two pairs of nodes in the graph. It is weakly connected if there is an undirected path between any two node pairs.

Clustering Co-efficient. This is a measure of the degree to which the graph nodes tend to cluster together. The global metric for the graph is based on the local clustering coefficient

[206] for each node. Clustering coefficient $C(v)$ of a node v is the fraction of edges between the vertices within the neighbourhood and the maximum number of edges that exist between them. Thus, the average clustering coefficient of the graph is given by $\bar{C}$, where $\bar{C} = \frac{1}{n}\sum_{i=1}^{n} C(v_i)$ and $n = |V|$.

Diameter. Diameter is the length of the maximum shortest path between any two nodes u and v.

Table 2.1 lists some common symbols used throughout the thesis. When applicable, each chapter also defines a set of symbols that are specific to that chapter.

Table 2.1: Table of common graph symbols used in notations

Symbol	Description
G	A graph
V, E	Set of vertices and edges, resp.
$\|V\|$ or n	Number of vertices
$\|E\|$ or m	Number of edges
$N_{in}(v)$	Incoming neighbours of vertex v
$N_{out}(v)$	Outgoing neighbours of vertex v
$d_{in}(v)$	Incoming degree of vertex v
$d_{out}(v)$	Outgoing degree of vertex v
$\bar{d}$	Average node degree
$\bar{c}$	Average Clustering Coefficient of the graph
(v, x)	An outgoing edge of v, or incoming edge of x
$p(u, v)$	Length of the shortest path between nodes u and v

2.2 Graph Representation

In this section we introduce data structures typically used to represent graph data. The graph database systems we study in this thesis encode the graphs as a variation of one of these fundamental structures. We use the example graph in Figure 2.2 to describe different graph representations below.

Edge list. The simplest way to represent raw data in a graph is via an edge list. This contains a list of all edges in the graph denoted as pairs of nodes: (u, v). For an undirected graph, the same edge may appear twice with two entries: (u, v) and (v, u). To simplify different computations, edge lists are generally converted to one of the following data structures.

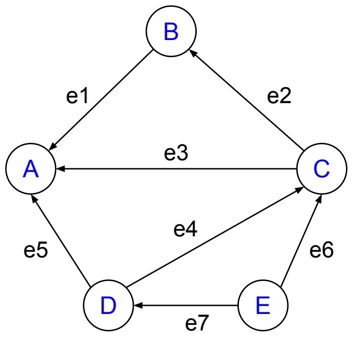


Figure 2.2: An example graph

	A	B	C	D	E
A	0	0	0	0	0
B	1	0	0	0	0
C	1	1	0	0	0
D	1	0	1	0	0
E	0	0	1	1	0

Figure 2.3: Adjacency Matrix representation of the graph

Adjacency Matrix. One way to represent a graph is via an Adjacency Matrix, denoted by an $n \times n$ matrix A. For a simple graph, $A(i, j) = 1$ if $(i, j) \in E$ and 0 otherwise. If the graph is weighted, the $A(i, j)$ position contains the value of the weight of the edge, denoted by $A(i, j) = w$. When the graph is undirected, the matrix is symmetric along the diagonal. If the graph does not contain self loops, the diagonal elements $A(i, i) = 0$. The matrix is said to be *sparse* when most of the elements are zero. Many of the real networks such as social networks and collaboration networks generate sparse matrices. Figure 2.3 is an adjacency matrix representation of the graph in Figure 2.2.

A matrix representation of a graph is efficient in addition and deletion of edges with $O(1)$ complexity and an operation to retrieve one's neighbours would be of complexity $O(n)$. Insertion and deletion of nodes would require restructuring the matrix. Systems favour this representation since many of the graph operations can be converted to a series of matrix multiplication functions. The drawback to this approach is space inefficiency, especially for sparse matrices, requiring quadratic $O(n^2)$ space, which is independent of the actual number of edges in the graph.

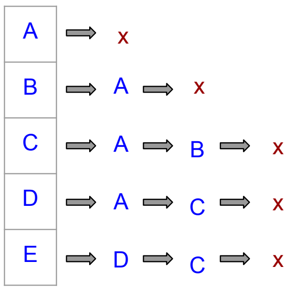


Figure 2.4: Adjacency List representation of the graph

Adjacency list. In this representation, every vertex v maintains a list of its adjacent neighbors. If the graph is undirected and an edge connects node u and v, then the list of u will contain the vertex v and vice versa. Figure 2.4 is an adjacency list representation of the graph in Figure 2.2. An operation to retrieve neighbours of a vertex v is proportional to the degree of the vertex: $O(d(v))$. Node insertion and deletion is much cheaper. Adding an edge would require adding an entry to the source node list performed in $O(1)$, while deleting an edge would require more restructuring of the adjacency list. For a sparse graph, an adjacency list is much more space efficient (compared to an adjacency matrix), requiring $O(V + E)$ space.

	e1	e2	e3	e4	e5	e6	e7
A	-1	0	-1	0	-1	0	0
B	1	-1	0	0	0	0	0
C	0	1	1	-1	0	-1	0
D	0	0	0	1	1	0	-1
E	0	0	0	0	0	1	1

Figure 2.5: Incidence Matrix representation of the graph

Incidence Matrix. Incidence matrix is a variation of the traditional adjacency matrix, with a size proportional to the number of nodes and edges, i.e. space cost of $O(VE)$. Here, the rows and columns of the matrix represent the nodes and the edges respectively. The value 1 in the matrix denotes that a column edge is incident on the row vertex and 0 otherwise. If the graph is directed, the edge type is denoted with a 1 (outgoing) or -1 (incoming). Figure 2.5 is an incidence matrix representation of the graph in Figure 2.2.

Compressed Sparse Row (CSR). CSR is a variation on storing the adjacency matrix A for fast access of rows. It represents A with three 1-dimensional vectors that contain the non-zero values in the matrix, the extent of rows and the column indices. Let NNZ denote the number of non zero values in A — for the graph represented in the matrix shown in Figure 2.6 NNZ=7. The first vector of size NNZ, AN shows the NNZ entries in A, in row-major order. The vector AI, of size $|V|+1$ essentially aggregates the number of NNZ values in each row. The final vector, also of length NNZ, denotes the column indices of the NNZ values in A. This representation allows fast matrix vector multiplications.

	A	B	C	D	E
A	0	0	0	0	0
B	7	0	0	0	0
C	5	4	0	0	0
D	3	0	6	0	0
E	0	0	7	2	0

AN	7	5	4	3	6	7	2
AI	0	0	1	3	5	7	
AJ	0	0	1	0	2	2	3

Figure 2.6: CSR representation of the graph on the left side

In practical applications, there are many variations and modifications of the above structures with the common objective of space-efficient storage and better performance of insert, update and query operations. In some cases, these basic structures are inadequate to represent certain characteristics of a graph – for example, a multi-graph, or properties attached to a node/edge cannot be represented in an adjacency matrix and thus require additional structures to capture them.

2.3 Real-world Graphs and Applications

In this section we review some common types of networks and interesting use cases and applications from the real-world.

Social Networks. Social networks in the real-world come in different flavours, e.g.: Facebook, an online social media and networking platform; Twitter, an online news, networking and microblogging platform; LinkedIn, a platform specialized for business and professional networking, and FourSquare, a location-based mobile social network. The kind of analyses and applications on these platforms are equally diverse with platform-specific goals and challenges. In the simplest form, a social network is made up of users as nodes and edges representing different semantics: a mutual, undirected connection for LinkedIn and Facebook, and a one-way directed *follows* connection to subscribe to user content in the case of Twitter.

Recommendation is a common goal in many of the social networks. Recommended items can be either content or other users based on interests/connections of existing users. The suggestive ability of these platforms is expected to be both efficient and effective; demanding that the right content is suggested promptly. Influence analysis is another useful application in many social networks that aim to understand a set of users who are most likely able to influence a large proportion of the network for the purposes of content propagation.

Review Networks. Users can post reviews about different types of content they consume; for example, product reviews in Amazon and Ebay, and movie reviews in platforms such as Netflix. In the simplest form, a review network can be represented as a bipartite graph (ref. Section 2.1) where node types are Users and Items (e.g. books, movies, products), and edges represent a review made by a user on some items. Collaborative filtering or recommender systems is the prominent application of these types of networks that aims to predict the likelihood of a purchase based on preferences of other users so that a recommendation can be made.

Program Dependency Graphs. Graphs have been used to represent data and control flow dependencies among software entities. For example, a 'call graph' is a type of control flow graph where nodes are subroutines in a program and a relationship (u, v) denote that procedure u calls procedure v. This intermediate representation of a software program facilitates a variety of useful applications including code optimizations, bug identification, defect prediction and program analysis.

Source code analysis and comprehension is one application of program dependency graphs. Depending on the use case, the graph captures program dependencies in varied precision and granularity corresponding to different levels of abstractions. In these graphs, queries such as impact estimation are performed to understand and explore the affected regions of the code if one seed function is changed. The dependency information stored enables comprehension of paths and its transitive effects on code repositories.

Evidently, the above discussion is a non-exhaustive list of graph types and use cases. There are many more types of networks such as knowledge graphs, biological networks, web graphs and transport networks running a diverse set of application scenarios.

2.4 Graph Data Management Systems

The advent of the scale and complexity of graph data in recent years has led to a demand for more sophisticated systems that can store, manage and query graph data. We introduce the 'property graph model' that better characterises real-world graphs, acknowledging labels and properties on both nodes and edges. Then we illustrate several options for modeling graph data and finally present the native graph database systems that simplify logical representation and facilitate efficient graph traversals.

2.4.1 Property Graph Data Model

Many of the graphs from the real-world require more information other than the plain nodes and edges. A property graph essentially consists of nodes, edges, labels and properties on both nodes and edges. A property graph demands no fixed schema and as such can contain any number of attributes. An example of a property graph from a hypothetical academic network is shown in Figure 2.7. This example is used to explain concepts throughout this chapter.

The property graph consists of four node types (labels) `author`, `publication`, publication `venue` and author `affiliation`. Among these nodes, five relationship types (labels) exist: `author_of`, `published`, `cites`, `affiliated` and `follow`. Additional key-value pairs of properties further explain the nodes and edges in the graph: for example, the `order` on the `author_of` edge, denotes the position at which the author appears in the publication. A property graph is generally queried starting at a given node and traversing the graph in either a depth-first or breadth-first direction, exploring the neighbourhood.

Definition 1 (Property graph)**.** A property graph G can be formally defined as $G = (V, E, L, A)$ with added characteristics L and A to a simple graph. V is a set of vertices and each edge in $E \subseteq (V \times V)$ connects two vertices. L is a set of labels, and A is a set of attributes of key-value pairs. A label is given to each vertex and edge where $L = \{ (o, l) \mid o \in (V \cup E),\ l \in \Sigma^* \}$ and Σ is a finite alphabet. The attributes set A contain key-value pairs where $A_i = \{(k_1, v_1), (k_2, v_2), ...\}$, assigning a key k_i to a value $v_i \in D$ where D may represent a valid data type such as int, boolean, string etc.

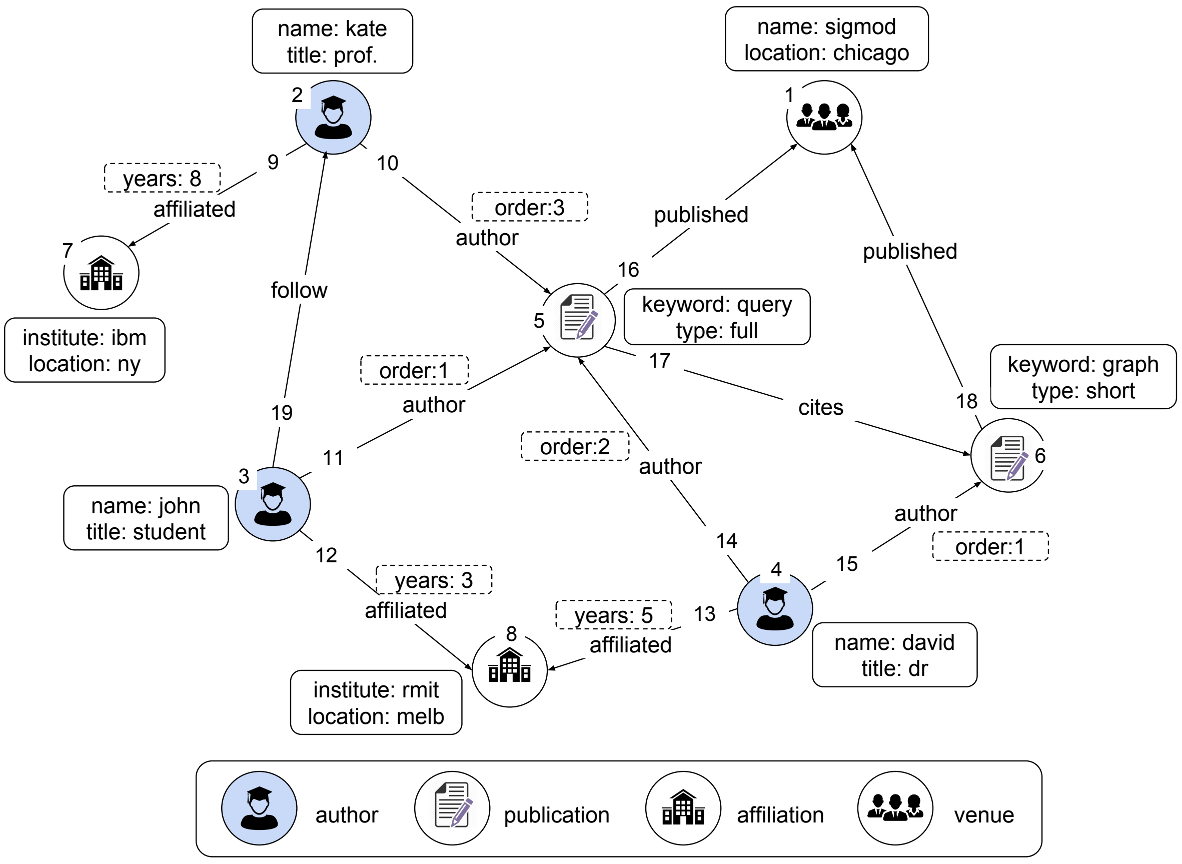


Figure 2.7: Property graph representation of academic network

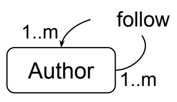
Modeling considerations. It must be noted there are several alternatives to modeling the above academic network. The affiliations for each author may be modeled as a property of the author node. Since the `affiliated` edge contains a property, this consequently becomes a property on the author node as well. Modeling decisions such as these are primarily driven by the types of queries that are executed on them. For example, if a certain workload requires aggregating all authors of a particular affiliation, and if we model the affiliation as part of the `author` node, this would require unnecessary traversal of all authors, filtering on the property. On the other hand, if the affiliation is modeled as a node, the same query translates to a simple 1-step in-neighbours of the `affiliation`.

2.4.2 Options for modeling graph data

The property graph that we described above can be modeled using either relational, RDF or a native graph model. To interact with the graph in each of these approaches, SQL, SPARQL and custom query languages are employed. Columnar- and object- oriented data models are also used in some representations. Next, we describe the logical representation of the graph model, indexing mechanisms and query system in each of these approaches.

2.4.2.1 Relational model with SQL queries

Let us consider the academic social network created in the above property graph example by way of `follow` edges among authors. One way to capture the information in the relational model is via two tables – one to hold the information of the authors of the network (`Author` table) and another to store the relationships among them (`Friendship` table) as shown in Figure 2.8. At the top, the Authors and their friendships are modeled in a traditional ER-diagram. Any additional properties on nodes and edges can also be held in each of the tables. Similarly, the other types of nodes and edges in Figure 2.7 are typically modeled in different tables.



Author		
AuthorId	Name	Title
1	Eric	Student
2	Kate	Professor
3	John	Student
...	...	...
6	Andrea	Dr.

Friendship	
from_Id	to_Id
1	2
1	3
1	4
2	5
2	3
3	6
3	2
4	2
5	1

Figure 2.8: Relational representation of an academic social network

Let us consider a few integral graph-based queries on the relational model and observe how they can be expressed and processed in SQL. A simple query to find details of Eric's direct neighbourhood of followers can be written as follows:

```
SELECT u1.Name, u1.Title
FROM Author u1 JOIN Friendship f
ON u1.AuthorId = f.to_id
JOIN Author u2
ON u2.AuthorId = f.from_id
WHERE u2.Name = ‘Eric’
```

This query is unnecessarily complicated. We could create an index on the `from_id` to speed up the query. The query in the reverse direction (assuming a non-reciprocal friendship) to find the details of Authors who are following ‘Eric’ may require having to go through all the rows in the Friendship table. Retrieving friends at greater depths becomes more complex both in terms of query expression and of processing which involve several recursive joins. For example, a query to find Eric’s friends-of-friends can be written as follows:

```
SELECT u1.Name, u1.Title
FROM Author u1 JOIN Friendship f1
ON u1.AuthorId = f1.from_id
JOIN Friendship f2
ON f1.from_id = f2.to_id
JOIN Author u2
ON u2.AuthorId = f2.to_id
WHERE u1.Name = ‘Eric’ AND f2.to_id <> u1.AuthorId
```

This fundamental query in any graph traversal operation is already computationally expensive to retrieve a 2-hop neighbourhood. Thus we need to explore representations that implicitly model graph relationships.

2.4.2.2 RDF model with SPARQL queries

With the origin of the semantic web, the Resource Description Framework (RDF) has been used to represent linked data. RDF models linked data and stores ‘triplets’; a *subject-predicate-object*. The *subject* denotes the resource, and the *predicate* denotes traits or aspects of the resource and expresses a relationship between the *subject* and the *object*. The resources are generally web sources, represented as nodes in the RDF graph by a unique URI. SPARQL [157] is the official W3C standard on querying RDF graphs. SPARQL query language is specialized for efficient pattern-matching tasks on RDF data.

From the property graph shown in Figure 2.7, the following triplets can be generated representing different semantics.

1. An attribute on a node: kate (subject) has the title (predicate) professor (literal object).
2. A relationship to a node: kate (subject) is affiliated to (predicate) ibm (resource object).

Specifically, the node data (triplet 1) and the graph topology (triplet 2) are both modeled the same way; thus RDF cannot distinguish between these two triplets. As a result, we cannot model attributes on the edges or multiple edges among the same pair of nodes. Also, since relationship instances cannot be qualified, traversal queries become tedious. There are several work-arounds to these issues, but they come at the cost of complex queries and/or modeling decisions that are counter intuitive. Path queries and reachability expressions were only introduced to the SPARQL specification recently (v.1.1), and have not yet been widely deployed in triplestores. The reason for this could also be that traversals are not fundamentally common and are not the focus of the RDF/SPARQL domain.

Triplestores are used to manage and provide the core infrastructure for RDF triplets and are queried using SPARQL. Some examples are AllegroGraph [4], Virtuoso [202], GraphDB [72] and RDF-3x [141]. Many of these stores are capable of accommodating triplets in the scale of billions. The key feature of these triplestores is to perform inferencing [203] for the SPARQL queries which discover new relationships based on the existing model and a set of rules.

2.4.2.3 Native graph model with custom query languages

While relational and the RDF data models are good at managing and querying one type of data, they fall short in management of connected data. In most of the native graph stores, the graph is generally represented with some variation to the methods described in Section 2.2. Nodes and edges are first-class citizens, and the layout of the graph on disk is optimized for fast graph traversals scaling to large graphs.

'Index-free adjacency' is a concept popularised by many of these native graph systems. This refers to the feature that the adjacent elements of a node can be retrieved efficiently from disk without having to look-up additional indexes. In other words, index-free adjacency enables looking up neighbours in constant $O(1)$ time and is only dependent on the edges emanating from the source vertex. In the relational model, to retrieve neighbours, it requires $O(log\, n)$ cost looking up a global index such as a B-tree, which is dependent on the total number of nodes n in the graph. A native graph store essentially makes the graph structure explicit where each vertex serve as a mini-index of its adjacent elements [167]. Like SQL for the relational model, there is no ubiquitous query language across all native graph stores. As a result, these graph database systems either expose an API with a set of primitives or makes use of a custom query language. In the next section, we discuss several systems that support native graph storage.

2.4.3 Graph Database Systems

For many years, relational database management system (RDBMS) technologies have been the de-facto standard in storage and management of a variety of data types. With the advent of the Web 2.0, the data that has been generated have become more complex and much bigger in size. The complexity and magnitude of the data were challenging existing technologies to meet growing and different requirements for data management and query processing. These new technologies were termed NoSQL (Not-only-SQL) and included broad categories catering to various types of data, namely, key-value stores, column-family stores, document-oriented databases and graph databases.

Until recently, many of the applications which were built using graph data have been stored and managed in non-graph data models such as the relational model. As discussed in the previous section, this resulted in counterintuitive approaches for understanding graph data and inefficient query processing schemes. Graph Database Management Systems have been proposed to classify a group of technologies that are explicitly storing relationships where the persistent graph is very similar to the logical data model they represent. We review some popular graph database systems and discuss their data model, storage mechanisms and methods of query processing.

2.4.3.1 Neo4j

Neo4j [140] is a graph database system developed by Neo Technologies described as an ACID-compliant transactional database. It is an open source platform supporting the property graph model running on the Java Virtual Machine (JVM). Like many other NoSQL databases, Neo4j is schema-free, which means that there need not be a pre-defined schema on the node, relationship or the attributes.

The high-level architecture of Neo4j is shown in Figure 2.9. Users can query the database via one of the three APIs: two imperative interfaces (Traversal and Core) or the API of the declarative query language, Cypher. Users are also able to interact through several application architectures, namely, embedded, server and server with extensions [166]. At the operating

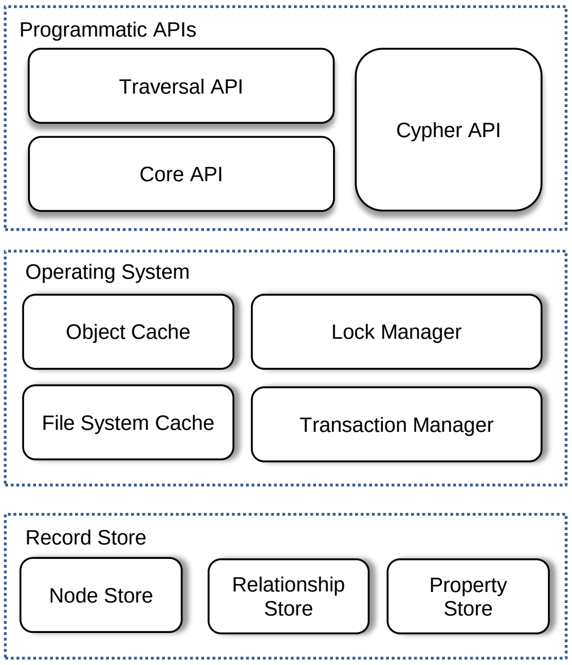


Figure 2.9: High-level architecture of Neo4j.

system level, a file system cache takes care of regions in the stored files while the object cache is optimized for traversals with node and relationship objects.

Node (14 bytes)

in use flag (1)	next relationshipID (4)	next propertyID (4)	labels (5)

Relationship (33 bytes)

in use flag (1)	source node (4)	target node (4)	type (4)	source node's previous relationship ID (4)	source node's next relationship ID (4)	target node's previous relationship ID (4)	target node's next relationship ID (4)	next property ID (4)

Figure 2.10: Neo4j layout of a storage record of nodes and relationships.

Storage. In Neo4j, graphs are stored as linked lists of fixed size records. In physical storage, separate records are maintained for nodes, relationships, properties and types. The layout of nodes and relationships are shown in Figure 2.10. Each record in the node store is of fixed size, to enable fast lookups on disk. An entity ID, multiplied by the record size, immediately gives the offset in the node stores. A node record holds information about the ID of its first relationship, ID of its first property followed by five bytes for information on its type. Each relationship of fixed size contains pointers to source and target nodes, a pointer to the relationship type (held in a type store), next and previous pointers for each source and target node, and finally a pointer to the first property of the relationship. Representing both next and previous nodes of a relationship enables traversal in either direction.

The property store is persistent as a key-value pair where the key is the name of the property as a string, and the value can be a primitive type, string or an array. Each property block can contain up to four property sub-blocks followed by a pointer to the next property [166]. Each property block holds three values: (a) the property type, (b) pointer to the property index file containing the property name, and (c) the property value—if the type is primitive, the value is held in-line, otherwise points to a dynamic store record.

Query Processing. The Core API in Neo4j is written in Java and gives users access to low level functions to interact with the database. Users are expected to have an understanding of the domain in order to fine-tune the query and achieve good performance. Traversal API is built on top of Core API enabling navigation in the graph structure. Retrieving the Authors of publication 5 (ref. Figure 2.7), using the Core API can be written as follows.

```
Node pub = graphDB.getNodeById(5);
Iterable<Relationship> rels = pub.getRelationships(Direction.INCOMING, AUTHOR_OF);
for (Relationship rel : rels){
   Node authorNode = rel.getStartNode();
   String authorName = authorNode.getProperties("name");
}
```

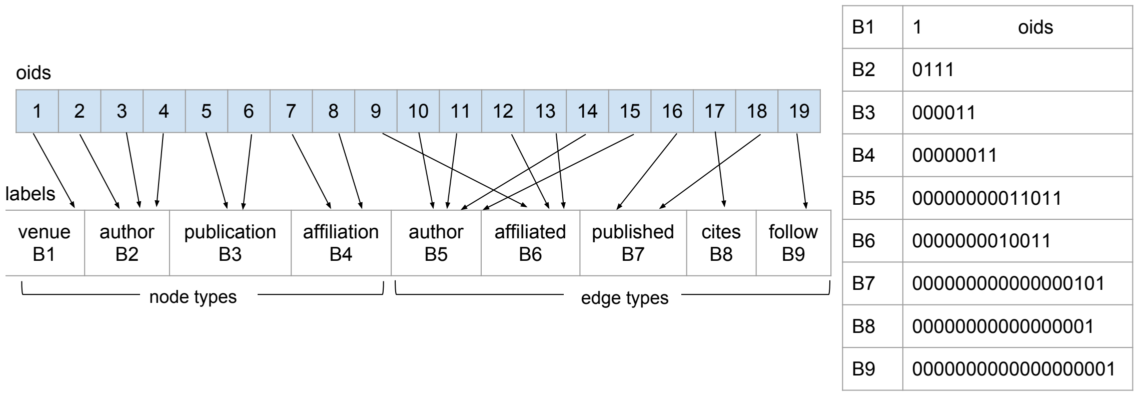
Using the Cypher query language, the same query can be written as follows.

```
MATCH (p:PUBLICATION {id:5})<-[:AUTHOR_OF]-(a:AUTHOR))
RETURN a.name
```

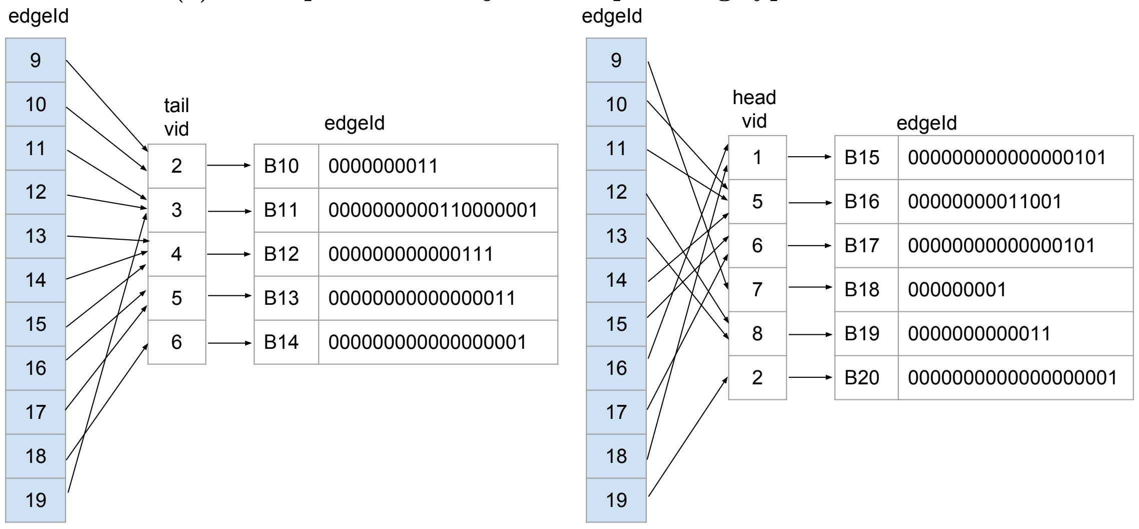
As with any declarative language, Cypher presents a set of primitives to express the information needs, making it less verbose and thus the preferred method of users.

2.4.3.2 Sparksee

Sparksee, formerly known as DEX [131], is a graph database system developed by Sparsity technologies written in C++ with ACID compliance and transaction support. Sparksee graph representation uses a combination of bitmap based data structures. It also supports directed, labeled attributed multi-graphs as the graph model. The motivation behind using a bitmap representation in Sparksee is two-fold [131]: first, bitmaps are able to hold large amounts of information in reduced amount of memory. Second, graph queries can be converted to a series of logic (bit) operations that can be performed very efficiently. Internally, each vertex $v \in V$ and edge $e \in E$ is identified by a unique object identifier, $\texttt{ID} \in \mathbb{Z}^*$. As with many graph systems, an internal id generator assigns a unique ID to each new vertex or edge when they are created and in the order they are inserted.



(a) Bitmaps for the Object Group storing type information



(b) Bitmaps for the Relationships Group with tail and head nodes

Figure 2.11: Property graph in a bitmap-based representation

Storage. Three groups of bitmaps capture the structure of an attributed multi-graph, namely the Objects, Relationship and Attribute groups. The *objects* group stores the type information for each node and edge; the *relationship* group consists of connectivity information recording the heads and tails for each edge, and the *attributes* group stores properties for both vertices and edges. The bitmap representation of the Object and Relationship group of the example property graph (Figure 2.7) is shown in Figure 2.11a and Figure 2.11b respectively. The network is stored similarly to an incident matrix described in Section 2.2.

Each group contains a mapping between an `ID` to a *value* and then a mapping between a *value* to a bitmap of `ID`s which contains the *value* (`ID` $\rightarrow$ *value* $\rightarrow$ bitmap). Each *oid* in the Objects group corresponds to a *value* of some type/label for nodes and edges. For example, all **`published`** edge types are denoted by the bitmap `B7`, where the 16th and 18th positions of the bitmap are set to one, which indicate the edge ids of that type. Similarly, the connections are stored in the relationship group— since node id 5 is the head of three edges, 10, 11 and 14, the `B16` bitmap in the right of Figure 2.11b, marks aces on the 10th, 11th and 14th positions. In terms of disk storage, a word-aligned scheme is used which can compress a long sequence of zeros. The mappings between ids, values and bitmaps are stored in a B+ tree for efficient retrieval [131].

Query Processing. Graph-based operations can be transformed into series of efficient bit operations. For example, a query to retrieve the authors of publication 5 can be expressed as a combination of set operations:

```
{lookup(TAILS, x) |
x ∈ objects( HEAD, objects(ID, 5)) ∩ objects( LABELS, ‘Author’))}
= {2, 3, 4}
```

The Sparksee API exposes a set of succinct functions for the users to manipulate more intuitively than at the set level. The **`neighbors`** operation below is used to navigate the neighbour nodes of a given identifier while an **`explore`** operation allows to navigate the edges incident to a given node. These two primary navigation operations can be used to retrieve neighbours with additional constraints on direction and edge type. The same query to retrieve authors of publication 5 can be written in the API as:

```
long input = g.findObject(attributeID, attributeVal);
int edgeType = g.findType("AUTHOR_OF");
Objects authorList = g.neighbors(input, edgeType, EdgesDirection.INGOING);
```

Further, **`TraversalBFS`** and **`TraversalDFS`** operations facilitate breadth-first and depth-first search and several filters on nodes and restrictions on the path can be specified on them.

2.4.3.3 Titan

Titan [12] is an open source distributed graph database built on top of Apache Cassandra. The nodes and edges may be distributed and replicated across a cluster of machines and provide support for thousands of concurrent users. Titan also allows other storage back-ends Apache HBase and Oracle BerkeleyDB. Titan has support for transactions and is ACID compliant.

Storage. Titan leverages an adjacency list representation on disk, co-locating a node with its adjacent edges. Depending on the choice for the back-end, exact storage method varies. With a column-oriented [1] back-end like Cassandra, the adjacency list is stored in a single column family where the row key is a vertex id. Each property and edge are stored in one column [25] while direction and labels are stored as a column prefix. The Blueprints framework (Apache TinkerPop 3.x[1] since Aug. 2017) on which Titan is built supports vertex queries. It ensures that the edges incident on a node are indexed by type, enabling efficient retrieval of edges of that type. Vertex-centric indexes [12] on the other hand are made possible by the underlying storage back-end for fine-grained retrieval of the vertex's incident edges.

Query Processing. Native integration with the TinkerPop graph stack gives access to the Gremlin query language. Gremlin is a path-oriented language which succinctly expresses complex graph traversals and mutation operations [73]. It is procedural, allowing the programmer to express queries as a set of steps or 'pipes'. A query to retrieve the authors of the publication 5 (ref. Figure 2.7) can be written in Gremlin as follows.

```
g.V.has('id', '5')
.in('author')
.name
```

Operations such as shortest paths traversals can be accomplished with a looping structure with constraints for the maximum depth of the path.

2.4.3.4 Graph database system summary

Apart from the graph database systems mentioned above, there are also other distributed systems such as InfiniteGraph [146]. OrientDB [29] is a graph database that also provides support for key-value and document object models. Databases such as HyperGraphDB [83] are specialized for directed hyper-graphs.

[1]Apache TinkerPop is an open source, vendor-agnostic, graph computing framework for both graph databases and graph analytic systems

Table 2.2: Comparison of features in relational, RDF and native graph stores

	Data Model	Query System	Consistency	Back-end	Language	In-memory
Neo4j [140]	property graph	Cypher	ACID	Doubly lists	Java	No
Titan [12]	property graph	Gremlin	ACID, Eventual	Cassandra, Hbase	Java	No
Sparksee [131]	property graph	API	ACID	Bitmaps	C++	No
OrientDB [29]	multi-model	Gremlin, SQL	ACID	Custom	Java	Multiple
InfiniteGraph [146]	property-graph	API, Gremlin	Flexible	Objectivity/DB	Java, C++	No
HyperGraphDB [83]	Hypergraph	API	MVCC	Berkeley DB	Java	Multiple
AllegroGraph [4]	RDF, XML	SPARQL, Prolog	ACID	Custom	C++, Lisp	No
Virtuoso [202]	RDF, relational	SQL, SPARQL, ..	ACID	Object-relational	C	Yes
RDF-3X [141]	RDF	SPARQL	Read-committed	RISC-style	-	No
Filament [62]	relational	API	-	PostgreSQL	Java	Yes
SQLGraph [190]	relational	SQL	ACID	Relational,JSON	Java	No

We summarise different features of relational, RDF and native graph stores in Table 2.2. We note the data model in these systems and provided query languages. Although not explicitly mentioned all systems expose an API to interact with the database. It can be observed that many of these systems provide support for transactions and different levels of consistency. Multiple in the 'in-memory' column denote that both in-memory and other disk-based modes are available.

2.4.4 Graph processing systems

To complete our discussion, we also distinguish the category of systems that have been developed primarily for *processing* very large graphs rather than for graph management. The graph database systems we presented above are more suitable for OLTP-like workloads while graph-processing systems are more focused on large-scale, off-line, analytical workloads. Typical workloads are long-running, such as machine learning tasks, statistical inference and collaborative filtering. Many of these systems have been developed over distributed storage to enable iterative and batch processing of the entire graph data.

Pegasus [95] and GBase [93] are graph mining platforms on which the graph is internally represented as a matrix and the matrix-based operations are run parallel on Hadoop / MapReduce. Due to limitations in data-parallel systems for graph algorithms, the concept of a 'graph-parallel' vertex-centric computation has been introduced. Computation that 'thinks like a vertex', extending the Bulk Synchronous Parallel (BSP) model [199], has been made popular by Google's Pregel [126] implementation. Consequently, many graph-parallel solutions have been developed in a distributed setting [65, 67, 174, 213] and on a single PC [110, 170, 76] with proven performance over its data-parallel counterparts.

2.5 Summary

In this chapter we first introduced graph preliminaries related to our thesis, including graph types, graph properties and different graph representations. We discussed types of real-world graphs and their applications such as social networks. We introduced the property graph model that can describe graphs, not only with nodes and edges of a single type, but also allowing different types of nodes and edges, and attributes on them. We comprehensively reviewed graph data management systems that can model, store and query property graphs. The above form the basis and background for our work described next.

In Chapter 3 we explore how graph database systems can be used in a social network setting and study a series of queries relevant for a microblogging scenario. In Chapter 4 we examine a code comprehension tool that captures dependency graphs and models them in a native graph database. We extend the capabilities of a system to enable versioning of dependency graphs when the underlying codebase changes over time. In Chapter 5 we investigate issues around storage of graph systems and propose edge re-labeling techniques to increase disk locality and thus improve query performance. In Chapter 6 we investigate how a textual search can be combined with graph traversals, integrating these dimensions in a generic graph database system.

Chapter 3

Graph Database Systems for Microblogging Analytics

With the inception of different types of social networks, a growing number of applications consume data collected from various Microblogging platforms. Twitter is one such platform where a myriad of research efforts have emerged studying different aspects of the Twittersphere. Each study exploits its own tools and mechanisms to capture, store, query and analyse Twitter data. Inevitably, frameworks have been developed to replace this ad-hoc exploration with a more structured and methodological form of querying and analysis. An analysis framework typically involves the following major components: data collection, pre-processing, data modeling and a language for querying tweets.

In this chapter we highlight the need for graph-based data models for Microblogging analytics by reviewing existing approaches. Addressing limitations of existing models, we propose a data model for the Twittersphere that captures different kinds of Twitter-specific interactions. We examine the feasibility of running analytical queries using graph database systems and offer empirical analysis of the performance of the proposed approach. Accordingly we observe how well graph database systems are able to drive the overall data management goals of a Twitter framework. In particular, we share our experiences on executing a wide variety of microblogging queries on two popular graph databases: Neo4j and Sparksee. The queries are executed on a large, real Twitter graph data set comprising nearly 50 million nodes and 326 million edges.

3.1 Introduction

The massive growth of data generated from social media sources has resulted in a growing interest on efficient and effective means of collecting, analysing and querying large volumes of social data. In particular, online social networking and microblogging platform Twitter has seen exponential growth in its user base since its inception in 2006, with now over 200 million monthly active users producing 500 million tweets daily[1]. A wide research community has been established since then with the hope of understanding interactions on Twitter. For example, studies have been conducted in many domains exploring different perspectives of understanding human behaviour. Prior research has focused on a variety of topics including opinion mining [15, 18, 84], event detection [113, 171, 222], spread of pandemics [40, 152, 181], celebrity engagement [212] and analysis of political discourse [45, 89, 196]. These types of efforts have enabled researchers to understand interactions on Twitter related to the fields of journalism, education, marketing, disaster relief etc.

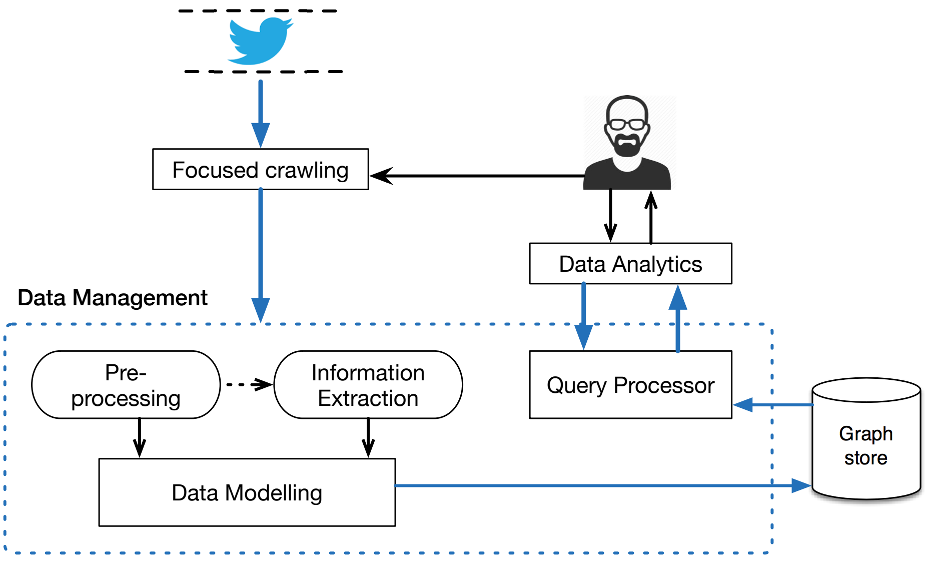


Figure 3.1: An abstraction of a Twitter data management platform

The systems that perform analysis in the context of these interactions typically involve the following major components: data collection, data management and data analytics. Here, data management comprises information extraction, pre-processing, data modeling and query processing components. Figure 3.1 shows a block diagram of such a system and depicts inter-

[1]http://tnw.to/s0n9u

actions among various components. Until now, there has been a significant amount of prior research around improving each of the components shown in Figure 3.1, but to the best of our knowledge, there have been no frameworks that propose a unified approach to Twitter data management that seamlessly integrates all these components. Following these observations, in the first part of this chapter we extensively survey the techniques that have been proposed for realising each of the components shown in Figure 3.1, summarise their drawbacks and describe the motivation for the need and challenges of a unified platform for managing Twitter data.

In our survey of existing literature, we observe ways in which researchers have tried to develop general platforms to provide a repeatable foundation for Twitter data analytics. We show the elements of our survey in Figure 3.2, primarily focusing on the following key elements.

- **Data Collection.** In Section 3.2 we describe mechanisms and tools that focus primarily on facilitating the initial data acquisition phase. These tools systematically capture the data using any of the Twitter's publicly accessible APIs.
- **Data management frameworks.** In addition to providing a module for crawling tweets, these frameworks provide support for pre-processing, information extraction and/or visualization capabilities. In Section 3.3 we review existing data management frameworks.
- **Languages for querying tweets.** A growing body of literature proposes declarative query languages as a mechanism of extracting structured information from tweets. Languages present end-users with a set of primitives beneficial in exploring the Twittersphere in different dimensions. In Section 3.4 we investigate declarative languages and similar systems developed for querying a variety of tweet properties.

As shown in Figure 3.2, for each of the components we make note of the data model and storage systems in use, dimensions explored and the types of analysis conducted with Twitter data. Armed with these observations, in Section 3.5 we consolidate the requirements of a data management platform for Twitter and highlight the importance of a graph-based approach to data management. As graph database management system is a good conceptual fit for our proposed data model; we conduct experiments to test the feasibility of running a series of interesting microblogging queries on them. Section 3.6 discusses preliminaries on the graph schema, query abilities of the tested graph systems and the pre-processing of the data. For the databases we do a feasibility analysis (Section 3.7) reporting on data ingestion and query processing. Finally, in Section 3.8 we discuss our findings on these two graph databases and propose improvements on them.

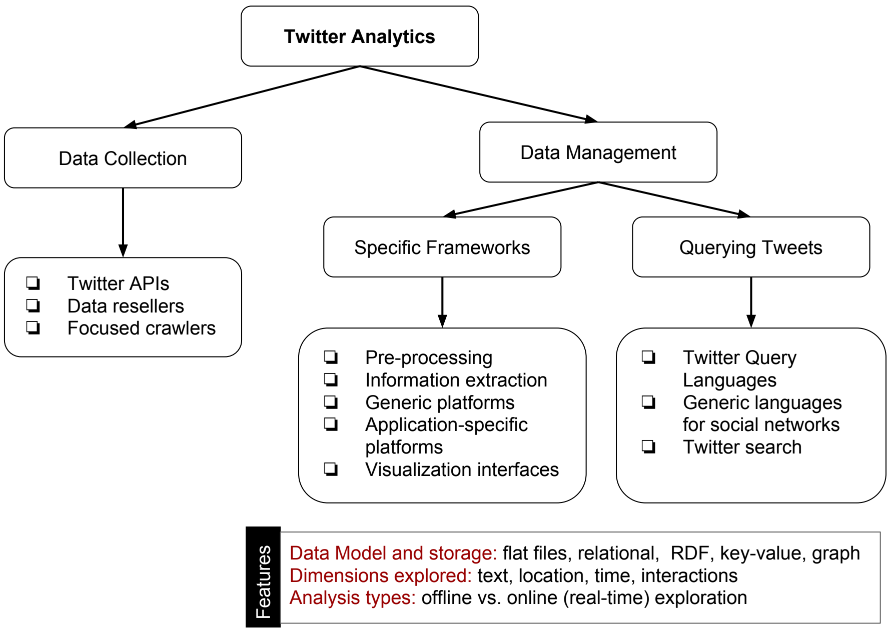

Twitter Analytics
Data Collection
Data Management
Specific Frameworks
Querying Tweets
Twitter APIs
Data resellers
Focused crawlers
Pre-processing
Information extraction
Generic platforms
Application-specific platforms
Visualization interfaces
Twitter Query Languages
Generic languages for social networks
Twitter search
Features
Data Model and storage: flat files, relational, RDF, key-value, graph
Dimensions explored: text, location, time, interactions
Analysis types: offline vs. online (real-time) exploration

Figure 3.2: Elements of the survey on Twitter analytics

Our contributions of this work can be summarised as follows.

- **Extensive Survey:** We conduct the first extensive review on existing approaches to primarily collect, represent, manage, and query twitter data. With these observations we consolidate the requirements of an integrated data management framework for Twitter.
- **Data Model and Queries:** We propose a data model for the Twittersphere that pro-actively captures Twitter specific interactions and properties. In this model, we suggest microblogging queries useful in a variety of application scenarios such as recommendation, co-occurrence and influence detection.
- **Experiments:** We conduct experiments on a large Twitter dataset, and examine how queries perform on existing GDBMS that use graph structures to represent data.
- **Lessons Learned:** We share our introspection on working with these graph database systems and discuss open problems and opportunities for future research.

3.2 Data Collection

Researchers have several options when choosing an API for data collection, i.e. the Search, Streaming and the REST API. Each API has varying capabilities with respect to the type and the amount of information that can be retrieved. The Search API is dedicated to running searches against an index of recent tweets. A request to the search API returns a collection of relevant tweets matching a user query. The Streaming API provides a stream to continuously capture public tweets where parameters are provided to filter the results of the stream by hashtags, keywords, twitter user ids, usernames or geographic regions. The REST API can be used to retrieve a fraction of the most recent tweets published by a Twitter user. All three APIs limit the number of requests within a time window and rate-limits are posed at the user and application levels. Responses obtained from Twitter API are generally in the JSON format. Third party libraries[2] are available in many programming languages for accessing the Twitter API. These libraries provide wrappers and provide methods for authentication and other functions to conveniently access the API.

Publicly available APIs do not guarantee complete coverage of the data for a given query as the feeds are not designed for enterprise access. For example, the streaming API only provides a random sample of 1% (known as the *Spritzer* stream) of the public Twitter stream in real-time. Applications where this rate limitation is too restrictive can rely on Twitter's enterprise APIs[3]. Alternatively, third party resellers like DataSift, KeyHole or TweetReach[4] provide various levels of access to the full collection of tweets, known as the *Twitter FireHose*. For a cost, resellers can provide access to archives of historical data, real-time streaming data, or both. It is mostly corporate businesses who opt for such alternatives to gain insights into their consumer and competitor patterns.

In order to obtain a dataset sufficient for an analysis task, it is necessary to efficiently query the respective API methods, within the bounds of imposed rate limits. Creating the users' social graph for a community of interest requires additional modules that crawl user accounts iteratively. Large crawls with more complete coverage were made possible with the use of whitelisted accounts [33, 109] and using the computation power of cloud computing [145]. Due to Twitter's current policy, whitelisted accounts are discontinued and are no longer an option as a means of large data collection. Distributed systems have been developed [22, 109] to make continuously running, large scale crawls feasible.

[2]https://developer.twitter.com/en/docs/developer-utilities/twitter-libraries
[3]https://developer.twitter.com/en/enterprise
[4]http://datasift.com/, http://keyhole.co, https://tweetreach.com

3.3 Data Management Frameworks

3.3.1 Focused Crawlers

The focus in studies such as TwitterEcho [22] and Byun *et al.*[28] is data collection, where the primary contributions are driven by crawling strategies for effective retrieval and better coverage. TwitterEcho describes an open source distributed crawler for Twitter. Data can be collected from a focused community of interest and it adapts a centralized distributed architecture in which multiple thin clients are deployed to create a scalable system. TwitterEcho devises a user expansion strategy in which the user's follower lists are crawled iteratively using the REST API. Byun *et al.*[28] proposed a rule-based data collection tool for Twitter with a focus on analysing sentiment of Twitter messages. It is a java-based open source tool developed using the Drools[5] rule engine. They stressed the importance of an automated data collector that also filters out unnecessary data such as spam messages.

3.3.2 Pre-processing and Information Extraction

Apart from data collection, several frameworks implement methods to perform extensive pre-processing and information extraction of the tweets. Pre-processing tasks of TrendMiner [156] take into account the challenges posed by the noisy genre of tweets. Tokenization, stemming and part-of-speech (POS) tagging are some of the text processing tasks that better prepare tweets for analysis. The platform provides separate built-in modules to extract information such as location, language, sentiment and named entities that are deemed very useful in data analytics. The creation of a pipeline of these tools allows the data analyst to extend and reuse each component with relative ease.

TwitIE [24] is another open-source information extraction NLP pipeline customized for microblog texts. For the purpose of information extraction (IE), the general purpose IE pipeline ANNIE is used. It consists of components such as sentence splitter, POS tagger and gazetteer lists (for location prediction). Each step of the pipeline addresses drawbacks in traditional NLP systems by addressing the inherent challenges in microblog text. As a result, individual components of ANNIE are customized. Language identification, tokenisation, normalization, POS tagging and named entity recognition are performed with each module reporting accuracy of tweets.

[5]http://drools.jboss.org/

Baldwin [14] presented a system designed for event detection on Twitter with functionality for pre-processing. JSON results returned by the Streaming API are parsed and piped through language filtering and lexical normalisation components. Messages that do not have location information are geo-located, using probabilistic models since it is a critical issue in identifying where an event occurs. Information extraction modules require knowledge from external sources and are generally more expensive tasks than language processing. Platforms that support real-time analysis [14, 222] require processing tasks to be conducted on-the-fly where the speed of the underlying algorithms is a crucial consideration.

3.3.3 Generic Platforms

There are several proposals in which researchers have tried to develop generic platforms to provide a repeatable foundation for Twitter data analytics. Twitter Zombie [20] is a platform to unify the data gathering and analysis methods by presenting a candidate architecture and methodological approach for examining specific parts of the Twittersphere. It outlines architecture for standard capture, transformation and analysis of Twitter interactions using the Twitter's Search API. This tool is designed to gather data from Twitter by executing a series of independent search jobs on a continual basis and the collected tweets and their metadata is stored in a RDBMS. One of the interesting features of TwitterZombie is its ability to capture hierarchical relationships in the data returned by Twitter. A network translator module performs post-processing on the tweets and stores hashtags, mentions and retweets, separately from the tweet text. Raw tweets are transformed into a representation of interactions to create networks of retweets, mentions and users mentioning hashtags. This feature captured by TwitterZombie, which other studies have paid little attention to, is helpful in answering different types of research questions with relative ease. Social graphs are created in the form of a retweet or mention network and they do not crawl for the user graph with traditional following relationships.

More recently, TwitHoard [185] suggested a framework of supporting processors for data analytics on Twitter with emphasis on selection of a proper dataset for the definition of a campaign. The platform consists of three layers; campaign crawling, integrated modeling, and the data analysis. In the campaign crawling layer, a configuration module follows an iterative approach to ensure the campaign converges to a proper set of filters (keywords). Collected tweets, meta-data and the community data (relationships among Twitter users) are stored in a graph database. This study should be highlighted for its distinction in allowing a flexible querying mechanism in addition to a data model built on raw data. The model is generated in the integrated modeling layer and comprises a representation of associations between terms (e.g. hashtags) used in tweets and their evolution in time. Their approach is interesting as it captures the often-overlooked temporal dimension. In the third, data analysis layer, a query language is used to design a 'target view' of the campaign data that corresponds to a set of tweets that contain, for example, the answer to an opinion mining question.

While including components for capture and storage of tweets, additional tools have been developed to search through the collected tweets. The architecture of CoalMine [210] presents a social network data mining system demonstrated on Twitter, designed to process large amounts of streaming social data. The ad-hoc query tool provides an end user with the ability to access one or more data files through a Google-like search interface. Appropriate support is provided for a set of Boolean and logical operators for ease of querying on top of a standard Apache Lucene index. The data collection and storage component is responsible for establishing connections to the REST API and to store the JSON objects returned in compressed formats.

In building support platforms, it is necessary to make provision for practical considerations such as processing big data. TrendMiner [156] facilitates real-time analysis of tweets and takes into consideration scalability and efficiency of processing large volumes of data. TrendMiner makes an effort to unify some of the existing text processing tools for Online Social Networking (OSN) data, with an emphasis on adapting to real-life scenarios that include processing batches of millions of data. TrendMiner envisioned the system to be developed for both batch-mode and online processing.

3.3.4 Application-specific Platforms

Apart from the above-mentioned general purpose platforms, there are many frameworks targeted at conducting specific types of analysis with Twitter data. Emergency Situation Awareness (ESA) [222] is a platform developed to detect, assess, summarise and report messages of interest published on Twitter for crisis coordination tasks. The objective of their work is to convert large streams of social media data into useful situation awareness information in real-time. The ESA platform consists of modules to detect incidents, condense and summarise messages, classify messages of high value, identify and track issues and finally to conduct forensic analysis of historical events. The modules are enriched by a suite of visualisation interfaces. Baldwin *et al.*[14] proposed another support platform focused on detecting events on Twitter. The Twitter stream is queried with a set of keywords specified by the user with the objective of filtering the stream on a topic of interest. The results are piped through text-processing components and the geo-located tweets are visualised on a map for better interaction. Clearly, platforms of this nature that deal with incident exploration need to make provision for real-time analysis of the incoming Twitter stream and produce suitable visualizations of detected incidents.

3.3.5 Support for Visualization Interfaces

There are many platforms designed with integrated tools predominantly for visualization, to analyse data in spatial, temporal and topical perspectives. One tool is tweetTracker [108], which is designed to aid monitoring of tweets for humanitarian and disaster relief. TweetXplorer [138] also provides useful visualization tools to explore Twitter data. For a particular campaign, visualizations in tweetXplorer help analysts to view the data in different dimensions; e.g. the most interesting days in a campaign (when), important users and their tweets (who/what) and important locations in the dataset (where). Systems like TwitInfo [127], Twitcident[2] and Torettor [171] also provide a suite of visualisation capabilities to explore tweets in different dimensions relating to specific applications such as fighting fires and detecting earthquakes. Web-mashups like Trendsmap [193] and Twitalyzer [197] provide a web interface and enterprise business solutions to gain real-time trends and insights of(or into) user groups.

3.3.6 Discussion: Data Model and Storage Mechanisms

Data models are not discussed in detail in most studies as a simple data model is sufficient to conduct a basic form of analysis. When standard tweets are collected, flat files [14, 210] are the preferred choice. Several studies that capture the social relationships [20, 28] of the Twittersphere employ a relational data model but do not necessarily store the relationships in a graph database. As a consequence, many analyses that can be performed conveniently on a graph are not captured by these platforms. Only TwitHoard [185] models co-occurrence of terms as a graph with temporally-evolving properties. Twitter Zombie [20] and TwitHoard [185] should be highlighted for capturing interactions including the retweets and term associations apart from the traditional follower/friend social relationships. TrendMiner [156] draws explicit discussion on making provision for processing millions of data and takes advantage of the Apache Hadoop MapReduce framework to perform distributed processing of the tweets stored as key-value pairs. CoalMine [210] also has Apache Hadoop at the core of its batch processing component responsible for efficient processing of large amounts of data.

Table 3.1: Overview of related approaches in data management frameworks.

	Pre-processing	Examples of extracted information	Social and/or other interactions captured?	Data Store
TwitterEcho [22]	✓	Language	Yes	Not given
Byun *et al.*[28]		Location	Yes	Relational
Twitter Zombie [20]	✓		Yes	Relational
TwitHoard [185]	✓		Yes	Graph DB
CoalMine [210]			No	Files
TrendMiner [156]	✓✓	Location, Sentiment, NEs	No	Key-value pairs
TwitIE[24]	✓✓	Language, Location, NEs	No	Not given
ESA [222]	✓	Location, NEs	No	Not given
Baldwin *et al.*[14]	✓✓	Language, Location	No	Flat files

Table 3.1 illustrates an overview of related approaches and features of different platforms. *Pre-processing* in Table 3.1 indicates if any form of language processing tasks such as POS tagging or normalization is conducted. Multiple ticks (✓) correspond to a task that is carried out extensively. *Information extraction* refers to the types of post-processing performed to infer additional information, such as sentiment or named entities (NEs). In addition to collecting tweets, some studies also capture a user's social graph while others propose the need to regard interactions of hashtags, retweets and mentions as separate properties. Backend data models supported by the platform shape the types of analysis that can be conveniently done on each framework. From the summary in Table 3.1, we can observe that each study on data management frameworks concentrate on a set of challenges more than others and graph-based models remain largely unexplored.

3.4 Languages for Querying Tweets

Next, we survey declarative languages that are available for querying different aspects of the Twittersphere paying attention to their underlying data models and query dimensions. The goal of proposing declarative languages and systems for querying tweets is to put forward a set of primitives or an interface for analysts to conveniently query specific interactions on Twitter, exploring the user, time, space and topical dimensions. High level languages for querying tweets extend the capabilities of existing languages such as SQL and SPARQL. Queries are either executed on the Twitter stream in real-time or on a stored off-line collection.

3.4.1 Generic Languages

TweeQL [128] provides a streaming SQL-like interface to the Twitter API and provides a set of user-defined functions (UDFs) to manipulate data. The objective is to introduce a query language to extract structure and useful information embedded in unstructured Twitter data. The language exploits both relational and streaming semantics. UDFs allow for operations such as location identification, string processing, sentiment prediction, named entity extraction and event detection. In the spirit of streaming semantics, it provides SQL constructs to perform aggregations over the incoming stream on a user-specified time window. The result of a given query can be stored in a relational fashion for subsequent querying.

Models for representing any social network in RDF have been proposed by Martin and Gutierrez [129] allowing queries in SPARQL. Their work explored the feasibility of adoption of this model by demonstrating their idea with an illustrative prototype but did not focus on a single social network such as Twitter in particular. TwarQL [135] extracts content from tweets and encodes it in RDF format using shared and well-known vocabularies (FOAF, MOAT, SIOC) enabling querying in SPARQL. The extraction facility processes plain tweets and expands its description by adding sentiment annotations, DBPedia entities, hashtag definitions and URLs. The annotation of tweets using different vocabularies enables querying and analysis in different dimensions such as location, users, sentiment and related named entities.

Temporal and topical features are of paramount importance in an evolving microblogging stream like Twitter. In the languages above, time and topic of a tweet (topic can be represented simply by a hashtag) are considered meta-data of the tweet and are not treated any differently from other metadata reported. Topics are regarded as part of the tweet content or what drives the data filtering task from the Twitter API. There have been efforts to exploit features that go well beyond a simple filter based on time and topic. Plachouras and Stavrakas [154] stressed the need for temporal modeling of terms in Twitter to effectively capture changing trends. A term refers to any word or short phrase of interest in a tweet, including hashtags or output of an entity recognition process. Their proposed query operators can express complex queries for associations between terms over varying time granularities, to discover the context of collected data. Operators also allow retrieving a subset of tweets satisfying these complex conditions on term associations. This enables the end-user to select a good set of terms (hashtags) that drive the data collection, and this has a direct impact on the quality of the results generated by the analysis.

Spatial features are another property of tweets often overlooked in complex analyses. Previously discussed studies use the location attribute as a mechanism to filter tweets. To complete our discussion, we briefly outline two studies that used geo-spatial properties to perform complex analysis using the location attribute. Doytsher *et al.*[53] introduced a model and query language suited for integrated data connecting a social network of users with a spatial network to identify places visited frequently. Edges named *life-patterns* are used to associate the social and spatial networks. Different time granularities can be expressed for each visited location represented by the life-pattern edge. Even though the implementation employs a partially synthetic dataset, it will be interesting to investigate how the socio-spatial networks and the life-pattern edges that are used to associate the spatial and social networks can be represented in a real social network dataset with location information, such as Twitter. GeoScope [26] finds information trends by detecting significant correlations among trending location-topic pairs in a sliding window. This gives rise to the importance of capturing the notion of spatial information trends in social networks in analysis tasks. Real-time detection of crisis events from a location in space, exhibits the possible value of Geoscope. In one of the experiments, Twitter is used as a case study to demonstrate its usefulness: a hashtag is chosen to represent the topic and city from which the tweet originates is chosen to capture the location.

3.4.2 Query Languages for Social Networks

To the best of our knowledge, there is no existing work focusing on high level languages operating on the Twitter's social graph. However it is important to note proposals for declarative query languages tailored for querying social networks in general [5, 179, 54, 129, 130, 168]. One of the queries supported are path queries satisfying a set of conditions on the path, and the languages in general take advantage of inherent properties of social networks. Semantics of the languages are based on Datalog [130], SQL [54, 168] or SPARQL [129]. Implementations are conducted on bibliographical networks [54], Facebook and social content sites like Yahoo! Travel [5] and are not tested on Twitter networks taking Twitter specific affordances into consideration.

3.4.3 Information Retrieval - Tweet Search

Another class of systems presents textual queries to efficiently search over a corpus of tweets. The challenges in this area are similar to that of information retrieval but also have to deal with peculiarities of tweets. The short length of tweets in particular creates added complexity to text-based search tasks as it is difficult to identify relevant tweets matching a user query

[16, 66]. Expanding tweet content is suggested as a way to enhance meaning. The goal of such systems is to express a user's information need in the form of a text query, much as in search engines, and return a tweet list in real-time with effective strategies for ranking and relevance measurements [55, 63, 209]. Indexing mechanisms were discussed in [36] as they directly impact efficient retrieval of tweets. The TREC microblogging track[6] is dedicated to calling participants to conduct real-time ad-hoc search tasks over a given tweet collection. Publications of TREC [149] document the findings of all systems in the task of ranking the most relevant tweets matching a pre-defined set of user-queries.

3.4.4 Discussion: Data Model and Storage for the Languages

Relational, RDF and Graphs are the most common choices of data representation. There is a close affiliation in these data models observing that, for instance, a graph can correspond to a set of RDF triples or vice versa. In fact, some studies like Plachouras and Stavrakas [154] have put forward their data model as a labeled multi digraph and have chosen a relational database for their implementation. None of these query systems models Twitter social network with following or retweet relationships among users. Doytsher *et al.*[53] implemented their algebraic query operators with the use of both graph and a relational database as the underlying data storage. They experimentally compared relational and graph database systems to demonstrate the feasibility of the model. Languages that operate on the twitter stream such as TweeQL and TwarQL generate the output in real-time; TweeQL [128] allows the resulting tweets to be collected in batches then stores them in a relational database, while TwarQL [135] at the end of the information extraction phase, encodes annotated tweets in RDF.

Table 3.2: Overview of approaches in systems for querying tweets.

	Data Model			**Explored dimensions**				
	Relational	RDF	Graph	Text	Time	Space	Social Network	Real-Time
TweeQL [128]	✓			✓	✓	✓		Yes
TwarQL [135]		✓		✓	✓	✓		Yes
Plachouras *et al.*[155]	✓			✓	✓✓			No
Doytsher *et al.*[53]*	✓		✓		✓	✓✓	✓	No
GeoScope *et al.*[26]*				✓	✓	✓ ✓		Yes
Languages on social networks*	✓	✓	✓			✓	✓✓	No
Tweet search systems	✓			✓✓	✓		✓	Yes

Table 3.2 illustrates an overview of related approaches in systems for querying tweets. Data models and dimensions investigated in each system are depicted. Systems that have made

[6]http://trec.nist.gov/

provision for the real-time streaming nature of the tweets are indicated in the *Real-time* column. Multiple ticks (✓) correspond to a dimension explored in detail. Note that the systems marked with an asterisk (*) are not implemented specifically targeting tweets, although their application is meaningful and can be extended to the Twittersphere. We observe there is a potential for developing languages for querying tweets that include querying by dimensions that are not captured by existing systems, especially the social graph.

3.5 Requirements of an Integrated Solution

It would be interesting to explore how we can assimilate individual efforts with the goal of providing a unified framework that can be used by researchers and practitioners across many disciplines. Integrated solutions should ideally handle the entire workflow of the data analysis life cycle from data management to presenting the results to the user. The literature we have reviewed in previous sections outlined efforts that support different parts of the workflow. In this section, we present our position with the aim of outlining significant components of an integrated solution addressing the limitations of existing systems. To complete our discussion, we also summarize key research issues in data management and present technical challenges that need to be addressed in the context of building a data management platform for Twitter. Olteanu et al [147] presents a detailed investigation on general challenges of research done on cross-disciplinary social data. In this recent work, complimentary aspects such as data quality, biases and ethical considerations have been extensively reviewed.

According to a review of literature conducted on the microblogging platform [39], a majority of published work on Twitter concentrates on the user domain and the message domain. The user domain explores properties of Twitter users in the microblogging environment while the message domain deals with properties exhibited by the tweets themselves. In comparison to the extent of work done on the microblogging platform, only few have investigated the development of data management solutions and query languages that describe and facilitate processing of social networking data. In consequence, there is an opportunity for improvement in this area for future research to address the challenges in data management. We elicit the following high-level components and envisage a platform for Twitter that encompasses such capabilities.

3.5.1 Focused crawler

Responsible for retrieval and collection of Twitter data by crawling the publicly-accessible Twitter APIs. A focused crawler should allow the user to define a campaign with suitable filters, monitor output and iteratively crawl Twitter for large volumes of data until its coverage of relevant tweets is satisfactory. An analysis may require the definition of one or more data collection campaigns:

- tweets for a specific time period, location or keyword(s);
- social graph originated from a set of seed users;
- social graph and all the tweets of those users, and
- users' profile information.

Challenges and Research Issues. Once a suitable Twitter API has been identified, we can define a *campaign* with a set of parameters. The focused crawler can be programmed to retrieve all tweets matching the query of the campaign. If a social graph is necessary, separate modules would be responsible to create this network iteratively. Exhaustively crawling all the relationships between Twitter users is prohibitive, given the restrictions set by the Twitter API. Hence the focused crawler must prioritize the relationships to crawl based on the impact and importance of specific Twitter accounts. Where the platform handles multiple campaigns in parallel, there is a need to optimize the access to the API. Typically, the implementation of a crawler should aim to minimize the number of API requests, considering the restrictions, while fetching data for many campaigns in parallel. Thus building an effective crawling strategy is a challenging task, in order to optimize the use of API requests available.

Appropriate coverage of a campaign is another significant concern and denotes whether all the relevant information has been collected. When specifying the parameters to define the campaign, a user needs a very good knowledge of the relevant keywords. Depending on those specified keywords, a collection may miss relevant tweets in addition to the tweets removed due to restrictions by APIs. Plachouras and Stavrakas' work [155] is an initial step in this direction as it investigated this notion of coverage and proposed mechanisms to automatically adapt the campaign to evolving hashtags. Other issues in collecting and processing generic social data discussed in [147] are all still applicable for Twitter data and must be cautiously dealt with.

3.5.2 Pre-processor

As highlighted in Section 3.3.2, this stage usually consists of modules for pre-processing and information extraction considering the inherent peculiarities of tweets: not all frameworks we have discussed provided this functionality. Features of a pre-processor may include basic text processing or more advanced modules useful in conducting analysis on tweets. Basic text processing on tweets may include normalisation, tokenisation and POS tagging.

Advanced information extraction attempts to derive more information from plain tweet text and their meta-data such as:

- named entity recognition;
- tweet and user location prediction;
- sentiment analysis, and
- language detection.

Ideally, end-users should be able to customize the modules to suit their requirements and integrate any combination of the components into their own applications.

Challenges and Research Issues. Many problems associated with summarization, topic detection and POS tagging in well-formed documents, e.g. news articles, have been extensively studied. Traditional named entity recognizers (NERs) depend heavily on local linguistic features of well-formed documents [163], such as capitalization and POS tagging of previous words. None of the characteristics hold for tweets with short utterances of tweets limited to 140 characters (testing 280 characters since Sept. 2017), which make use of informal language, undoubtedly making a simple task of POS tagging more challenging. Besides the length limit, heavy and inconsistent usage of abbreviations, capitalizations and uncommon grammatical constructions pose additional challenges to text processing. Any effort that uses Twitter data needs to make use of appropriate twitter-specific strategies to pre-process text, addressing the challenges associated with intrinsic properties of tweets.

Similarly, information extraction from tweets is not straightforward as it is difficult to derive context and topics from a tweet that is a scattered part of a conversation. There is separate literature on identifying entities (references to organizations, places, products, persons) [115, 165], languages [31, 71], and sentiment [150] present in the tweet text for a richer source of information. Location is another vital property representing spatial features either of the tweet or of the user. The location of each tweet may be optionally recorded if using a GPS-enabled device. A user can also specify his or her location as a part of the user profile and is often reported in varying granularities. The drawback is that only a small portion of about 1% of the tweets are geo-located [38]. Since analysis almost always requires the location property, when absent, studies conduct their own mechanisms to infer location of the user, a tweet, or both. There are two major approaches for location prediction: content analysis with probabilistic language models [38, 44, 79] or inference from social and other relations [35, 50, 176].

3.5.3 Data Model

Much of the literature presented (Section 3.3.6 and 3.4.4) does not emphasize or draw explicit discussions on the data model in use. The logical data model greatly influences the types of analysis that can be done with relative ease on collected data. A physical representation of the model involving suitable indexing and storage mechanisms of large volumes of data is an important consideration for efficient retrieval. We notice that current research pays little attention to queries on Twitter interactions, the social graph in particular. A graph view of the Twittersphere is consistently overlooked and we recognize there is much potential in this area. The graph construction on Twitter is not limited to considering the users as nodes and links as following relationships; embracing useful characteristics such as retweet, mention and hashtag co-occurrence networks in the data model will create opportunities to conduct complex analyses on these structural properties of tweets. We envision data models that consist of the following properties:

- tweets and their meta-data: timestamp, location, text, keywords and language etc.
- users and their meta-data: profile_name, verified accounts and location etc.
- social connections among users: user–follow–user, user–mention–user;
- propagation connections among twets: tweet–retweet–tweet;
- topical behaviour of tweets: tweet–has–hashtag;
- connections among users and tweets: user–posts–tweet, and
- co-occurrence behaviour among topics: hashtag–cooccur–hashtag.

As shown in Figure 3.3, the above requirements can be modeled as a directed attributed multi-graph (i.e. property graph) with three types of nodes: `user`, `tweet` and `hashtag`. A multi-graph allows two nodes to be connected with more than one edge. Users following each other are represented by a `follows` relationship, while posting is represented by a `posts` edge between a `user` and `tweet`. A retweet of an original tweet is denoted by a `retweets` edge, while mentions of a tweet by a particular user are captured by a `mentions` edge. If a particular tweet contains a hashtag, the `tags` edge is used to represent this information.

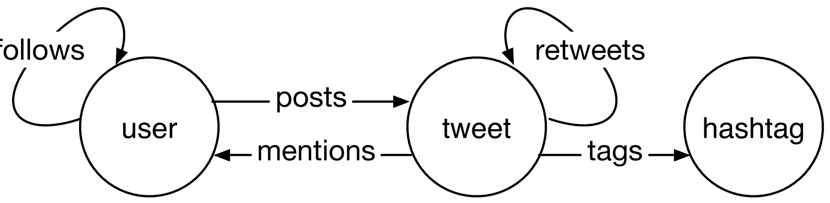


Figure 3.3: Graph-based data model for the Twittersphere.

In Sections 3.3.6 and 3.4.4 we outlined several alternative approaches in literature for a data model to characterise the Twittersphere. The relational and RDF models are frequently chosen while graph-based models are acknowledged but, not realized concretely at the implementation phase. With a data model such as Figure 3.3, graph data management in Twitter can be extremely diverse and interesting; different types of networks can be constructed apart from the traditional social graph. The advent of a graph view to model the Twittersphere gives rise to a range of queries that can be performed on the structure of the tweets, essentially capturing a wider range of use case scenarios used in typical data analytics tasks. We need to investigate prevalent technologies such as graph database systems that can conveniently persist graphs with the above schema facilitating graph traversals on them.

3.5.4 Query Language

Languages described in Section 3.4, define both simple operators to be applied on tweets and advanced operators that extract complex patterns, which can be manipulated in different types of applications. Some languages provide support for continuous queries on the stream or queries on a stored collection, while others offer flexibility for both. The advent of a graph view makes crucial contributions in analysing the Twittersphere, allowing us to query twitter data in novel and varying forms. It will be interesting to investigate how typical functionality [211] provided by generic graph query languages can be adapted to Twitter networks.

As discussed in Section 3.4.2, there are already languages similar to SQL which have been adapted to social networks. Many of the techniques mentioned in the literature are for generic social networks under a number of specific assumptions. For example, social networks satisfy properties such as the power law distribution, sparsity and small diameters [133]. We envision queries that take this a step further and execute on Twitter graphs. Simple query languages FQL [57] and YQL [216] provide features to explore properties of Facebook and Yahoo APIs but are limited to querying only part(usually a single user's connections) of the large social graph. Considering the features we included in the data model in the previous section, a query system on Twitter should be able to *efficiently* execute queries on the following dimensions:

- attributes (meta-data) on users and tweets;
- different types of connections: user–follow–user, tweet–retweet–tweet etc. and
- search through keywords, terms and hashtags within tweets.

Visualizing the data retrieved as a result of a query in a suitable manner is also an important concern. Another interesting avenue to explore is the introduction of a ranking mechanism of the query result. Ranking criteria may involve relevance, timeliness or network attributes like the reputations of users in a social graph. Ranking functions are a standard requirement in the field of information retrieval [36, 85] and studies such as SociQL [179] report the use of visibility and reputations metrics to rank results generated from a social graph. A query language with a graph view of the Twittersphere along with capabilities for visualizations and ranking will certainly benefit efforts to analyse Twitter data.

3.5.5 General Challenges in Data Management

One of the predominant challenges is the management of large graphs that inevitably results from modeling users, tweets and their properties as graphs. With the large volume of data involved in any practical task, a data model should be information-rich, yet also be a concise representation that enables expression of useful queries. Queries on graphs should be optimized for large networks and should ideally run independently of the size of the graph. There are already approaches that investigate efficient algorithms on very large graphs [94, 91, 92, 173]. Efficient encoding and indexing mechanisms should be in place, taking into account variations of indexing systems already proposed for tweets [36] and indexing of graphs [217] in general. We need to consider maintaining indices for tweets, keywords, users, hashtags for efficient access to data in advance queries.

Besides the above challenges, tweets impose general research issues related to big data. Challenges should be addressed in the same spirit as any other big data analytics task. In the face of challenges posed by large volumes of data being collected, the NoSQL paradigm should be considered an obvious choice for dealing with them. Developed solutions should be extensible for new requirements and should indeed scale well. With respect to implementation, it is necessary to investigate paradigms that scale well, like MapReduce which is optimized for offline analytics on large data partitioned on hundreds of machines. For OLTP-like workloads which require low-latency access to small portions of the graph schema, MapReduce-based graph models may not be an ideal candidate. Consequently, we argue that database systems such as Titan [12], Sparksee [131], and Neo4j [140] should be compared for graph implementations. In the next section, we investigate how well graph database systems can drive the data management goals of a Twitter framework.

3.6 Graph Database Systems for Microblogging Queries

In the previous sections, we highlighted the need for efficient querying and management of large collections of Twitter data modeled as graphs. In the second part of this study, we model the basic elements of the Twittersphere as graphs, and determine the feasibility of running a set of microblogging queries in graph database systems and present our introspection. As shown in the graph schema in Figure 3.3, Twitter can be modeled as a labeled, directed, attributed multi-graph. Graph database systems support management of property graphs, which consolidate the above features (Detailed in Section 2.4.1), thus become a good conceptual fit to test our model.

In existing work around the topic, analyses of general data management queries on graph database systems have been widely reported [8, 200, 88], but none have demonstrated the feasibility of analyzing microblogging queries using such databases. It is noteworthy that most of the prior studies have focused on either executing MLDM (machine learning data mining) algorithms over large graphs [132] or on performing graph data management queries using relational databases [123]. In many of these studies, the goal is to create benchmarks for graph database management systems in terms of computational [132] or data management query workloads [200, 88, 125]. In addition, a large number of existing research has focused on using RDF stores or relational databases [123, 200] to store Twitter data [70]. Most of the relational queries are written with self-joins, requiring many optimizations in order to achieve an acceptable performance. Different from these approaches, we study the feasibility of using a graph database system to query Twitter data. We believe that graph data management systems are better equipped to test the particular type of microblogging data workloads used in this chapter. We define queries relevant to microblogging and share our introspection on executing them using graph management systems; thereby perfectly complementing those prior works.

For our analysis, we have carefully chosen queries pertinent to several applications of microblogging data. For example, our queries are relevant to applications such as providing friend recommendations, analyzing user influence, finding co-occurrences and shortest paths between graph nodes. In addition, we have analyzed fundamental atomic operations like selection and retrieving the neighbourhood of a node. For executing the aforementioned queries, we have chosen two popular open-source graph database systems: Neo4j [140] and Sparksee [131]. Such systems are typically able to efficiently answer data management queries concerning attributes and relationships exploiting the structure of the graph. We particularly want to find answers to the following questions.

- How efficiently can graph systems ingest a large graph dataset?
- Can graph systems model the Twittersphere with all the required properties?
- Can microblogging workloads be effectively translated to graph queries?
- How efficient are the queries when running them in a declarative and procedural fashion?
- What are the limitations of graph database systems and future research directions?

The goal of this work is not to perform a full benchmark of the two systems or recommend one over the other. Instead our objective is to report our experiences working on these two graph database systems, as a way forward for us to understand the capabilities of graph database systems for data management.

3.6.1 Database Schema

The data model we proposed in Section 3.5.3 is what we use in this study. Here, in Figure 3.4 we describe it further with attributes, and discuss a few alternate data modeling options. The figure only shows a few attributes attached to each of the nodes and edges; `User` and `Tweet` nodes particularly has many more properties on them. Many of the edges may have the timestamp as an attribute. A Twitter dataset collected from an API would require pre-processing to create many of these relationships: `follows` relationship may be directly returned by the Twitter's REST API while a Tweet may have to be processed to extract the `hashtag` nodes and `retweet` relationships. Although we specify multiplicity on the edges, they are generally enforced at the application level since many graph database systems cannot defined such constraints on the schema.

Some applications would require tweet text to be tokenized and stored in an inverted index, in order to be able to efficiently search keywords or hashtags within tweet text. If a keyword search is conducted on any of these attributes, it is necessary to create a separate text index. Next, let us consider a few more alternate modeling options. Depending on the analysis, we may or may not model the hashtags as a separate vertex (and tags edge) in the graph schema. Hashtags could be simply modeled as an attribute on the Tweet node itself. On the other

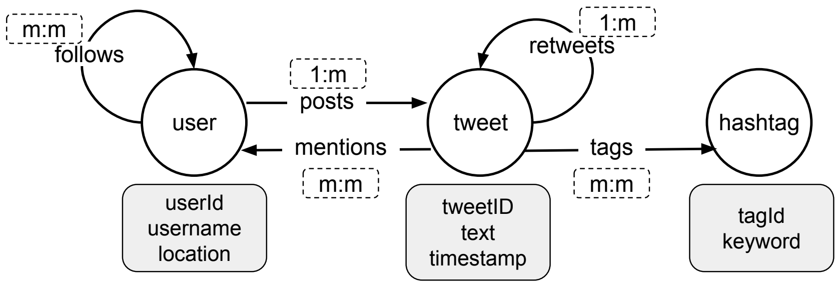


Figure 3.4: Data model of the schema with properties and multiplicity of edges.

hand, modeling hashtags in this way enables us to efficiently express queries on co-occurrence as discussed in Section 3.7.3.

3.6.2 Graph Databases

For our analysis we chose two leading open-source graph management systems, namely, Neo4j and Sparksee. These systems not only support all the features needed for analyzing Twitter data, but also support declarative query languages and API interfaces to interact with the property graphs. **Neo4j** as introduced in Section 2.4.3.1 is a fully transactional graph management system implemented in Java. It supports a declarative query language called *Cypher*. Using the above schema, a query that retrieves the tweets of a given user with id 531 can be written in Cypher as:

```
MATCH (u:USER uid:{531})-[:POSTS]->(t:TWEET)
RETURN t.text;
```

Another method of interaction is by using its core API. The core API offers more flexibility through a *traversal framework*, which allows the user to express exactly how to retrieve the query results. Cypher supports caching the query execution plans. This reduces the cost of re-compilation at run-time when a query with a similar execution plan is executed more than once. We have often used Cypher's profiler to observe the execution plan and determine which query plan results in the least number of database hits (db hits) and have rephrased the query for better performance. It is noteworthy that all the queries can be alternatively written using the Java API exploiting the traversal framework. However, as with any imperative approach, the performance is dependent on *how* the query is translated into a series of API calls.

Sparksee, as introduced in Section 2.4.3.2, is a graph database management system implemented in C++. Different to a declarative query language, imperative approach in Sparksee provides APIs in many languages. We choose the Java API for our experiments. As an example, the query that retrieves the tweets of a given user 531 can be written in Sparksee's API as:

```
int nodetype = g.findType("USER");
int attrID = g.findAttribute(nodetype, "uid");
Value attrVal = new Value();
attrVal.setInteger(531);
long input = g.findObject(attrID, attrVal);
int edgeType = g.findType("POSTS");
Objects userTweets = g.neighbors(input, edgeType, EdgesDirection.OUTGOING);
```

Sparksee queries have two primary navigation operations: `neighbours` and `explode`, which return an unordered set of unique node and edge identifiers that are adjacent to any given node ID. When translating the queries using Sparksee's API, we made use of most of the constructs provided by the developers.

For this study, with the objective of understanding the diverse functionality of different graph database systems, we opted to run our queries with the declarative interface for Neo4j and the core API interface with Sparksee.

3.7 Data Ingestion and Query Processing

In this section we will analyse the feasibility of executing a wide variety of relevant microblogging queries on Neo4j and Sparksee. In Section 3.7.1 we start by discussing the details of the dataset we used. In Section 3.7.2 and Section 3.7.3, we share our experience in importing a large dataset and executing microblogging queries respectively. All the experiments were conducted on a standard Intel Core 2 Duo 3.0 GHz and 8GB of RAM with a non-SSD HDD. For Neo4j we used Version 2.2.M03, and for Sparksee Version 5.1 was used. The research license for Sparksee could accommodate up to 1 billion objects. We used the respective APIs in Java embedding the databases.

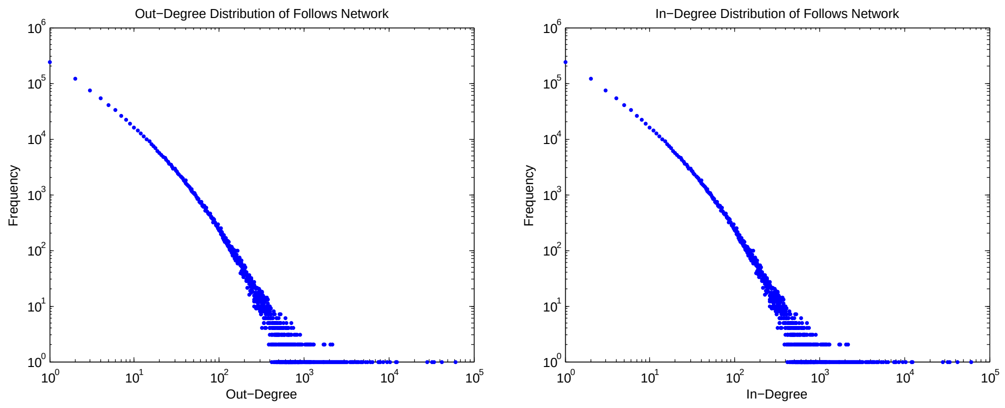
3.7.1 Dataset and Pre-processing

For our experiments, we processed a dataset [116] consisting of 284 million `follows` relationships among 24 million users. For a subset of 140,000 users who have at least 10 followees, this dataset contains 500 tweets per user. We retain only 200 tweets per user from this set. By processing the tweets, we reconstruct all the edges and nodes of the schema shown in Figure 3.4. Unfortunately, this dataset does not have exact information on retweets, therefore we could not reconstruct the `retweets` edges. Although the dataset is not complete with tweets of all users, it satisfies the requirement of being able to model the schema with a reasonable number of nodes and edges. A summary of the characteristics of the dataset is shown in Table 3.3.

Table 3.3: Characteristics of the dataset depicting types of nodes and edges.

Nodes		Relationships	
user	24,789,792	follows	284,000,284
tweet	24,000,023	posts	24,000,023
hashtag	616,109	mentions	11,100,547
		tags	7,137,992
Total	49,405,924	**Total**	326,238,846

As is the case with many social networks, the in-degree and out-degree distributions of the `follow` network shows a power-law distribution (cf. Figure 3.5).



(a) Out-degree **(b) In-degree**

Figure 3.5: Degree distribution of the `follows` network.

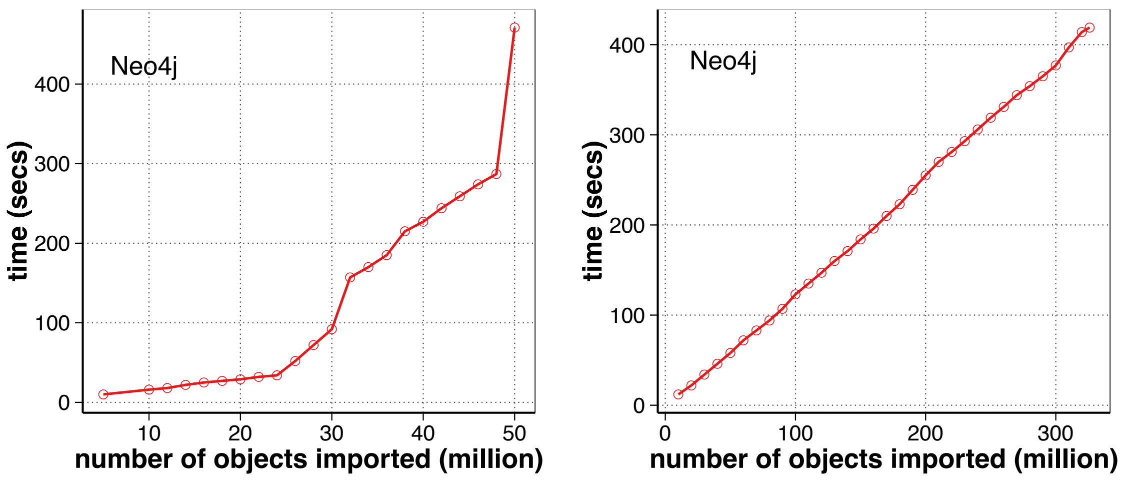
3.7.2 Data Ingestion

We use batch-loading procedures offered by both graph systems. The same source files containing the nodes and edges were used with both databases.

3.7.2.1 Neo4j

We used the Neo4j's import tool for importing the workload. We decided to use the import tool after trying several other options. A main reason was that the tool effectively manages memory without explicit configuration. However, it cannot create additional indices (on node IDs) while importing takes place. Indices were created after the data import is complete. Neo4j's import tool writes continuously and concurrently to disk.

We plot the time taken for importing nodes and edges in Figure 3.6(a) and (b) respectively. The plot shows the number of objects (nodes or edges) inserted against the time taken to import every 1 million objects. Observe that the insertion of edges is smoother as compared to the nodes. The jumps in Figure 3.6(a) are mainly due to the time taken to flush the nodes to disk, which slightly slows down the import. After the node import is complete, Neo4j performs additional steps, for example, computing the dense nodes, before it proceeds with importing the edges. These intermediate steps require approximately 10 minutes. Then we create indices on all unique node identifiers, which took about 8 minutes. Since a node could be of type `user`, `tweet`, or `hashtag`, these indices give us the flexibility to efficiently query the aforementioned node types. Overall, importing the workload required a total of 45 minutes, taking 20.8 GB of disk space.

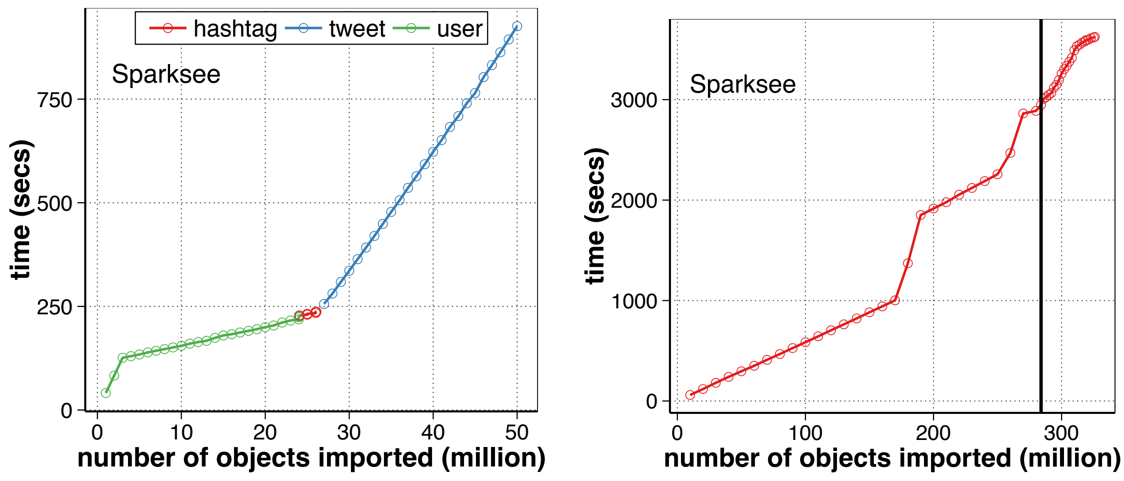


(a) Nodes (b) Edges

Figure 3.6: Import times for nodes and edges using Neo4j. Jumps in the import time in (a) are due to the time take to flush the cache onto disk.

3.7.2.2 Sparksee

Sparksee scripts, which is an importing mechanism available in Sparksee, has been used to define the schema of the database. A script also specifies the IDs to be indexed and source files for loading data. Recovery and rollback features were disabled to allow faster insertions. The extent and cache size are another two parameters that could be configured. With lower extent sizes, we found that the insertions were fast initially but slows down as the database grows. In the final configuration, the extent size was set to 64 KB and cache size to 5GB. Sparksee recommends to materialise neighbours during the import phase. This creates a neighbour index that can be used for faster querying. But with this option enabled, it took us a long time to import and we aborted the import after waiting for 8 hours. With materialisation turned off, Sparksee required 72 minutes, taking 15.1 GB of disk space.



(a) Nodes **(b) Edges**

Figure 3.7: Import times for nodes and edges using Sparksee. The vertical line in (b) refers to the end of the import of `follows` edges.

The load times for nodes and edges are shown in Figure 3.7. The three coloured regions in Figure 3.7(a) correspond to the three node types imported with different pay loads. The import times of the nodes can be separated into the three regions marked in Figure 3.7(a). Since the payload of the `tweet` nodes is larger than the other node types, Figure 3.7(b) refers to the end of the import of `follows` edges, which make up 80% of the edges. The remaining edge types add up to only 20% of the edges. Sharp jumps in the insertion time of edges is when the cache is full and has to flush edge to disk before insertions can be continued. Notice that the jumps in Figure 3.7(a) are bigger than the jumps in Figure 3.6(a). This is because Neo4j concurrently

Table 3.4: Query workload. Experience for queries marked with (⋆) is discussed in detail.

		Category	Example
	Q1.1	Select	All Users with a follower count greater than a user-defined threshold
	Q1.2	Keyword-Search	All tweets containing a given keyword
	Q2.1	Adjacency (1-step)	All the followees of a given user `A`
	Q2.2	Adjacency (2-step)	All the Tweets posted by followees of `A`
	Q2.3	Adjacency (3-step)	All the hashtags used by followees of `A`
(⋆)	Q3.1	Co-occurrence	Top-n users most mentioned with user `A`
	Q3.2	Co-occurrence	Top-n most co-occurring hashtags with hashtag `H`
(⋆)	Q4.1	Recommendation	Top-n followees of `A`'s followees who `A` is not following yet
	Q4.2	Recommendation	Top-n followers of `A`'s followees who `A` is not following yet
	Q4.3	Recommendation (topic)	Top-n users who have used the same hashtag as A
	Q5.1	Influence (current)	Top-n users who have mentioned `A` who are followees of `A`
(⋆)	Q5.2	Influence (potential)	Top-n users who have mentioned `A` but are not direct followees of `A`
(⋆)	Q6.1	Shortest Path	Shortest path between two users where they are connected by `follows` edges

writes to disk, while Sparksee waits for the cache to be full before flushing it to disk. The plots for data ingestion are not consolidated as the batch loaders for the two database systems operate on different settings. At the time of writing this chapter, both Neo4j and Sparksee could not import additional data into an existing database (i.e. incremental loading), hence all data was loaded in one single batch.

3.7.3 Query Processing

In this section we propose a set of relevant microblogging queries and share our experiences in executing these queries on Neo4j and Sparksee. We proposed a set of non-exhaustive queries that were designed keeping in mind the typical analysis that is performed with microblogging data. The survey conducted in the previous sections, helped us to identify different dimensions in which queries were executed in a varied set of applications. We also observed typical queries for general social networks as proposed by [8] and extended the query classification by introducing queries useful in the context of microblogs.

The queries were classified into six categories as shown in Table 3.4. For each category we used 2-3 exemplar queries for performing the analysis. The systems were left in its default configurations. We started by executing a query and, once the cache was warmed-up and the execution time stabilized, we reported the average execution time over 10 subsequent runs. For queries 2-5 we ran the queries varying the degree of the source node in concern. This gave an idea of queries performance when the given node or subsequent nodes in the traversal involved dense nodes. Although we present our experience with implementing all query types in both systems, we do not report on the performance of all query types as Q1, Q2 are deemed to be simpler than the rest. Hence, in our discussion we focus on Q3, Q4, Q5, Q6, as indicated with a ($\star$) in Table 3.4. We discuss these queries since they exhibit interesting behaviour of the two systems.

We intentionally explore the ability of two different approaches in the two databases: one that uses a declarative query system (Cypher in Neo4j) and another that can manipulate the inbuilt features of the exposed API (Sparksee). Executing Cypher queries may involve overhead with processing the declarative syntax. Due to the use of the different approaches, the results of the queries may not be directly comparable. Therefore, for reasons of fairness, we report on the performance of these two systems separately.

3.7.3.1 Basic Queries

We start by presenting our experience with selection and adjacency queries. Using these atomic operations, we construct new, complex microblogging queries.

Q1 – Select queries. These queries select nodes or edges based on a predicate over one or more of their properties. The combination of selection conditions can be easily expressed in Cypher with logical operators. Sparksee does not directly support filtering on multiple predicates. Therefore, to evaluate a disjunctive or conjunctive query, we have to evaluate its predicates individually and combine the results appropriately to construct the final result.

It is often necessary to search for tweets with a given keyword, or filter tweets from a given time period. Performance of a partial match is tested by way of giving a keyword to search for in the tweet text. Full-text indices are supported by Neo4j for text retrieval, while string indexing is not yet offered by Sparksee.

Q2 – Adjacency queries. Adjacency queries retrieve the immediate neighbourhood of a node specified with a different number of hops. This is different from traversal queries as it requires a recursive exploration for a given edge type. A k-step neighbourhood can be explored via different edge types. Adjacency queries form the basis of almost all other queries mentioned in Table 3.4. As shown in this table, we have used 1-, 2- and 3-step adjacency queries in both systems. Q2.1 is a simple query to retrieve the direct followers of User `A`. Q2.2 use the 2-hop adjacency which first retrieves the direct followees of `A` and then retrieves their corresponding tweets. Q2.2 with a condition on the time of tweet is a classic *timeline* query which gives all the tweets posted by the followers of User `A`. Q2.3 will give an idea about what the followers of a User `A` are talking about, retrieving the hashtags posted by the followers of that user.

3.7.3.2 Advanced Queries

Next, we discuss our experiences in executing queries Q3-Q6. Queries Q3, Q4, and Q5 are top-n queries. Such queries can be expressed and executed in Cypher using `COUNT`, `ORDER BY` and `LIMIT` clauses. For Sparksee, a map structure is used for maintaining the required counts. These counts are then sorted to obtain the final result. Its API does not provide the functionality to limit the returned results so should be done programatically.

Q3 – Co-occurrence queries. Two nodes of any type are said to be *co-occurring* with each other if there is another node that connects both of them. Elements tested here are co-occuring hashtags and mentions. Co-occurrence is a special type of adjacency query useful for finding recommendations for users. Finding co-occurrences is a 2-step process. For example, in Q3.1 the steps are (1) find the users who mention a User `A` in their tweet set `T`, and (2) find other users that are mentioned in the tweet set `T`. Figure 3.8 shows the two-step process for the top-n most co-occurring hashtags with `H`, in Q3.2. First, the tweets `T` that contain the hashtag `H`, where `T = {T1, T2, T3, ... Tn}` should be filtered. In the second step, hashtags tagged in each $t \in$ `T` should be queried, aggregated and counted to retrieve the top-n results.

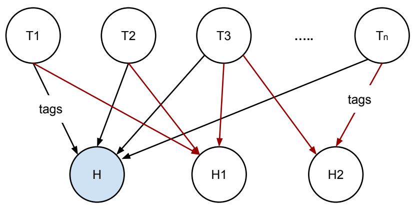


Figure 3.8: Co-occurrence example.

The results of the query execution for query Q3.1 are shown in Figure 3.9(a) and (b). They show a straightforward, increasing trend. However, when the number of rows returned are low the results for both systems seem to fluctuate, but become more predictable with increase in rows returned. Perhaps this fluctuation is due to the random disk accesses that require a different portion of the graph for a new parameter in the query.

Q4 – Recommendation queries. Recommending users to follow, often involves looking at a user's 1- and 2-step followers/followees, since recommendations are often useful when obtained from the local community. Users can be recommended i) based on other users `A` follows (Q4.1 and Q4.2), and (ii) based on common topics used in `A`'s neighborhood (Q4.3). We propose recommendation queries based on the incoming and outgoing neighbourhood of `A` as illustrated in Figure 3.10. Q4.1 finds all the 2-step followees of a User `A`, who `A` is not following. Such

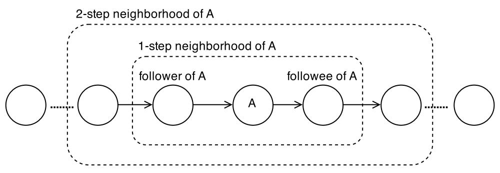
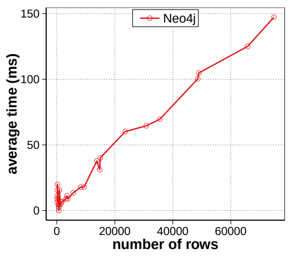
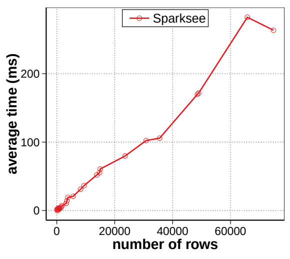


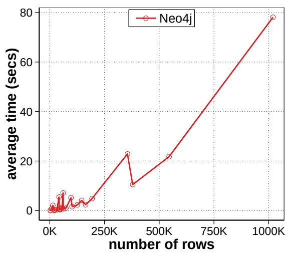
Figure 3.10: Neighborhood of A.



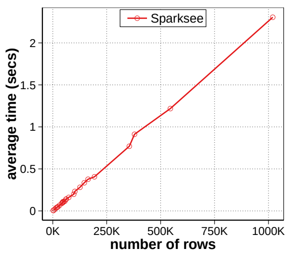
(a) Q3.1 – Neo4j



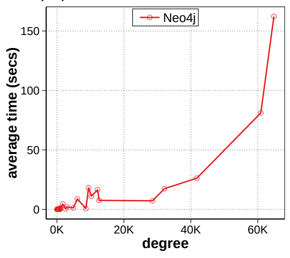
(b) Q3.1 – Sparksee



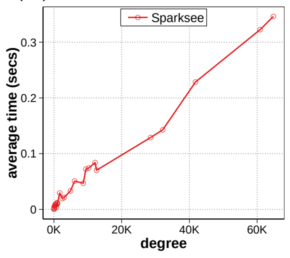
(c) Q4.1 – Neo4j



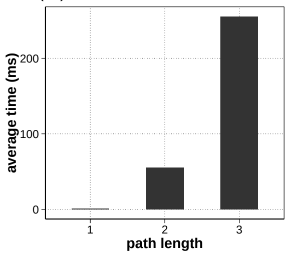
(d) Q4.1 – Sparksee



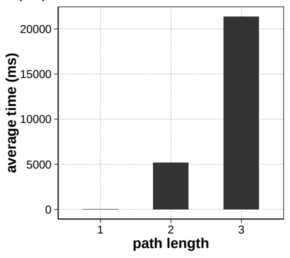
(e) Q5.2 – Neo4j



(f) Q5.2 – Sparksee



(g) Q6.1 – Neo4j



(h) Q6.1 – Sparksee

Figure 3.9: Query execution results. (a) and (b) show co-occurrence query (Q3.1), while (c) and (d) show recommendation query (Q4.1). Influence query (Q5.2) is shown in (e) and (f) and shortest path query (Q6.1) is shown in (g) and (h).

followees are recommended to `A`. Another variant of this query is Q4.2 that finds 1-step followers of `A`'s 1-step followees. Only the top-n users who are followees of A's followees are considered relevant for recommendation thus the 2-step neighbourhood must be aggregated to return the most relevant users. Users are considered relevant for a recommendation if they are followed by many of the followees of User `A`. Suggestions can also be given to `A` based on users who use the same popular hashtags with `A` (Q4.3), assuming they are talking about similar topics. These queries essentially test the ability of the database to return different forms of 'friend of a friend' (FoF) queries.

Recall that for retrieving the neighbours of a particular node, Sparksee provides the `neighbours` operator. For answering Q4.1, a separate `neighbours` call has to be executed for each 1-step followee of `A`, which makes the execution of this query expensive. A separate `neighbours` call is required since we are interested in the popularity (in terms of outlinks) in addition to the identity of `A`'s 2-step followees.

The results of executing Q4.1 are shown in Figure 3.9(*c*) and (*d*). Finding 2-step followees results in an explosion of nodes when 1-step followees have high out-degree. This forces the systems to keep a large portion of the graph in memory. The sudden spike in the plot for Neo4j is due to the fact that the direct degree of the node in concern is much higher even though the number of rows returned are lower. It is noteworthy that (a) Neo4j's performance degrades with a large intermediate result in memory, and (b) Sparksee is able to take advantage of the graph already in memory, as we observe fewer fluctuations with the output.

Q5 – Influence queries. In many use cases, it becomes necessary to discover current and potential influence a particular user has on her community. As an example, for targeting promotions a retail store (with a Twitter account) might be interested in the community of users whom they can influence. Although there exists research on many models of influence propagation, we focus our attention on a set of intuitive queries that can be used for examining influence.

Influence queries are defined based on current and potential influence with respect to a given user. *Current* influence of `A` in our setting is defined as the most frequent users who mention `A` and who are already followers of `A`. The *potential* influence of a person is defined as people who are most mentioning `A` without being direct followers of `A`. In both Neo4j and Sparksee this translates to finding the users who mentioned `A`, and removing (or retaining) the users who are already following `A`. Figure 3.11 shows an example of first, by a 2-hop, retrieving the users mentioned by `A`, and then filtering in(current) or out(potential) the subset of users followed by `A`. Although not tested in our study, top-n retweets where `A` is mentioned is also useful for `A` to know what tweets are going viral or being popular when `A` is mentioned.

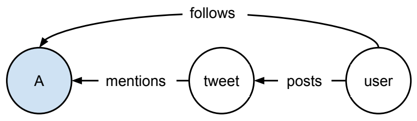


Figure 3.11: Influence of A.

The result of executing Q5.2 is shown in Figure 3.9(e) and (f). The degree of a user mention(x-axis) is defined as the number of times that user is mentioned in the collection. Notice that the degree is low, demonstrating behaviour similar to that of the first portion of the plots for co-occurrence queries (refer Figure 3.9(a) and (b)).

Q6 – Shortest path queries. Shortest path queries find the shortest path between two given nodes in a graph. In addition to finding a path, they can also handle restrictions on the type of node that these queries can return as a part of the shortest path. An example of a shortest path query is Q6.1. Shortest path queries can be the basis of a query that needs to target a particular user or a community of users, essentially finding the degrees of separation from one person to another.

In practice, it is necessary to limit the number of returned paths and/or depth of the traversal (maximum hops); otherwise it could lead to an exhaustive search for all paths. The Cypher function `shortestPath` was used for writing the query, while the native function `SinglePairShortestPathBFS` was used for Sparksee, where the maximum length of the shortest path was set to 3 hops. The average time required to compute the shortest path between two randomly selected users is shown in Figure 3.9(g) and (h). In our experience, Neo4j seems to perform shortest path queries more efficiently. Since our experiments, Sparksee has introduced (in Version 5.2) a more efficient `SinglePairShortestPathBFS` operation where to find the shortest path, a bi-directional BFS is conducted traversing both incoming and outgoing edges simultaneously.

3.7.3.3 Deriving Other Queries

It is noteworthy that many other interesting questions can be answered by using different combinations of the aforementioned queries in Table 3.4. As an example, suppose User `A` is interested in a topic (represented by a hashtag `H`) and is looking for users to learn more about the topic. Such a query can be answered combining other queries as follows:

1. Get all the hashtags that co-occur with the given hashtag `H` (Q3.2);
2. Get the most retweeted tweets mentioning those hashtags (Q2.3);
3. Get the original users of those retweets (given that `retweets` relationships is modeled in the database), and
4. Order the users based on the shortest path length from `A` (Q6.1).

Similarly, User `A` may be recommended to topics (instead of recommending users) to tweet based on the popular co-occuring topics used most frequently by `A` and his followees.

1. Get the most frequent hashtags by `A` (3-hop adjacency);
2. Get the most frequent hashtags by followees of `A` (3-hop adjacency), and
3. Get the hashtags that co-occur with hashtags above most frequently and recently (Q3.2).

Combining these queries, we can generate many new and interesting use cases that can help a Twitter analytics task.

3.8 Discussion

In this section we summarise our findings. We also discuss open problems and opportunities for future research.

3.8.1 Efficiency of alternate solutions

The queries in Cypher can also be written using the Neo4j API with a combination of constructs in the traversal framework and the core API. For queries that we did translate to the API, we observed a slight improvement in performance compared to the Cypher queries version. But the benefit of a declarative language is lost, if they have to be re-written using the less expressible API from scratch for performance gains. It must also be noted that significant effort was required to translate some of the queries in the traversal description when they can be very conveniently expressed in Cypher.

We also noticed benefit in performing query optimisations in Cypher. We observed performance differences in queries (returning the same results) depending on the way they were expressed in Cypher. For example, a recommendation query can be written in three similar ways:

(a) going through the follows relationships for depth 2 using `[:follows*2..2]`;
(b) collecting the intermediate results and checking them against the results at depth 2, and
(c) expanding the `follows` relationship to depth 2 and removing the friends at depth 1.

Method (b) performed the best. Methods (a) and (b) resulted in different execution plans, although with a similar number of total database accesses. It was not clear why Method (c) failed to return a result in a reasonable time. As such, some queries had to be rephrased in order to achieve gains in performance. Ideally a query optimizer in Cypher should be converting a query plan to a consistent set of primitives at the back end. With every new release, Cypher is being improved with a lot of emphasis on cost-based optimizers to cater for this. While the expressiveness is a great advantage in Cypher, an optimizer must take care in converting it to an efficient plan based on the cost of alternate traversal plans. A good speedup can be achieved by specifying parameters, because it allows Cypher to cache the execution plans.

On the other hand Sparksee requires sole manipulation of mainly navigation operations (`neighbors`, `explode`) to retrieve results. Even though this gives a lot of flexibility, this might end up in a series of expensive operations as was in the case of recommendation queries. In Sparksee, queries can also be translated to a series of traversals using the `Traversal` or `Context` classes. Our preliminary findings show that using the raw navigation operations (`neighbors` and `explode`) are slightly more efficient than expressing the query as a series of traversal operations. This is perhaps due to the overhead involved with the traversals. Sparksee can certainly also benefit from a query language to complement its current API.

3.8.2 Overhead for aggregate operations

For many queries, users are often only interested in finding the top-n results. For example, it is not useful to show more than 5 recommended followers for a given user. Cypher performance increases with the removal of the additional burden of having to order the results by a count after the grouping. Removing ordering (`ORDER BY`), de-duplication (`DISTINCT`) and limiting the number of results returned (`LIMIT`) are all factors that contribute to performance gains in Cypher. In Sparksee, in order to limit the returned results, the entire result set must be retrieved and filtered programatically to display only the top-n rows.

3.8.3 Problems with the cold cache

We noticed that Neo4j takes a long time to warm up the caches for a new query. This time is even longer if we do not allow the execution plans to be cached. The time taken for the first run is significant even for queries exploring a small neighbourhood. It might not always be possible to allow the caches to be warmed up, if a large number of queries access the cold parts of the graph. As the degree of the source node increases, the time taken to warm the cache increases dramatically as the system attempts to load a large portion of the graph into memory.

3.8.4 Processing keyword search on graphs

In our survey, particularly in Section 3.4.3, we encountered many studies focusing on tweet search. In the Twittersphere, the tweet text (including its terms and hashtags) is an ideal candidate for keyword search which requires maintaining an inverted index for efficient retrieval. The inverted index is a simple data model to locate relevant documents on the basis of user input terms. As mentioned before, for full-text search features, Neo4j provides access to an external Lucene index while Sparksee's capabilities only go as far as searching text via regular expressions.

The studies we found in the survey maintained ad-hoc approaches to maintain and query the corpus of tweets. Many of the existing works focused only on either tweet text or the social graph but never both. We observe great potential in combining these two dimensions and expressing interesting queries on both the graph and the text dimensions. More importantly, we want to explore how well graph database systems are able to handle this type of combined queries efficiently. Detailed investigation on this topic is available in Chapter 6.

3.9 Summary

In this chapter, we first highlighted the need for new data management and query language frameworks for Twitter. We reviewed the tweet analytics space by exploring mechanisms primarily for data collection, data management and languages for querying and analyzing tweets. In this first part, we outlined the research issues and challenges associated with integrated solutions and propose a graph-based model for the Twittersphere.

In the second part of this chapter we investigated how well graph database systems are able to drive the data management goals of a Twitter framework. Then we used a graph-based model of Twitter and proposed a set of queries relevant for a microblogging applications scenario. We chose two representative systems Neo4j and Sparksee and shared our experiences, noting the limitations in running the microblogging queries on these data management systems. Our contributions of this work can be summarised as follows:

- **Extensive Survey:** We conducted the first extensive review on existing approaches to primarily collect, represent, manage and query Twitter data. Armed with these observations we consolidated the requirements of an integrated data management framework for Twitter: Section 3.2—Section 3.4.
- **Data Model and Queries:** We proposed a data model for the Twittersphere that proactively captures Twitter specific interactions and properties Section 3.5. In this model, we suggested microblogging queries that is useful in a variety of application scenarios, such as recommendation, co-occurrence and influence detection.
- **Experiments:** We conducted experiments on a large Twitter dataset, and examined how queries performed on existing GDBMS that use graph structures to represent data Section 3.6—Section 3.7.3.
- **Lessons Learned:** We shared our introspection on working with these graph database systems and discussed open problems and opportunities for future research: Section 3.8.

Chapter 4

Evolving Dependency Graphs for Multi-versioned Codebases

In Chapter 3 we explored how graph database systems can be used to model a large scale social network application such as Twitter. On this proposed model we implemented a series of microblogging queries and observed the ability of graph systems to drive data management goals in a social network setting. Dependencies in software source code repositories can also be modeled as an attributed multi-graph and graph databases become a good conceptual fit to manage and query this type of data. In this chapter we study software dependency graphs exhibiting characteristics different to that of social networks, with hundreds of node and edge types representing software entities. A software dependency graph can be captured for several reasons, one useful objective is code comprehension.

Frappé, is a code comprehension tool developed by Oracle Labs that extracts the code dependencies from a codebase and stores them in a graph database, enabling advanced code comprehension tasks. We study this established project from industry that captures the code dependencies and extend it to create, manage and query versioned graphs when the underlying codebase evolves over time. Unique challenges associated with versioned graph construction in multiple code revisions were addressed by leveraging efficient entity resolution strategies. In this chapter we explore how a graph database system addresses these challenges and facilitate representation, construction and querying of versioned code repositories.

4.1 Introduction

As the size and scope of a codebase grows, code querying and comprehension tools become crucial in understanding and navigating tens of millions of lines of code. This is especially true for C/C++ codebases with their complex language features and custom-built systems. Text editors in combination with text-based tools, such as Cscope[46] and Grep, support searching for references of a particular symbol, navigating from definition to a declaration and fuzzy text searches. However, these text-based searches and fuzzy parsing results lack context sensitivity and have limited knowledge of the semantics of the search symbol. Consequently, the user needs to navigate through a large number of results to manually filter out appropriate answers. Integrated Development Environments (IDEs) such as Eclipse have more complete compilation-level support and also have access to a richer set of language-dependent structural information. They are not favoured on large C/C++ codebases for reasons of build integration complexity, performance and tradition.

It is often useful to show the relationships connecting the query symbols to better understand the context of the symbol for purposes of debugging or further analysis. As such, capturing the different dependencies within the source code is an important aspect of building context-sensitive comprehension tools. Code dependencies can be naturally modelled as a *dependency graph* representing call graphs, type graphs and inheritance hierarchies. Frappé [77] is a source code querying tool developed by Oracle Labs that supports code comprehension tasks for large C/C++ codebases. Frappé extracts graph-structured data from underlying codebases and in addition to symbol search, is able to answer navigational queries in the form of:

- *Does function X or something it calls, write to global variable Y?* and,
- *How much code could be affected if I change macro M?*

The extracted dependency information is stored in a Neo4j [140] graph database (Detailed in Section 2.4.3.1). Graph databases provide a platform that allows native support for code comprehension use cases expressed as efficient graph-based queries. In Neo4j, comprehension queries are expressed in their declarative query language Cypher.

In a collaborative software development environment, the codebases are constantly changing. Developers are continuously adding new features, fixing bugs and refactoring code generating new versions of the code. Frappé captures the dependency graph based on the most recent snapshot of the codebase. Each new version of the codebase results in a modified version of the dependency graph. Existing comprehension queries can be issued against a specific version of the dependency graph in isolation. In this chapter, we extend Frappé, focusing on strategies to enable advanced code comprehension when the underlying codebase evolves over time, and queries may span multiple versions. Any tool that captures a codebase dependency graph can benefit from our experiences in building versioned graphs for multiple revisions of the codebases.

Storing dependency graphs corresponding to each version of the source code is not an ideal solution. In larger software projects, a significant proportion of the graph remains unchanged and thus can have redundant information. Separate graphs also present a complex challenge for implementing user queries that span multiple versions. Existing graph database systems do not have in-built support for efficient storage and management of versioned graphs. As such, end-users of graph databases need to either make copies of each different version of the data or investigate user-defined representations of storing the deltas. A graph *delta* is a history of graph differences over time, e.g. node additions and removals. In this chapter, we sought an efficient representation of the dependency graph to store and query multiple revisions. We also study in detail how the version graph can be built, dealing with the inherent traits of C/C++ codebases. The proposed model must be scalable and performant and be able to seamlessly integrate the current Frappé workloads.

Existing work in generic graph evolution and management [101, 187, 177] is relevant to our work. In many of these studies, the changeset between two graph snapshots (i.e. delta) can be easily determined perhaps by the use of a consistent ID in all successive snapshots. Due to the peculiarities of software entities that we discuss later in Section 4.5, determining the equivalence of entities is a challenging task. As such, versioning of codebase dependency graphs have unique building characteristics and pose new management and query challenges, giving rise to new research directions in graph database systems. In studies from software engineering research, different program meta-models [112, 169, 51] are constructed from source code and, additionally, meta-information extracted from Source Control Management (SCM) repositories (such as git). Our goal with extending Frappé has been to encode and version all supported semantic information in the graph, not restricting the change information to that directly available from the SCM. Some projects [68, 160] extract dependency graphs in varying levels of granularity, dependent on their use case. For the current use cases of Frappé, the dependency graph is more coarse-grained than a full Abstract Syntax Tree (AST), enabling a storage-efficient graph-based comprehension query platform that scales to very large codebases.

In this work, we propose a model of the dependency graph representing n versions of the codebase. On this versioned graph we are able to perform code comprehension tasks on a version specified by the user with the added benefit of facilitating queries across versions. For example, we are able to view how a function has changed over the last five revisions of the code, and thus identify and flag changes that have a wider dependency impact. We also present a systematic study on how graph databases can facilitate versioning code dependencies in terms of its representation, construction and query processing. Our contribution in this chapter can be summarised as follows:

- Presents methods of conducting resolutions of entities across versions, with and without location information.
- Using the resolutions, propose a scalable model to represent a versioned code dependency graph capturing code evolution.
- Evaluates a large codebase (≈13 million lines of code) and show the rate of growth and the storage benefit with a versioned graph over maintaining individual snapshots.
- Recommends new comprehension queries that can be performed as a result of the proposed versioned graph.

4.1.1 Chapter Organisation

In this chapter we first present the background of the Frappé project in Section 4.2 including its architecture, graph schema and code comprehension use cases. A review of relevant work is presented in Section 4.3. In Section 4.4 we explore some possible solutions for capturing code dependencies and introduce the proposed unified model. Node and edge resolution is an important step in building a versioned graph and this is discussed in Section 4.5. We evaluate our approach in Section 4.6 and discuss how the model can be further improved. Finally, Section 4.7 evaluates existing and new representative queries on the proposed versioned graph.

4.2 Frappé background

This section introduces the background to Frappé starting with a brief description about the architecture and how the dependency information are extracted, stored and visualized. A graph model is explained to understand what is represented as the nodes, edges and properties of the dependency graph. Finally, representative code comprehension queries are explained.

4.2.1 Architecture

The architecture of Frappé is shown in Figure 4.1. The wrapper scripts in the extractor run a modified version of the Clang compiler to capture the precise dependency information from

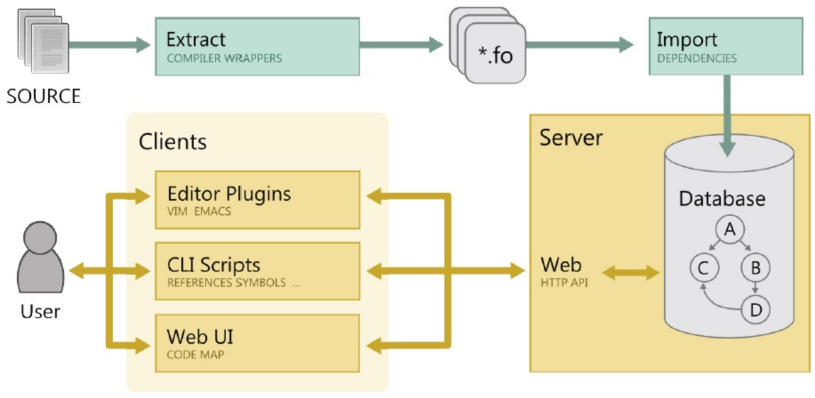


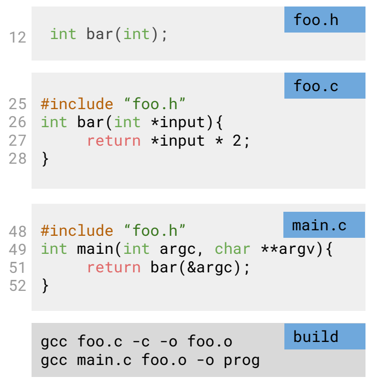
Figure 4.1: Frappé architecture

the source code and generate a set of intermediate .fo files for each unit of compilation. The extracted dependencies are then imported into a graph repository. The zoomable 2D spatial visualization available in the web UI, known as the 'codemap' of the query results may include individual source entities, display paths through the code or transitive closures in the call graph. In a code comprehension task, all this additional information gives the end-user a clear idea of the location, locality, structure and quantity of the results which help immensely in filtering out irrelevant results [77].

4.2.2 Graph Model

The dependency information in a codebase is encoded as a combination of nodes and edges (collectively known as *entities*) in the Frappé graph model. A code segment and its corresponding dependency graph is shown in Figure 4.2. The nodes of the dependency graph represent entities within the code, such as variables, functions and macros. The directed edges represent the associations among these entities, such as `calls` edges between function nodes and `has_param` edges between a function and parameter nodes. As shown in Figure 4.2b, the nodes and edges originate from various data sources and at different steps of compilation. For example, modules, files and the linking information between them come from the *build system*; file inclusions macro definitions, expansions and interrogation links are a result of the *pre-processor*; and other directories, source files from the *file system* and general symbols within the code.

In addition to the name and type attributes of the entity shown in Figure 4.2b, the database stores further attributes on the nodes and the edges. A node of type `enumerator` would have an attribute 'value' to denote the integer representation of the enumerator. An edge type `has_param` would contain an 'index' attribute to indicate the position of parameter in



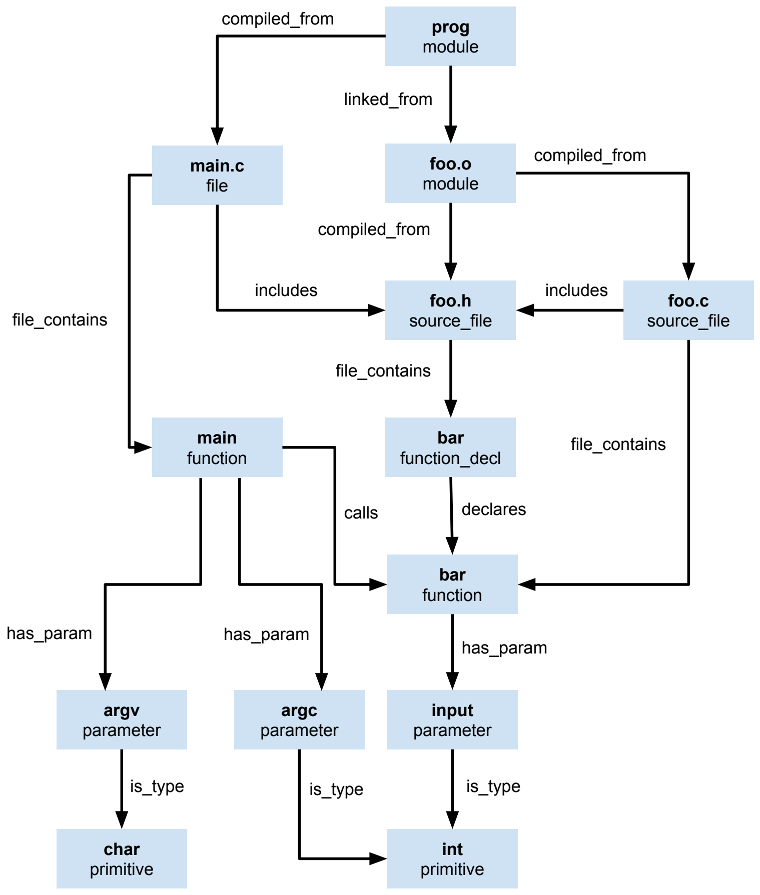
```
foo.h
12  int bar(int);
```

```
foo.c
25  #include "foo.h"
26  int bar(int *input){
27      return *input * 2;
28  }
```

```
main.c
48  #include "foo.h"
49  int main(int argc, char **argv){
51      return bar(&argc);
52  }
```

```
build
gcc foo.c -c -o foo.o
gcc main.c foo.o -o prog
```

(a) Code Segment



(b) Dependency graph

Figure 4.2: Example of a code dependency graph

the function signature. All `file_contains` and reference edge types describe the location information by a set of attributes to precisely identify the position at which an entity appears in the underlying code. The location information stored on the edges is an important part of the model as this information is used later for cross-referencing and visualizations in the 'codemap'. This location information is also crucial in distinguishing different instances of entities.

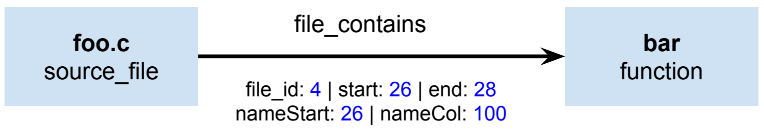

foo.c
source_file
file_contains
file_id: 4 | start: 26 | end: 28
nameStart: 26 | nameCol: 100
bar
function

Figure 4.3: Example of an edge with location information

Each reference edge captures the ‘use’ and ‘spelling’ location (used in Clang terminology, sometimes referred to as ‘expansion’ and ‘name’ location) of adjacent nodes (Figure 4.3). To illustrate with the code segment in Figure 4.2a, the `file_contains` edge between `foo.c` and `bar` has the following properties: (1) File Id of `foo.c`, (2) **use location**: the start and end line numbers at which the `bar` function begins and ends, and (3) **spelling location**: the start line number (nameStart) and column number (nameCol) which the `bar` name token appears in. As we see in Section 4.5.1, the location information becomes an important modeling consideration when the code evolves. Note that the schema of the graph does not have a one-to-one mapping from the nodes represented in an AST – it is at a more coarse-grained level of granularity and does not reproduce all the detailed syntactic structures in an AST.

4.2.3 Code Comprehension Queries

Using the graph model described, high-level code comprehension questions can be translated into graph queries. Queries can range from simple index lookups to complex pattern-matching queries that require traversing a significant portion of the dependency graph. The queries are written using Neo4j’s query language – Cypher. A few representative categories of queries are discussed below.

Code Search. The most common feature of any comprehension tool is its ability to quickly search for a given symbol. Any simple text editor allows a user to search for a symbol, but that may produce a large number of results. A more advanced IDE will allow a user to filter the symbol by its type or the location in which it is defined given that entity types are identified in advance. External index can be built for fast retrieval of symbols and fuzzy searches. A user can provide additional constraints on type, attributes or the location in which a symbol is defined so that results which are more relevant can be retrieved.

For example, to return only fields named ‘id’ present in the module `preprocess.elf`, any fields that are not reachable from the node representing that module via a sequence of file_contains, compiled_from and linked_from edges can be eliminated [77]. The code snippet below shows this query in Cypher.

```
START m = node:node_auto_index(‘name: preprocess.elf’)
MATCH m -[:compiled_from|linked_from*]-> f WITH DISTINCT f
MATCH f -[:file_contains]->(n:field{name: ‘id’}) RETURN n
```

Code Navigation. Another useful feature in a comprehension tool is its ability to move between source files. *Go-to-definition* allows navigation from a symbol to where it is defined. This can be done either by entering the symbol along with any conditions to filter on or by clicking a symbol hyper-link in the visualisation. In contrast, *find-references* retrieves all locations from which a given symbol is referenced and presents the user with a list to filter further. The references are searched by returning all the incoming reference locations to a given symbol. These navigation features facilitate the general purpose of debugging.
Both these search and navigation functionalities are staples of modern IDEs. However Frappé can provide these functionalities for large C/C++ codebase environments where an IDE is not available or is impractical.

Code Path Comprehension. First, shortest path queries in this category enable better understanding of how two nodes in the graph are connected by a given edge type or that they are not reachable at all. Second, developers are often interested in exploring how a seed statement or a region in the code affects the rest of the code (known as a *program slice* [208, 182, 215]). For example, given a seed function, the call graph can be traversed transitively to show the impact of the function either by incoming (forward slice) or outgoing (backward slice) `calls` edges. Moreover, the same affected regions can be found for source file inclusions or macro expansions both features particularly useful in debugging.

While the above queries facilitate comprehension of large codebases, providing more context to the user, the current model is unable to perform queries over a history of changes. Some motivating use cases for extending Frappé with multi-versioned functionality are discussed next.

Motivating use cases for multi-versioned querying. Augmenting the current graph model with versions enables a series of new and interesting queries over these versions. A standard code review typically involves careful and detailed investigation of the changes that have been made to the code over revisions. A reviewer's job is greatly facilitated if, as a part of the code comprehension tool, the changes, such as the function dependencies, are highlighted in advance. Further analysis on multiple revisions, such as which areas of the code are prone to change, how often they change, and which areas tend to change together, can all yield more insight into the codebases.

4.3 Related Work

In this section, we discuss several categories of existing work that are closely related to versioning of code dependencies. Relevant work originate from diverse research areas pertaining to graph evolution, source code analysis for software evolution and projects from industry.

4.3.1 Evolving Graphs

There are many works from the graph domain addressing the general problem of evolving graph sequences [106]. Frameworks and systems have been proposed [164, 102, 101] focusing on efficient snapshot retrieval and analytics. Recent work [111, 187] present distributed frameworks designed for data management of large-scale temporal graphs. LLAMA [124] focus on storage of an evolving graph with the emphasis on data layout augmenting the compressed sparse row representation to store mutating graphs. G* [111] takes advantage of the commonalities in successive snapshots and stores them in compact form. None of these works are specifically focused on the evolution of code dependency graphs which presents a set of unique challenges.

Several works stem from the field of social networks [37, 32, 105] where temporal aspects are introduced to the graphs to enable interesting social queries over the time dimension (e.g. historical queries). Chen et al. [37] proposed a storage model for temporal social networks and indexes to speed up temporal queries on users, relationships and their activities. Proximity networks are modelled in [32], where nodes represent users and edges represent timed interactions between users originating from wearable sensors. Semertzidis and Pitoura [177] discuss alternative methods of storing generic graph snapshots in a native graph database, and present historical graph queries on it. They discuss the possibility of representing time either as an attribute on node/edge or as a different edge type corresponding to each time-point (2012,2013 etc.). Some studies attempt to index the graph in a way that helps specific graph queries such as the historical reachability [178] and shortest path [81, 3] queries.

Instead of consuming individual snapshots, some research [101, 105] process the graph delta. Queries require reconstructing the graph by applying the correct delta on the current snapshot and [105] show how the performance of historical queries can be improved by materialising more than one snapshot, partial reconstruction and indexing deltas. DeltaGraph [101] proposes efficient ways to retrieve a single or multiple snapshots using an hierarchical index structure of the deltas.

Evolving graphs is the most relevant body of work although to the best of our knowledge there has been no work specifically focused on the evolution of code dependency graphs. It must also be noted that for all of the above-mentioned approaches, there is a known implicit association between the entities across versions (perhaps with a persistent id), but in the code dependency graphs we need to compute it.

4.3.2 Industry Projects

Projects have originated from industry for advanced code querying, analysis and comprehension. In addition to text-based tools such as CScope [46], several more advanced tools have been designed through lexing and parsing the source code at different levels. IDEs are one such example providing developers with features for basic code navigation, class browsing and refactoring tools. With no support for incremental indexing built-in to these tools, management of multiple revisions of the source is up to the user; each code repository must be analysed individually and the query results should be compared manually.

Prototype tools such as Wiggle [198] represent the graph in a graph database running queries at a mixture of syntax-tree, type, control-flow-graph or data-flow levels. The AST is stored and queried for several use cases, but there is no discussion about the storage cost or query performance on individual repositories. OpenGrok [148] and Google Kythe [68] are more large-scale projects capturing code structure in varying granularity and complexity with different goals in mind. OpenGrok[148] is a project initiated at Oracle with the aim of providing developers with a tool for searching and cross-referencing in various source code repositories with support for different program file formats. OpenGrok does not build a dependency graph in the backend–it includes parsers for several languages, maintains a text index and uses regular expressions for search tasks. It provides support for version control histories such as Mercurial and Git, allowing the user to select the version of the source to be indexed.

A closer match to Frappé is Google Kythe [68]. The core of the Kythe project is in defining language-agnostic protocols and data formats for representing, accessing and querying source code information as data. This standardisation effort provides protocols for inter-operable developer tools. Extractors in Kythe pull compilation information from the build system, and index the retrieved information in a language-agnostic graph. The graph is then used to answer queries related to code browsing, review and document generation. The Kythe graph schema captures more information than the model in Frappé, thus making the graph more complex and querying difficult. For each new revision of the code, the repository needs a complete re-index, possibly in parallel. For Kythe, indexing every version is an adequate solution, considering their focus is on interoperability with cross-referencing in each version. Many of these tools from the

Software Engineering domain allow code comprehension in multiple versions in the sense that search results may be provided as an aggregated view of results in individual repositories. In this work, our objective is to allow cross-version querying involving de-deduplicating entities and efficient storage.

4.3.3 Source code analysis and other program meta-models

Source code analysis and manipulation has a long standing history in software engineering research. Some early work built tools to understand a single software repository and to query it using declarative [153, 75, 104] and natural languages [103]. Several approaches in literature have analysed software repositories for the purpose of understanding their evolution over time [48]. Data is collected from different sources including versioning and bug tracking systems, the retrieved data is modeled, stored and analysed for use cases such as change impact propagation, hotspot analysis, developer effort and fault prediction to name a few [48]. Program meta-models are built [112, 169, 51], adding a layer of indirection to the software at hand, with the objective of performing different types of analysis regarding its evolution. The meta-model CHA-Q [169] in particular persists the elements of the model in Neo4j. Since the Frappé meta-model already builds a storage-efficient dependency graph that scales to very large codebases, we investigated versioning the existing graph model with an emphasis on maintaining this storage scalability without compromising the existing querying capabilities.

Many forms of query languages have also been developed to enable querying a versioned software project in a declarative manner. In QWALKEKO [189], a git repository is directly viewed as a graph and queried using a combination of regular path expressions and logic query languages. ABSINTHE [98] is a general purpose tool for querying versioned software and the history is modeled as a directed acyclic graph. SysEdMiner [137] is another tool that uses mining algorithms on change histories with the specific goal of finding unknown systematic edits. These approaches share a similar goal of enabling querying of versioned software. Our objective is to build versioned graphs that can be queried with a general purpose language, but these approaches use domain-specific languages and change information from the SCM.

4.3.4 Syntactic and Semantic Differencing

Literature on syntactic and semantic differencing is also relevant research since we need to determine the delta of successive snapshots by identifying equivalent entities (Section 4.5). Algorithms operate on graphs of different granularities such as program representation graphs

[80], parse trees [220] and fine-grained Program Dependency Graphs (PDG) [107] (for duplicate code detection) thus producing varying levels of accuracy and semantics of the differences. The coarse-grained model in Frappé introduced a storage-efficient graph model for a different purpose of code comprehension on very large codebases. The performance and scalability of differencing algorithms on large codebases is uncertain. For example, Dex [160] employs a graph matching algorithm operating on ASTs and the algorithm reports a complexity of $O(n^4)$ in the worst case: it will be expensive dealing with codebases with millions of LOC.

What sets us apart from alternative approaches?

In Table 4.1 we compare our graph-based proposed solution (Section 4.4.1.4) with the following features of alternative approaches: (a) complexity of the model; (b) Scalability to millions of LOC; (c) Granularity of the model and storage efficiency for a source code comprehension use case; (d) Support for C/C++ codebases, and finally, (e) the ability to capture code changes precisely.

Table 4.1: Feature based comparison with alternative approaches

	Complexity	**Scalability**	**Granularity**	**Support**	**Precision**
Evolving graphs	simple	n/a	n/a	n/a	n/a
Industry tools	complex	✓	✗	✗	✓
Source code analysis	complex	?	✓	✗	✓
Differencing algorithms	moderate	✗	✗	✓	varied
Proposed Solution	moderate	✓	✓	✓	varied

The essence of our solution is to incorporate the principles of evolving graphs for versioning code dependencies. For reasons mentioned above we cannot employ an approach that assumes the availability of a graph delta. Instead our approach is a variation where the delta is calculated by means of node and edge resolutions (details in Section 4.5). For most of the projects from industry, although scalable, the model becomes too complex or the level of granularity is not suitable for the types of code comprehension queries that we deals with. Also, in existing approaches we have not seen particular support for C/C++ codebases. The approaches in source code analysis evaluates much smaller codebases (100-800k entities) compared to systems supported by Frappé (5M nodes, 80M edges). Due to the inherent complexity of the algorithms, differencing algorithms do not scale to large graphs with millions of LOC.

4.4 Versioning Dependency graphs

A ‘version’ or a ‘snapshot’ of the codebase is user-defined; incremental revisions to the code may be distinguished in terms of commits or an aggregated set of commits, and is generally associated with a revision number and/or a timestamp. In the rest of this chapter we use the terms version and snapshot interchangeably to refer to a particular point in time, either in the codebase or the corresponding dependency graph. Notations used in definitions and algorithms are given in Table 4.2.

4.4.1 Potential solutions to versioning dependencies

We describe several methods for representing an evolving dependency graph capturing multiple code revisions. In each of the approaches, we discuss trade-offs between implementation, storage efficiency and query simplicity.

4.4.1.1 Autonomous storage

With an autonomous storage solution, each individual snapshot is stored independently. In the case of querying a single instance at time point t_i, the required snapshot S_i can be made available on demand. However, with this approach, queries across versions become problematic: the results for each individual snapshot can be returned in isolation, but the answers from disconnected snapshots would lack context, having no association between entities across the versions. Identifying the equivalent nodes becomes a part of the query processor as these individual snapshots are disconnected.

Another major drawback is storage inefficiency. For large codebases, autonomous storage represents a significant storage cost with the size of each snapshot dependency graph exceeding by orders of magnitude the size of the source codebase. To give an idea of the approximate size, a single graph store we experiment with in this work is 12GB in size on disk (refer Table 4.4). We need to take into account that, for nightly snapshots of a codebase, only a few files would be changed and as a result, a major fraction of its dependency graph would remain the same.

4.4.1.2 Delta storage

In this approach, the first version of the graph is stored along with only the *changeset* or *delta* for each new version. Instead of keeping all the contents of the subsequent versions, only the delta is stored. The delta is a log of all the graph changes from S_i to S_{i+1} such as the addition and removal of nodes and edges. This approach addresses the storage inefficiency of the previous autonomous storage solution where identical parts of the graph need not be stored repeatedly and separately. Querying the graph generally involves a two-step process: first, the required snapshot must be reconstructed by applying the appropriate delta on it and, as a second step, the reconstructed graph is queried. When the number of versions increase, instead of storing only the initial version, intermediate snapshots may also be materialised.

Many studies of the graph domain [187, 101, 177] are a variation on this idea, where the delta is implicitly known for each graph evolution. In the case of a social network, there is a one-to-one mapping between the changeset/delta and the modification in the network, i.e. a new friendship created directly gets translated into addition of a link between those user nodes in the graph. In contrast, for dependency graphs we capture, particularly for C/C++ source with pre-processor macros, the entire graph may need to be computed for a given source code changeset. In such scenarios, identifying the delta for a changeset is one of the key challenges.

4.4.1.3 Use of an existing program meta-model

Several approaches in the software literature build program meta-models [112, 169, 51] of class-based software systems. They support versioning incorporating sufficient information from an AST, static and dynamic program state and version control systems, to form a first class representation for modeling software evolution. The software systems analyzed in these approaches are substantively smaller (100-800K entities) than the systems supported by Frappé (5M nodes, 80M edges). We do not believe these more precise encodings of evolving code graphs will successfully scale for our use cases.

4.4.1.4 Proposed unified model

We propose the versioned graph model shown in Figure 4.4 that represents an evolving graph while unifying and fully retaining the information of the individual snapshots. Similar designs have been employed to model memory graphs [183] and temporal graphs [187, 105], but have raised a unique set of challenges when applied to representing source code histories.

In this solution, we need to first identify the equivalent entities across versions; once corresponding nodes such as A in version 1 and version 2 are resolved (deemed equivalent), each node and edge in the graph can hold information of the temporal version interval for which that entity is valid (i.e. its lifespan). `ver_from` and `ver_to` denote the start and the end version of an entity (can represent a timestamp or a revision number). In Figure 4.4, node D's

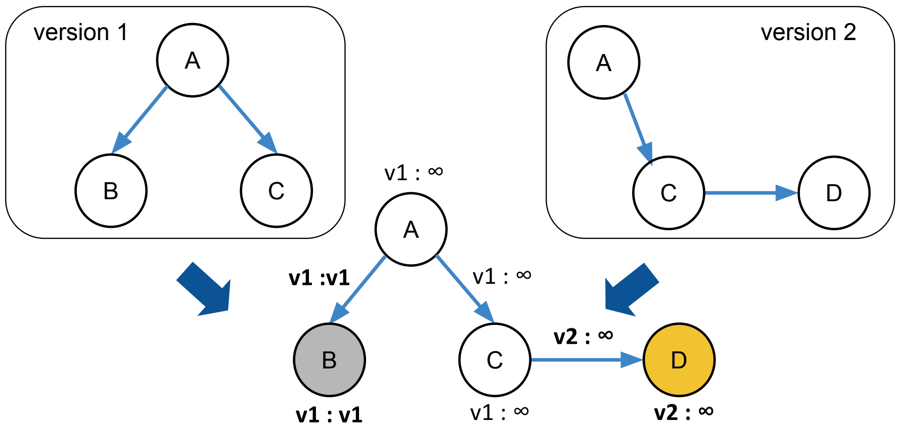


Figure 4.4: Versioned graph model with lifespan attributes on nodes and edges

`ver_from` = `v2`, as it was added in version 2. Node B and all its associated edges are denoted by `ver_to` = `v1`, indicating that B was deleted between `ver_from` = `v1` and `ver_from` = `v2`. The proposed versioned graph has the following advantages.

- All the information of the individual snapshots can be succinctly captured.
- Reduces memory footprint by preserving parts of the graph that have not changed.
- Finds equivalent entities, enabling queries across several versions.

However, this solution is limited to a linear history of versions. The crucial part of this model is determining equivalent software entities across two versions. In Section 4.5 we discuss how we addressed these challenges. A slight variation of this model is proposed [177] in which the lifespan is represented as multi-edges (i.e. additional edge corresponding to each timestamp/revision) instead of as an attribute. Although they observed faster query execution with multi-edges, we did not opt for this approach since when we version both nodes and edges, it would result in a graph having many more multi-edges scaling with the number of time-points that should be recorded.

4.4.2 Preliminaries of the Unified model

A snapshot graph S can be defined as $S(V, E, A_V, A_E)$ where V and E represents the nodes and edges, and A_V and A_E are sets of attributes on the nodes and edges respectively. The attributes include a specialised string label to characterise the type of the node or the edge. When the graph evolves, a sequence of these snapshot graphs is generated. We define an evolving graph that represents the combined union of all the individual snapshots.

Table 4.2: Summary of Notations in Definitions and Algorithms

Symbol	**Description**
$G(V, E)$	evolving graph
G_i, G_{ij}	subgraph of G at time t_i and time $[t_i, t_j]$ resp.
$S_i(V_i, E_i)$	snapshot graph at time t_i
A_V, A_E	sets of attributes on nodes and edges
$A_{ts}, A_{\text{verFrom}}$	attribute that holds the start time, version resp.
A_{te}, A_{verTo}	attribute that holds the end time, version resp.
N, R	Added nodes or edges after the resolutions
`idMap`	Dictionary of $\{id_i : id_{i-1}\}$ pairs
nodeMap	Dictionary of {hashkey:id} pairs for nodes
edgeMap	Dictionary of {hashkey:id} pairs for edges
$id(v), id(e)$	id of a node or edge

Definition 2 (Evolving Graph). An evolving graph G in time interval $[t_i, t_j]$ (or revisions r_i to r_j) is the linear collection of snapshot graphs. $G = S_i, S_{i+1}, ..., S_{j-1}, S_j$. G is characterised by the union of all attributes of individual graphs and two temporal attributes on each node and edge, A_{ts} (start) and A_{te} (end) to denote that the entity is valid in time interval $[t_s, t_e]$ where $A_{ts}(v), A_{te}(v) \in A_V$ and $A_{ts}(e), A_{te}(e) \in A_E$. Until a given entity is deleted, the attribute $A_{te}(v) = A_{te}(e) = \infty$.

4.4.3 Queries in the Unified Model

The queries performed on these evolving graph sequences can be categorised in two main types: queries that are performed on a single graph at a particular time point t_i and queries that are performed on a series of subgraphs valid across a given time interval $[t_i, t_j]$. A time-point query is a query on snapshot S_i. On the evolving graph, the required subgraph at time point t_i, G_i needs to be filtered before the query can be performed. A time-interval query runs on a subset of n graphs valid in time-interval t_i and t_j. Formally, the two types of queries are defined as follows.

Definition 3 (Time-point Query). The subgraph G_i at time instance t_i of the evolving graph $G = (V, E, A_V, A_E)$ is defined as $G_i = (V_i, E_i)$ where $\forall v \in V_i : A_{ts}(v) \leq t_i \leq A_{te}(v)$ and $\forall e \in E_i : A_{ts}(e) \leq t_i \leq A_{te}(e)$. A query that executes on subgraph G_i is a time-point query.

Definition 4 (Time-interval Query). The subgraph G_{ij} at time interval $[t_i, t_j]$, $i < j$ of the evolving graph $G = (V, E, A_V, A_E)$ is defined as $G_{ij} = (V_{ij}, E_{ij})$ where $\forall v \in V_{ij}$:

$A_{ts}(v) \leq t_j \wedge A_{te}(v) \geq t_i$ and $\forall e \in E_{ij} : A_{ts}(e) \leq t_j \wedge A_{te}(e) \geq t_i$. A query that executes on subgraph G_{ij} is a time-interval query.

4.5 Node and Edge Resolutions

Dependency graphs generated corresponding to two versions of the codebase are represented as two complete graphs. It is necessary to resolve entities in the graph that are equivalent across two graphs so they can be marked and need not be stored redundantly. In this section we discuss methods to identify and resolve entities in our dependency graph model and demonstrate why the location of an entity is a significant aspect of resolutions.

4.5.1 Resolution Rules

4.5.1.1 Resolutions in a single version

Before a node (or an edge) can be resolved across two versions, we need to first be able to uniquely resolve all nodes in a single version. When a symbol is processed multiple times as a result of multiple compilation units, Frappé already has an approach to de-duplicate such nodes in a single version. For example, if structure `A` is defined in a header file and if both `foo.c` and `bar.c` include the header file, it is de-duplicated and only a single node is created representing the structure `A` irrespective of the number of times `A` is seen as a result of the header file inclusions. To uniquely identify a structure, the combination of the symbol name, type, source file in which the structure exists and use location attributes are used (Ref. Section 4.2.2 for use location). Next, we present some examples that illustrate the importance of including location information to resolve entities.

```
#ifdef BLAH
    struct foo{
        int a;
    }
#else
    struct foo{
        int b;
    }
#endif
```

Figure 4.5: Challenges with the pre-processor

Example 1. Figure 4.5 shows an example that highlights the importance of use location to uniquely distinguish nodes when a pre-processor is involved. The use location of the structure is required to uniquely distinguish it due to the pre-processor as illustrated in Figure 4.5. Depending on whether the macro `BLAH` is defined, we need two different nodes created for the structure `foo`; if `BLAH` is defined, a node must be created with '`a`' as a field and with '`b`' otherwise. Use location (found on the containment edge) is added to name, type, source file attributes to identify them as two structures. In this example, the intricacies of the pre-processor make location an important property to distinguish nodes.

Example 2. A common local variable `i` may appear multiple times within the same parent. The parent refers to the parent container (function, structure, union etc.) in which the local variable is defined. Since the current Frappé model does not store scoping information, the combination of symbol name, type, source file, parent id and the variable location is used to uniquely identify a local variable. If a local variable is a part of a macro, the spelling location (Ref. Section 4.2.2 for definition) is also required to uniquely identify it.

Table 4.3: Attributes used to resolve each NodeType apart from `name` and `type`. Function GENERATEHASH(v)

NodeType	Resolution attributes in addition to `name,type`
source_file, module, primitive, directory	_
namespace	parent_id
parameter	parent_id, index
macro	source_fileid, start_line
local static_local	parent_id, source_fileid, start_line, end_line, name_start_line, name_start_column, name_fileid
other	parent_id, source_fileid, start_line, signature

Table 4.3 summarises the attributes used to resolve specific node types. A hash function, GENERATEHASH(v), combines these attributes. Name, type and signature are attributes on nodes, while all others require traversing different incident edges of a node type. For example, in order to retrieve the parent_id, respective containment edges (such as `contains`, `has_param`, `has_local`) of a node must be visited and all file ids and location information are recovered from the attributes on the `file_contains` edge (Figure 4.3). Once the nodes are resolved, resolving

the edges is fairly straightforward. Frappé graph model is a multi-graph having multiple edges between the same source destination nodes; thus source, destination, edge type combination is not sufficient to uniquely identify an edge. Therefore all the attributes on an edge, including location information, are used to uniquely identify it (in function GENERATEHASH(e)) as it is guaranteed to have no duplicate edges with exactly the same properties.

Once all the nodes within a single version are resolved, we define a method to resolve entities across two versions. The same rules for a single version can be applied for multiple versions. The only difference between distinguishing two nodes in a single version and a particular node across versions is the notion of time. In principle, we should be able to identify them with the same rules. In one of our experiments (Section 4.6.2) we verify the feasibility of this approach.

However, we also need to ask several fundamental questions in order to agree on the equivalence of entities across versions. The effect of refactoring code in the new version may yield different interpretations for equivalence among end-users. For example, if a parameter was added to a function, do we identify the function to be a new one in the new version? If a function is renamed, can we claim that the renamed function is equivalent to the function in the previous version? The answers to these questions are goal-specific. Our objective is to provide end-users with the right level of abstraction that satisfies a reasonable set of use cases. In next sections, we demonstrate the extent to which we support these different use cases. In Section 4.5.3 we discuss how we improve the model accounting for none or relative location information. Constructing a versioned model of code dependency graphs presents a unique set of challenges not available in other graphs and are summarised below.

- Finding the deltas. A textual change in the source files may or may not result in a change in dependency in the corresponding graph. Thus the computation of the delta should take place as a post-processing step involving a mechanism to first find equivalent entities.
- Storage cost of the proposed model. The current Frappé model is lightweight, storing only the most critical components required from the build process capturing essential semantics. Additional information such as the AST could be maintained but the cost of storage need to be considered.
- Pre-processor. Many of the challenges associated with resolving nodes within one version can be attributed to the pre-processor. As we have seen with examples, the structure of the source will have very different meaning depending on the source path of conditional compilations.

- Right level of abstraction for multiple versions. Determining what makes two entities equivalent across two versions that are acceptable to a majority of use cases is also challenging because users will have divided assessments on what the equivalence is in the case of refactoring code.

In the following sections, we present our experiences in building a versioned dependency graph and discuss how the above challenges were addressed in the process.

4.5.2 Versioned graph construction

Here we describe a simplified model constructing the versioned graph in Figure 4.4 and then improve it (in the next section).

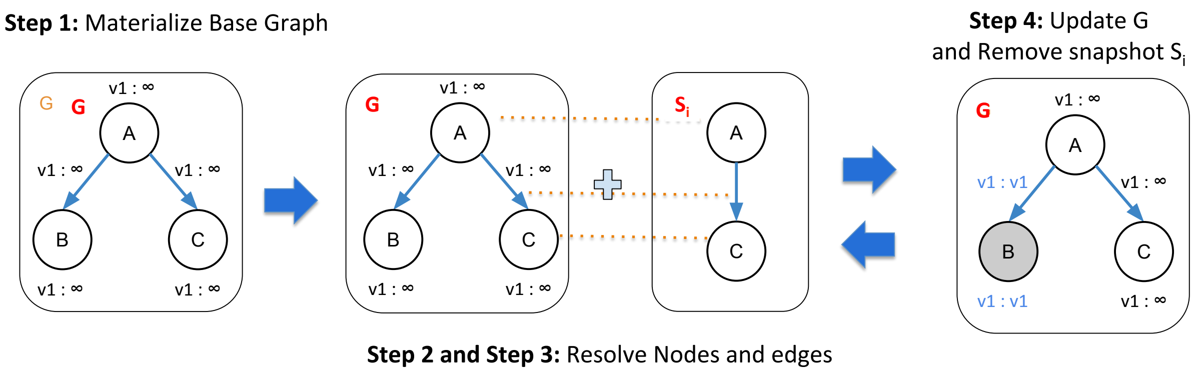

Step 1: Materialize Base Graph
Step 4: Update G and Remove snapshot S_i
G
G
v1 : ∞
A
B
C
S_i
v1 : v1
Step 2 and Step 3: Resolve Nodes and edges

Figure 4.6: Steps in constructing the versioned graph

Figure 4.6 illustrates the steps involved in building a versioned graph. In *Step 1*, a graph is selected to be the base graph $G = (V, E)$ (which later evolves with versions) and materialised with additional node and edge attributes to represent the version interval an entity is valid in, such that initially, $\forall v \in V$, $A_{\text{verFrom}}(v) = 1$ and $A_{\text{verTo}}(v) = \infty$ (similarly $\forall e \in E$, $A_{\text{verFrom}}(e) = 1$ and $A_{\text{verTo}}(e) = \infty$) assuming base version $i = 1$. The base graph is incrementally updated, with subsequent snapshot graphs incorporating the resolutions below.

Step 2 involves resolving entities and updating G with node additions and removals. Algorithm 1 outlines the steps involved in resolving the nodes in the new graph S_i with the base graph G where S_i is the snapshot graph for version i. Depending on the type of the node v, GENERATEHASH(v) function generates a unique key with a combination of the attributes given in Table 4.3. Hash functions are generated for each node belonging to $i-1$ version graph extracted from the base graph G and stored in nodeMap as a map of hashkeys to ids. (lines 5-9). The validity of a node in the base graph is tested in function ISVALID. The `idMap` in

Algorithm 1 Node resolutions

```
Input: Base $G = (V, E)$, New $S_i = (V_i, E_i)$ for version $i$
Output: Added nodes $N$, Removed nodes $R$, idMap
 1: $N \leftarrow \{\}, R \leftarrow \{V\}$
 2: nodeMap $\leftarrow \{\}$                    ▷ dictionary for <hashkey:id>pairs
 3: idMap $\leftarrow \{\}$                      ▷ dictionary for $<id_i : id_{(i-1)}>$pairs
 4: /* Generate map of nodes for version $i - 1$ from $G$ */
 5: for $v \in V$ do
 6:     if isValid$(v, i - 1)$ then
 7:         nodeMap $\leftarrow$ nodeMap $\cup \{($GenerateHash(v)$, id(v)\}$
 8:     end if
 9: end for
10: /* Generate hash for each node in $S_i$ */
11: for $v \in V_i$ do
12:     key $\leftarrow$ GenerateHash$(v)$
13:     if key $\in keys$(nodeMap) then
14:         idMap $\leftarrow \{ id(v)$,nodeMap.get$(key) \}$
15:         $R \leftarrow R \setminus \{v\}$
16:     else
17:         $N \leftarrow N \cup \{id(v)\}$
18:     end if
19: end for
20: function isValid$(v$, current_version$)$
21:     if $A_{\text{verFrom}}\ (v) \leq$ current_version $\leq A_{\text{verTo}}(v)$ then
22:         return true
23:     end if
24: end function
```

Algorithm 1 maintains a mapping between the id attribute in S_i to corresponding (differing) ids in G, if a mapping exists. Iterating through nodes in the snapshot graph (lines 11-19), the mapped id in base graph is retrieved from `idMap` to generate the key. A crucial and a less apparent aspect of this iteration of nodes is its order – the parents must always appear and thus be resolved before its children. For example, all source_file nodes must be resolved (and corresponding ids included in the `idMap`) before function nodes are processed.

In *Step 3*, a similar approach is taken for resolving the edges (Algorithm 2) of the graph using the function GenerateHash(e). In order to generate the key, when processing the edges in S_i (i.e. E_i), the same `idMap` output in Algorithm 1 is used to find the matching ids for source, destination and other ids related to the edge. For example, for attributes `use_file_id` and `name_file_id` that appear on the edge, the global id that corresponds to the previous version (as resolved in `idMap`) must be used when generating the hash key.

Algorithm 2 Edge resolutions

```
Input: Base $G = (V, E)$, New $S_i = (V_i, E_i)$ for version $i$, idMap
Output: Added edges $N$, Removed edges $R$
 1: $N \leftarrow \{\}, R \leftarrow \{E\}$
 2: /* Generate hash for each edge per node, in $S_i$ */
 3: for $v \in V_i$ do
 4:     edgeMap ← CreateEdgeMap($v$)
 5:     if edgeMap is null then
 6:         mark all edges of $v$ as new
 7:         continue
 8:     end if
 9:     for $e \in v$.getRelationships() do
10:         key ← GenerateHash($e$)
11:         if key $\in$ $keys$(edgeMap) then
12:             $R \leftarrow R \setminus \{e\}$
13:         else
14:             $N \leftarrow N \cup \{id(e)\}$
15:         end if
16:     end for
17: end for
18: function createEdgeMap($v$)
19:     edgeMap ← {}                          ▷ dictionary for <hashkey:id>pairs
20:     $u \leftarrow$ idMap.get($v$)
21:     if $u$ is null then                   ▷ no matching id found
22:         return null
23:     end if
24:     for $e \in u$.getRelationships() do
25:         if isValid ($e$, $i - 1$) then
26:             edgeMap ← edgeMap $\cup$ {(GenerateHash($e$), $id(e)$)}
27:         end if
28:     end for
29:     return edgeMap
30: end function
```

For each node v in the S_i graph, an edgmap is created (line 4) with the corresponding nodes' edges in the previous version. CreateEdgeMap function (lines 18-30) does this by first finding the corresponding global id (line 20) and creating the above map with the hash function generatehash(e). An edgemap would not be created if a matching id was not found in version $i - 1$ (lines 21,5). Otherwise, a hash is generated for each edge of S_i, and checked against the edgeMap to find if a matching edge exists.

Considering that $|E| >> |V|$ it would be not be efficient to first generate the hashmap of edges for version i before generating the map for version $i + 1$ (as we have done for nodes).

Instead, as shown in Algorithm 2 without arbitrarily iterating through edge list, edges were processed per node to exploit caching behavior for better efficiency.

After the resolution, nodes and edges that are identified to be equivalent in two versions will continue to remain valid in the current version i being examined. Entities that no longer exist (R) will be closed with `ver_to` $= i - 1$. Entities that do not get resolved to an entity in the previous version(N) will be added to the graph with `ver_from` $=$ `i`. Finally, in *Step 4*, snapshot graph S_i is no longer required and can be removed as all the information is captured in G. Steps 2 and 3 are repeated until all snapshot graphs for all versions have been processed.

The node resolutions take up time complexity $O(|V|)$ as hash function generation executes in $|V|$ and $|V_i|$ (resulting in $|V + V_i|$) for snapshot and base graphs respectively. Edge resolution runs in similar time complexity, resulting in the overall resolution algorithm time complexity of $O(V + E)$.

4.5.3 Model Improvements

The current definition of resolutions adds new nodes and edges that cannot be resolved due to location changes. Section 4.5.1 discussed why locations are important to identify entities precisely. This may be suitable when de-duplicating entities in a single version, but, when considering new source versions the location information becomes highly volatile. We investigate if resolutions can be increased when the locations are ignored without a loss in precision.

```
# File foo.c in version 1

13 void foo(int a){
        ...
18 }
```

```
# File foo.c in version 2

15 void foo(int a, int b){
        ...
20 }
```

Consider a function `foo` in source file `foo.c` (version 1) with an integer parameter `a`. In version 2, a new parameter `b` has been added and function is moved further down in the same source file `foo.c`. Due to the change in location attribute, the function in version 2 will not be matched to a function in version 1, thus a new node will be added. The resulting subgraph is shown on the left of Figure 4.7, as the foo function in version 2 will be treated as a new function, foo′. However, in many use cases, the user would like to see an abstraction where foo′ is a *changed* version of foo and not identified as a new symbol.

No Locations. In order to achieve this, we consider the subset of attributes excluding location attributes. If there are no other functions in foo.c with the name foo, the function can be

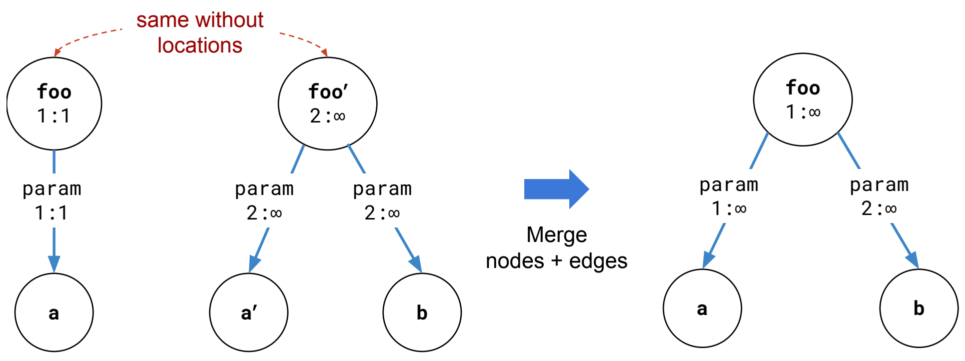
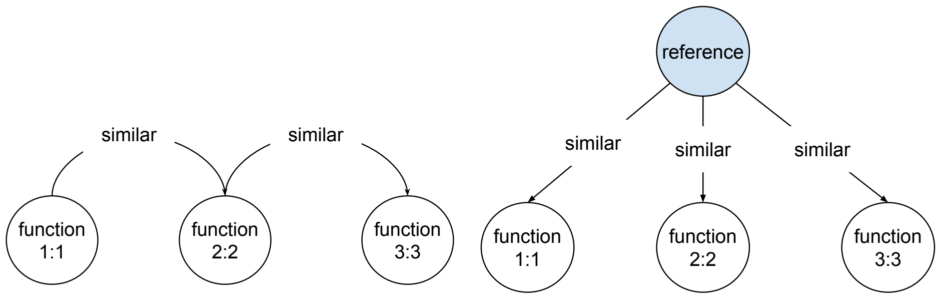


Figure 4.7: Merging the nodes and edges

resolved and we can safely decide that these two foo functions are equivalent, even when the line numbers differ. As a result, nodes foo and foo′ can be merged. Similarly, parameters `a` and `a`′ can also be merged to create a more simplified versioned graph (right of Figure 4.7). The lifespan attributes on the param edge will denote that parameter `a` is valid in both versions, while `b` was introduced in version 2. Observe that as a result of merging nodes, we may lose the information on edges connecting them: `file_contains` edges of foo and foo′, and `param` edges of `a` and `a`′ have different values describing their locations. When merging, in order to preserve information on edges, we propose two approaches.

1. Introduce `similar` edges – Without collapsing nodes, a `similar` edge would be introduced between foo and foo′ and all the edges would remain intact (Figure 4.8a). Alternatively, all the different versions of subsequent foo nodes can be connected via a single *reference* node (Figure 4.8b). The advantage of the latter method is that all the different 'versions' of foo would be indexed via a reference node and would not need to iteratively traverse through `similar` edges.



(a) Connected Similar edges **(b) Connected by a reference node**

Figure 4.8: Alternate Similar edge representations

2. Move location information – Another approach is to merge both node and respective edges while the location information on the edges would be moved into separately versioned storage. The merged edge id would contain multiple versioned records/rows corresponding to each version. To illustrate: assuming the merged `file_contains` edge id is 25, the location can be stored in two versioned records corresponding to the two (or more) versions.

```
edgeId | version | st_line | end_line |...
  25   |   v1    |   13    |   18     |...
  25   |   v2    |   15    |   20     |...
```

The latter approach requires an additional lookup on a table to retrieve any attribute on the edge while with `similar` edges, information is found within the graph itself. The latter solution is also not favorable when only a graph database is utilized for all data management needs.

Relative Order. With the current model in Frappé, some entities cannot be resolved if we ignore the location information entirely. For the code segment below, as the local `i` appears multiple times within the same function `bar`, it will be impossible to distinguish them without absolute or at least relative locations.

```
# File bar.c in version 1

   void bar(){
15     int i;
       ...
24     int i;
       ...
27     int i;
   }
```

```
# File bar.c in version 2

   void bar(){
20     int i;
       ...
29     int i;
       ...
32     int i;
   }
```

As in the example above, taking only the name, type, source_fileid, parent_id subset cannot distinguish the local variable `i` as there are three such instances. The solution we propose here distinguishes them by the relative order as illustrated in Figure 4.9. Retrieving relative locations require sorting the locations instances in each version and matching them in order. Note that at this stage we assume the function `bar` in the two versions has been already resolved to be equivalent functions. If the parents of these locals (i.e. bar function) are not identified to be equivalent, we cannot match `i` at all. In the next section we evaluate our approaches of the initial model a) with locations; b) ignoring locations; and c) using relative locations.

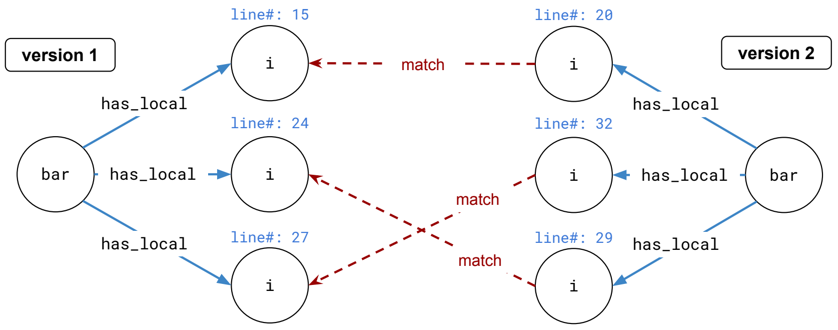


Figure 4.9: Taking relative locations

4.6 Evaluation

In this section, we explore the following research questions relating to constructing the versioned graph. Evaluation of queries in the versioned model is discussed in Section 4.7.

RQ1. What percentage of entities can be resolved with existing rules for resolutions?

RQ2. Can the entity resolution be improved by incorporating more rules to account for changing location information?

RQ3. How fast does the graph grow and what is the storage benefit of having a versioned graph compared to an autonomous storage solution?

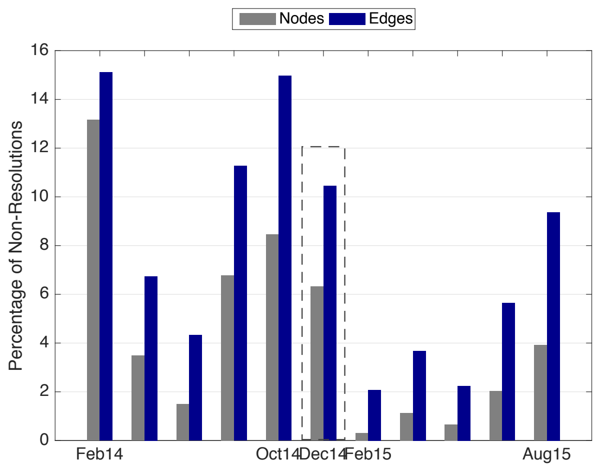
Table 4.4: Dataset characteristics and description

Dataset	Nodes	Edges	Files processed	KLOC	Size on Disk (+index)	Description
Quake^3	114,465	934,220	764	384	122 MB (+62 MB)	Quake online game
OIC	5,279,131	76,909,072	13,963	13,524	7.3 GB (+4.7 GB)	Large Oracle internal codebase

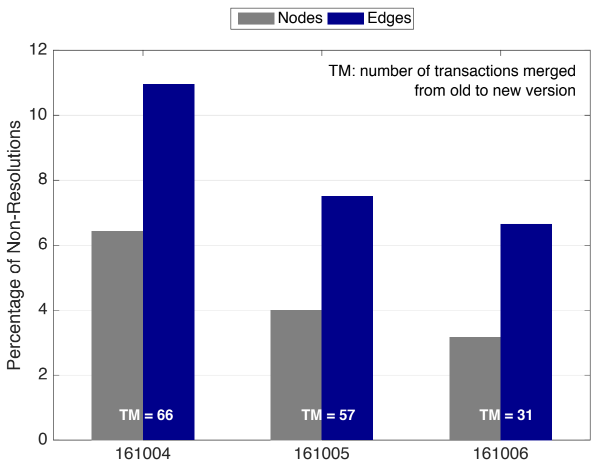
4.6.1 Datasets

Two datasets have been used in our analysis. We use the open source Quake^3 dataset [159] representative of a long-lived, stable, medium-sized codebase. The dependency graphs are created from 12 snapshots of the Quake source, each using a commit that is roughly four months apart starting from February 2014 to August 2015. The Quake codebase has a slow rate of change, with four month intervals chosen to match a rate of change comparable to the nightly changes of a more active codebase.

We also use an Oracle internal codebase (OIC), representing a large codebase with approximately 13,000 Kilo lines of code (KLOC) similarly is a long-lived, stable codebase with a comparable relative rate of change. We have experimented with four nightly integration builds (snapshots) of this codebase. The dataset characteristics are shown in Table 4.4. KLOC are shown as an indicator of the size of the code repository. The values include only code lines in C/C++ source and header files and exclude any blank and comment lines. The nodes, edges, and size on disk describe the dependency graphs that are created in Neo4j. The index includes the `name`, `type` properties of the node and the `file_id` property on the edges.



(a) Non-resolutions for Quake



(b) Non-resolutions for OIC

Figure 4.10: Non-resolutions for Quake and OIC datasets

4.6.2 Resolution Evaluation

We first investigate the percentage of entities we cannot resolve (to be equivalent) across versions using the simplified model described in Section 4.5.2. The percentage of nodes and edges that could *not* be resolved in Quake and OIC databases are shown in Figure 4.10a and Figure 4.10b respectively. The y-axis shows the percentage of non-resolutions compared to its previous version in each dataset and the x-axis shows the version date. For example, in Figure 4.10a, the highlighted group for Dec. 2014 denotes that 6.2% of nodes and 10.2% of edges could not be resolved compared to its previous version in Oct. 2014. In Figure 4.10b for the Oracle internal codebase, the bars are annotated with the number of changesets merged from the old to the new version as an indication of code changes. Note that this model uses absolute location information; if a function has been moved down by a couple of lines, the key combination will not match a node in the previous version and is thus a non-resolution. The accuracy of the resolutions was confirmed by manual inspection on the Quake database before advancing to the larger codebase.

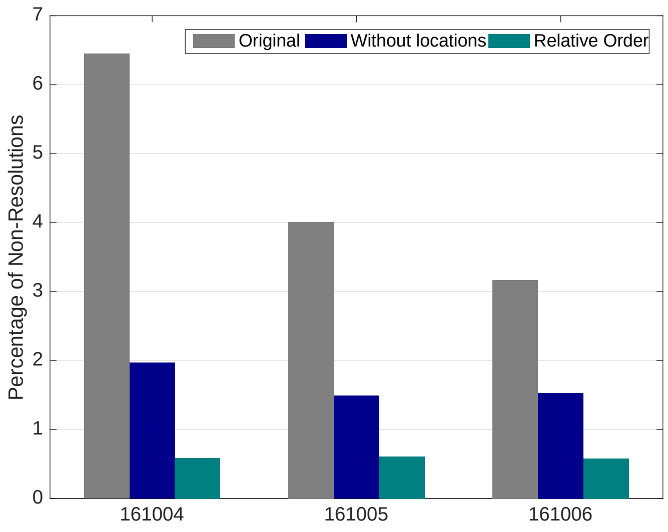


Figure 4.11: Non-resolutions for OIC with improved resolution strategies

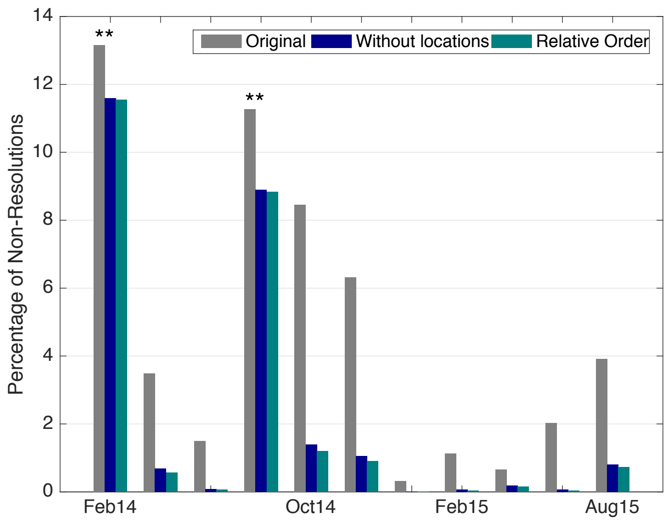


Figure 4.12: Non-resolutions for Quake with improved resolution strategies

Next we incorporate the model improvements discussed in Section 4.5.3 and monitor how the non-resolutions are affected. The results are presented in Figure 4.11 and Figure 4.12 for the OIC and Quake respectively. The bars in each group show the percentage of non-resolutions using the original definition with locations, without absolute locations and using relative order respectively.

For OIC, it can be observed that a large proportion (98%) can be resolved (non-resolutions, 2%) without location information. The resolutions can be refined even further by using relative locations reducing the non-resolutions to about 0.5%. However, a limitation of this approach is that we are unable to detect renames. For example, if a function is renamed it is considered to be a new function. The remaining percentage of entities that could not be resolved (i.e. marked as new nodes/edges) reflects the actual nodes and edges that were added in the subsequent version representing code changes.

For Quake, the results show a similar trend of improved resolutions without locations and relative orders, with the exceptions marked with an asterisk(*). It was revealed that in these particular cases the code in Quake has been refactored specifically by upgrading third party modules, thus replacing the module names. As a result, all child nodes that depend on these modules will be marked as new nodes, hence there is only a minor improvement in the non-resolutions.

Graph growth and storage benefit. The graph for the OIC codebase grew, with 4.5% more nodes and 8.3% more edges on average added to the graph for each nightly snapshot with the basic resolution strategy. The values for Quake are 4.8% and 7.4%. With our existing system capability, we compare the storage benefit of the proposed unified model to independently storing n different snapshots graphs. Storing all 4 snapshots of OIC require a total 55.9GB compared to the 18.6GB for the unified graph, demonstrating a 66.7% reduction in storage. For a versioned graph of 12 smaller Quake databases, an 86.1% reduction is observed.

4.6.3 Discussion

Due to the fact that we use the symbol name to resolve nodes, we are unable to handle renames; if a function is renamed in a subsequent version, we identify that function to be a new one. In some contexts, some may argue that the renamed function indeed needs to be regarded as a new one. We can make use of other attributes to resolve entities: by comparing the hash of the function source, we can conclude if two functions are identical or not, but the hashes alone will not facilitate identifying changes. Due to the volatile nature of the locations, one may be tempted to ignore locations altogether, but as we highlighted in Section 4.5.1, location information plays an important role in precisely distinguishing entities even within a single version. Alternatively, we need to store more information in the dependency graphs, such as scoping details, hash of the source or the complete AST. Despite having to incur the additional cost of storage, this information may also make the dependency graph more complex. This is a trade-off that may need to be taken into account in future work.

It would be interesting to investigate more advanced resolution strategies such as [214] and addressing the renaming problem [186]. Systematic study of exact precision and recall measures for the resolutions is another area for future work.

4.7 Queries on Versioned Graphs

The objective of this section is to investigate how the existing and new queries will perform on the multi-versioned graph model. Specifically, we explore the following research questions.

RQ1. What is the overhead of running current Frappé use cases as a time-point query (Definition 3) on the versioned graph? Are the results of the versioned model correct compared to running them on independent versions?

RQ2. As a benefit of the proposed model, what are the additional time-interval queries (Definition 4) that can be written on the versioned graph?

4.7.1 Time-point queries

Now that we have a single versioned graph in place of n different snapshot graphs, our first objective is to run the same use cases we had for Frappé (ref. Section 4.2.3) in the graph on a user-specified version. We note the accuracy and performance of queries on the versioned graph compared to running them on individual snapshots. We have selected two fundamental queries in all the workloads: one that searches a text symbol, and another that retrieves the shortest path between two nodes (reachability) to cover path navigations.

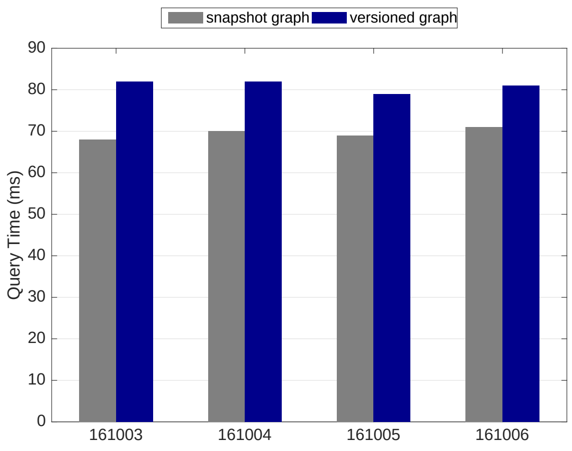
Version selection. When querying the unified model, we need to filter the nodes and edges that belong only to the requested version. The search query on the versioned graph translates to an additional condition on the node to check if the symbol is valid in a given version. An additional constraint is added to the type of the node, to limit the number of results that are returned. A search query in Cypher that looks up a function `calculateBFS` in version 5 can be written as follows.

```
START n = node:node_auto_index('name: calculateBFS')
MATCH (n)
WHERE n.type='function' AND n.ver_from <= 5 AND x.ver_to >= 5
RETURN n
```

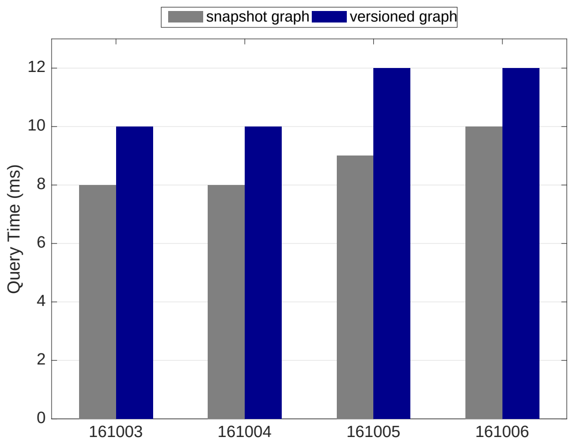
The reachability query on a versioned graph involves traversing a series of intermediate nodes and edges, checking for conditions to filter the nodes and edges that belong only to the requested version. Our query involves returning the path between two function nodes via `calls` edges that are at a maximum of 40 hops from each other. A query in Cypher to search paths in version 5 is given below.

```
START n = node(23), m = node(58)
MATCH p = shortestPath (n-[r:calls*..40]->m)
WHERE all(x IN r
    WHERE x.ver_from <= 5 AND x.ver_to >= 5)
RETURN p, n, m
```

We show results of searching for frequent symbol '`i`' that returns around 32K results. For reachability, we have shown query results for two functions that are 40 hops apart. For accuracy, we verified that for each query, the results returned querying the versioned graph are the same as the results returned from running the query on the individual snapshots. Next, we observe the extra overhead the additional filter adds to the current queries. The query results for search and reachability are shown in Figure 4.13a and Figure 4.13b respectively for OIC. In both figures, the orange bars represent the average query times over 100 runs in the versioned graph compared to the average query times running the same query on individual snapshots (grey bar). The query times for Quake are omitted as they are in the order of milliseconds, making it too small to compare across snapshots.



(a) Search Query. Symbol = 'i'



(b) Reachability Query. path length = 40

Figure 4.13: Time-point queries for OIC

Observing Figure 4.13a and Figure 4.13b, it can be seen that there is a small overhead in query times due to the additional filter. However, it does not seem to increase with the number of versions added. In future work it would be interesting to confirm this conjecture with more data points (versions). The lifespan attributes `ver_to`, `ver_from` were not indexed as the cardinality for each version number would be low, thus not helping the reduction of the intermediate result set.

4.7.2 Time-interval Queries

With the entities across versions being identified, we examine the questions we can ask on the new versioned graph. A code review is one such use case that involves results from n versions ($n \geq 2$). A standard code review typically involves careful and detailed investigation of the changes that have been made to the code since the last snapshot. With the current information available on the versioned graph, out of hundreds of functions available we are able to highlight those that changed from one version to another. A reviewer may be interested to focus on the highlighted functions where:

- Calls/parameters have been added and removed, and/or
- Reads and writes have been added to global variables.

With the emphasis on the changes, a reviewer can examine these functions to confirm that there have not been constraints that are violated or that the changes made were all intentional.

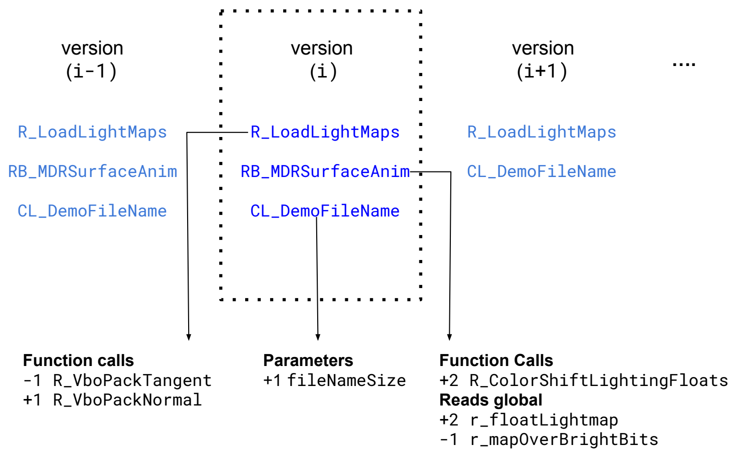


Figure 4.14: A function history showing modified functions

Although the more common scenario for a code review use case is to compare the changes in two versions, we can easily extend the time interval to show the changes in multiple versions. For example, with the information on versioned graphs, a user can be presented with a timeline or history of the functions that have changed, across n versions.

Figure 4.14 shows functions that were changed between two versions in Quake graphs. If a listed function is selected, the pre-configured or all of the changes can be highlighted. `RB_MDRSurfaceAnim` function in version i shows a few changes compared to its previous version $i-1$: 2 new `calls` are added(+) to function `R_ColorShiftLightingFloats` and, 1 `reads` is removed(-) from global variable `r_mapOverBrightBits` etc. Note that a change in function here is characterised only by any change in connecting edges, which may not reflect some syntactical changes to the function that the Frappé model does not store. To account for this limitation, the function source may have to be stored within the graph.

Implementation of a function history:

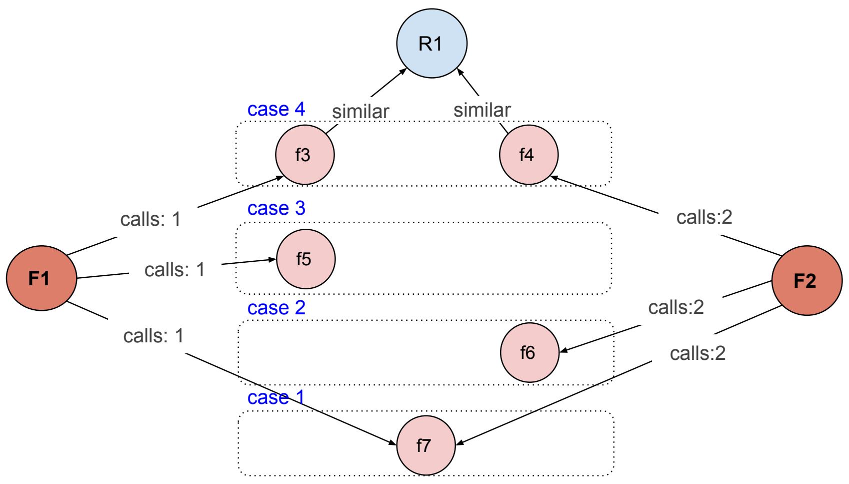


Figure 4.15: Possible cases of change in function calls between two versions

As discussed in Section 4.5.3, with the introduction of similar edges with reference nodes, we illustrate the steps in involved in extracting the above function history. Assume that `F1` and `F2` correspond to a function in two versions; $v1$ and $v2$. For this example, we consider that the function `calls` between these two versions are of interest. We describe the four possible cases of modifications in Figure 4.15.

- **case 1:** Calls the same function $f7$ in both versions.
- **case 2:** A new call to function $f6$ has been added in $v2$.

- **case 3:** The call to function $f5$ no longer exists in $v2$.
- **case 4:** Calls function $f3$ and $f4$ that are connected by a `reference` node `R1`. This means that $f3$ and $f4$ differ by locations, but refer to equivalent functions.

When displaying a history of a function, we are most often interested in filtering out cases 1 and 4 and showing the instances in case 2 and 3, where function calls were added and removed moving from $v1$ to $v2$. A variation on the above cases is when there may be multiple calls to the same function. For example, although $f7$ is referenced in both versions, the number of times it has been called may differ. Whether this information is of interest is up to the user. The following query in Cypher returns a list of function calls for both versions of `F1` and `F2`.

```
MATCH (x)
WHERE ID(x) = 30
MATCH (x)-[c:calls]->(n)
WHERE c.fromId = 4
WITH x, COLLECT([n.name,COUNT(c)]) AS v1
MATCH (y)-[c:calls]->(n)
WHERE ID(y) = 40
WHERE c.fromId = 5
WITH y, from1 + COLLECT([n.name, COUNT(c)]) AS v2
UNWIND v2 AS row
WITH row[0] AS function, row[1] AS count
RETURN function, SUM(count)
```

(a) Cypher retrieving function calls

```
function, v1, v2
f3, 1, 0
f4, 0, 1
f5, 1, 0
f6, 0, 1
f7, 1, 1
```

(b) Query Output

Figure 4.16: Cypher Query and that retrieves function calls in two versions

Once the output above is retrieved, we can filter out (a) functions with matching function calls (case 1) and (b) functions that are connected by `reference` nodes (case 4).

```
START n = node(99)
MATCH (n)<-[:similar]-(r:reference)-[:similar]->(x)
RETURN x
```

For each function node in the above output, the idea is to check if a similar node from the other version exists in the output list. If it does, the function node pair is omitted from the final list. If a matching similar node does not exist, and the $v1$ or $v2$ count is zero, it is denoted as a function call addition or a removal respectively.

4.8 Summary

In this chapter we described our graph-based approach to managing multiple revisions of a codebase. All of the approaches are conducted on a graph database system demonstrating the feasibility and performance of constructing and querying versioned graphs. The contributions of our work is summarised below.

- **Resolve entities across versions:** We have shown how entities across two versions can be resolved and discussed the extent to which the resolutions can be improved without changing location details (Section 4.5). We also presented our scalable solution for constructing the versioned graph. (Section 4.5.2).
- **Experiments on a large codebase:** We evaluated a real codebase consisting of around 13 million lines of code. We presented the rate of growth in the versioned graph and the storage benefits of upto 86% as evidence of a feasible and effective solution (Section 4.6).
- **Proposed queries on the versioned model:** We showed that current comprehension workloads can be easily integrated into versioned graphs with only a marginal overhead in query time. The proposed versioned graphs also enable new use cases involving querying across versions (Section 4.7).

Code comprehension tools need to consider the inherent revisions in codebases over time and incorporate strategies in their underlying models to manage changes effectively, without compromising on efficiency and performance. Our experiences with Frappé pave the way for the development of such tools.

Chapter 5

Edge Labeling Schemes for Graph Data

In Chapter 4 we discussed how well graph databases such as Neo4j are able to handle evolving graphs representing software code dependencies. We explored the types of useful queries that can be run on a versioned graph model and discussed storage benefits of the proposed model. While the data model we choose has an effect on the efficiency of the queries, the internal graph databases themselves take measures to facilitate efficient query processing.

Given that the throughput of many graph queries can be significantly affected by disk performance, graph database systems need to focus on effective graph storage for optimizing disk operations. While many graph database systems (Neo4j, Sparksee, etc.) take approaches to define memory hierarchies for efficient query processing, in this chapter we investigate how graph data storage can be improved at the physical level. In this work, our goal is to optimally assign edge labels coupled with edge indices to achieve improved disk locality for efficiently answering typical graph queries, without modification to the storage internals of the graph system at hand.

We propose edge-labeling schemes GrdRandom and FlipInOut, to label edges with integers based on the premise that edges should be assigned integer identifiers exploiting their consecutiveness to a maximum degree. We provide extensive experimental analysis on real-world graphs, and compare our proposed schemes with other labeling methods based on assigning edge IDs in the order of insertion or even randomly, as traditionally done. We show that our methods are efficient and result in significantly improved query I/O performance leading to faster execution of neighborhood-related queries.

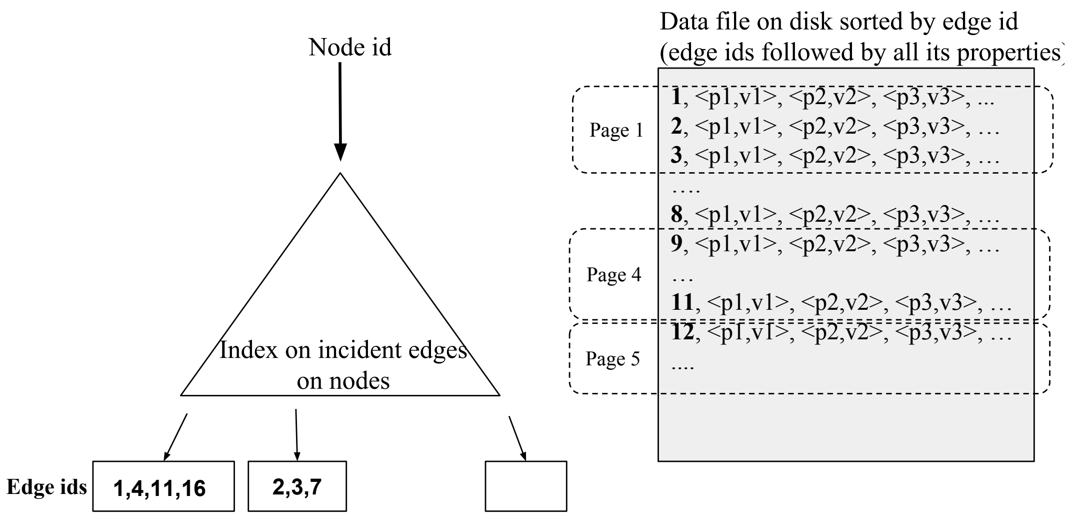
5.1 Introduction

By leveraging advancements in graph management tools, researchers have been able to gain more insights by asking new questions (queries) about their graph data. While these graph database systems enable users to effectively express their graph-based queries, users also expect to have their queries answered as efficiently as possible. Disk performance is one of the crucial factors affecting the throughput of many graph queries. A graph management system typically assigns internal identifiers (IDs) to vertices and edges at insertion time to allow their fast reference and indexing. In many systems, IDs are simply assigned based on the order of insertion, which is typically dependent on the data source: a web graph could be labeled in the lexicographic order of web pages, and a social network in the order in which users are crawled. A graph is generally represented as an adjacency list or matrix. Other systems such as Sparksee [131] and SNAP [114] also maintain a graph as list of edges, especially useful for indexing edges with rich attributes and managing multigraphs. Representing edges of a graph is therefore an important aspect and devising optimal representations has an impact on the performance of such systems.

Our motivation of this work stems from optimizing such systems, in particular for improving the efficiency of answering edge queries. For instance, given a node of a graph, a query could ask for its k-hop neighboring attributed edges (both incoming and outgoing) which possess a value of a particular edge attribute like the timestamp. In a friendship social network, such a query could be finding a person's friendship details (with both followers and followees) established at the date of 01/01/2017. Other examples of neighborhood queries are finding mutual ties in a co-authorship network and recommendations in a product network.

Figure 5.1 displays the retrieval of neighbourhood edge properties via edge indexes (left of the figure), while the underlying stored data file (right of the figure) is sorted by labeled edge IDs. The incident edges for a given node are indexed and contain pointers to the actual location of the edge records on disk. Each edge record consists of the assigned edge ID, along with a number of property value pairs that describes the edge. We argue that having consecutive edge IDs (over all node neighbourhoods) ensures edges are located in closer pages on disk, thus leading to better I/O performance for answering neighbourhood queries.

In this work, our goal is to optimally assign edge labels coupled with edge indices to achieve improved disk locality for efficiently answering these typical graph queries, *without modification* to the storage internals of the graph system at hand. While node labeling has been widely studied, one should not overlook the related edge-labeling problems aiming at improving query



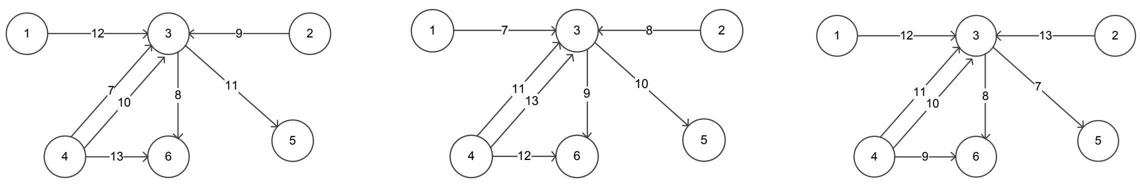
Node id
Index on incident edges on nodes
Edge ids
1,4,11,16
2,3,7
Data file on disk sorted by edge id (edge ids followed by all its properties)
Page 1
Page 4
Page 5
1, <p1,v1>, <p2,v2>, <p3,v3>, ...
2, <p1,v1>, <p2,v2>, <p3,v3>, …
3, <p1,v1>, <p2,v2>, <p3,v3>, …
….
8, <p1,v1>, <p2,v2>, <p3,v3>, …
9, <p1,v1>, <p2,v2>, <p3,v3>, …
…
11, <p1,v1>, <p2,v2>, <p3,v3>, …
12, <p1,v1>, <p2,v2>, <p3,v3>, …
….

Figure 5.1: An example of indexing attributed edges.

performance of some current and future graph analysis systems that store edge lists for different reasons. We focus on edge-labeling schemes for *directed* graphs, and demonstrate that even simple labeling of edges alone can significantly improve the performance of some typical workloads of graph applications, by lowering the number of disk reads. Retrieving neighborhood of a node is at the heart of many operations conducted on graph data. We posit that better disk locality of outgoing and incoming edges incident to a particular vertex would result in significant speedup of the neighborhood query, and consequently, in better execution times for a vast majority of queries which are based on it.

We approach this problem as labeling (*encoding*, *ordering* or *numbering*) of edges, where each edge in a set of edges E is given a number (an edge identifier, ID or eid) between 1 and $|E|$. The encoding is performed so that outgoing and incoming edges of nodes are given eids as consecutively as possible, thereby putting outgoing and incoming neighboring edges close together on disk based on these eids. Most existing graph databases (Neo4j, Sparksee, etc.) take a much more simplistic approach at dealing with the memory hierarchy than what we are used to from relational databases. Therefore it is important to look at the physical level of graph databases in terms of how to better manage graph data.

Figure 5.2 illustrates different edge-labeling strategies for a directed multigraph. Random ordering as shown in Figure 5.2a cannot guarantee consecutiveness of the eids assigned to edges. If edges are ordered by source nodes (a directed edge points from a source node to a target node), as in Figure 5.2b, all outgoing edges from a source node are guaranteed to be assigned



(a) Random order: $\texttt{C}_{\texttt{in}}(\texttt{G}) = 0.4$, $\texttt{C}_{\texttt{out}}(\texttt{G}) = 0.4$. **(b) Source-based order:** $\texttt{C}_{\texttt{in}}(\texttt{G}) = 0.4$, $\texttt{C}_{\texttt{out}}(\texttt{G}) = 1.0$. **(c) Perfect order:** $\texttt{C}_{\texttt{in}}(\texttt{G}) = 1.0$, $\texttt{C}_{\texttt{out}}(\texttt{G}) = 1.0$.

Figure 5.2: Illustration of different ordering strategies.

consecutive IDs. However an obvious drawback of this approach is that incoming edges of target nodes are overlooked, resulting in these edges possibly scattered across a range of IDs, undesirable for the underlying physical data storage. On the other hand, Figure 5.2c shows a perfect edge numbering such that both incoming and outgoing edges are given consecutive numbers. While for real graphs it is often impossible to perform such a perfect labeling, we can attempt to maximize the overall consecutiveness.

In this chapter, we formulate edge-labeling as an optimization problem, and present two scalable approaches, GrdRandom and FlipInOut, to label edges in a way that maximizes the total edge consecutiveness of graph, i.e., maximize the number of sequentially labeled edges to enable sequential storage, thereby increasing the locality of disk accesses. GrdRandom is based on the idea that numbering should be alternated between incoming and outgoing neighbors to strike a balance between the edge directions. FlipInOut extends this idea by taking into account the neighborhood information, and prioritizing high-degree nodes. Our contributions in this chapter can be summarized as follows:

- **Formulation:** We propose an edge consecutiveness metric on directed graphs (that takes into account both outgoing and incoming edges) and formulate edge-labeling as a maximization problem of this metric.
- **Methods:** We introduce GrdRandom and FlipInOut as two edge-labeling algorithms that focus on the balance between numbering outgoing and incoming edges.
- **Experiments:** We conduct extensive experiments on real graphs, and show significant benefits of our approaches over baselines in disk I/Os and query times.
- **Applications:** We demonstrate a case study of our methods to be applied in streaming graph partitioning.

We conduct experiments to evaluate disk I/O performance, and the subsequent speedup of various graph operations (e.g. friend-of-friend queries and shortest paths). Among the systems that index edges, we use Sparksee as a representative graph analysis system. Other systems, such as Unicorn [47] can also take advantage of edge-labeling. Unicorn has several types of edges (i.e., relationships among users, posts etc. in Facebook), and any index built on these edges based on edge properties can leverage a good labeling scheme to achieve disk locality and efficient edge indexing.

5.1.1 Chapter Organisation

In this chapter we begin by presenting existing research in Section 5.2 that is relevant to the edge labeling problem. In Section 5.3 we first formally define our edge labeling problem and introduce our proposed methods. We evaluate our approaches in Section 5.4 on real datasets presenting timing, scalability and disk I/O performance of different types of fundamental queries. We also make a connection between the consecutiveness measure we define theoretically and the experimental page accesses. Finally in Section 5.5 we investigate an application area that can benefit from an improved edge labeling, namely streaming graph partitioning.

5.2 Related Work

In this section we first discuss several categories of existing work that are closely related to the edge-labeling problem, and then review the implementation of Sparksee, which stores its edges as bitmaps on disk.

5.2.1 Node arrangement

The most relevant body of our work is node reordering – with different optimization objectives in mind. SlashBurn [119], for example, is a recent approach for renumbering the nodes so that the non-zero elements of the adjacency matrix are grouped together. The objective is to maximize the number of non-zeros (i.e. smaller number of denser blocks) within a matrix block in order to enable lower disk I/O, faster execution of matrix-based graph operations and better compression. SlashBurn investigates the 'no good cut' problem [119] for power law graphs and propose techniques for node reordering. The nodes are reordered such that matrix has smaller number of denser blocks facilitating lower disk I/O and better compression. Shingle Ordering [41] groups similar nodes to form dense communities by exploiting the link reciprocity of social networks. They approach the problem as a variation of the graph bandwidth problem [42]. It focuses on solving MLOGA and MLOGGAPA minimization problems, whereas we focus on solving a maximization problem.

Recently there have been several studies investigating graph ordering, focused on improving CPU cache performance. Wei et al. [207] exploits node ordering by finding an optimal permutation of nodes such that it minimizes the CPU cache miss ratio. Frequency based clustering and compressed sparse row segmenting approaches [227] have also been proposed to improve cache performance. The basic idea of the frequency based clustering is to prioritize popular nodes that are frequently accessed within the cache to reduce runtime overhead. Although node reordering or relabeling can result in improved performance in many node-oriented queries, it does not necessarily guarantee good performance for edge-oriented ones.

5.2.2 Graph compression and Space filling curves

Existing work exploits the property of locality in graphs with graph compression as a primary objective. Early approaches have focused on compressing web graphs with similarity and locality features [162], lexicographic localities [23]—later extended to social networks [41, 52]—, or a BFS approach [10]. Recent work on compression has also experimented with different ordering schemes [180] to improve locality. Graph summarisation approaches discussed in [121] are also closely related to graph compression. It must be noted that compression is not our main focus, although a benefit in compression may be a side-effect of our proposed encoding schemes.

Hilbert and other curves [136] are also closely related if we view the edge encoding problem as a mapping of edges to IDs. Hilbert curves generate a mapping between a 1-dimensional and a 2-dimensional space known to achieve good locality of reference. Different types of space filling curves are widely used to index spatial objects based on proximity. Intuitively, Hilbert curves recursively partition an (x,y) coordinate in a 2-dimensional space such that it can be mapped to a single integer. With a different goal in mind, a recent study has adopted a Hilbert ordering on graph edges as a way of improving the graph layout [134]. Their goal was to compare the reported performance of graph processing systems with a comparable single threaded implementation of the same datasets. Previous works have also used the Hilbert orderings to improve Sparse Matrix-Vector Multiplications (SpMV) [225, 224].

In our case, for every edge we can calculate a Hilbert index using the combination of the adjacent endpoints. We can then use this index to label the edges. The Hilbert index of an edge is sensitive to the neighboring node labels. In the original use of the space filling curve, the two endpoints refer to an actual (x, y) coordinate of a point in space. But in the graph space unless the x, y node IDs are 'close' (distance-wise in the graph space), we cannot guarantee that the edges will be assigned consecutive, or even close numbers.

5.2.3 Graph partitioning, Community detection and Clustering

The well-studied problem of graph partitioning is also pertinent to our work. The objective of partitioning algorithms is to reduce the number of edges crossing partitions, i.e., *edge cuts*, so that the nodes belonging to the same partition can be grouped together as a coherent unit of storage. A multitude of partitioning algorithms have been developed over the years in response to variations of the classic partitioning. METIS [96], one of the most widely used in practice, belongs to the category of multi-level partitioning strategies [99, 61]; many distributed algorithms [161, 205] work well with large graphs; DIDIC [64] and EvoCut [7] require only local computations eliminating expensive global operations on the graph. To avoid splitting high-degree vertices across multiple machines in a distributed setting, PowerGraph [67] proposes greedy approaches for placing edges in machines with balanced 'vertex-cuts'.

Community detection and clustering algorithms [34, 30, 143, 21, 151] have a very similar objective of grouping densely connected regions of a graph (e.g. cliques and bipartite cores) which are loosely connected with the rest of the graph.

All of these algorithms achieve some degree of locality within the graph by considering homogeneous regions in the network. One may be tempted to leverage a partitioning or clustering approach to derive an edge numbering. For example, a method such as METIS (known to have good edge-cuts) can be used to partition the graph, and then guide the arrangement of the edges on disk. For instance, the inter-edges can be assigned to the partition of the source or destination node, decided at random, while the intra-edges can be numbered in some arbitrary sequence. However, once a node numbering (or edge placement in the case of PowerGraph [67]) is known it is not straightforward to define a method that labels the edges in a way that their *consecutiveness* is maximized. Moreover, techniques like SlashBurn and METIS operate on *undirected* graphs, so only the existence of the edge is sufficient to obtain the final numbering. Naturally edge directionality is ignored when placing a node in a partition, cluster or community, while for us directionality is of utmost importance.

5.2.4 Sparksee

In Section 2.4.3.2 we introduced the Sparksee graph database management system. This systems primarily store edges for query processing thus becomes the testbed in the experiments. As detailed in Section 2.4.3.2, vertices, edges, attributes are stored internally as a combination of compact bitmaps enabling efficient bit operations for query processing. In this section, we reiterate and emphasize important concepts in Sparksee related to our edge-labeling problem.

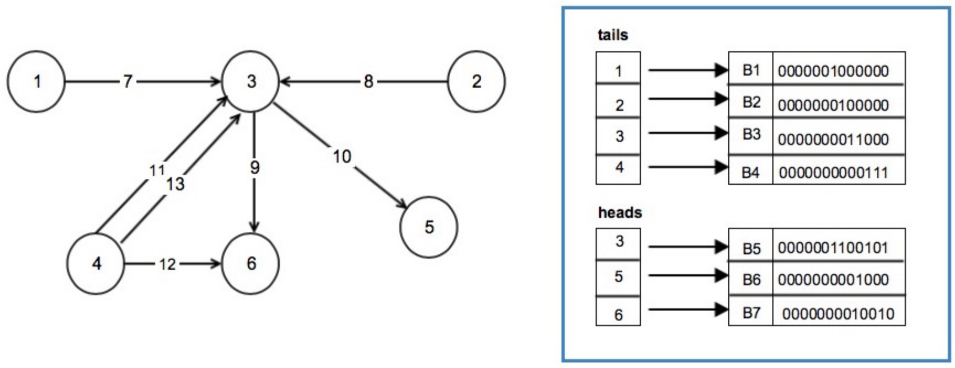


Figure 5.3: Bitmaps representing relationships for a graph with edges sorted by the source

Each vertex $v \in V$ and edge $e \in E$ is identified by a unique object identifier, *oid* $\in \mathbb{Z}^*$. As it is with many graph systems, its internal id generator assigns unique *oid*s when nodes and edges are created and in the order they are inserted. The assignments of *oid*s create the compactness of the variable-length sequences of 1s and 0s in bitmaps (see bitmaps `B1` to `B7` in Figure 5.3). Regards to the underlying storage, bitmaps are stored under a word aligned scheme where the bitmaps are split and aligned into 32-element chunks. Having as many consecutive 0s and 1s also makes bitmaps compression friendly.

Let us illustrate edge labeling order and its effect on bitmaps. Figure 5.3 shows the corresponding bitmaps (LSB on left) for storing relationships of a simple graph with edges sorted by the source. The `tail`/`head` group shows the IDs of all edges of which each node is a tail/head. Let us consider node ID 3 as an example: Node ID 3 is the tail for edges with IDs 9 and 10. Therefore, in the bitmap for 3 in the tails group (`B3`), the 9th and the 10th bits are set to one. Similarly, node ID 3 is the head for edges with IDs 7,8,11 and 13. Hence, its bitmap, `B5`, has the bits 7,8,11 and 13 set to one. Notice that a different edge-labeling will result in a different bitmap in the relationships. As illustrated in Figure 5.3, when the graph edges are sorted by source, 1s in the `tail` group will be grouped together however, the 1s in the `head` group are scattered across the bitmap requiring more pages to represent the bitmap on disk. Consequently, a query that involves retrieving the incoming neighbors of node ID 3 will incur more disk I/O operations. Maximizing consecutiveness of edge IDs is crucial to performance, since achieving consecutive 1s at a maximum will allow compression of bitmaps and therefore faster bit-level operations as well as less storage.

5.3 Edge-labeling schemes

Answering a query on a directed graph may either involve traversing through a node's outgoing edges, incoming edges, or a combination of both. This neighborhood query is the basis of most graph queries, if not all, used in practice. Consider a graph-based recommendation query on a who-follows-whom network on Twitter. Recommending users to follow to a user `u` may involve finding the 2-step followees (2-hop outgoing neighbors) who `u` is not following (1-hop outgoing neighbors). Moreover, in label propagation, at any round, both the incoming and outgoing edges of a node need to be accessed as messages are exchanged between neighbors. As such, our focus is primarily on improving the locality of the neighborhood query. Although an optimal arrangement of the edges is dependent on the characteristics of the graph and the type of query, retrieving the 1-hop neighborhood is fundamental to (almost) all graph operations.

Accessing neighboring edges of a node on disk requires reading pages from disk. If the neighboring edges are placed closer together on disk, this will reduce the costly random reads required for the fundamental neighborhood query. As such our objective is to place neighboring edges with improved locality independent of the type of graph. In Sparksee and other systems, this translates to assigning numbers to both outgoing and incoming edges as consecutively as

Table 5.1: Some graph related notations used in algorithms.

Symbol	Description
E'	reordered set of edges in a graph
V^R	random permutation of the nodes V
T	edge type: 'in' or 'out'
T'	an inverse edge type
L	set of unlabeled edges in a graph
eids	edge identifiers
$N_T(v)$	neighbors of vertex v of type T
$deg_T(v)$	degree of vertex v of edge type T
E_v	set of unlabeled edges incident to v
(v, x)	an outgoing edge of v, or incoming edge of x
visited $[T]$	set of visited vertices of type T

possible. Since there are many ways to number the edges in a graph an exhaustive search is not feasible; we propose a 'balanced labeling' technique that alternately numbers edges of opposite types (incoming and outgoing). As we confirm in the experiments, how well we can achieve consecutiveness depends on graph characteristics and the type of query workload. In the following, we first present the edge-labeling problem and then our proposed methods. We provide descriptions of some graph related notations in Table 5.1.

5.3.1 Problem formulation

Let $G(V, E)$ be a directed (multi-)graph with a set of vertices, V, and a set of edges, E. Internally in any graph system, each vertex $v \in V$ and edge $e \in E$ is identified/labeled by a unique integer ID. We consider G stored as edge lists (for reasons outlined in Section 5.1). First, we define the edge consecutiveness metric for any vertex as follows.

Definition 5 (Edge Consecutiveness)**.** Given a directed graph $G = (V, E)$, and a mapping $\pi : E = \{u, v\} \rightarrow \mathbb{Z}^*$ of edges to integer eids, the incoming edge consecutiveness (in-consecutiveness) of a vertex v, $C_{in}(v)$, is defined as the total number of pairs of its incoming edges with consecutive eids under the numbering π. Formally, let $N_{in}(v) = \left\{u_0, u_1, \ldots, u_{deg_{in}(v)-1}\right\}$ be the incoming neighbors list of a node v sorted in ascending numbering according to π such that $\forall i \in [1, deg_{in} - 1(v)] : \pi(u_i, v) > \pi(u_{i-1}, v)$, then we have:

$$C_{in}(v) = \begin{cases} \sum_{i=1}^{deg_{in}(v)-1} I\left(\pi(u_i, v) - \pi(u_{i-1}, v)\right) & \text{if } deg_{in}(v) > 1 \\ 1 & \text{if } deg_{in}(v) = 1 \\ 0 & \text{if } deg_{in}(v) = 0 \end{cases}$$

where I is an indicator function dictating the 'consecutiveness':

$$I(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & otherwise \end{cases}$$

The boundary/special cases of $C_{in}(v)$ above are when incoming degree of a node is 1 or 0. The outgoing edge consecutiveness (out-consecutiveness) $C_{out}(v)$ can be defined similarly.

Based on the above definition, for the whole graph G we naturally have two *total* in- and out- consecutiveness scores, respectively: $\sum_{v \in V} C_{in}(v)$ and $\sum_{v \in V} C_{out}(v)$. The following lemma upper bounds these consecutiveness values:

Lemma 1. *For a directed graph $G = (V, E)$, under the above consecutive labeling scheme, the maximum values of the total in- and out- consecutiveness scores are $|E|-n_{in}$ and $|E|-n_{out}$ respectively where n_{in} and n_{out} are the numbers of nodes with at least two incident edges of their respective edge types.*

Proof. In a directed graph, an edge can be both treated as incoming or outgoing depending on its incident end points. We first focus on the type of incoming edges and partition the graph accordingly as $E = \{E_0, E_1, ..., E_{|V|}\}$, i.e. the union of all nodes $\{v_0, v_1, ...\}$'s incoming edges. According to Definition 5, there are three cases of node v based on the value of its incoming degree $deg_{in}(v)$. Let's fix a node v_i,

- case 1) when $deg_{in}(v_i) > 1$: we have $C_{in}(v_i) \le |E_i|-1$;
- case 2) and 3) when $deg_{in}(v_i) = 1$ or 0: from Definition 5, $C_{in}(v_i) = |E_i|$.

Summing up these three cases of nodes over $v_i \in V$ and reflecting on the fact that $|E|= |E_0|+|E_1|+... + |E_{|V|}|$, we then have an upper bound of $|E|-n_{in}$ as only case 1) makes the difference between $C_{in}(v_i)$ and $|E_i|$. The upper bound holds similarly for the total out-consecutiveness. The lemma then follows.

With the previous consecutiveness definition, we then have the following consecutiveness optimization criterion for the whole graph edge labeling.

Definition 6 (Edge-Labeling)**.** Given a directed graph $G = (V, E)$, the goal is to find the best labeling $\pi^* = \arg\max_\pi C(G)$, i.e. the 'total' normalized in- and out-consecutiveness of the graph $C(G)$ as below, is maximized:

$$C(G) = \underbrace{\frac{1}{|E|-n_{in}} \sum_{v \in V} C_{in}(v)}_{C_{in}(G)} + \underbrace{\frac{1}{|E|-n_{out}} \sum_{v \in V} C_{out}(v)}_{C_{out}(G)} \quad (5.1)$$

where $C_{in}(v)$, $C_{out}(v)$, n_{in} and n_{out} were defined in Definition 5 and Lemma 1.

The scaling factors in the above consecutiveness formulation is to ensure that $C(G) \in [0, 2]$, since the maximum value of the in-consecutiveness (out-consecutiveness) of G is $|E|-n_{in}$ ($|E|-n_{out}$).

Example 5.1. **Consecutiveness of a simple graph.** Let us take the graph in Figure 5.4 (same as Figure 5.2b) as an example to show how consecutiveness values are calculated in Table 5.2. The $C_{in}(v)$ and $C_{out}(v)$ columns in the table display the edge consecutive values at each node $1 - 6$. For the whole graph, the normalized total consecutive values are therefore $C_{in}(G) = 0.4$ and $C_{out}(G) = 1$ and overall $C(G) = 1.4$.

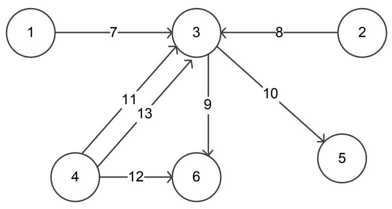


Figure 5.4: Graph with edges sorted by source nodes

Node	$C_{in}(v)$ (x=in)	$C_{out}(v)$ (x=out)
1	0	1
2	0	1
3	1	1
4	0	2
5	1	0
6	0	0
$\sum_{v \in V} C_x(v)$	2	5
$\|E\| - n_x$	$7 - 2 = 5$	$7 - 2 = 5$
$C_x(G)$	$2/5 = 0.4$	$5/5 = 1.0$

Table 5.2: Calculation example for in- and out-consecutiveness

Note that although we focus on labeling directed graphs our schemes can be readily extended to undirected networks as well. The theoretical maximum for perfect in-/out- consecutiveness is 1 which means either in-/out- edges of the graph are consecutive. For a given graph G, the out- (or in-) consecutiveness can be easily made 1 by labeling all the outgoing (or incoming) edges of each vertex i consecutively. However, it is unlikely that even in the optimal case, *both* incoming and outgoing edges can be made perfectly consecutive, because in directed graph labeling one type of edges also labels the edges of the inverse type. For example, consider labeling the outgoing edges of vertex `A` in sequence in order to improve $C_{out}(A)$. Labeling `A` $\rightarrow$ `B` and `A` $\rightarrow$ `D` with eids $e1$ and $e2$ would affect the incoming labeling of the nodes `B` and `D`, inheriting the numbers already assigned. An intuitive attempt to maximizing $C(G)$ is to locally maximize $C_{out}(v)$ and $C_{in}(v)$ for each node $v \in V$ by 'taking turns' in labeling edges of opposite directions. This is the core idea of our proposed algorithms to be presented next.

5.3.2 Labeling schemes

In this section we describe the baseline methods for labeling the edges of a graph, and then give the details of our proposed methods. Our solution seeks to maximize the consecutiveness at each individual vertex $C(v)$ so that local decisions greedily make progress towards the global optimal.

5.3.2.1 Baselines for labeling

There are three natural ways that the edges of a graph can be ordered in the input file, independent of the type of graph. We use the following methods to form our baselines:

- **Random**. The edges are listed in an order given by a random permutation.
- **consecIN**. The incoming edges of each node are labeled consecutively, edge IDs are sorted over the edge target/destination nodes.
- **consecOUT**. The outgoing edges of each node are labeled consecutively, edge IDs are sorted over the edge source nodes. In most of the datasets, this is also the natural ordering of the edges.

As explained in Section 5.2, existing node reordering methods are not directly applicable in solving our graph consecutiveness maximization problem with edge-labeling. However, correlations may exist between node and edge ordering methods for serving different types of graph queries. We leave such correlation studies for future work and instead focus on solving the edge-labeling problem here.

5.3.2.2 Proposed Method: GrdRandom

The two baseline methods consecIN and consecOUT are biased towards labeling the respective edge type consecutively. We propose an intuitive algorithm, GrdRandom, in which the labeling does not favor a single edge type. In this greedy approach we first consider a random permutation of the nodes to inform the visit order. For each of the nodes we flip a coin to decide if the outgoing or the incoming edges of that node should be labeled consecutively. Once an edge is given a number, it is not changed. The idea is that we alternate between the type of edge we number so that we do not favor a single edge type.

Algorithm 3 shows the general idea of the GrdRandom algorithm. The algorithm randomly numbers the edges and can complete fairly quickly. In the case that not all edges are labeled (due to randomness), a restart procedure can be performed for labeling unlabeled edges. The algorithm takes O($|V|$) time to run the random permutation, and performs the number assignment in O($|E|$). Therefore the complexity of the algorithm is $O(|V|+|E|)$.

5.3.2.3 Proposed Method: FlipInOut

Our first approach, GrdRandom, is simple and easy-to-implement, and, as we show in our experiments, it outperforms the baselines. In our proposed method, FlipInOut, we further

Algorithm 3 GrdRandom

```
Input: Graph $G = (V, E)$
Output: Re-labeled edge list $E'$
 1: $E' \leftarrow \{\}$                                   ▷ labeled edge list to return
 2: $L \leftarrow E$                                       ▷ list of unlabeled edges
 3: $V^R \leftarrow$ random_permutation($V$)
 4: for $v \in V^R$ do
 5:     if $rand() > 0.5$ then T ← 'out' else 'in' end if
 6:     /* $E_v$: unlabeled edges incident to $v$ */
 7:     if T == 'out' then                                ▷ unlabeled edges
 8:         $E_v \leftarrow \{(v, x) \in E\} \cap L$                ▷ outgoing
 9:     else
10:         $E_v \leftarrow \{(x, v) \in E\} \cap L$                ▷ incoming
11:     end if
12:     for $e \in E_v$ do
13:         $E' \leftarrow E' \cup \{e\}$                      ▷ edge added in order
14:         $L \leftarrow L \setminus \{e\}$
15:     end for
16:     exit if $|L| == 0$
17: end for
```

advance GrdRandom's main idea, and carefully incorporate more features to consciously improve the edge consecutiveness. We first give a simplified example of our proposed algorithm in Figure 5.5 and then discuss its main features in detail.

Example 5.2. **Illustrative Example.** As shown in Figure 5.5(a), the algorithm starts with the vertex of the highest total degree (node `B`) and numbers the edge type that has most unlabeled edges. For `B` we start numbering its incoming edges. From the in-neighbors of `B` (*candidate* vertices: `A`, `E` and `F`), it then picks the in-neighbor with the most unlabeled edges of the inverse (flipped) edge type (i.e., `edgeType` = `out`). For each node the algorithm keeps track of the number of unlabeled out- and in-neighbors. Node `E` is the selected vertex for the next iteration as neighbor `A` has no more out-edges, while `E` has 2 (`EF`, `EC`) and `F` has 1 (`FC`). `E`'s remaining outgoing edges are labeled in sequence (4(b)). Our method continues by considering the neighbors of E and selecting the vertex with the highest unlabeled edges of `edgeType` = `in` etc. (Figure 5.5(c)). If all neighboring nodes have their edges labeled, the algorithm restarts with the node having highest remaining degree. After numbering `C`'s incoming edges, there are no more neighbors of `C` that are unlabeled, hence the algorithm restarts with the only remaining node–`D`.

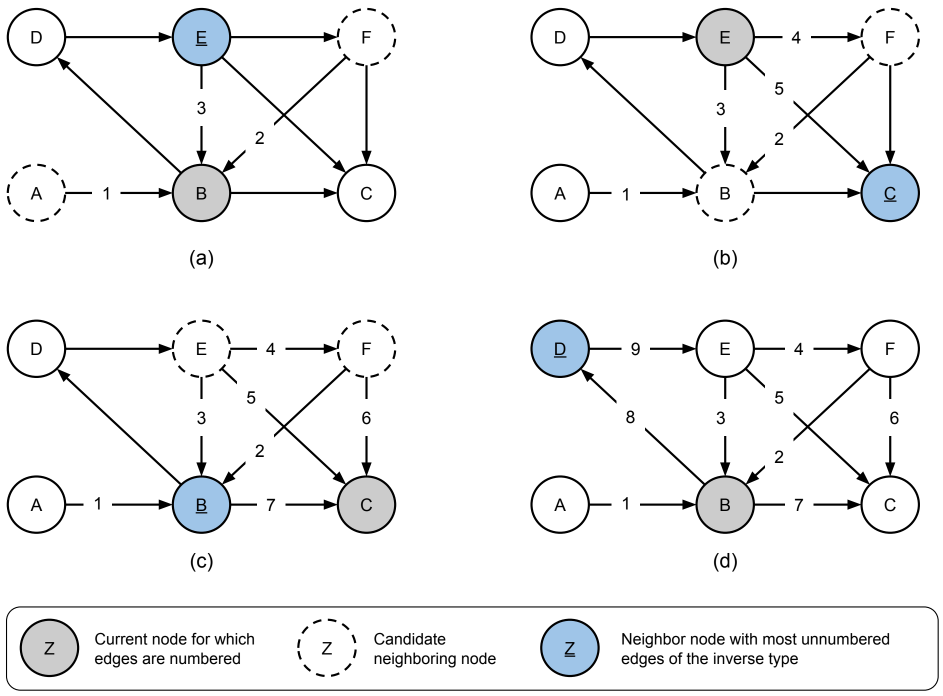

(a)
(b)
(c)
(d)
Current node for which edges are numbered
Candidate neighboring node
Neighbor node with most unnumbered edges of the inverse type

Figure 5.5: FLIPINOUT Algorithm: Example.

As more edges are labeled there are many node instances with all their neighbors labeled which results in the algorithm to restart often. Our proposed algorithm, FLIPINOUT shown in Algorithm 4, incorporates the following three main ideas with the goal of labeling edges in both directions as consecutively as possible:

I1. Alternate. Similar to GRDRANDOM, at each iteration, the algorithm flips between labeling outgoing and incoming edges (hence the name Flip-In-Out) to balance the consecutiveness. The node visit order of FLIPINOUT is based on the number of unlabeled edges of the flipped type, while GRDRANDOM is neighborhood agnostic. For example, if the incoming edges of a node was the last to be labeled, FLIPINOUT will examine the *outgoing* edges of the current node's neighbors, and vice versa. A swap procedure explained later (after I3) ensures the continuity in labeling.

I2. Prioritize. High-degree nodes are given high priority, and are labeled earlier in the algorithm. As in the example, when presented with a choice, FLIPINOUT numbers the edges of the highest-degree neighboring node. The intuition is that locality is especially important for high-degree nodes; If the edges of high in- and out-degree nodes are assigned consecutive

Algorithm 4 FLIPINOUT

```
Input: Graph $G = (V, E)$
Output: Re-labeled edge list $E'$
 1: $E' \leftarrow \{\}$                                    ▷ re-labeled edge list to return
 2: $L \leftarrow E$                                        ▷ list of unlabeled edges
 3: visited[out] $\leftarrow \{\}$ and visited[in] $\leftarrow \{\}$
 4: $[v, T]$ = CHOOSEVERTEX($L$, visited)                   ▷ starting vertex
 5: while $|L| \neq \emptyset$ do                           ▷ There are still unlabeled edges
 6:     /* $E_v$: unlabeled edges incident to $v$ */
 7:     if T == 'out' then
 8:         $E_v \leftarrow \{(v, x) \in E\} \cap L$                ▷ outgoing
 9:     else
10:         $E_v \leftarrow \{(x, v) \in E\} \cap L$                ▷ incoming
11:     end if
12:     for $(x, y)$ in $E_v$ do
13:         $L \leftarrow L \setminus \{(x, y)\}$
14:         $E' \leftarrow E' \cup \{(x, y)\}$                      ▷ edge added in order
15:         visited[T] $\leftarrow$ visited[T] $\cup\{v\}$
16:         $deg_T(x)$−=1; $deg_T(y)$−=1                            ▷ current degree
17:     end for
18:     T $\leftarrow$ T$'$                                 ▷ flip the edge type ('in' or 'out')
19:     /* $N_T(v)$: neighbors of node $v$ of type $T$*/
20:     if $|\{(x, y) \in E | x \in N_T(v)\} \cap L| \neq \emptyset$ then
21:         /* Find the next vertex to visit, from $v$'s neighbors */
22:         $v = \text{argmax}_{x \in N_T(v) \setminus \text{visited[T]}} \{deg_T(x)\}$
23:     else
24:         $[v, T]$ = CHOOSEVERTEX($L$, visited)
25:     end if
26: end while
27: /* Choose the new starting vertex */
28: function CHOOSEVERTEX($L$, visited)
29:     $v = \text{argmax}_{v_i \in V \setminus \{\text{visited[T]} \cap \text{visited[T']}\}} \{deg_T(v_i)\}$
30:     if $deg_{out}(v) > deg_{in}(v)$ then T $\leftarrow$ 'out' else 'in'
31: end function
```

numbers, a larger proportion of edges will be consecutive. As a result, it is more important that edges incident to "popular" nodes are closer together on disk, compared to a node with only a couple of incident edges. Any query that involves accessing the neighborhood at a depth greater than one (e.g. shortest paths) is more likely to reach a high-degree node due to its large number of connections. If the neighbors of these high-degree nodes are not close on the disk, a query can quickly become very inefficient. We therefore seek to minimize the overall I/O cost for accessing the graph by minimizing the I/O activity of the high-degree nodes.

I3. Terminate Early. This idea is applicable and particularly important for large graphs, where it is common to have frequent vertex restarts after a significant portion of the edges are labeled. For perspective, the frequent restarts problem leads to 95% slower runtime for Flickr dataset compared to FLIPINOUT with early termination (1988 vs. 88 seconds). The idea behind early termination is to differently order the last $\delta\%$ of the edges. Specifically, we employ a neighborhood-agnostic approach, which decides the visit order of the vertices with unlabeled incident edges and terminates the 'flipping edge' idea. Each node v is represented as a set of at most two pairs: (i) $(v, deg_{in}(v))$, if it has unlabeled incoming edges; and (ii) $(v, deg_{out}(v))$, if it has unlabeled outgoing edges. The resulting pairs are ordered in decreasing order of degree (in or out) to inform the order in which the vertices will be visited. Their incident edges of the corresponding type are then labeled consecutively. In our experiments, $\delta = 12\%$ achieved good performance in the largest graphs that we used and eliminated the frequent restart problem. The percentage δ is set once for all the remaining edges. For brevity, we have excluded the early termination process from Algorithm 4 (the criterion on line 5 would change to $|L| > \delta|E|$).

The **alternate** and **prioritize** steps are employed to locally maximize the individual consecutiveness $C(v)$ (Equation 5.1) for each vertex v. When this greedy approach terminates, the algorithm is making progress towards an optimal solution. As we mentioned in the **alternate** step, we employ a **swap procedure** to ensure continuity. As described in Example 5.2, at any given step, out of the candidate neighboring nodes, the next vertex to visit is selected based on the number of unlabeled edges a vertex has of the flipped edge type. When the vertex is chosen, the selected vertex inherits at least one edge from the current node. Before labeling the edges of the selected vertex a condition is tested: If the edge connecting the current vertex and the selected vertex (i.e., the common edge) does not have the highest edge number seen so far, the edge number is swapped with the maximum sibling edge ID. This ensures the continuity in numbers assigned to the inherited edge and the edges of the selected vertex that are about to be labeled. This is another strategy to ensure that the local consecutiveness of a given node is at the maximum.

EXAMPLE 5.3. **Swap procedure example.** As illustrated in Figure 5.6, outgoing edges of `A` are being numbered consecutively (edge IDs 11–98). Assume that in the next iteration, node u_1 will be selected, since it has the highest number of unlabeled incoming edges thus

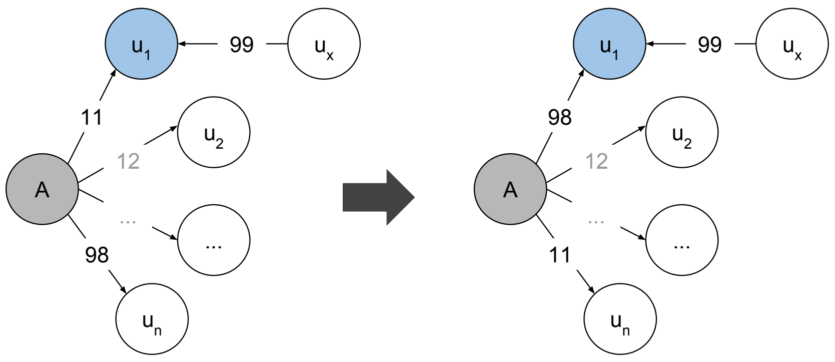


Figure 5.6: Swapping procedure in FlipInOut algorithm

the numbering will start at edge ID 99. Without a swap procedure, the labeling is shown on the left of Figure 5.6 resulting in a gap of 88 between two incoming edges. A simple $O(1)$ check can swap the edge IDs such that all the outgoing and incoming edges will be numbered consecutively for `A` and u_1 respectively.

Runtime Complexity. Line 4 in Algorithm 4 spends time $O(|V|)$. Per node v, we execute lines 12-17 in $O(deg(v))$, and either line 22 or 24, which are $O(deg(v))$ and $O(|V|)$, respectively. So, FlipInOut is $O(|V|+|E|+max\{|E|, |V|^2\})$. Its worst case complexity, $O(|V|^2)$, occurs only when it keeps restarting (line 24—i.e., the graph consists of disconnected stars). In practice, restarts happen only towards the end of the algorithm, and FlipInOut is very efficient needing only 2.8 and 4.9 minutes to label 33M and 69M edges, respectively.

5.4 Experimental Evaluation

We conduct experiments to demonstrate the performance of our encoding methods on a variety of real graphs. In the following subsections, we answer the questions:

- Can we speed up popular graph queries using FlipInOut edge-labeling and how does it compare to the baselines?
- Can we observe improved disk I/O performance as a result of better locality when storing the graph on disk?
- Do GrdRandom and FlipInOut show benefit in storage compared to a random ordering and other baseline schemes?

Before we answer these questions, we describe the experimental setup for our analysis.

5.4.1 Experimental Setup

Environment. The experiments were conducted on a Linux machine with 4.00GHz Intel Core i7-4790K, 8GB of memory and 60GB SSD.

Setup. Our experiments use Sparksee 5.2 (cf. Section 5.2) for the creation of databases. Given a dataset, node labeling is identical for all labeling methods, but the edge-labeling differs. For every graph, a Sparksee database is created for each of the labeling methods including the baselines—Random, consecIN, consecOUT, GrdRandom and FlipInOut. In order to run queries on edge properties, we have augmented each of the datasets by adding randomly generated integer attributes on all the edges, representing weight or timestamp property.

Datasets. We conduct experiments on six directed real-world graphs with edges ranging from 100,000 to 69 million with varying characteristics, which we obtained from SNAP[1] and KONECT[2]. In Table 5.3 we summarize basic statistics of each graph considered. For each dataset, we give the number of nodes and edges, the number of edges in the largest strong connected component (LCC), the average clustering coefficient (ACC), and a short description of the graph representation.

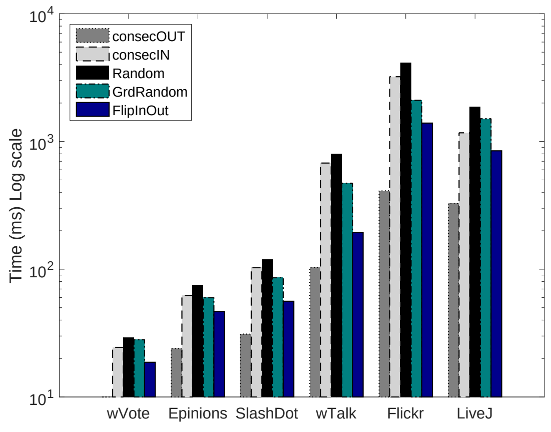
Table 5.3: Summary of Datasets: Number of nodes and edges, the number of edges in the largest strong connected component (LCC), the average clustering coefficient (ACC), and a graph description.

Dataset	Nodes	Edges	LCC	ACC	Description
WikiVote	8,297	103,690	0.38	0.14	who-votes-whom
Epinions	75,879	508,837	0.87	0.14	who-trusts-whom
Slashdot	82,168	948,464	0.96	0.06	social network
WikiTalk	2,394,385	5,021,410	0.29	0.05	Wiki talk network
Flickr	2,585,568	33,140,018	0.82	0.10	social network
LiveJ	4,847,571	68,993,773	0.95	0.27	social network

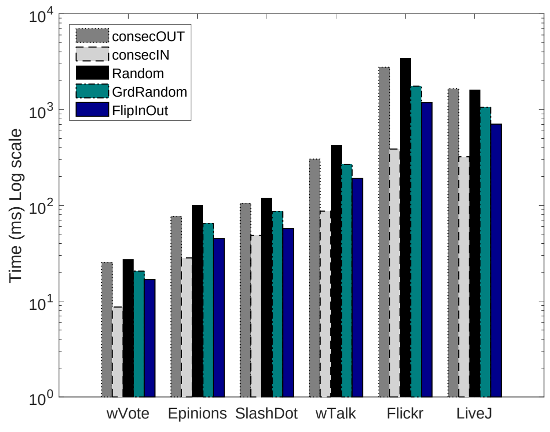
5.4.2 Speedup of Queries

We perform experiments to demonstrate the speedup of some popular graph queries: (i) friend-of-friend queries, which explore a node's 2-hop neighborhood (ii) shortest path queries, (iii) queries that retrieve edge properties, and (iv) queries that retrieve the entire neighborhood at a given depth. The performance for all queries is shown in Figure 5.7. Each plot shows the average execution time (y-axis) of running the query for 100 instances (node IDs). For a

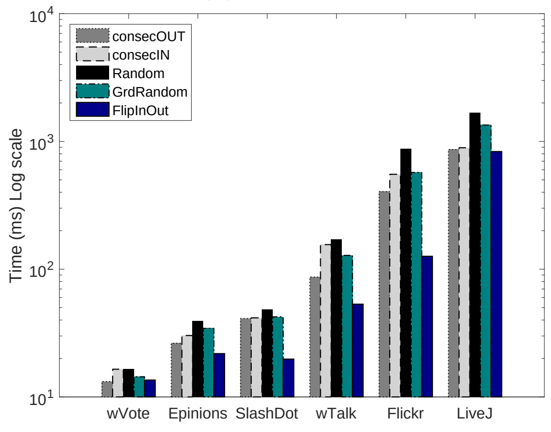
[1]https://snap.stanford.edu/data/index.html
[2]http://konect.uni-koblenz.de



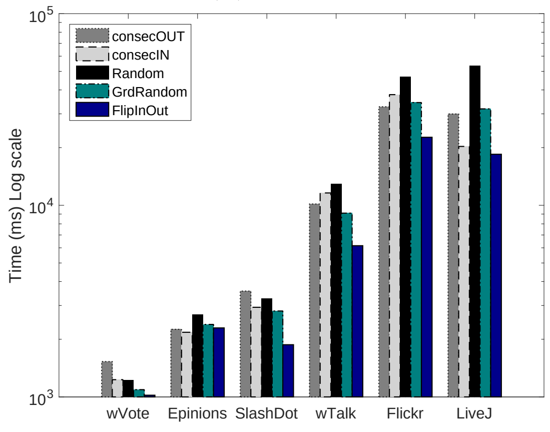
(a) **FoF-out**



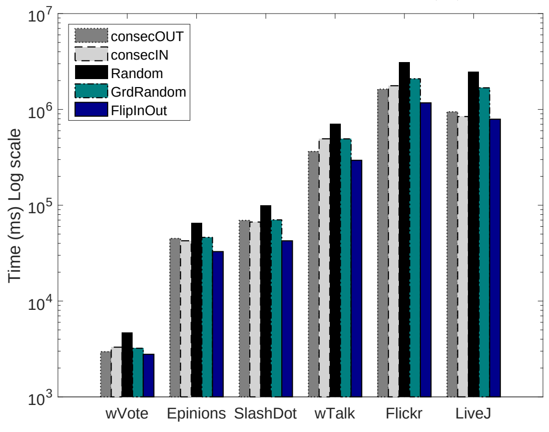
(b) **FoF-in**



(c) **Shortest Path**



(d) **Edge-property**



(e) **Neighborhood at Depth-2**

Figure 5.7: Query Performance (in ms) in real networks: FLIPINOUT and GRDRANDOM have the best combined performance for in- and out-specific queries. The runtime is measured as the average execution time over 100 runs.

dataset, the same instances (node IDs for FoF, property queries and ID pairs for shortest paths) are used for all queries and labeling methods.

5.4.2.1 Friend-of-Friend (FoF) Queries

We perform two types of friend-of-friend queries, which explore neighborhoods at depth 2, to inspect both directions: FoF-in and FoF-out. These two uni-directional queries are chosen to show the behaviour of different schemes when the query is biased. For high-degree nodes these queries involve having to traverse through a large neighborhood. The FoF query performance for different labeling schemes is shown in Figure 5.7a-b.

We observe that for a FoF-out query (Figure 5.7a), consecOUT numbering has the lowest execution time reported across all graphs. This is expected as consecOUT is the optimal arrangement of edges for an FoF-out query with all outgoing edges having consecutive numbers ($C_{out}(G) = 1.0$). However, the same query performed on a database with consecIN edge encoding ($C_{in}(G) \approx 0$), has performance close to that of a random ordering. Similarly, a consecIN numbering performs best for an FoF-in query (Figure 5.7b), but performs similar to a random ordering when running an FoF-out query. To put this in context, for the Flickr graph, while consecOUT is $8\times$ faster compared to consecIN for FoF-out queries, it becomes $7\times$ slower for FoF-in queries. Thus, consecIN and consecOUT are biased towards a single direction (in/out, resp.) and are only suitable for the query type on which the database is built.

If the query workloads are known apriori to be only of one type, certainly these approaches work very well. But in practice this is a strong assumption and rarely the case. Therefore, we need methods to strike a balance between achieving locality of both incoming and outgoing edges. Across a variety of query workloads, our approaches meet halfway between the biased labeling and stay consistent with the best-performing methods irrespective of query type.

Observation 1. FlipInOut is closer to the best performing consecOUT for FoF-out query (Figure 5.7a) and also closer to the best performing consecIN for FoF-in queries (Figure 5.7b). For FlipInOut, the average relative performance improvement for FoF-out (FoF-in) queries ranges from 36% to 76% (38% to 66%, resp.) compared to random numbering.

Although GrdRandom does not perform as well as FlipInOut, its execution is still consistent across different query types. It outperforms the random and consecIN schemes for out-specific queries, and the random and consecOUT schemes for in-specific queries. Recall that GrdRandom was a fairly straightforward and easy-to-implement approach which gives acceptable results. In Section 5.4.4 we confirm that the main reason behind the better timing in our methods is improved disk I/O operations.

5.4.2.2 Shortest Path Queries

We chose shortest path queries to represent the category of queries that employ both outgoing and incoming edges. The shortest path from the source vertex s to the target vertex t involves a bidirectional breadth-first strategy, which leads to significant speedup in the algorithm by reducing the number of visited vertices. The idea is to perform a forward search from s via its outgoing edges, and a backward search from t via its incoming edges until a common node is processed. Thus, the query requires going through the outgoing and incoming edges of a graph simultaneously. The shortest path query performance of all labeling methods is shown in Figure 5.7c.

Observation 2. For shortest path queries, FlipInOut outperforms all the encoding methods, which are biased towards edges in one direction.

Thus, if a query needs to retrieve both outgoing and incoming neighbors (e.g. the optimized shortest path query), a balanced numbering clearly results in better query performance. Overall, the average performance improvement for FlipInOut ranges from 18% to 86% compared to random numbering, and shows up to 7× speedup (Flickr). Upon closer inspection, we note that for WikiVote and LiveJornal datasets, the timing difference between FlipInOut and the fastest baseline is marginal. For WikiVote, this can be attributed to the small path lengths between node pairs (most lengths are 1-2, and 19% of them are 0—non reachable node pairs). In Table 5.3 we see that only a small fraction of edges–0.38 belong to the LCC thus some nodes within the graph were not reachable.

For LiveJournal, we attribute the marginal difference to its high average clustering coefficient (0.27) compared to other graphs. The clustering coefficient of a vertex indicates how well-connected the neighborhood of that vertex is, and is defined as the ratio of actual edges between neighbors over the maximum number of potential edges. If its neighbors are well-connected, it is likely that the nodes required to perform an FoF-out, FoF-in, or a shortest path query are already available in memory, and thus leading to low execution time.

5.4.2.3 Edge-Property Queries

For the next set of experiments we select a pattern matching query on edges. Any query that filters the incident edges of a node based on a given property value is an example. We use a query that filters the incident neighborhood (both edge directions) of a given node v based on a edge property value x. The value of x is selected such that the selectivity is around 5%-10%. The query involves retrieving both the outgoing and incoming incident edges of v. The edge-property query performance for the different encoding schemes is shown in Figure 5.7d.

We observe that FlipInOut numbering results in lowest average execution time across all graphs. Similar to shortest path queries, since FlipInOut attempts to balance the numbering it results in better performance when dealing with queries that involve incident neighborhood irrespective of direction. Overall, the average performance improvement for FlipInOut ranges from 10% to 78% compared to a random numbering.

Observation 3. For edge property queries, FlipInOut outperforms all the encoding methods with an average relative performance improvement ranging from 10% to 78% compared to a random numbering.

The behavior of consecOUT and consecIN varies depending on the query mix – if nodes along query paths of a certain depth have more (less) outgoing neighbors than incoming, consecOUT (consecIN) would perform better.

5.4.2.4 Neighborhood Queries

The friend-of-friend queries that we investigated are uni-directional and we showed that the respective consecutive labeling schemes work well when the query only explores a single direction. Next we evaluate the behaviour of a query that explores the full neighborhood of a give node. A neighborhood query is a fundamental operation in algorithms such as community detection, where the direction of the neighborhood (either incoming or outgoing neighbors) is dispensable. We consider the full-neighborhood at depth-2 and the query performance for different labeling schemes are shown in Figure 5.7e.

We observe that for a full neighborhood query, FlipInOut has the lowest execution time followed by either consecIN or consecOUT labeling methods. Relative performance improvement of FlipInOut ranges from 6% to 36%, compared to its next best labeling scheme. Table 5.4 helps explain which method comes closer to FlipInOut. This table shows, as an average, the fraction of outgoing edges to total edges for each of the datasets. As an example, the value 0.57

Table 5.4: Percentage of average outgoing edges at depth-1 and depth-2.

	avg. out at depth-1	avg. out at depth-2
WikiVote	0.55	0.58
Epinion	0.45	0.50
SlashDot	0.49	0.47
WikiTalk	0.82	0.81
Flickr	0.57	0.54
LiveJournal	0.36	0.37

for Flickr means that for a node, on average, 57% of edges are outgoing and, thus, 43% edges are incoming. WikiTalk and LiveJournal are heavy on outgoing and incoming edges respectively. As a result, in Figure 5.7e, for WikiTalk, consecOUT run time is closer to FlipInOut than consecIN – the queries are heavy on outgoing edges, so consecOUT contributes more for the full-neighborhood than consecIN. In Section 5.4.5 we further explain this observation with analytical consecutiveness values for each of the methods.

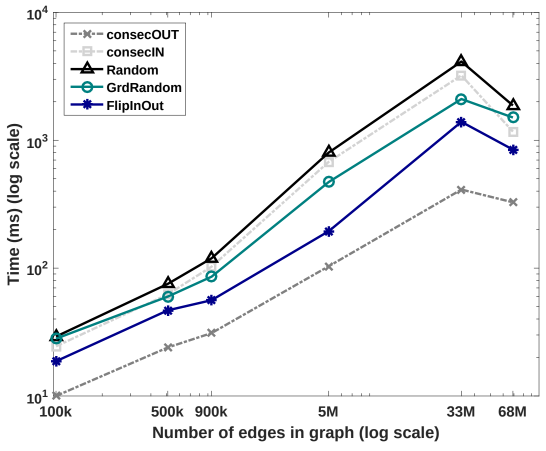
Observation 4. For a full neighborhood query, FlipInOut has the lowest execution time followed by either consecIN or consecOUT labeling methods. Relative performance improvement of FlipInOut ranges from 6% to 36%, compared to its next best labeling schemes.

Recall that GrdRandom on the other hand, follows a random ordering of the nodes to labels. It seems that although it takes turns numbering both directions, for the full neighborhood in most datasets, GrdRandom cannot beat the consecOUT and consecIN methods.

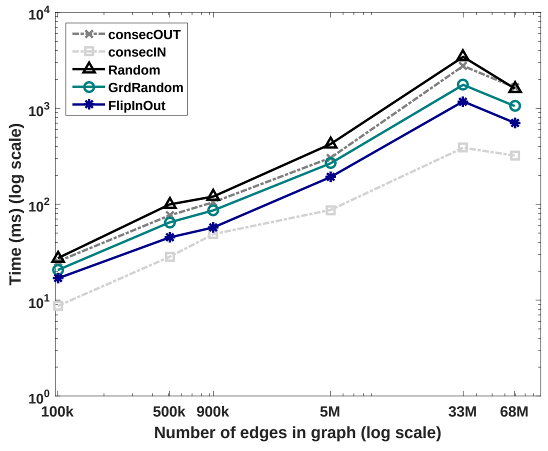
5.4.3 Scalability

Figure 5.8 presents the average execution time of the encoding schemes as a function of the size of graph edges. The x-axis in the plot corresponds to the number of edges in each of the datasets (Table 5.3) in increasing order and the y-axis to the average time in log scale. For FoF-out and FoF-in, the dark blue star line representing FlipInOut remains consistent across the graphs regardless of the query direction. For shortest path, edge property and neighborhood queries, we observe FlipInOut to have lowest execution times, scaling well with graph sizes.

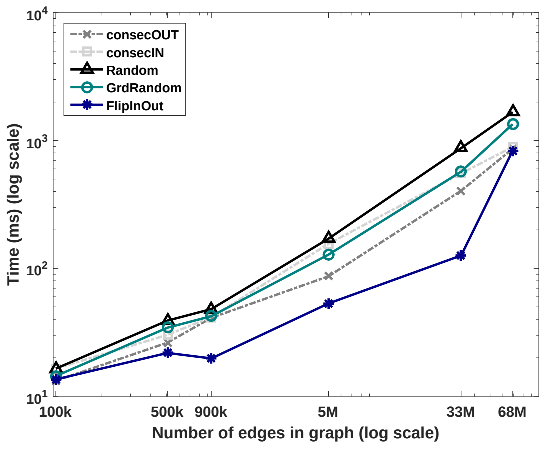
Observation 5. FlipInOut scales well with the size of the input graph, and its relative improvement is robust to the graph size regardless of the edge direction in the queries.



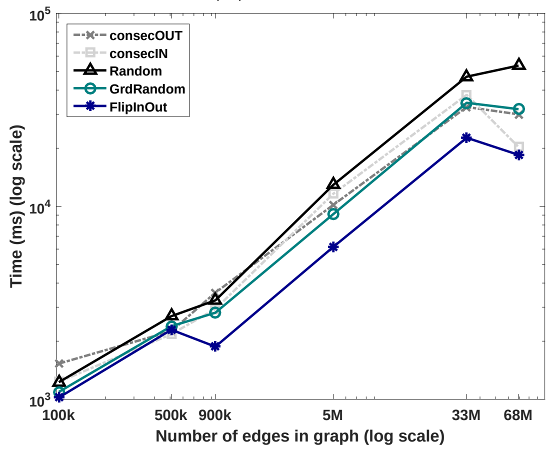
(a) FoF-out



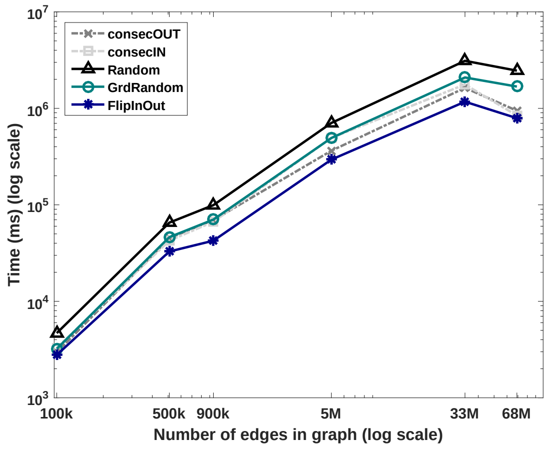
(b) FoF-in



(c) Shortest Path



(d) Edge-property



(e) Neighborhood at Depth-2

Figure 5.8: Query Performance (in ms) vs. number of edges in each input graph (log-log scale)

As explained before, the drop in timing for the LiveJournal graph is likely related to the high connectivity between the neighbors that are already loaded into memory.

5.4.4 Disk I/O Performance

To measure locality preservation on disk and confirm our hypothesis that better edge-labeling improves the number of disk accesses, we monitored the disk I/O performance of each query.

In Figure 5.9 (for smaller datasets) and Figure 5.10 (for larger datasets), for each type of query and encoding scheme, we show the total number of persistent page reads from disk (y-axis) over 100 instances of node IDs. The disk I/O for FoF-out, FoF-in, shortest path, edge property and neighborhood queries are shown in each row. In Figure 5.9 the query results for WikiVote, Epinion and SlashDot are shown in columns a, b, c respectively and the same columns in Figure 5.10 shows results for WikiTalk, Flickr and LiveJournal datasets. The reported number of pages read refers to all the internal structures, including the indexes and actual data stored in bitmaps. All the statistics are recorded the first time a query is run, on a cold cache.

These figures shows how the page accesses vary across different labeling schemes and their relative differences. For FoF-out and FoF-in queries, a consecOUT and consecIN layout clearly show a benefit consistently across all the graphs. FLIPINOUT is closer to the winner in each of the respective methods. Both the shortest path and edge property queries seem to benefit from having a more non-biased edge layout on disk. Peculiar behaviour in the LiveJournal graph is also exhibited in these plots. Variations on these plots help explain the behaviour of query times in the previous section.

OBSERVATION 6. The number of persistent page reads correlates with the query time of the encoding schemes.

For WikiVote, there appears to be a big difference in the number of reads between the methods (which should affect its runtime), but the disk page reads are consistent across methods, ranging from 15 to just 60. The small number of reads is likely due to the network's small size—its 100K edges can be cached. Furthermore, with the promising results shown with improved disk I/O, we believe that the proposed labeling schemes will also be useful in a distributed setting where the graph is partitioned across machines. Having locality in neighboring edges would mean less communication overhead across the network.

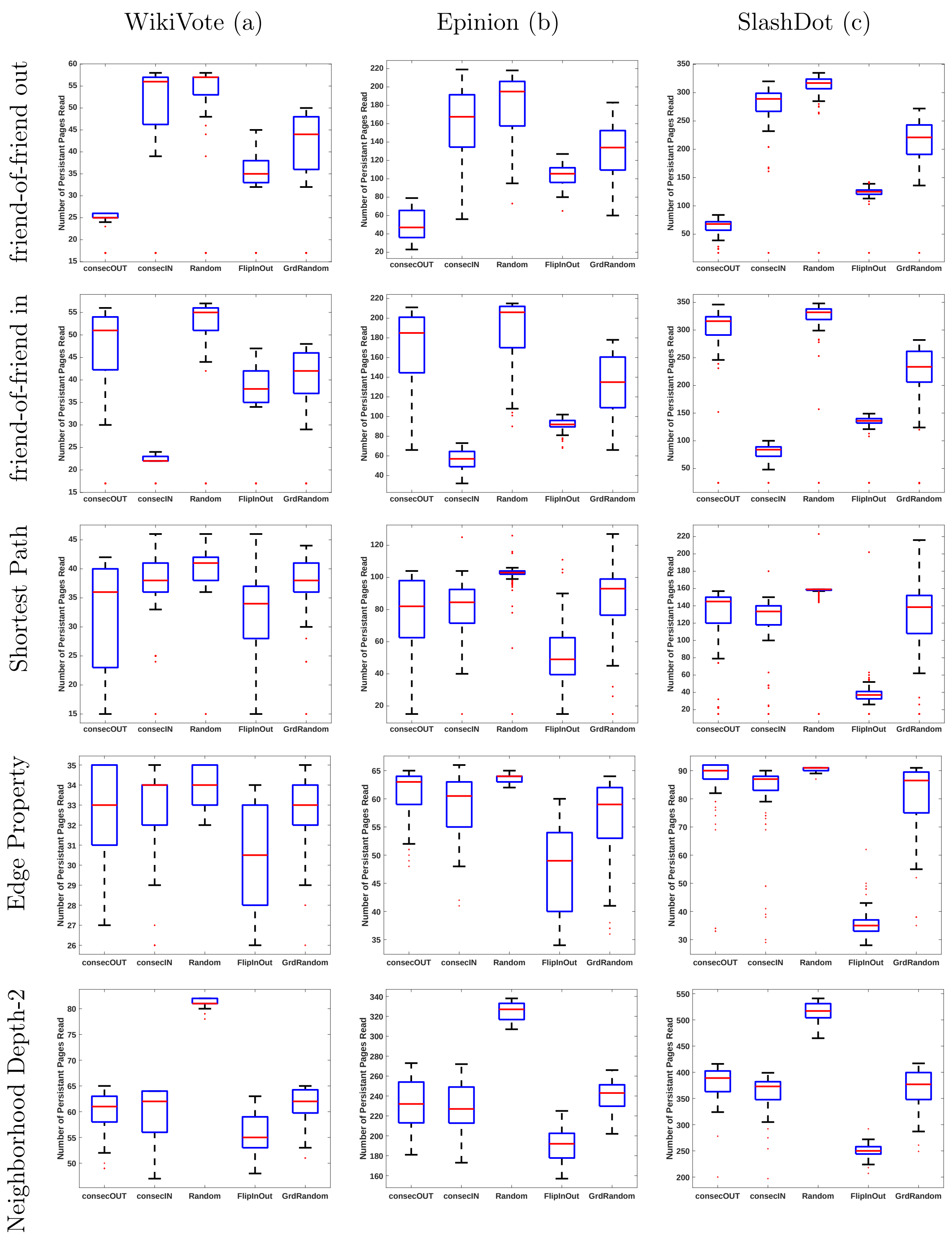


Figure 5.9: Disk I/O Performance of Queries Smaller Datasets

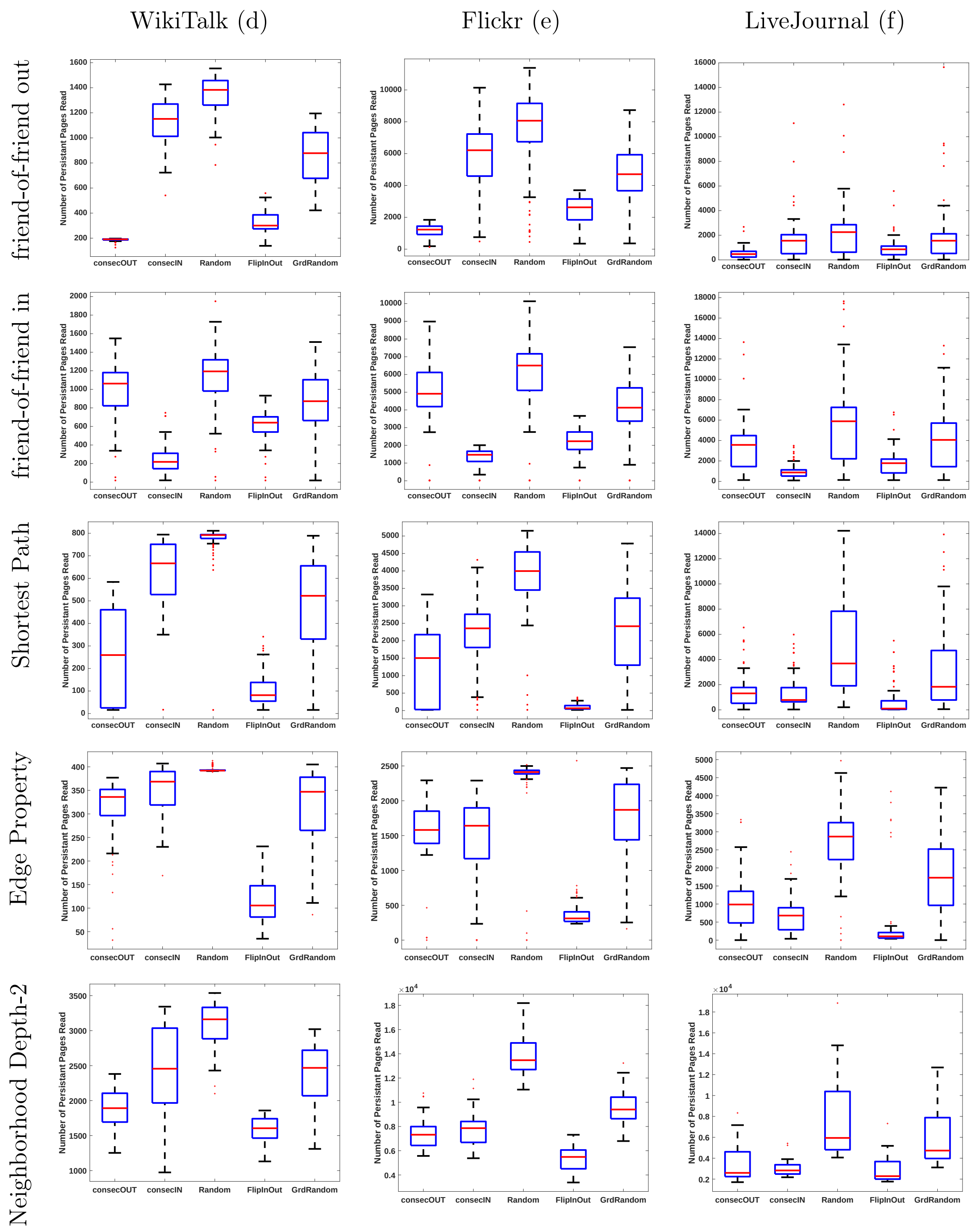
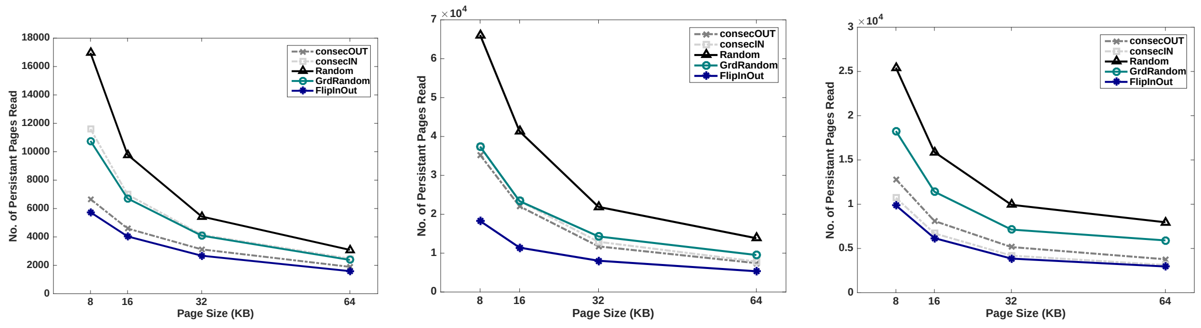


Figure 5.10: Disk I/O Performance of Queries Larger Datasets

5.4.4.1 Varying Page Sizes

To better understand the disk I/O performance, we evaluate it with varying page sizes. Databases are created with varied physical page sizes 8, 16, 32, 64 (KB). For the experiments, full-neighborhood query is run on the three larger datasets – WikiTalk, Flickr and LiveJournal and the results are shown in Figure 5.11.



(a) WikiTalk **(b) Flickr** **(c) LiveJournal**

Figure 5.11: Varying page size for the neighborhood query

As the page size is increased from 8 to 64, the number of disk page accesses decrease. With larger page sizes, more information can be fit per page and, thus, there are fewer page reads. This behaviour is observed across all the datasets, but, the relative performances of the datasets are mixed.

Observation 7. With increasing page sizes, we observe lower page reads. Although FlipInOut has lowest page accesses across the datasets, consecOUT and consecIN becomes close contenders for WikiTalk and LiveJournal respectively.

Recall that the neighborhood query involves accessing both incoming and outgoing neighbors at 2-hops. Although FlipInOut has the lowest number of page accesses across the datasets, consecOUT and consecIN become close contenders for WikiTalk and LiveJournal, respectively. As discussed in Section 5.4.2.4, since the WikiTalk graph (LiveJournal) is heavy on outgoing edges (incoming edges), the neighborhood query explores many outgoing (incoming) edges and, thus, the benefit of FlipInOut becomes closer to consecOUT (consecIN) methods.

On these larger datasets, with smaller page sizes (8KB, 16KB) we note that the number of page reads (thus timing) is significantly higher, compared to the number of page accesses (and timing) for large page sizes.

5.4.4.2 Disk Storage Benefit

We also compare the schemes with respect to the raw sizes of the databases they created with the different encoding methods. As shown in Table 5.5, FlipInOut and GrdRandom achieve comparable or better storage benefit than consecIN and consecOUT, ranging between 10%–27% reduction compared to a database with random ordering.

Observation 8. FlipInOut and GrdRandom achieve comparable or better storage benefit than consecIN and consecOUT, ranging between 10%–27% reduction compared to a database with random ordering.

The reduction in size compared to a graph with consecOUT or consecIN is marginal (at most 8%). The reason is that when the edges are sorted by source (in consecOUT), the bitmaps in the tails group (Figure 5.3) is optimal, but the disarray in the heads cancels out its storage benefit. The low compression benefit of consecIN for WikiTalk is due to its high in-ratio (Table 5.3), which means that many nodes have only incoming edges. In Sparksee (Section 5.2), this translates to sparse `head` group bitmaps, and numbering the edges of such a graph with consecIN is almost equivalent to a random numbering.

Table 5.5: Storage Benefit (%) compared to the Random encoding scheme. Higher is better.

Method	WikiVote	Epinions	SlashDot	WikiTalk	Flickr	LiveJ
FlipInOut	19.6	19.5	23.2	15.4	26.7	26.5
GrdRandom	18.5	18.7	21.0	10.4	25.2	24.4
consecIN	19.6	16.6	18.9	1.9	24.5	24.0
consecOUT	18.5	16.9	18.5	15.0	24.6	24.1

In conclusion, our methods generally have better and balanced runtime and disk I/O performance for a wide range of queries, and also have the side-benefit of better or comparable storage benefit to the baseline encoding schemes.

5.4.5 Analytical Cost of Varying Depth Neighborhood Queries

In this section, we study the connection between an analytical cost model built on the consecutiveness definition (Definition 5 and Equation 5.1) and the experimental page access numbers. Our goal is to observe the behaviour of the page accesses we find experimentally, when the edges encountered by a query exhibit certain consecutiveness metrics. We consider the full neighborhood query, representing a query type that explores both incoming and outgoing neighbors of a node, which is fundamental in many types of analyses. As we have seen in Section 5.4.2.4, a

d-depth neighborhood query encounters a certain number of nodes in both directions as part of the query execution. We can calculate an edge consecutiveness measure observed during one query run. Having consecutive edges facilitates storing these pages sequentially. Thus, consecutiveness becomes a dominant measure in determining the number of page accesses. The cost relative to a given query node q_x, is given by the relative cost, $RC(q_x)$:

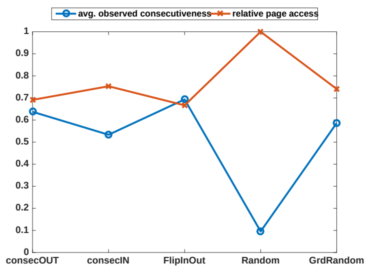
$$RC(q_x) = \frac{1}{2d} \sum_{i=1}^{d} \left(\frac{\sum_{v_j \in V_{in}^{i-1}} \frac{C_{in}(v_j)}{|E_{in}(v_j)-1|}}{|V_{in}^{i-1}|} + \frac{\sum_{v_k \in V_{out}^{i-1}} \frac{C_{out}(v_k)}{|E_{out}(v_k)-1|}}{|V_{out}^{i-1}|} \right) \tag{5.2}$$

where attached to a query q_x, V_{in}^i and V_{out}^i are the encountered incoming and outgoing nodes sets respectively at depth i ($V_{in}^0 = V_{out}^0 = q_x$); out of these encountered nodes, $E_{in}(v_j)$ and $E_{out}(v_k)$ denote their incident respective incoming and outgoing edge sets. $C_{in}(v_j)$ and $C_{out}(v_k)$ denote the number of consecutive incoming and outgoing pairs of edges, for nodes v_j and v_k respectively. For example, for $d = 1$, the value of the formulation then becomes $\frac{1}{2}\left(\frac{C_{in}(q_x)}{|E_{in}(q_x)-1|} + \frac{C_{out}(q_x)}{|E_{out}(q_x)-1|}\right)$. For convenience, the cost model is normalized in the range of $[0, 1]$.

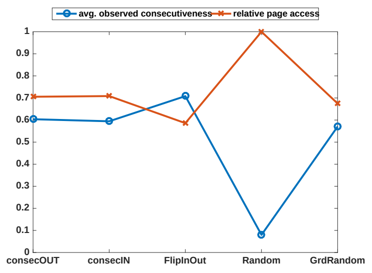
The relative cost in Equation 5.2 above, is essentially the number of consecutive edges as a fraction of total edges (both incoming and outgoing for the neighborhood query) over the whole paths of query execution. The cost for a uni-directional query is simply derived by ignoring either the incoming or the outgoing terms above. Observe that this formula also closely resembles the consecutiveness definition in Equation 5.1. Then, we can estimate an average consecutiveness number (query cost, QC_{avg}) of n query runs as:

$$QC_{avg} = \frac{1}{n} \sum_{x=1}^{n} RC(q_x) \tag{5.3}$$

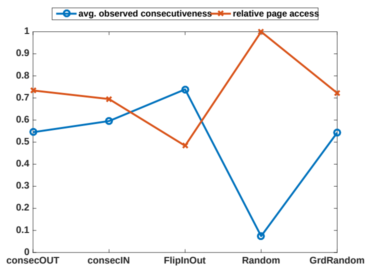
In our experimental analysis, for each labeling scheme, we also test the average page accesses over n runs for $d = 2$. Figure 5.12 shows the inverse relationship between average observed consecutiveness and the page accesses. The red line depicts the relative page accesses and the blue line depicts the average observed consecutiveness (QC_{avg}). In this figure, the page accesses is taken relative to the Random labeling scheme so that can be normalized in the range [0,1]. Figure 5.12 clearly demonstrates that methods with higher consecutiveness have lower page accesses.



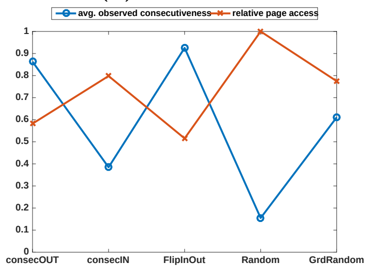
(a) WikiVote



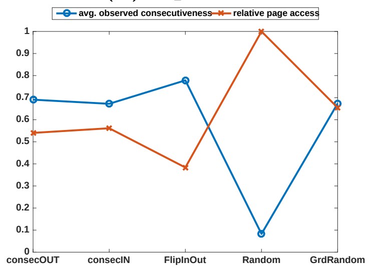
(b) Epinions



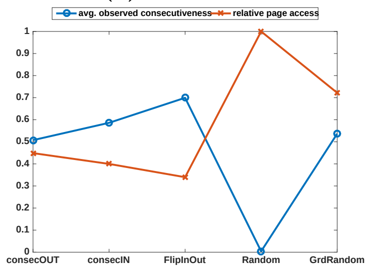
(c) SlashDot



(d) WikiTalk



(e) Flickr



(f) LiveJournal

Figure 5.12: Inverse correlation between observed Avg. Consecutiveness vs. the relative page accesses.

Observation 9. The higher the average consecutiveness, better performance is observed for disk I/Os. For the neighborhood query, FlipInOut demonstrates the highest consecutiveness (corresponding to the lowest page accesses) across all the datasets. QC_{avg} is a good predictor for estimating the page accesses.

For all the datasets, FlipInOut appears to have better results with the highest consecutiveness scores and lowest number of page accesses for neighborhood queries. With the exception of Epinions, the calculated QC_{avg} of consecIN and consecOUT become closer to the performance of FlipInOut. consecIN and consecOUT also clearly behave different for two datasets WikiTalk and LiveJournal which are heavy on outgoing and incoming edges respectively (as shown in Table 5.4).

It must be noted that the consecutive edges contribute directly to the pages required to store the edges. However the exact number of pages occupied is dependent on a variety of factors including the compression scheme of the internal graph system. On the other hand, it must be noted that the actual number of disk I/O for a given query is also dependent on the caching behavior and the buffering systems in place. We leave the development of advanced analytical cost models for the prediction of experimental page accesses as future work.

5.4.6 Balance of Labeling

The edge consecutiveness of graph G defined in Section 5.3.1 is a combination of individual metrics $C_{\text{in}}(G)$ and $C_{\text{out}}(G)$. As discussed in Section 5.3, our proposed methods attempt to maximize the consecutiveness and have a balance between the in- and out-consecutiveness. The method consecIN performs perfectly on the $C_{\text{in}}(G)$ metric but penalizes $C_{\text{out}}(G)$. Methods that are non-biased to a single edge direction possess the property that $C_{\text{in}}(G) \approx C_{\text{out}}(G)$, i.e., the *balance* $C_{\text{in}}(G)/C_{\text{out}}(G) \approx 1$. For query workloads that are uniform with respect to accesses of incoming and outgoing edges, it is desirable to increase total $C(G)$ while maintaining the balance of labeling.

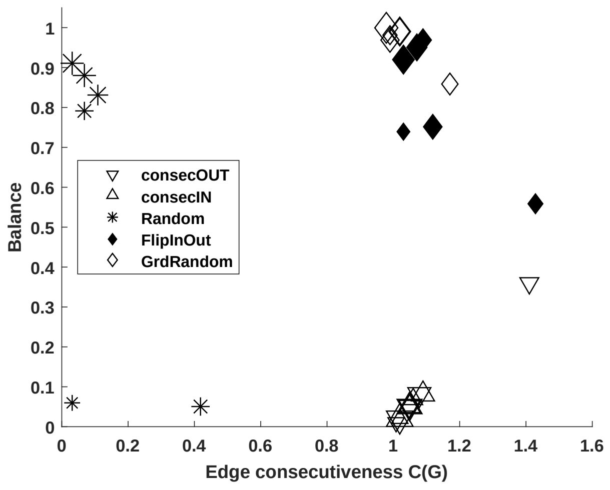


Figure 5.13: Trade-off between consecutiveness and balance. Markers are scaled according to graph size.

Figure 5.13 shows the trade-off between these two properties (balance on y-axis and total graph consecutiveness on x-axis) for all labeling methods on the datasets. The markers on the figure are scaled with the graph size. We aim to be at the upper right corner of the matrix maximizing on both consecutiveness and balance. In the bottom right, consecIN and consecOUT algorithms exhibit high $C(G)$, however fail to balance the consecutiveness. We observe that our proposed methods are consistently placed in the upper right corner demonstrating the desired balanced property of these algorithms.

5.5 Application: Streaming Graph Partitioning

In this section we investigate an application area that can benefit from an improved edge labeling, namely streaming graph partitioning. In the application of partitioning, our conjecture is that if the input stream is already pre-processed to ensure locality, better performance (e.g. lower edge cuts) can be achieved. For example, the FlipInOut edge labeling and its locality benefits can be achieved via small modifications in the crawling procedure by leveraging the true or estimated (via sampling) marginal degree distribution of the input graph.

We start by briefly summarizing the basics of the streaming partitioning model. For an *undirected* graph $G = (V, E)$, the vertices arrive in a stream, each with the set of its adjacent neighbors. The goal is to divide the set of vertices into k disjoint partitions $(P_1, ..., P_k)$ such that inter-edges, i.e., the *edge cuts*, are minimized. Streaming algorithms [184, 195, 144], which focus on scalable partitioning solutions to large graphs with time and space constraints, assign each vertex to a partition using local graph information (e.g. the existing partitions), and never move it. Existing work, such as LDG [184] and Fennel [195], shows that the produced edge cuts are comparable to the ones created by offline versions with access to the whole graph, such as METIS [96].

Given that the node assignment is based on increasing amount of information (i.e., the partitions at time t), the *streaming order* is an important consideration that affects the performance of the greedy node assignments to partitions. Existing work in this area usually considers three *node* streaming orders; Random, Breadth First Search (BFS), and Depth First Search (DFS). The Random ordering is practical, does not involve pre-processing, and is preferred for very large graphs.

Based on FlipInOut, we introduce a streaming graph partitioning method, FlipCut, which is (almost) agnostic to the neighborhood of each incoming vertex. We define as k the number of partitions with capacity C, and $P^{(t)}(i)$ the i^{th} partition at time t. Unlike other methods, which consider a streaming order of *vertices*, FlipCut considers one *edge*, (v_1, v_2), at a time in the FlipInOut order, and assigns its endpoints to partitions based on the following three rules:

1. If v_1 and v_2 are already assigned to partitions, FlipCut ignores the incoming edge.
2. If v_1 and v_2 have not been assigned to a partition yet, both of them are assigned to the partition with the minimum load at time t, i.e., $argmin_{i\in\{1,2,...,k\}}\frac{|P^{(t)}(i)|}{C}$.
3. If only v_1 is not assigned to a partition, it is assigned to the partition of its neighbor v_2, i.e., $arg_{i\in\{1,2,...,k\}}P^{(t)}(i)\cup v_2$. If that partition is full, rule 2 is applied, and v_1 is assigned to

the current smallest partition, i.e., $argmin_{i \in \{1,2,\ldots,k\}} \frac{|P^{(t)}(i)|}{C}$. If v_2 is the only unassigned endpoint, it is handled similarly.

In other words, the endpoints of any streaming edge dictate the order in which FlipCut will visit the vertices. We strive to keep a set of simple rules assuming that we are already working on an edge set that has improved locality. An advantage of FlipCut over other baselines is that it only inspects one edge at a time, and decides on the placement of the incoming edges' vertices *without* accessing the subgraph of already seen vertices. FlipCut does only one pass over the edges of a directed graph, and thus runs in $O(|E|)$ time.

5.5.1 Baseline Methods and Methodology

To evaluate the performance of our streaming graph partitioning method, FlipCut, with respect to the percentage of edge cuts, we compare it to three state-of-the-art methods:

• BFS + LDG [184]: This is the best performing method among 3 node orderings and 7 partitioning heuristics in [184]. The nodes are being read in BFS order, and assigned to partitions according to the Linear Deterministic Greedy (LDG) heuristic. The idea of LDG is to assign an incoming node v to the partition with most of its neighbors, while penalizing larger partitions and imposing size of $\sim |V|/k$ vertices in each partition. In order to make the decision for a given node v, LDG looks at the partitions of all the neighbors of v.

• Random + LDG [184]: Nodes arriving in a random order is another streaming order tested due to its simplicity and scalability to large graphs. On average, it has been observed [184] to report comparable performance to BFS + LDG.

• Hashing [122, 95]: A node is hashed to a partition independent of the graph structure. The vertices can be distributed evenly across the partitions and the expected fraction of edge cuts for $k \geq 1$ partitions is $1 - \frac{1}{k}$. This technique is widely used in practice because it is simple and can efficiently determine the partition of a node without maintaining a mapping table. The performance of hashing also acts as a classic upper bound.

5.5.2 Results

To compare FlipCut with the baseline methods, graphs were made undirected for the LDG baselines. As the graph size grows, it is more sensible to test with higher value for the number of partitions, k. Thus, for small graphs, we test with partition sizes 4 and 8, and for the larger graphs (Flickr, LiveJ) we vary k from 12 to 100.

The percentage of edge cuts is shown in Table 5.6 for $k = 4$ and $k = 8$. We see that with the exception of Epinions, LDG with BFS and random ordering exhibit similar performance, which has been confirmed on other datasets as well [184].

Observation 10. For $k = 4$ partitions, FlipCut has 8% to 42% reduction in edge cuts compared to the LDG variants, and 26%-50% reduction compared to the Hashing method.

For 8 partitions, the benefit compared to BFS+LDG becomes smaller, and is very similar in the case of Epinions and WikiTalk. FlipCut outperforms the Hash partitioning by a large margin, and also has potential for practical use, given that it requires observing only a single edge at a time, without accessing the whole graph. As expected, for all the methods, the fraction of edge cuts increases with more partitions.

Table 5.6: Percentage of edge cuts for 4 and 8 partitions. Lower (in bold) is better. Italics indicate near-ties.

Data	BFS+LDG	Random+LDG	FlipCut	Hashing
		k = 4		
WikiVote	63.63	69.93	**41.41**	75.0
Epinions	26.94	64.98	**24.87**	75.0
SlashDot	63.32	65.20	**36.48**	75.0
WikiTalk	56.45	54.47	**48.54**	75.0
		k = 8		
WikiVote	78.35	82.09	**65.73**	87.5
Epinions	**41.33**	76.56	*42.5*	87.5
SlashDot	76.52	76.81	**66.85**	87.5
WikiTalk	**62.46**	64.17	*63.12*	87.5

Table 5.7 shows the fraction of edge cuts for Flickr and LiveJ. As done in [184], for the larger graphs we compare our method to the natural ordering provided in the original dataset. In addition, we also test with a random node permutation.

In the case of Flickr, our method shows a reduction in edge cuts compared to all the other methods. For LiveJ, although the edge cuts are improved compared to the Random and Hashing counterparts, this is not the case compared to the Natural Order + LDG. We speculate that one reason for this discrepancy is the high clustering coefficient of LiveJ compared to the other graphs (Table 5.3). It may have had an adverse effect on FlipCut, since it does not use the graph structure information (other than the current edge) when deciding the vertex assignments. We note that independent work also reports that the LiveJ graph exhibits different behavior from other social networks; Chierichetti et al. [41] focus on network compression and claim that the natural crawl order outperforms their Shingle ordering method.

Table 5.7: Percentage of edge cuts for the largest graphs with higher number of partitions, k. Lower (in bold) is better.

Data	k	Natural + LDG	Random + LDG	FLIPCUT	Hashing
Flickr	12	55.85	85.45	**27.04**	91.7
	24	61.65	89.74	**54.12**	95.8
LiveJ	24	**41.01**	87.88	63.67	95.8
	50	**46.99**	90.56	70.16	98.0
	100	**51.74**	92.04	75.00	99.0

This experiment shows the potential applicability of FLIPCUT on generating partitions in a streaming setting. Our results are promising, as they display consistently better cuts compared to LDG with random ordering. We emphasize that FLIPINOUT, which was designed with a different goal in mind and was not optimized for reducing the edge cuts, has the side-benefit of supporting streaming graph partitioning. Lastly, existing algorithms generally work on undirected graphs inspecting $2|E|$ edges while FLIPCUT can produce comparable results observing only half the edges for a directed graph.

5.6 Summary

In this chapter we proposed the problem of effective edge labeling techniques on directed graphs. The contributions of our work can be summarised as follows.

- **Formulation of edge labeling problem:** We propose an edge consecutiveness metric on directed graphs (that takes into account both outgoing and incoming edges) and formulate edge-labeling as a maximization problem of this metric (Section 5.3.1). In a query cost model (Section 5.4.5) we show the inverse relationship between the theoretical consecutiveness measure and the average disk I/Os for a query run.

- **New Algorithms:** In (Section 5.3) we introduce GRDRANDOM and FLIPINOUT as two edge-labeling algorithms that focus on the balance between numbering outgoing and incoming edges. These two novel and efficient edge-labeling schemes maximizes the consecutiveness at every individual vertex so that local decisions greedily make progress towards the global optimal.

- **Experiments on real large-scale graphs:** We conduct extensive experiments and show that our edge-labeling schemes did in fact lead to significantly improved query times and disk I/O performance by achieving a better layout and locality of edges on disk (Section 5.4).
- **Other applications of improved labeling:** We demonstrate a case study of our methods to be applied in streaming graph partitioning. Based on FlipInOut, we introduced FlipCut, an effective one-pass, neighborhood-agnostic strategy for streaming graph partitioning which resulted in reduced edge cuts compared to state-of-the-art methods (Section 5.5).

Chapter 6

Social-Textual Query Processing on Graph Database Systems

In Chapters 3 and 4, we investigated the use of GDBMS in social network analytics and evolving code dependency application settings respectively. We explored how existing query mechanisms of graph database systems can express and efficiently perform interesting queries in each of the application scenarios. In this chapter we study in detail a specific query that involves the integration of keyword search and graph traversals. In previous chapters we made use of different attributes that were attached to both nodes and edges of property graphs. Apart from the standard attributes, a graph may have text associated with it in some shape or form. For example, the social network on Facebook connecting the users has text associated with the posts that they share, or in a co-authorship network such as DBLP there may be research interests attached to each author.

In this chapter we investigate a new query that requires a combined graph traversal and text search in a graph database system. In a social network context, given a query user u and a keyword w, our objective is to retrieve k users with the highest ranking scores, measured as a combination of their social distance to u and the relevance of the text description to the query keyword w. We leverage graph partitioning strategies in our proposed approach to speed-up query processing along both dimensions. We conduct experiments on real-world large graph datasets and show benefits of our algorithm compared to several other baseline schemes.

6.1 Introduction

A property graph model is able to capture many of the graphs emerging from the real-world, with different types of nodes, edges, and attributes describing them. Most of the graph-structured data, particularly those generated from social networks, may have specialized text attributes associated with the nodes. For example as shown in Chapter 3, in Twitter, users are connected via the "follow" relationship and those user nodes may be associated with tweets or hashtags they generate. Examples of text and other attributes on real-world graphs from different domains are shown in Table 6.1. We need to treat the text attributes (terms, hashtags, research interests) as a specialized type of attribute as queries performed on these unstructured text items can be more complex involving relevance ranking than simple predicates on other attributes returning exact answers.

Table 6.1: Nodes, Edges and text attributes of graphs from different domains.

	NodeType (Attributes)	**Text attributes**	**EdgeTypes (from-to)**
Twitter	User (name, location)	keywords, hashtags	follows (user-user)
LinkedIn	User (name, place of work, location)	skills, profile summary	connected (user-user)
DBLP	Author (name) Publication (title)	affiliation, research interests abstract	cites (author-paper) co-author(author-author)
FLARN	Points of interest (coordinates x,y)	POI type	path (POI-POI)

Graph database systems are increasingly being used to store and manage large-scale property graphs with complex relationships. Standard graph database systems such as Neo4j and Sparksee are optimized for graph traversals with 'index-free adjacencies'. Although these systems have support for indexing of attributes, there have been no comprehensive studies on how different dimensions stored with a graph can work well together. The dimensions of a graph can be the connectivity of the network, predicates on entities (nodes and edges) and other textual attributes on them. For efficient query processing, these dimensions are fundamentally supported very well by different storage models — queries on the topology by graph database systems and specialised indices; queries on predicates by relational and key-value stores; and queries on text search by information retrieval systems having specialized full-text indexing schemes. Our goal is to investigate how a full-text search can be seamlessly integrated into graph traversals within a graph database system.

The objective of graph database systems is to be able to scale graph type traversals queries on very large graphs. Existing graph database systems such as Neo4j [140] and Titan [12] provide support for indexing on both node and edge properties for exact search on graphs, while full-text search capabilities are supported by an external text search engine such as Lucene [9]. Recently, APOC procedures in Neo4j also provided enhanced features to access the indexes. Full-text searches are not first-class citizens of graph systems and thus are not yet fully integrated into the graph schema. However, real-world graphs and practical queries on them demand graph database systems that can facilitate the integration of text search with graph traversals. For example, in Twitter, if a user is looking for people interested in a particular topic, he/she is more likely to respond to and make friends with users in his/her close neighbourhood talking about this topic; In LinkedIn, if a professional is looking for users with a set of skills or interests, it is easier to be introduced to suggested users who are, for example, 2-steps away.

Motivated by the requirements above, in this chapter we focus on a query that can process graph traversal and text search in combination. We introduce a query that retrieves k objects that are both socially close to a query user u and are textually relevant to a query keyword w. We denote this query as a *Social Textual Ranking Query* (kSTRQ). This can be the basis of queries that involve graph traversals with conditions on the structural content on the nodes.

Example 6.1. *kSTRQ example.* A user may be interested in finding friends who are interested in going to the 'Australian Open'. In Twitter, this translates to finding the top-10 users of a user u_5 (perhaps from his close neighbourhood) who have mentioned terms relevant to the query hashtag `#AusOpen`. The objective of the query is to suggest 10 users who are socially close to the query user (more likely to be friends) and who have used terms relevant to query keyword `#AusOpen`. This can be answered with kSTRQ where $u = u_5$, $w =$ `AusOpen` and $k = 10$.

Related work on social media platforms are specifically tailored for different types of social query workloads that are not necessarily focused on graph traversals. Facebook's Unicorn system [47] proposes methods for performing the 'typeahead' search on the social graph. The typeahead query enables users to search other users by typing the first few characters of a person's name who are within his close network. The prefix search is essentially a different query and their solution is focused on returning users from direct friends or friends-of-friends. EarlyBird engine [27] focused on rapid data ingestion enables Twitter's real-time search service. In addition to other features, the ranking function only utilizes the user's local social graph to compute the relevance score for a Tweet. Our solution is more generic, involving a user's n-step neighborhood with the ability of varying the preference to each dimension thereby perfectly complementing these prior works. Graph keyword search has been studied on RDF [192, 56, 142] and XML [97, 90, 74] graphs which exhibit different characteristics with an output that includes subgraphs while kSTRQ requires a ranked list of node entities. Several proposals for answering the *nearest* keyword search have been studied [49, 13, 158], in which an approximate distance between two users has been considered. Specialized index structures have been proposed in other works [158, 86, 117] to overcome the major drawback of distance estimation errors in approximate approaches. More details of related work are provided in Section 6.2.

Different from these approaches, we propose our algorithm PART_TA to efficiently answer kSTRQ queries in a graph database setting, aiming at a general solution based on the graph data model. We wish to construct a combined index along the graph and text dimensions similar to an IR-tree [43] designed for answering spatial-textual queries. We believe that a good graph partitioning can represent a generic solution for a graph index and we combine this with text lists that map to the partitions. Our intuition is that this graph partitioning approach serves as an index that can quickly find the socially close users, by placing them within the same partition. These smaller components of the graph enable us to run kSTRQ locally, searching for k results, traversing and expanding as few partitions as possible. Then we assemble the results from the locally generated partitions to construct the final answer to the kSTRQ. Our key contributions of this work can be summarized as follows.

- **Methodology.** We reviewed existing algorithms and adopt relevant techniques appropriate for a graph database setting. To the best of our knowledge, we are the first to conduct a detailed investigation on running the kSTRQ in a graph database system.

- **Algorithm.** We designed an algorithm PART_TA to efficiently process kSTRQ by introducing graph partitioning to optimize the use of existing techniques on a decomposed graph.

- **Experiments.** We conducted experiments on three real-world datasets, Twitter, Aminer and Flicker with different characteristics. Our proposed solution performs well on large graphs demonstrating performance gains up to 76% compared to other baselines.

6.1.1 Chapter Organisation

The rest of the chapter is organised as follows. In Section 6.2 we review existing work relevant to our problem. Then, in Section 6.3 we give background to the kSTRQ and provide formal definitions of the query, relevance ranking and proximity measures. Section 6.4 introduces the baseline approaches we implemented on a graph database system. We present our proposed partition-based approach in Section 6.5 and discuss its variations and optimisations. Finally we experimentally evaluate our approach and discuss our findings in Section 6.6 on real datasets.

6.2 Related Work

In this section, we discuss several categories of existing work that are closely related to the graph keyword search problem. We present relevant work on graph-keyword query processing, solutions from social media platforms and orthogonal work that integrates different query dimensions.

6.2.1 Social Graph Queries – Twitter and Facebook

Closely related work from industry exists from research at Twitter and Facebook but are not focused on performing graph traversals. The main focus on Twitter search as described by the EarlyBird [27] engine, is rapid data ingestion and enables text to be immediately available for search with real-time results. The engine performs filtering and personalization to retrieve the most relevant results using static and dynamic signals. Static signals such as information the user's local social graph are added during indexing time. Dynamic signals such as the user's language and query timestamp also contribute to the relevance score. Unicorn [47] describes the system for searching the social graph in Facebook and is the primary back-end for Facebook Graph Search. It provides the primary infrastructure for Facebook's 'typeahead' search which enables users to find other users by typing the first few characters of the person's name. The query essentially performs a prefix search that returns a ranked list of relevant users from either direct social circle or friends-of-friends. The social graph is partitioned (based on geographic similarity) and also maintains postings lists for every name prefix up to a predefined character limit. The query engine defines a set of primitives that involve fast set operations. Unicorn's architecture is highly customized for the particular typeahead query, scaling to large volumes of data. In both of these platforms the frameworks are highly customized for very specific query workloads and are not focused on graph traversals beyond a 1- or 2-step neighborhood.

6.2.2 Keyword search on graphs

Keyword search has been studied with a focus on specific types of graphs such as RDF and XML. The knowledge bases in RDF model are represented as subject-object-predicate triples and are queried with well defined languages such as SPARQL. Keyword search on RDF collections

[192, 56, 142] are explored to facilitate querying without having expertise on languages like SPARQL. The general idea is that the models retrieve a set of RDF subgraphs matching a query keyword and rank them using statistical language models which take the distribution of terms into account. The type of workloads involve efficient retrieval of triples while our workloads are fundamentally different. The schema defined in these corpuses may involve several thousand different edge types, whereas in our work the node and edge types are limited. Keyword search over XML data [97, 90, 74] has also been studied. XML data is a specialised tree-structured graph and the keyword search returns snippets of XML documents as the query result. The most relevant results are generally defined to be the smallest XML subtrees containing the keywords. The focus of these existing studies is returning a subgraph while we want to retrieve an ordered set of graph nodes.

Different from keyword search over XML, a category of related works [78, 204, 17, 100] studied search with keywords over graphs that may not necessarily be tree structured. The idea is to efficiently explore a graph, returning the sub-structures with distinct roots containing query keywords. The results are ranked using a scoring function that involves node scores, the extent of the match and edge scores reflecting the strength of the connection. Although not operating on XML graphs, similar to keyword search over XML, these queries return graph substructures containing the keywords. Another specialized form of a graph keyword search query known as the *typeahead* query has also been investigated [47, 60] involving efficient ways to perform prefix searches in the graph.

There have been several proposals for answering the *nearest* keyword search query where an approximate distance between two users/nodes are considered [49, 13, 158]. This is the most relevant work for our problem that is not operating on specialized graphs such as RDF or XML. The approximate distances in the Partitioned Multi-Indexing (PMI) scheme [13] are calculated based on 'distance oracles' [49] which is used to estimate the distance between two nodes. A major drawback in this approach as observed in later studies [158, 86, 117] is that the distance estimation error is large in practice, thus affecting the ranked result list. Alternative tree-based strategies [158] have been proposed but they are memory-based and do not scale well to large graphs and it is not clear how ranking in other dimensions (such as textual similarity) can be incorporated into those schemes. Problem-specific indexes are proposed in some studies [117, 158] and they are not easily extensible to the generic graph database system scenario we address.

6.2.3 Orthogonal work in multiple domains

There have been several studies on the combination of querying the graph, along with predicates (attributes) on nodes and edges. G-SPARQL [172] proposed a SPARQL-like language for querying attributed (property) graphs. The general idea is that the topology of the graph is in memory, while attributes are on disk in a de-normalized relational model. A query optimizer decomposes the query and determines the order to execute parts of the query such that the intermediate results are minimized. Horton+ [175] is another system that can express reachability queries with predicates on nodes and edges to match graph paths.

There has been related works that combines the efficient processing of queries in other domains such as spatial-textual [43, 118, 59] and spatial-social [139, 11]. IR-tree [43] is an index introduced to efficiently perform queries that involve the spatial and the textual dimensions. IR-Tree enables a query that is composed of a keyword and a location, and retrieves documents that are both geographically and textually close to the query object. In the IR-tree, a traditional spatial R-tree index is augmented with an inverted index having the posting lists at each level of the tree.

As there is no universally agreed index to query graph data (like the R-tree for spatial data), initially, we examined graph indexes built with the objective of speeding up different types of queries such as subgraph matching [218, 228, 226, 191], shortest path and reachability [194, 87, 221]. Similar to the IR-tree, our goal was to review if these indexes can be extended with text information. We did not continue with this approach for several reasons; many of these graph indexes involved a large pre-computation overhead [191] and was focused on efficient processing of specific queries.

6.3 Problem Definition

In this section we formally introduce the problem of top-k graph keyword search in a network. Let $G(V, E)$ be a social graph with a set of users represented as vertices $v \in V$ and a set of edges $e \in E$ connecting them. An edge can represent any social interaction among users. Each node $v \in V$ contain a set of zero or more keywords associated with it, denoted by $D(v)$. The set of vertices containing a given keyword w is denoted by $V(w)$ where $V(w) \subseteq V$. Notations we use in definitions and later in algorithms are listed in Table 6.2. A user v is ranked based on the combination of distance to query $q.u$ and textual descriptions relevant to query term $q.w$ as illustrated next.

Social Proximity: A path $p = (v_1, ...v_l)$ is a sequence of l vertices in V such that for each v_i $(1 \leq i \leq l)$, $(v_i, v_{i+1}) \in E$. The length of the path is the number of edges along the path. Social proximity between any two users v_i and v_j, denoted as $s(v_i, v_j)$ is based on their shortest path distance:

$$s(v_i, v_j) = \frac{sdist(v_i, v_j)}{sdist_{max}} \tag{6.1}$$

where $sdist(v_i, v_j)$ is the length of the shortest path connecting v_i and v_j. The $sdist_{max}$ is the largest shortest path length between any pair of vertices in the graph, used to normalise $s(v_i, v_j)$ between [0,1]. We adopt the shortest path approach as a measure of social proximity as previous work [223, 201] has shown that it effectively captures the influence between two users. A higher value of social proximity $(1 - s(q.u, v))$ for node v indicates better social relevance to query node $q.u$.

Textual Relevance: A user v is considered relevant to the query iff v contains the query term $q.w$ at least once, i.e. $q.w \in D(v)$. The textual relevance denotes the similarity between a query term $q.w$ and $D(v)$. We adopt the standard tf-idf model.

$$t(q.w, D(v)) = \frac{tf(q.w, D(v)) \times idf(q.w)}{tdist_{max}} \tag{6.2}$$

where $tf(q.w, D(v))$ denotes the number of occurrences of term $q.w$ in $D(v)$(i.e $freq$) and calculated as $freq/(|D(v)|)$. $idf(q.w)$ denotes the inverse document frequency of $q.w$ ($docFreq$)

Table 6.2: Some notations used in definitions and algorithms.

Symbol	Description
V, E	set of vertices (nodes) and edges, resp.
$D(v)$	set of keywords associated with node v
$V(w)$	set of vertices containing the keyword w
$s(v_i, v_j)$	social proximity between users v_i and v_j
$t(q.w, D(v))$	textual relevance between query term $q.w$ and terms used by user v
$sdist_{max}$, $tdist_{max}$	maximum possible social and textual scores resp.
R	final result set with k nodes
f_k	the k-th highest f value in the result set R
$p(v)$	partition for node v
$S(q.u)$	social list with decreasing social proximity to $q.u$
$T(q.w)$	text list with decreasing text relevance to $q.w$
P	set of partitions for graph G
$S(P_i, q.u)$	social list for partition P_i for query user $q.u$
$T(P_i, q.w)$	text list in partition P_i with decreasing text relevance to $q.w$
$B(P_i)$	set of boundary nodes for partition P_i
$C(v)$	subset of boundary nodes closest to v

in the entire document collection calculated as $(numDocs/(docFreq+1))$. $tdist_{max}$ denotes the maximum score for the term, used to normalize the text score to [0,1], which is denoted by:

$$tdist_{max} = \max_{v \in V} t(q.w, D(v)) \tag{6.3}$$

A higher value of textual relevance $t(q.w, D(v))$ for node v indicates better textual relevance to query keyword $q.w$.

Overall ranking function. Following common practice in combining rankings from different domains, we apply a linear function [43, 118] over the normalized social and textual proximity to rank objects. Given a query user $q.u$ and a keyword $q.w$, the ranking of $v \in V$ is determined by function f as:

$$f(v) = \alpha \cdot (t(q.w, D(v))) + (1-\alpha) \cdot (1 - s(q.u, v)) \tag{6.4}$$

where $0 \leq \alpha \leq 1$ denotes the relative significance of the individual components in the two domains.

Definition 7 (kSTRQ)**.** Top-k Social Textual Ranking Query on a graph G can be expressed as a triple $q = (u, w, k)$ where $u \in V$ is the query vertex in G, w is a keyword and k is a positive integer denoting the number of output records. kSTRQ query returns a result set R that contains k users $v \in V - \{q.u\}$ with the highest $f(v)$ values.

6.4 Baseline Algorithms

kSTRQ is a first attempt at providing a solution to the social-textual ranking queries in a graph database system. Although variations on this query processing exist, dealing with approximate distance calculations, operating on different graphs, and introducing specialized structures (as discussed in Section 6.2), they are not directly applicable to solving kSTRQ. This is either because they do not solve the exact query or they make use of customized index structures that are not easily extensible in a graph database setting. Thus there is no straightforward way to use any of these approaches as our baselines. Instead we resort to other approaches that we can modify and implement in our current setting.

A naive approach is to go through all the objects v in the graph calculating the combined $f(v)$ score (Equation 6.4) for each node, sorting it and returning the top-k elements in the list. This computation may be prohibitively expensive considering that compared to the total number of nodes, only a small subset of nodes may include the query term. As a result, our first baseline involves traversing the posting list with the query term and calculating the combined score f.

6.4.1 Text First Algorithm (TFA and TFA_ET)

kSTRQ can be processed using a Text First Algorithm (TFA) that iterates through the text list that contains the query term (also known as the postings list). For a given keyword $q.w$, it processes users who have used the term $(V(q.w))$ in decreasing text relevancy. For each user $v \in V(q.w)$, TFA calculates the social proximity between $q.u$ and v (Equation 6.1) and in turn computes the combined value f. If v is the last user in set $V(q.w)$, the expression $\theta = \alpha.t(q.w, D(v))$ (lower) bounds the f value of every non-encountered user.

Retrieving a sorted list of text scores is a fundamental operation in any text index. However, the calculation of the shortest path between two random users (for social proximity) may be a more costly operation. We rely on the ability of the graph database system to calculate the shortest path; alternatively one can utilize an external index for efficient shortest path calculation. The TFA approach works well when the frequency of the query keyword is low. The complexity of this algorithm is determined by the size of the postings list $O(|V(w)|)$.

Inspired by the Threshold algorithm [58], an early termination condition on TFA (named, TFA_ET) enables traversing a sorted postings list partially. At each iteration of TFA, the algorithm keeps track of the score of the current k-th object in the result set, denoted by f_k. The best possible *sdist* is when any user v, is 1-step (direct neighbours) from query user $q.u$ and thus the right of the '+' operand of Equation 6.4 can be upper bounded to the maximum social score $maxSS$. For a new object, if f_k exceeds the text relevancy score combined with $maxSS$ (from Equation 6.4), the algorithm can terminate. The reason is that it is guaranteed that the f-score of unseen objects is always lower than the current kth score.

6.4.2 Social First Algorithm (SFA and SFA_ET)

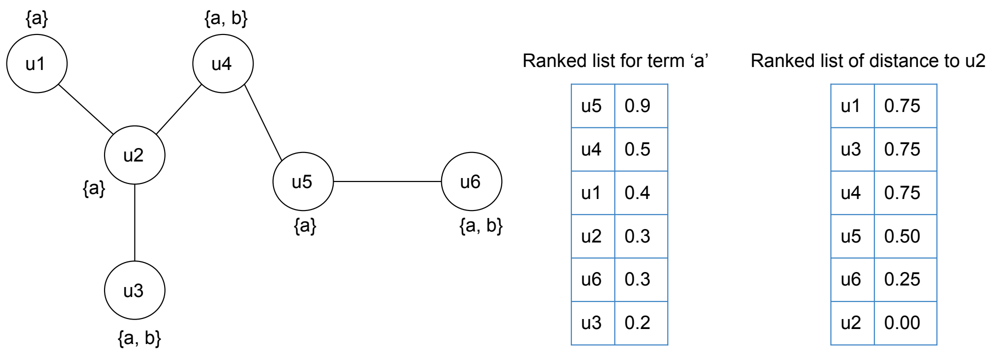
The main idea of SFA is to consider users in increasing social distance to the query user $q.u$. A Breadth-first-search (BFS) and a Dijkstra algorithm around $q.u$ would be required for unweighted and weighted graphs respectively. For every encountered user v, the text relevance of query keyword $q.w$ is computed (if v exists in $V(q.w)$), and then calculates the combined score f. Finding the text score requires a random access to the postings list $V(q.w)$. The first top-k users are placed in the interim result R. For any subsequent user v, if $f(v) > f_k$,

v is added to the interim result. To efficiently perform this, a forward index which keeps the text relevance score of each user/term is helpful. The number of iterations of this algorithm is upper bounded by $O(|V|)$. If the graph is well connected with a small diameter, BFS can be computed efficiently.

SFA can also be terminated early by keeping track of the f_k value. Early termination of SFA (named SFA_ET) may exit early, without having to perform a full BFS. The best possible text score is $tdist_{max}$ (Equation 6.3). For any calculated $sdist$, the left of the '+' operand in Equation 6.4 can be upper bounded to the maximum textual score $maxTS$. For a new object, if f_k exceeds the social proximity score combined with $maxTS$ (from Equation 6.4), the algorithm can terminate. The reason is that it is guaranteed that unseen objects are always lower than the current kth score.

The drawback of both TFA- and SFA- based algorithms is that they are ignorant of the social or the textual dimension respectively. SFA would be unnecessarily traversing nodes that either have a low textual score or worse, not be in the postings list at all. Similarly, TFA may traverse nodes that are socially distant to the query node. To overcome this limitation, the threshold algorithm we describe next, iterates through both dimensions simultaneously for efficient pruning of results.

6.4.3 Threshold algorithm (TA)



Ranked list for term 'a'

u5	0.9
u4	0.5
u1	0.4
u2	0.3
u6	0.3
u3	0.2

Ranked list of distance to u2

u1	0.75
u3	0.75
u4	0.75
u5	0.50
u6	0.25
u2	0.00

Figure 6.1: Social network and ranked lists for term a and distance to u_2

In the threshold algorithm [58] (TA), two ranked lists are maintained for each of the social and text dimensions. The social list is in increasing social distance (decreasing proximity) to $q.u$ denoted by $S(q.u)$, while the textual list is in decreasing textual relevance to $q.w$ denoted by $T(q.w)$. Figure 6.1 shows the ranked lists based on the social proximity from query user

u_2 ($s(u_2, u_i)$) and text relevancy for query term a ($t(q.a, D(v))$). Sorted and random access to each of the lists are required to calculate the candidates overall score f.

Algorithm 5 Threshold Algorithm

```
Input: Sorted social list $S(q.u)$, sorted text list $T(q.w)$, $k$
Output: Result set $R$ containing the top-$k$ users
 1: visited ← {}, position ← 0, threshold $\theta$ ← 0
 2: R ← getNewPriorityQueue()
 3: while top-$k$ elements in R > $\theta$ do
 4:     $v$ ← S.getS(position)                          ▷ sorted access to $S$
 5:     if $id(v) \notin$ visited then
 6:         sScore ← $score(v)$
 7:         socialThreshold ← $score(v)$
 8:         tScore ← T.getR($id(v)$)                    ▷ random access to $T$
 9:         if tScore $\neq$ 0 then                     ▷ will be 0 if $id(v) \notin$ T
10:             $f$ ← COMBINE(sScore, tScore)
11:             R.Enqueue($id(v)$, $f$)
12:             visited ← visited ∪ {$id(v)$}
13:         end if
14:     end if
15:     $v$ ← T.getS(position)                          ▷ sorted access to $T$
16:     if $id(v) \notin$ visited then
17:         tScore ← $score(v)$
18:         textThreshold ← $score(v)$
19:         sScore ← S.getR($id(v)$)                    ▷ random access to $S$
20:         $f$ ← COMBINE(sScore, tScore)
21:         R.Enqueue($id(v)$, $f$)
22:         visited ← visited ∪ {$id(v)$}
23:     end if
24:     $\theta$ ← COMBINE(socialThreshold, textThreshold)
25:     position ← position + 1
26: end while
```

The TA algorithm is outlined in Algorithm 5. In order to calculate a combined score, for every sorted access (getS) in one ranked list it requires a random access (getR) to the other. The COMBINE function in the algorithm uses Equation 6.4 to calculate the total score. It terminates once enough results that satisfy a particular threshold have been processed. TA requires that the preference function f (Equation 6.4) is increasingly monotone on all m attributes (in our case, $m = 2$) probing the ranked lists in a round robin fashion. For each element pulled from a list, it computes the f value of the corresponding tuple by fetching its $m-1$ attributes from the other lists via random access. It maintains an interim result of top-k tuples seen so far, it also keeps a threshold θ computed as the value of f over the last attribute value pulled from each of the m repositories. Essentially θ is an upper bound on the f value of any non-encountered tuple further down. TA terminates when θ is no smaller than any of the f values in the interim results, which is reported as the final result.

EXAMPLE 6.2. *Threshold algorithm:* The social network and the corresponding terms of six users are shown in Figure 6.1. Query user is u_2, query term is a and $\alpha = 0.5$. TA retrieves the top-2 as follows. TA first accesses u_1 in the social domain (u_1's text score from random access) with f value 0.575 and places into the interim result, $R = \{u_1\}$. Sorted postings list gives u_5 with f value of 0.7 (social score from random access) and sets $R = \{u_5, u_1\}$. Next, u_3 with $f = 0.475$ and u_4 with $f = 0.625$ is added to the list $R = \{u_5, u_4, u_1, u_3\}$. At this point the threshold values for social and textual domain are 0.75 and 0.5 yielding a threshold $\theta = 0.625$. The current top-2 results are no smaller than θ, the algorithm terminates returning top-2 $\{u_5, u_4\}$ with the highest scores.

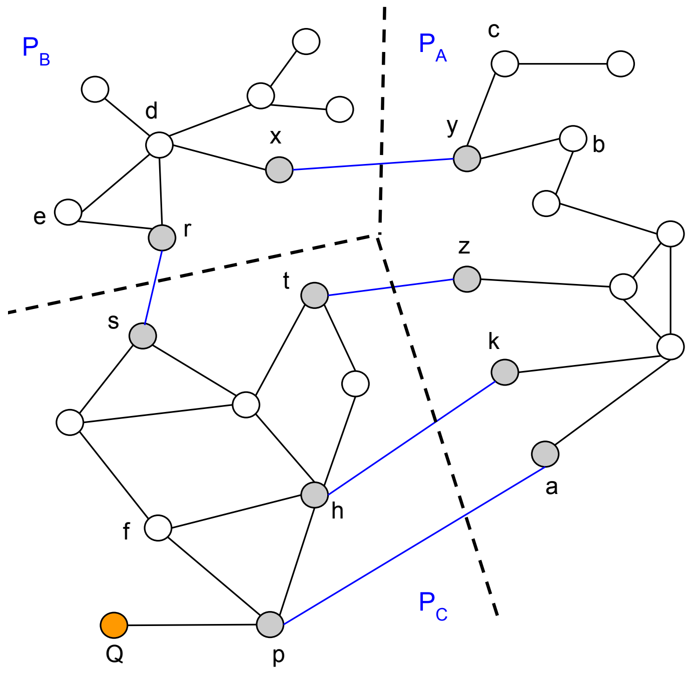
Table 6.3: Sorted and random access to ranked lists

Ranked list	sorted access	random access
users sorted by distance to $q.u$ ($S(q.u)$)	S.getS(i)	find shortest path *sdist* S.getR($id(v)$)
users sorted by text scores with $q.w$ ($T(q.w)$)	T.getS(i)	given user id, get text score T.getR($id(v)$)

Table 6.3 lists how sorted and random access is performed on each of the lists. Sorted access to $S(q.u)$ list means that, given a position i, we are able to sequentially retrieve the user in that position (S.getS(i)). Given a position i, the sorted access to text list $T(q.w)$ means we can sequentially traverse this list (T.getS(i)). In order to run the TA, efficient random access to each of the lists is necessary. Random access in the social domain requires that, given any user v, the shortest distance between query node $q.u$ and v is found. We depend on the graph database system to return the shortest path between two given nodes efficiently. Random access in the textual domain means that, given a term w, and a user v, we need to quickly find its score (i.e. *tf-idf* score). To retrieve this from $T(q.w)$, in the worst case, a full scan on the posting list is required. Random access to the $S(q.u)$ and $T(q.w)$ lists may not necessarily have the same cost. The shortest path between two random nodes further apart may be a more expensive operation compared to finding the text score of a user in the sorted text list. The number of iterations in TA, is upper bounded by the size of the postings list $|T(q.w)|$ as $|T(q.w)| \leq |S(q.u)|$.

6.5 Proposed PART_TA algorithm

The main idea in the proposed algorithm is that the data is decomposed along the social dimension by performing a graph partitioning, and the text indexes are also maintained to map the social partitions. The rationale behind graph partitioning is to enable users who are socially close, to be placed within the same partition. In some way our goal is to construct a combined index like the IR-tree [43]. The graph index is represented by a set of partitions in combination with postings lists indexed for all users within the partition. This way, for a given query user, most of this user's n-step neighbourhood will be found by traversing only a few partitions. Our algorithm is inspired by the threshold algorithm running on partitions, hence the name PART_TA. When a kSTRQ is issued, the objective is to expand as few partitions as possible to retrieve the k nodes with highest f scores. In the next sections we discuss the pre-processing steps involved in PART_TA, the query processing algorithm and some possible optimisations.



partition	$T(p_i, q.w)$ for keyword w
P_A	a: 5, b: 5, c: 1
P_B	d: 8, e: 2
P_C	f: 2

Figure 6.2: A partitioned social network with corresponding postings lists for keyword w. The edges crossing partitions are in blue and the boundary nodes are highlighted in grey. Query node is Q.

Figure 6.2 shows the social network of users decomposed into three partitions. The edges crossing the partitions (*edgecuts*) are marked in blue. The nodes that have edges cut by the partitions, known as *boundary nodes* are noted for each partition and are marked in grey. Each partition P_A, P_B and P_C maintains separate inverted indexes of terms with posting lists, denoted by $T(P_i, w)$, containing only terms associated with nodes in the given partition. For example, each partition P_i maintains postings list for a term w, if $v \in T(P_i, w)$ such that $\{v \in V(w) \cap v \in P_i\}$.

6.5.1 Precomputation

At the partitioning phase, we also note the boundary nodes for each partition. For example, the boundary nodes for each partition P_A, P_B and P_C in Figure 6.2 are $\{a, k, y, z\}$, $\{r, x\}$ and $\{h, p, s, t\}$ respectively. For r edge cuts, there could be a maximum of $2r$ unique boundary nodes for a graph. But in real graphs, since a single node may become a boundary to multiple partitions, the total number of boundary nodes are much less than $2r$. A partition that a node v belongs to is denoted by $p(v)$ and the boundary node set for a given partition P_i is denoted by $B(P_i)$. A single partition will have a maximum of $max(|B(P_i)|) = \frac{2r}{|P|}$ boundary nodes.

The objective of keeping track of the boundary nodes is to pre-calculate the minimum distance to reach each of the partitions from any given node $v \in V$. For each node v in the graph, the closest boundary nodes to v, denoted $C(v)$, along with the distance to v is precomputed where $C(v) \in B(p(v))$ and generally $|C(v)| < |B(p(v))|$. In Section 6.5.2 we explain why we do not require maintaining the distance to all boundary nodes. For a specific node v, any node $c \in C(v)$ and $c \in p(v)$, the minimum distance to reach any other partition P_i is $sdist(v, c) + 1$ via some other boundary node in partition P_i. This feature allows us to precompute and store a small subset of only the boundary nodes in the partition that v belongs to $B(p(v))$, instead of calculating distances to all the boundary nodes in other the partitions. In a fully-connected graph, once the closest boundary nodes for $p(v)$ is known, we can easily find the corresponding boundary nodes in the rest of the partitions. The node sets along with their social scores, denoted by $S(P_i, v)$ for each partition p_i, specific to a node v, act as the closest entry point to reach each of the partitions from v. $S(p(v), v)$ is calculated via a BFS from node v.

Example 6.3. *Distances to boundary nodes pre-computation.* For query node Q in the example Figure 6.2, $P(Q) = P_C$. The boundary nodes $B(P_C) = \{p, h, s, t\}$. In this simplified example, $C(Q) = B(P(Q))$. The closest boundary nodes with their distances are {p:1, h:2, s:4, t:4}. From this we can derive the closest entry points to reach each of the partitions: $P_A = \{a : 2, k : 3, z : 5\}$ and $P_B = \{r : 5\}$ denoted by $S(P_A, Q)$ and $S(P_B, Q)$ respectively.

6.5.2 Query Processing algorithm

Algorithm for PART_TA is outlined in Algorithm 6. The intuitive idea is to run a variation of the threshold algorithm locally on each partition until the global top-k results are found. A priority queue is used to keep track of the results. An element in the queue can either represent a score (f) for a partition (*type*=‘partition’) or represent a score for a user (*type*=‘user’) in the network.

Algorithm 6 PART_TA

```
Input: query user $q.u$, keyword $q.w$, requested number of users $k$,
    $S(P_i, q.u)$ ranked social list, $T(P_i, q.w)$ ranked text list
Output: Result set $R$ containing the top-$k$ users
 1: R ← {}
 2: Queue ← getNewPriorityQueue()          ▷ type (‘partition’ or ‘user’)
 3: /* For each partition queue top-1 partition score */
 4: for partition $p_i \in P$ do
 5:     $f$ ← TRAVERSE($p_i$, 1)            ▷ $f$ is the max. score for $p_i$
 6:     Queue.Enqueue($p_i, f$)                  ▷ type=‘partition’
 7: end for
 8: /* Processing the Queue */
 9: while |R| ≤ $k$ do
10:     element ← Queue.Dequeue()
11:     if type(element) == ‘partition’ then
12:         /* expand partition */
13:         $x$ ← $k$−|R|              ▷ Remaining no. of elements to find
14:         localQueue ← TRAVERSE(id(element), $x$)      ▷ partition id
15:         Queue ← Queue ∪ localQueue               ▷ add $x$ elements
16:     else
17:         R ← R ∪ {id(element)}
18:     end if
19: end while
20: Return R
```

The algorithm starts by enqueueing partition elements with f scores representing the top-1 scores for each partition. The top-1 scores are calculated for each partition by running the threshold algorithm locally (line 4-7 in Algorithm 6). The TRAVERSE function (Algorithm 7) runs a modified threshold algorithm locally within the partition to find the top-x elements. TRAVERSE requires two sorted lists of the social scores and text scores. $T(P_i, q.w)$ is a ranked postings list recored for partition P_i filtered by the query keyword $q.w$. $S(P_i, q.u)$ is a ranked list of social proximity from user $q.u$ to enter a partition P_i as described in Section 6.5.1. Each

Algorithm 7 TRAVERSE: Local Threshold algorithm

```
Input: partitionId $i$, $x$ no. of top elements to find, sorted social list
    $S(P_i, q.u)$, sorted text list $T(P_i, q.w)$
Output: Local Queue $LQ$, topScore $f$
1: function TRAVERSE(i, x)
2:     Run threshold algorithm until top-$x$ is found
3:     Return $LQ, f$
4: end function
```

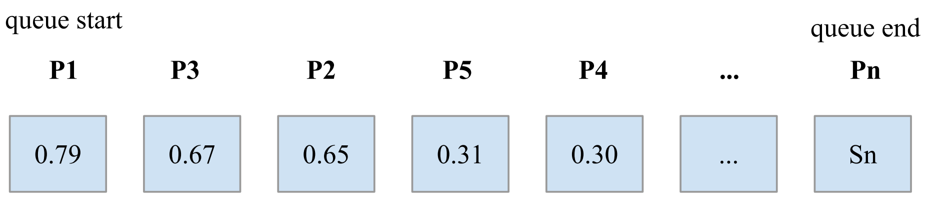


Figure 6.3: Initial Queue with $k = 10$, |R|={}

partition also records its local queue, and an iteration position. After this step, the initial queue will consist of maximum f scores of only partition elements as shown in Figure 6.3. This will guide the order in which the query will be processed. Once the top-1 positions have been found, the query is processed in this order. For example, in Figure 6.3, partition P1 will be expanded first since it has a maximum f score of 0.79. The first element is dequeued, and P1 partition is expanded (Line 11) first to find x elements where initially, $x = k$. TRAVERSE continues traversing the lists restarting from the position in which the calculation stopped to find top-1. The algorithm continues to expand partitions in the order they appear in the queue, until k users have been found.

The complexity of TA is upper bounded by the size of the postings list since, in the worst case, it will only run for n iterations, where $n = |T(q.w)|$. Assuming a similar distribution of keywords among all partitions, a partition on average has a postings list of size $l = \frac{|T(w)|}{|P|}$. Using this heuristic, the size of the social list in a partition, $|(S(P_i, q.u))|$ or the subset of boundary nodes to be maintained ($|C(q.u)|$), can be upper bounded to l where w above is a non-stopword term with the highest frequency.

There are several optimisations within the kSTRQ algorithm to terminate early. Figure 6.4 shows an intermediate position of the queue where $k = 10$ and the top-3 elements have already been found. The next element in the queue to be dequeued is a node that represents the partition P2 with a top-1 score of 0.65. At this position, P2 will be expanded in the TRAVERSE function until either of the two following conditions are met.

1. Found top-x in the partition where $x = k - |R|$. In Figure 6.4, in the worst case, P2 should be expanded to find the top-7 elements.
2. The local threshold in the partition does not go lower than the score in the $x-$th position in the global queue, i.e. 0.16. The local TRAVERSE function can terminate if a score of an element goes below 0.16 as it is guaranteed that an element below that threshold cannot be part of the top-k elements in the global queue.

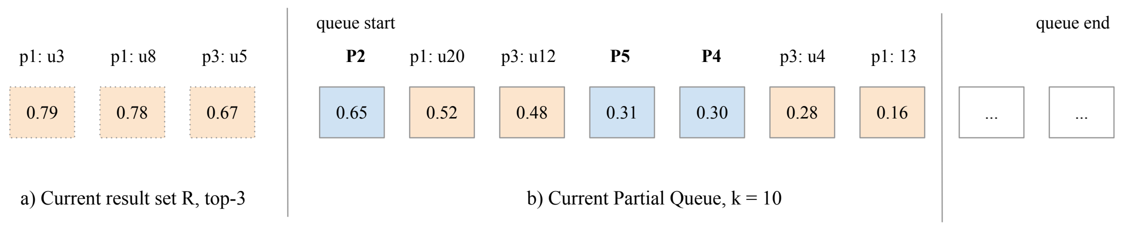


Figure 6.4: Intermediate Global Queue with $k = 10$. Partitions expanded so far are P1 and P3.

The local threshold algorithm can terminate with the local queue if one of those conditions is met. The result set |R| is then populated with the top-x elements in the local queue. The PART_TA algorithm guarantees that only the partitions that appear in the top-k result set will be expanded.

6.5.3 Graph partitioning strategy

There are several approaches to partition the graph. We perform a n-way graph partitioning algorithm using METIS [96] to decompose the graph. Graph partitioning divides the set of vertices V into n disjoint partitions $(P_1, ..., P_n)$ such that $P_i \cap P_j = \emptyset$ for $i \neq j$, $|P_i| \approx \frac{|V|}{n}$, and inter-edges, i.e., the edge cuts, are minimized. Partitioning algorithms such as METIS create partitions similar in size, enabling a balanced workload for each partition. The drawback of this approach is that n may not represent the natural clusters within the graph.

Alternatively, we can adopt community detection or clustering algorithms, but they are not guaranteed to generate equal sized partitions. Another improvement on the decomposition is to extend existing clustering approaches that take into account the homogeneity of attribute values along with the graph topology when deciding its clusters [219, 229]. This may yield a better final result (expanding even fewer partitions), as the graph is essentially partitioned in both dimensions, however comes at a much higher pre-processing cost. These are two trade-offs to be considered in selecting algorithms to decompose the graph. In this chapter, we resort to graph partitioning methods that are known to be efficient and effective in terms of reducing edge cuts on large graphs. Instead of arbitrarily selecting n, we choose n to be dependent on $|V|$, i.e. $n = log(|V|)$. As seen in other work [120, 82] the idea is that, to take advantage of a decomposition, we need to consider reductions that are an order of magnitude less than the original graph.

6.6 Experiments

We conducted experiments to demonstrate the performance of the proposed PART_TA algorithm over the baseline approaches. We first describe the experimental setup for our analysis. All algorithms were implemented in Java. The experiments were conducted on a Intel Core i7-4790K at 4.00GHz with 16GB RAM and 60GB SSD. The details of the datasets and graph database system used are given next.

6.6.1 Datasets

We used three real-world datasets (maintained by the AMiner project [6]) demonstrating different characteristics of the graph and text associated with the nodes. Table 6.4 shows a summary of each of the datasets considered.

- *Twitter* graph is created from 87K users with the "following" relationship among them. The 99,696,204 Tweets recorded for these users have been processed to attach hashtags as the text content of each user.
- *AMiner* is a co-authorship network representing the collaboration relationships of academic authors. Each user has a set of keywords describing his/her research interests.
- *Flickr* is a photo sharing network of users where the links represent friendship relationships. From this dataset, we extracted relationships of around 400k users. Flickr user nodes are given group names, representing the groups they belong to sharing common interests of photography such as Wildlife and Landscapes.

Since Twitter and Aminer graphs had many isolated components, we have preserved only users that belong to the largest connected component of each of these networks. Setting a

Table 6.4: Dataset Description

Dataset	Nodes	Edges	Avg/Max Degree	Diameter	No. of unique keywords	Description
Twitter	87,349	306,249	7.8 / 230	15	1.3M	who-follows-whom social network
AMiner	1,057,194	3,634,124	4.9 / 551	24	2.9M	Academic Co-authorship network
Flickr	424,169	8,475,790	39.6 / 11,930	8	340K	Friendship social network

default partitioning to be around log(|V|), Twitter, AMiner and Flickr have been partitioned into 6, 8, and 6 partitions respectively.

6.6.2 Graph Database System

We chose Neo4j as the graph database system to conduct our experiments for several reasons. Neo4j is optimized for graph traversals and is suitable for efficiently performing graph-based operations of the kSTRQ. Neo4j also allows manipulation of text via a Lucene index, to perform a full-text search on node properties. The combination of these features in Neo4j facilitates our kSTRQ query, making it an ideal test bed for our experiments. Lucene enables us to construct the inverted index with keywords as terms, node ids corresponding to the document ids, and we made use of the tf-idf based similarity measures provided by Lucene. The databases were created using Neo4j 3.1 Community Edition. For all experiments page cache in Neo4j was set to 4GB.

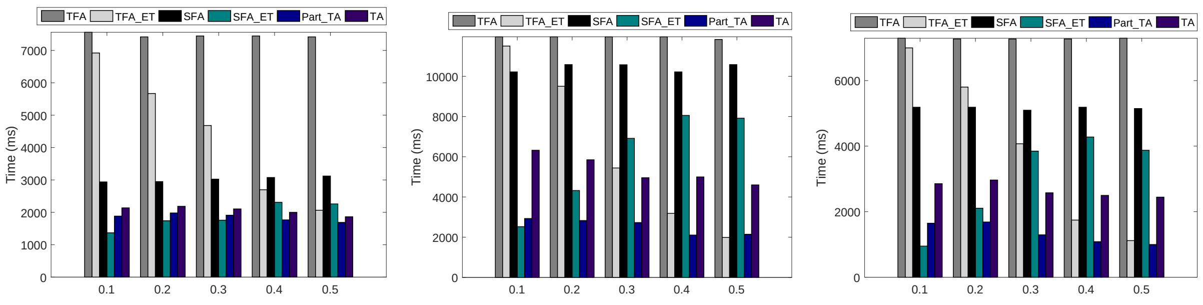
The partitions created from METIS have been modeled in the graph as a node attribute ranging from 1 to |P|. As PART_TA requires the inverted indexes to be partitioned; we simulate this behaviour in Neo4j by storing different Lucene indexes corresponding to each partition. The local similarity measures of each partition were configured to match the similarity of a global inverted index – otherwise, PART_TA operating on partitions would yield a different result set to the baseline algorithms.

Table 6.5: Parameter Variations and default values

Parameter	Range	Default
α	0.1, 0.2, 0.3, 0.4, 0.5	0.3
k	5, 10, 15, 20	10

6.6.3 Performance Evaluation

To generate a query, we randomly pick a user in the dataset and randomly pick a term out of the keywords attached to this user to be the query keyword. The reported timing measurements is an average timing of running 100 such queries. The proposed PART_TA algorithm is compared with the baselines, TFA, TFA_ET, SFA, SFA_ET and TA. We study the performance on different datasets under various parameter settings listed in Table 6.5. In particular, we investigate the effect of varying the preference parameter α (Equation 6.4), and the number of objects k returned from the query.



(a) Twitter **(b) AMiner** **(c) Flickr**

Figure 6.5: Effect of preference parameter α.

Varying α. Figure 6.5 shows the results of varying the preference parameter value α from 0.1 to 0.5; this setting is following existing work [47, 27] where preference is given to social proximity. Y-axis shows the average timing in milliseconds. A smaller value of α indicates that more preference is given to the social proximity of a user v to the query user u. Objects closer to the query user in graph distance become more eligible to be included in the final result set. In real social network applications, it is appropriate to retrieve and suggest socially close users rather than far away users who are ranked higher in text scores. In all the datasets, TFA and SFA approaches do not vary much, as these baselines require traversing their full respective textual and social lists, irrespective of the α value set. We retain their performance to observe the relative difference in their respective early termination variations.

Let us first discuss some general observations across the datasets. A larger value in α leads to better performance in the TFA_ET algorithm as the text relevance becomes more important: the termination threshold would be reached faster having to go through only a few iterations in the text list. Conversely, for SFA_ET, timing becomes worse with increasing α, as the weight on the social proximity is reduced, and thus, has to traverse more iterations in the social lists.

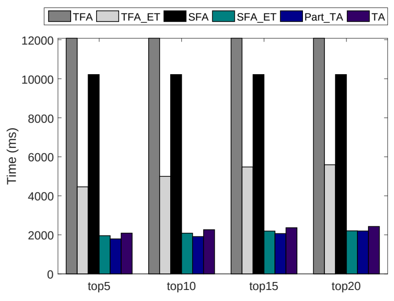
As we have noted in Section 6.4, a random unit operation in TFA_ET (i.e. shortest path calculation) is much more expensive than a random unit operation in SFA_ET (i.e. retrieving a text score in a list) which helps to explain the large timing difference between TFA_ET and SFA_ET in lower α values.

Although better than most baselines, for the smaller Twitter graph (Figure 6.5a), the timing between PART_TA and TA does not show a significant difference. Averaging across varying α values, the performance improvement of PART_TA compared to TA is 10.2%. SFA_ET performs better than PART_TA and TA for small α values operating on the graph with only 87K nodes. A possible reason could be that, due to the small size of the graph, it does not fully utilize the benefits of partitioning the graph. Comparing AMiner (Figure 6.5b) and Flickr (Figure 6.5c) graphs, the relative difference between TFA and SFA is larger for Flickr. This behaviour can be explained by characteristics of the datasets (Refer Table 6.4). Flickr is a more dense graph with a diameter of 8 and a much larger average degree which indicate that nodes are more clustered together, thus requiring less time to perform a full BFS (as in SFA) compared to AMiner.

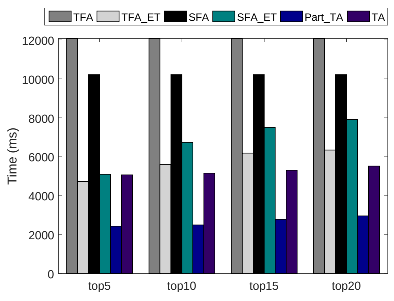
For larger graphs, PART_TA demonstrates better performance irrespective of the changing α. For the Aminer dataset, comparing with TA, the best performance is observed at $\alpha = 0.4$, with a performance improvement of 57.8% and an improvement of 52.2% on average. For the Flickr dataset, the best performance is observed at $\alpha = 0.5$, with a performance improvement of 59.1% and on average, an improvement of 50.1%. PART_TA performs even better compared to TFA- and SFA- based early termination variations (upto 76% improvement), with the exception of the two edge cases in AMiner. For the edge case of 0.1 we observe that SFA_ET performs better but gets worse as α is further increased. Similarly, at the other end, at 0.5, TFA_ET becomes a close contender to PART_TA. Again, it gets worse when α is decreased. This unstable behaviour of the early termination algorithms, makes them not suitable for the general case. The TA algorithm also seems robust to changing α, similar to PART_TA, however demonstrates worse timing as quantified above.

Varying k. Figure 6.6 shows the results of varying the number of output records k from 5 to 20. Y-axis shows the average timing in milliseconds. The TFA and SFA baselines are about the same as they are not sensitive to the value of k. As the value of k is increased, more processing and traversals are required to retrieve the final result set.

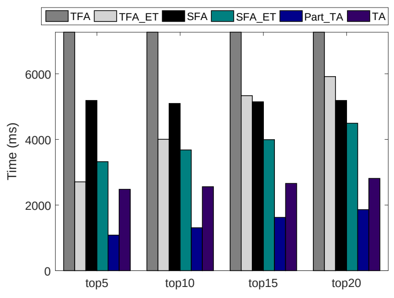
The Twitter graph (Figure 6.6a) does not show a significant increase in the methods as the costly shortest path calculation step can be efficiently executed on a smaller graph. In



(a) Twitter



(b) AMiner



(c) Flickr

Figure 6.6: Effect of preference parameter k.

all datasets, with increasing k, the rate of growth for TFA_ET and SFA_ET algorithms is higher than TA and PART_TA. An interesting behaviour of the larger AMiner (Figure 6.6b) and Flickr (Figure 6.6c) graphs, is that our proposed PART_TA approach does not vary much when k is increased. Our conjecture is that once a partition is expanded to search for x objects, an additional $x + \delta$ values would be found from a smaller, localised partition, whereas other algorithms deal with the whole graph making it a much larger search space to find the additional values.

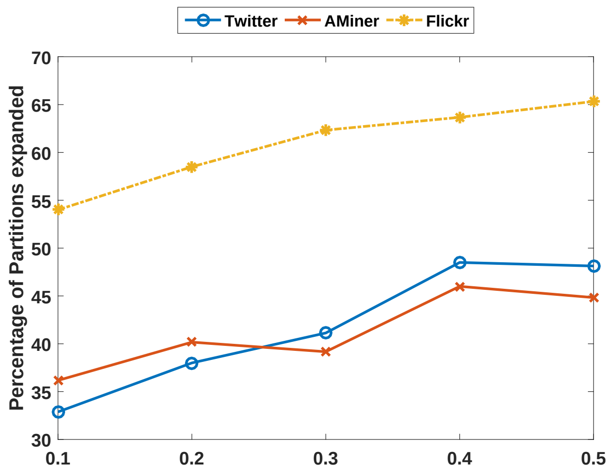


Figure 6.7: Percentage of partitions expanded in PART_TA with varied α.

Partitions expanded. For each of the datasets, we investigate the percentage of partitions expanded when α is increased, as shown in Figure 6.7. As the graphs are of different characteristics, we want to examine the effect the density of the graphs have on the percentage of expanded partitions. Smaller values of α indicate that more preference is given to the social proximity of a user v to the query user u. Since partitioning aims to co-locate a query user's neighbourhood within the same partition, socially close users would be found by expanding only a few partitions. As α is increased, more partitions need to be expanded to retrieve the query result, as socially distant users may also be eligible for the final result set, if their text is relevant. For the AMiner and Twitter graphs, the percentage of partitions expanded is increasing with α as expected, however it stabilises at around 48% when $\alpha = 0.5$. For the denser Flickr graph, the percentage of partitions expanded is much higher at first, however do not increase much with a growing α. For this experiment, we increased α until 0.9 to examine if the percentage of partitions expanded became closer to 100% which is not desired behaviour for PART_TA. At $\alpha = 0.9$, the percentage only increased to 66.3, 54.5 and 49.8 for the Flickr, Twitter and Aminer graphs respectively, which is acceptable, not expanding all partitions.

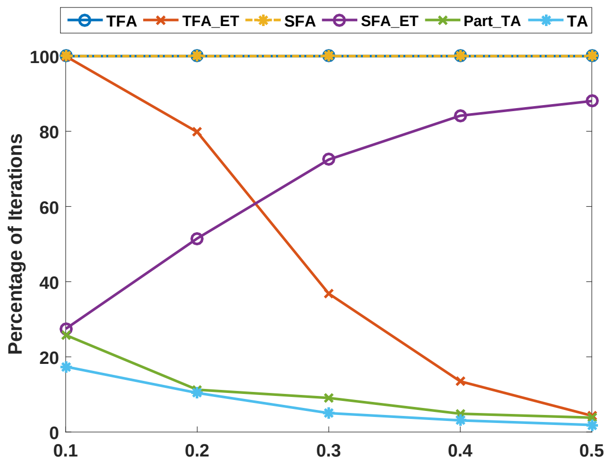


Figure 6.8: Percentage of iterations traversed (actual iterations / maximum possible iterations) for AMiner.

Percentage of iterations traversed. In Figure 6.8, for each of the algorithms we examine the number of iterations run as a fraction of the maximum possible iterations in the AMiner graph. For example, the maximum possible iterations for SFA and TFA algorithms is $|V|$ and $|V(q.w)|$ respectively. Since these algorithms are not optimized, they run $|V|$ and $|V(q.w)|$ iterations producing a percentage of 100% in the above figure. On the other hand, the early termination variations run only a fraction of $|V|$ and $|V(q.w)|$ for SFA_ET and TFA_ET respectively. Both TA and PART_TA, are executed as a fraction of $|V(q.w)|$ (Discussed in Section 6.4.3).

Figure 6.8 clearly demonstrates the reverse behaviour of TFA_ET and SFA_ET with increased α values. When α is small, SFA_ET can terminate by examining only about 27% of graph nodes while TFA_ET shows no improvement over the TFA algorithm. TFA_ET stands out only at higher α values, traversing only about 5% of the text lists to retrieve the top-k results. TA demonstrates slightly better results compared to PART_TA; this difference is acceptable considering that PART_TA only performs on local partitions.

6.6.4 Discussion

In our experiments we observe better performance and robustness of the PART_TA algorithm. We observe benefits especially on larger graphs, which are desired in many applications. The generated partitions are simulated on Neo4j to localise our computation. We believe that if graph database systems had support to manage the partitions physically in storage, we would reap more benefits of our approaches, dealing with much smaller, autonomous graphs for computation. However the platform should ideally not incur much overhead (i.e. communication costs) for processing parts of the graph spanning multiple partitions. The percentage of partitions expanded above is indicative of the fraction of partitions that would have to be processed in order to retrieve the final result set k.

On the aspect of text indexing, although the Lucene index has different strategies to partition and shard the index, it is not clear how these functionalities can be deployed via Neo4j. In future work, one can also investigate the effects of using different social and text relevance metrics and the effects of varying number of partitions has on the final result.

6.7 Summary

In this chapter we investigated the kSTRQ query that requires combined graph traversal and text search in a graph database system. The contributions of our work are as following.

- **Methodology:** We reviewed existing work related to our problem (Section 6.2) and have adopted algorithms (Section 6.4) that can be extended to a graph database setting. We are the first to conduct a detailed investigation into performing the kSTRQ in a graph database system.

- **New algorithms:** In Section 6.5 we introduce our algorithm PART_TA that partitions the graph and efficiently processes the kSTRQ. The algorithm operates on a smaller, localised graphs looking for a subset of results, expanding as few partitions as possible, until the users with the highest global ranking scores have been found.

- **Experiments on real graphs:** We conducted experiments on large real graphs (Section 6.6) having text associated to their nodes from different domains exhibiting diverse characteristics. We observed that our PART_TA algorithm did in fact lead to improved query performance over the baselines and it also demonstrated robust behaviour under changing parameters.

Chapter 7

Conclusions and Future Directions

This thesis focused on modeling, storage and query processing aspects of graph database management systems gaining insights on different large-scale graph application scenarios. Decades of research have contributed to the development and improvement of relational systems. We focus on the above important topics to gain more insights into the evolving genre of graph database systems. In this chapter we first summarise the contributions of this thesis and conclude with interesting research directions stemming from our work.

7.1 Summary of Contributions

In this thesis we modeled large-scale and complex applications to study how effectively graph database systems can support features relevant to them. For this purpose, we adopted two application scenarios constructing graphs which exhibit very different characteristics.

GDBMS for Microblogging Analytics. In our first study we investigated the use of GDBMS for Microblogging analytics representing a social network application setting. We conducted the first extensive review on data models and query systems in existing approaches for microblogging analyses. The requirement for graph-based data models were emphasized to answer novel and interesting queries. A graph-based data model was proposed for the Twittersphere which facilitate graph queries such as user recommendations, co-occurrence and influence finding. For our empirical analysis we chose two representative systems Neo4j and Sparksee. On a large Twitter dataset, we shared our experiences on different aspects of using these graph database systems including performance of data ingestion, expressiveness of query languages and efficiency of processing the above queries.

GDBMS for Evolving Software Code Dependencies. Dependencies in software code repositories can also be modeled as an attributed property graph depicting different types of dependencies among software entities. For our second large-scale graph application, an established project from industry, Frappé, was studied. This extracts the code dependencies from the most recent snapshot of a codebase and stores them in a graph database enabling advanced code comprehension queries. We investigated graph-based strategies to enable advanced code comprehension when the underlying codebase evolves over time. Unique challenges associated with versioned graph construction with multiple code revisions were addressed by leveraging efficient entity resolution strategies. We examined how well GDBMSs are able to model and query evolving graphs: any tool that models code evolution would benefit from our experience on building versioned graphs addressing the challenges associated with it. Evaluations were conducted on a very large codebase of around 13 million lines of code. On a versioned graph built on this codebase, we demonstrated how existing code comprehension queries can be efficiently processed and also showed the benefit of running queries across multiple versions.

Improving storage and disk I/O performance of GDBMS. Graph storage is an important factor affecting disk I/O performance and efficiency of query processing of a graph system. As such, the problem of effective graph storage was addressed in this study, for optimizing disk operations. This study introduced the novel edge-labeling problem on directed graphs, which aims to label both incoming and outgoing edges of a graph maximizing the 'edge-consecutiveness' metric. The edge-labeling problem has been formulated as a maximisation problem of the consecutiveness metric with the goal of optimally assigning edge labels to efficiently answer typical graph queries. We proposed two new edge labeling schemes, FLIPINOUT and GRDRANDOM and provided extensive experimental analyses on real-world graphs. Our methods resulted in significantly improved disk I/O performance by achieving a better layout and locality of edges on disk, leading to faster execution of neighbourhood-related queries. Based on FLIPINOUT, we also introduced FLIPCUT, an effective one-pass, neighbourhood-agnostic strategy for streaming graph partitioning.

Integrated processing of graph and keyword search on GDBMS. Although graph systems provide support for efficient graph-based queries, there have been no comprehensive studies on how other dimensions (such as text) stored with a graph can work well together on graph-based queries. In our final study, we addressed this problem of top-k social textual ranking queries (kSTRQ) in a graph database system setting. An algorithm Part-Ta was proposed, that can efficiently process graph and textual queries in combination. Graph partitioning strategies have been employed in our proposed approach to optimize and speed-up query processing along both dimensions. Our methods have been evaluated using real-world graph datasets that have text associated on the nodes and show significant benefits in our approaches compared to the baselines.

7.2 Future Research Directions

Armed with the observations we have of existing GDBMSs, we discuss some promising future research directions.

Optimisation of graph query languages. A set of graph query languages such as Cypher, Gremlin and SPARQL have been developed to query specialized graph structures. While a standardisation of graph query languages is a long-term goal of the graph database community, research efforts can explore optimisations of existing languages. As we have investigated a few declarative and imperative query systems, we observe there is potential in looking into cost-based optimizers for graph queries. Similar to SQL, it would be interesting to investigate how an optimizer can construct an efficient plan based on alternate traversal plans.

Query support for graph evolution. Although graph evolution has been studied in literature, it has not yet been adopted in graph database systems. This research direction is motivated by real-world graphs such as software code dependencies and social networks exhibiting inherent behaviour of evolution. Support for graph evolution can be two-fold, essentially recognising the temporal dimension. First, graph systems can focus on storage of evolving graphs e.g. LLAMA [125], with emphasis on the data layout by augmenting traditional graph representations. These approaches can also provide better support for the construction of versioned graphs. Secondly, graph systems can provide in-built functionality for query languages to facilitate versioning. It would be interesting to investigate how query languages can be developed or languages such as Cypher can be extended, to natively and efficiently support time-point and time-interval queries.

Edge labeling on dynamic graphs. Current research on node [41, 119] and edge labeling methods [69] for improved disk-locality focuses on offline ordering/labeling algorithms. This means that ordering of nodes or edges is performed as a pre-processing step and require a re-run of the algorithm when the graph structure is altered. Labeling methods that deal with dynamic graphs, conducted online, is an open and interesting research direction.

Seamless combination of multiple query dimensions. In this thesis we paid attention to the integration of keyword search and graph traversals within graph database systems. Support and integration of other dimensions (such as the spatial and temporal domain) remain largely unexplored. Specialised systems have been developed for performing graph queries with node/edge predicates [172, 175], systems for running spatial-social [139, 11] queries and augmenting existing distributed platforms with temporal capabilities [188, 111]. Consolidating these research efforts in a graph database system setting is challenging, but would be beneficial in wider adoption of graph database systems, focused on a myriad of queries on these dimensions.

Bibliography

[1] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980, 2008.

[2] F. Abel, C. Hauff, and G. Houben. Twitcident: fighting fire with information from social web streams. In *Proceedings of the 21st World Wide Web Conference, WWW*, pages 305–308, 2012.

[3] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, pages 237–248, 2014.

[4] Allegrograph: A semantic graph database. https://franz.com/agraph/allegrograph/.

[5] S. AmerYahia;, L. V. Lakshmanan;, and Cong Yu. SocialScope : Enabling information discovery on social content sites. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[6] AMiner: Search and Mining of academic social networks. https://aminer.org/data-sna.

[7] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, pages 235–244, 2009.

[8] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on*

Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS, pages 1–7, 2013.

[9] Apache. Apache lucene. http://lucene.apache.org, 2017.

[10] A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[11] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. *Proceedings of the VLDB Endowment*, 6(10):913–924, 2013.

[12] Aurelius. Titan: A distributed graph database, 2017. http://thinkaurelius.github.io/titan.

[13] B. Bahmani and A. Goel. Partitioned multi-indexing: Bringing order to social search. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 399–408, New York, NY, USA, 2012. ACM.

[14] T. Baldwin, P. Cook, and B. Han. A support platform for event detection using social intelligence. In *Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 69–72, 2012.

[15] L. Barbosa and J. Feng. Robust sentiment detection on Twitter from biased and noisy data. In *23rd International Conference on Computational Linguistics: Posters. Association for Computational Linguistics*, pages 36–44, aug 2010.

[16] M. S. Bernstein, B. Suh, L. Hong, J. Chen, S. Kairam, and E. H. Chi. Eddi: interactive topic-based browsing of social status streams. In *23nd annual ACM symposium on User interface software and technology - UIST*, pages 303–312, Oct. 2010.

[17] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering, ICDE*, pages 431–440, 2002.

[18] A. Bifet and E. Frank. Sentiment knowledge discovery in twitter streaming data. *Discovery Science. Springer Berlin Heidelberg*, pages 1–15, Oct. 2010.

[19] N. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory, 1736-1936*. Clarendon Press, New York, NY, USA, 1976.

[20] A. Black, C. Mascaro, M. Gallagher, and S. P. Goggins. Twitter Zombie: Architecture for capturing, socially transforming and analyzing the Twittersphere. In *International conference on Supporting group work*, pages 229–238, 2012.

[21] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. *J. Stat. Mech. Theor. Exp.*, 2008(10):P10008, 2008.

[22] M. Boanjak and E. Oliveira. TwitterEcho - A distributed focused crawler to support open research with twitter data. In *International conference companion on World Wide Web*, pages 1233–1239, 2012.

[23] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[24] K. Bontcheva and L. Derczynski. TwitIE: an open-source information extraction pipeline for microblog text. In *International Conference on Recent Advances in Natural Language Processing*, 2013.

[25] M. Broecheler. Titan: Big Graph data with Cassandra. https://www.datastax.com/dev/blog/boutique-graph-data-with-titan, 2012.

[26] C. Budak, T. Georgiou, and D. E. Abbadi. GeoScope: Online detection of geo-correlated information trends in social networks. *Proceedings of the VLDB Endowment*, 7(4):229–240, 2013.

[27] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1360–1369, Washington, DC, USA, 2012. IEEE Computer Society.

[28] C. Byun, H. Lee, Y. Kim, and K. K. Kim. Twitter data collecting tool with rule-based filtering and analysis module. *International Journal of Web Information Systems*, 9(3):184–203, 2013.

[29] OrientDB: Distributed Graph, Document, Multi-model Database. http://orientdb.com.

[30] J. J. Carrasco, D. C. Fain, K. J. Lang, and L. Zhukov. Clustering of bipartite advertiser-keyword graph. In *ICDM 03*, 2003.

[31] S. Carter, W. Weerkamp, and M. Tsagkias. Microblog language identification: Overcoming the limitations of short, unedited and idiomatic text. *Language Resources and Evaluation*, 47(1):195–215, June 2012.

[32] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch. Time-varying social networks in a graph database: a Neo4j use case. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES*, page 11, 2013.

[33] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM*, pages 10–17, 2010.

[34] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully Automatic Cross-associations. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 79–88, 2004.

[35] S. Chandra, L. Khan, and F. B. Muhaya. Estimating twitter user location using social interactions–a content based approach. In *IEEE Conference on Privacy, Security, Risk and Trust*, pages 838–843, Oct. 2011.

[36] C. Chen, F. Li, C. Ooi, and S. Wu. TI : An efficient indexing mechanism for real-time search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*, pages 649–660, 2011.

[37] X. Chen, C. Zhang, B. Ge, and W. Xiao. Temporal social network: Storage, indexing and query processing. In *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops*, 2016.

[38] Z. Cheng, J. Caverlee, K. Lee, and C. Science. A content-driven framework for geolocating microblog users. *ACM Transactions on Intelligent Systems and Technology*, 2012.

[39] M. Cheong and S. Ray. A literature review of recent microblogging developments. Technical report, Clayton School of Information Technology, Monash University, 2011.

[40] C. Chew and G. Eysenbach. Pandemics in the age of twitter: content analysis of tweets during the 2009 H1N1 outbreak. *PloS one*, 5(11), 2010.

[41] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 219–228, 2009.

[42] P. Z. Chinn, J. Chvatalova, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices - a survey. *Journal of Graph Theory*, 6(3):223–254, 1982.

[43] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[44] B. O. Connor, N. A. Smith, and E. P. Xing. A latent variable model for geographic lexical variation. In *Conference on Empirical Methods in Natural Language Processing*, pages 1277–1287, 2010.

[45] M. Conover, J. Ratkiewicz, M. R. Francisco, B. Gonçalves, F. Menczer, and A. Flammini. Political polarization on twitter. In *Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, July 17-21, 2011*, 2011.

[46] Cscope home page. http://cscope.sourceforge.net/.

[47] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. B. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[48] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. *Analysing Software Repositories to Understand Software Evolution*, pages 37–67. 2008.

[49] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 401–410, New York, NY, USA, 2010. ACM.

[50] J. David. That's what friends are for: inferring location in online social media platforms based on social relationships. In *Proceedings of the Seventh International Conference on Weblogs and Social Media, ICWSM*, 2013.

[51] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1—the famoos information exchange model, 2001.

[52] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544, 2016.

[53] Y. Doytsher and B. Galon. Querying geo-social data by bridging spatial networks and social networks. In *2nd ACM SIGSPATIAL International Workshop on Location Based Social Networks*, pages 39–46, 2010.

[54] A. Dries, S. Nijssen, and L. De Raedt. A query language for analyzing networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009*, pages 485–494, 2009.

[55] M. Efron. Hashtag retrieval in a microblogging environment. In *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR*, pages 787–788, 2010.

[56] S. Elbassuoni and R. Blanco. Keyword search over rdf graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 237–242, New York, NY, USA, 2011. ACM.

[57] FaceBook Query Language(FQL) overview. https://developers.facebook.com/docs/technical-guides/fql.

[58] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[59] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE*, pages 656–665, 2008.

[60] P. Ferragina, F. Piccinno, and R. Venturini. Compressed indexes for string searching in labeled graphs. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 322–332, 2015.

[61] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 175–181, 1982.

[62] Filament: Graph management toolkit. http://filament.sourceforge.net/index.html.

[63] S. Frénot and S. Grumbach. An in-browser microblog ranking engine. In *International conference on Advances in Conceptual Modeling*, volume 7518, pages 78–88, 2012.

[64] J. Gehweiler and H. Meyerhenke. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *24th IEEE International Symposium on Parallel and Distributed Processing,IPDPS*, pages 1–8. IEEE, 2010.

[65] Apache giraph: Iterative graph processing system. http://giraph.apache.org.

[66] G. Golovchinsky and M. Efron. Making sense of Twitter search. In *Proc. CHI2010 Workshop on Microblogging: What and How Can We Learn From It?*, 2010.

[67] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[68] Kythe. http://www.kythe.io/docs/kythe-overview.html.

[69] O. Goonetilleke, D. Koutra, T. Sellis, and K. Liao. Edge labeling schemes for graph data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, pages 12:1–12:12, 2017.

[70] O. Goonetilleke, T. Sellis, X. Zhang, and S. Sathe. Twitter analytics: A big data management perspective. *SIGKDD Explorations*, 16(1):11–20, 2014.

[71] M. Graham, S. A. Hale, and D. Gaffney. Where in the world are you? geolocation and language identification in twitter. *CoRR*, abs/1308.0683, 2013.

[72] GraphDB. http://graphdb.ontotext.com.

[73] Gremlin Query Language. http://s3.thinkaurelius.com/docs/titan/0.5.4/gremlin.html.

[74] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over xml documents. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 16–27, New York, NY, USA, 2003. ACM.

[75] E. Hajiyev, M. Verbaere, and O. de Moor. *codeQuest:* scalable source code queries with datalog. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7*, pages 2–27, 2006.

[76] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM.

[77] N. Hawes, B. Barham, and C. Cifuentes. Frappé: Querying the linux kernel dependency graph. In *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES*, pages 4:1–4:6, 2015.

[78] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 305–316, 2007.

[79] B. Hecht, L. Hong, B. Suh, and E. Chi. Tweets from Justin Bieber's heart: the dynamics of the location field in user profiles. In *Conference on Human Factors in Computing Systems*, pages 237–246, 2011.

[80] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 234–245, 1990.

[81] W. Huo and V. J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 38:1–38:4, 2014.

[82] T. İnkaya. A parameter-free similarity graph for spectral clustering. *Expert Syst. Appl.*, 42(24):9489–9498, Dec. 2015.

[83] B. Iordanov. Hypergraphdb: A generalized graph database. In *Web-Age Information Management - WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010, Revised Selected Papers*, pages 25–36, 2010.

[84] B. J. Jansen, M. Zhang, K. Sobel, and A. Chowdury. Twitter power: Tweets as electronic word of mouth. *Journal of the American Society for Information Science and Technology*, 60(11):2169–2188, Nov. 2009.

[85] J. Jiang, L. Hidayah, T. Elsayed, and H. Ramadan. BEST of KAUST at TREC-2011 : Building effective search in Twitter. *TREC*, 2011.

[86] M. Jiang, A. W. Fu, and R. C. Wong. Exact top-k nearest keyword search in large networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 393–404, 2015.

[87] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 595–608, 2008.

[88] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *Social-Com*, pages 708–715. IEEE, 2013.

[89] P. Jürgens, A. Jungherr, and H. Schoen. Small worlds with a difference: new gatekeepers and the filtering of political information on Twitter. In *International Web Science Conference-WebSci*, pages 1–5, June 2011.

[90] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 505–516, 2005.

[91] U. Kang, D. H. Chau, and C. Faloutsos. Managing and mining large graphs : Systems and implementations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, volume 1, pages 589–592, 2012.

[92] U. Kang and C. Faloutsos. Big graph mining : Algorithms and discoveries. *SIGKDD Explorations*, 14(2):29–36, 2013.

[93] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos. GBASE: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference*

on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011, pages 1091–1099, 2011.

[94] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: An efficient analysis platform for large graphs. *Proceedings of the VLDB Endowment*, 21(5):637–650, June 2012.

[95] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining*, ICDM, pages 229–238, 2009.

[96] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[97] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 779–790, 2004.

[98] A. Kellens, C. D. Roover, C. Noguera, R. Stevens, and V. Jonckers. Reasoning over the evolution of source code using quantified regular path expressions. In *18th Working Conference on Reverse Engineering, WCRE*, pages 389–393, 2011.

[99] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49, 1970.

[100] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. *Proceedings of the VLDB Endowment*, 6(3):181–192, 2013.

[101] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *29th IEEE International Conference on Data Engineering, ICDE*, pages 997–1008, 2013.

[102] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT*, pages 65–76, 2016.

[103] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *26th IEEE/ACM International Conference on Automated Software Engineering ASE*, pages 376–379, 2011.

[104] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Intl. Working Conf. on Source Code Analysis and Manipulation*, pages 168–177, 2009.

[105] G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. *CoRR*, abs/1302.5549, 2013.

[106] A. Kosmatopoulos, K. Giannakopoulou, A. N. Papadopoulos, and K. Tsichlas. An overview of methods for handling evolving graph sequences. In *ALGOCLOUD Workshop*, pages 181–192, 2015.

[107] J. Krinke. Identifying similar code with program dependence graphs. In *8th Working Conf. on Reverse Engineering*, pages 301–309, 2001.

[108] S. Kumar, G. Barbier, M. Abbasi, and H. Liu. TweetTracker: An analysis tool for humanitarian and disaster relief. In *Proceedings of the Fifth International Conference on Weblogs and Social Media ICWSM*, pages 661–662, 2011.

[109] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010*, pages 591–600, 2010.

[110] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46, 2012.

[111] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. Hwang, and W. Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4):479–514, 2015.

[112] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 76(12):1177 – 1193, 2011. Special Issue on Software Evolution, Adaptability and Variability.

[113] C.-H. Lee, H.-C. Yang, T.-F. Chien, and W.-S. Wen. A novel approach for event detection by mining spatio-temporal information on microblogs. In *International Conference on Advances in Social Networks Analysis and Mining*, pages 254–259, July 2011.

[114] J. Leskovec and R. Sosic. SNAP: A general-purpose network analysis and graph-mining library. *ACM TIST*, 8(1):1:1–1:20, 2016.

[115] C. Li, J. Weng, Q. He, Y. Yao, and A. Datta. TwiNER: named entity recognition in targeted twitter stream. In *The 35th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR*, pages 721–730, 2012.

[116] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pages 1023–1031, 2012.

[117] Y. Li, Z. Bao, G. Li, and K. Tan. Real time personalized search on social networks. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 639–650, 2015.

[118] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. L. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[119] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Trans. Knowl. Data Eng.*, pages 3077–3089, 2014.

[120] J. Liu, C. Wang, M. Danilevsky, and J. Han. Large-scale spectral clustering on graphs. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1486–1492. AAAI Press, 2013.

[121] Y. Liu, A. Dighe, T. Safavi, and D. Koutra. A graph summarization: A survey. *CoRR*, abs/1612.04883, 2016.

[122] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[123] H. Ma, J. Wei, W. Qian, C. Yu, F. Xia, and A. Zhou. On benchmarking online social media analytical queries. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS*, pages 1–7, 2013.

[124] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: efficient graph analytics using large multiversioned arrays. In *31st IEEE International Conference on Data Engineering, ICDE*, pages 363–374, 2015.

[125] P. Macko, D. Margo, and M. Seltzer. Performance introspection of graph databases. In *6th Annual International Systems and Storage Conference, SYSTOR*, pages 1–10, 2013.

[126] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[127] A. Marcus, M. Bernstein, and O. Badar. Tweets as data: demonstration of TweeQL and Twitinfo. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 1259–1261, 2011.

[128] A. Marcus, M. Bernstein, and O. Badar. Processing and visualizing the data in tweets. *SIGMOD Record*, 40(4), 2012.

[129] M. S. Martín and C. Gutierrez. Representing, querying and transforming social networks with RDF/SPARQL. In *European Semantic Web Conference*, pages 293–307, 2009.

[130] M. S. Martín, C. Gutiérrez, and P. T. Wood. SNQL: A social networks query and transformation language. In *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011*, 2011.

[131] N. Martínez-Bazan, M. A. Águila Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J.-L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering and Applications Sysmposium*, pages 110–119, 2012.

[132] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, 2014.

[133] M. McGlohon, L. Akoglu, and C. Faloutsos. Statistical properties of social networks. In *Social Network Data Analytics*, pages 17–42. 2011.

[134] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[135] P. Mendes, A. Passant, and P. Kapanipathi. Twarql: tapping into the wisdom of the crowd. In *Proceedings of the 6th International Conference on Semantic Systems*, pages 3–5, 2010.

[136] M. F. Mokbel and W. G. Aref. Space-filling curves. In *Encyclopedia of Database Systems*, pages 2674–2675. 2009.

[137] T. Molderez, R. Stevens, and C. De Roover. Mining change histories for unknown systematic edits. In *proc. of the Intl. Conference on Mining Software Repositories*, pages 248–256, 2017.

[138] F. Morstatter, S. Kumar, H. Liu, and R. Maciejewski. Understanding Twitter data with TweetXplorer. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pages 1482–1485, 2013.

[139] K. Mouratidis, J. Li, Y. Tang, and N. Mamoulis. Joint search by social and spatial proximity. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1578–1579, 2016.

[140] Neo4j graph database. https://neo4j.com/product/.

[141] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[142] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *Proceedings of the VLDB Endowment*, 19(1):91–113, Feb. 2010.

[143] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[144] J. Nishimura and J. Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1106–1114, 2013.

[145] P. Noordhuis, M. Heijkoop, and A. Lazovik. Mining Twitter in the cloud: A case study. In *IEEE 3rd International Conference on Cloud Computing*, pages 107–114, July 2010.

[146] Infinitegraph: Distributed graph database. http://www.objectivity.com/products/infinitegraph/.

[147] A. Olteanu, C. Castillo, F. Diaz, and E. Kiciman. Social data: Biases, methodological pitfalls, and ethical boundaries. *SSRN Electronic Journal*, 2016.

[148] A wicked fast source browser. https://opengrok.github.io/OpenGrok/.

[149] I. Ounis, C. Macdonald, J. Lin, and I. Soboroff. Overview of the TREC-2011 Microblog Track. In *20th Text REtrieval Conference (TREC)*, 2011.

[150] A. Pak and P. Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *International Conference on Language Resources and Evaluation*, pages 1320–1326, 2010.

[151] S. Papadimitriou, J. Sun, C. Faloutsos, and P. S. Yu. Hierarchical, Parameter-Free Community Discovery. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2008.

[152] M. J. Paul and M. Dredze. You are what you tweet: Analyzing twitter for public health. In *Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, July 17-21, 2011*, 2011.

[153] S. Paul and A. Prakash. Querying source code using an algebraic query language. In *Proceedings of the International Conference on Software Maintenance, ICSM*, pages 127–136, 1994.

[154] V. Plachouras and Y. Stavrakas. Querying term associations and their temporal evolution in social data. In *International VLDB Workshop on Online Social Systems*, 2012.

[155] V. Plachouras, Y. Stavrakas, and A. Andreou. Assessing the coverage of data collection campaigns on Twitter: A case study. In *On the Move to Meaningful Internet Systems: OTM 2013 Workshops*, pages 598–607. 2013.

[156] D. Preotiuc-Pietro, S. Samangooei, and T. Cohn. Trendminer : An architecture for real time analysis of social media text. In *Workshop on Real-Time Analysis and Mining of Social Streams*, pages 4–7, 2012.

[157] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF W3C recommendation. W3C, URL= https://www.w3.org/TR/rdf-sparql-query/,, 2008.

[158] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *Proceedings of the VLDB Endowment*, 6(10):901–912, Aug. 2013.

[159] ioQuake3 codebase. https://github.com/ioquake/ioq3.

[160] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *20th International Conference on Software Maintenance ICSM*, pages 188–197, 2004.

[161] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *, 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 51–60. IEEE, 2013.

[162] K. H. Randall, R. Stata, J. L. Wiener, and R. G. Wickremesinghe. The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*, DCC '02, 2002.

[163] L. Ratinov and D. Roth. Design challenges and misconceptions in named entity recognition. In *Conference on Computational Natural Language Learning (CoNLL)*, number June, pages 147–155, 2009.

[164] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[165] A. Ritter, S. Clark, and O. Etzioni. Named entity recognition in tweets : an experimental study. In *Conference on Empirical Methods in Natural Language Processing*, pages 1524–1534, 2011.

[166] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2 edition, 2015.

[167] M. A. Rodriguez. Problem-solving using graph traversals. AT&T Technical Talk - Glendale, California, 2010.

[168] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *Proceedings of the 25th International Conference on Data Engineering, ICDE*, pages 1595–1602, Mar. 2009.

[169] C. D. Roover, C. Scholliers, V. Jonckers, J. Pérez, A. Murgia, and S. Demeyer. Implementation of the CHA-Q meta-model: A comprehensive change-centric software representation. *ECEASST*, 65, 2014.

[170] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[171] T. Sakaki. Earthquake shakes twitter users : Real-time event detection by social sensors. In *Proceedings of the 19th International Conference on World Wide Web, WWW*, pages 851–860, 2010.

[172] S. Sakr, S. Elnikety, and Y. He. G-sparql: A hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 335–344, New York, NY, USA, 2012. ACM.

[173] S. Salihoglu and J. Widom. GPS : A graph processing system. In *International Conference on Scientific and Statistical Database ManagementSSDBM*, pages 1–31, 2013.

[174] S. Salihoglu and J. Widom. GPS: a graph processing system. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 22:1–22:12, 2013.

[175] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proceedings of the VLDB Endowment*, 6(14):1918–1929, Sept. 2013.

[176] A. Schulz, A. Hadjakos, and H. Paulheim. A multi-indicator approach for geolocalization of tweets. In *Proceedings of the Seventh International Conference on Weblogs and Social Media, ICWSM*, pages 573–582, 2013.

[177] K. Semertzidis and E. Pitoura. Time traveling in graphs using a graph database. In *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops*, 2016.

[178] K. Semertzidis, E. Pitoura, and K. Lillis. Timereach: Historical reachability queries on evolving graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT*, pages 121–132, 2015.

[179] D. Serrano, E. Stroulia, D. Barbosa, and V. Guana. SociQL: A query language for the social Web. In E. Kranakis, editor, *Advances in Network Analysis and its Applications*, chapter 17, pages 381–406. 2013.

[180] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference, DCC*, pages 403–412, 2015.

[181] A. Signorini, A. M. Segre, and P. M. Polgreen. The use of Twitter to track levels of disease activity and public concern in the U.S. during the influenza A H1N1 pandemic. *PloS one*, 6(5), Jan. 2011.

[182] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[183] V. Singh, R. Gupta, and I. Neamtiu. MG++: memory graphs for analyzing dynamic data structures. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER*, pages 291–300, 2015.

[184] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012.

[185] Y. Stavrakas and V. Plachouras. A platform for supporting data analytics on twitter challenges and objectives. *Intl. Workshop on Knowledge Extraction & Consolidation from Social Media*, (Ict 270239), 2013.

[186] D. Steidl, B. Hummel, and E. Juergens. Incremental origin analysis of source code files. In *11th Working Conference on Mining Software Repositories, MSR*, pages 42–51, 2014.

[187] M. Steinbauer and G. Anderst-Kotsis. Dynamograph: A distributed system for large-scale, temporal graph processing, its implementation and first observations. In *Proceedings of the 25th International Conference on World Wide Web, WWW, Companion Volume*, pages 861–866, 2016.

[188] M. Steinbauer and G. Anderst-Kotsis. Dynamograph: extending the pregel paradigm for large-scale temporal graph processing. *IJGUC*, 7(2):141–151, 2016.

[189] R. Stevens and C. D. Roover. Querying the history of software projects using QWALKEKO. In *30th IEEE International Conference on Software Maintenance and Evolution*, pages 585–588, 2014.

[190] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901. ACM, 2015.

[191] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[192] A. Tonon, G. Demartini, and P. Cudré-Mauroux. Combining inverted indices and structured search for ad-hoc object retrieval. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 125–134, New York, NY, USA, 2012. ACM.

[193] TrendsMap, Realtime local twitter trends. http://trendsmap.com/.

[194] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 845–856, 2007.

[195] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014.

[196] A. Tumasjan, T. O. Sprenger, P. G. Sandner, and I. M. Welpe. Predicting elections with twitter: What 140 characters reveal about political sentiment. In *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010*, pages 178–185, 2010.

[197] Twitalyzer: Serious analytics for social business. http://twitalyzer.com.

[198] R.-G. Urma and A. Mycroft. Source-code queries with graph databases-with application to programming language usage and evolution. *Sci. Comput. Program.*, 97(P1):127–134, 2015.

[199] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[200] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Annual Southeast Regional Conference*, ACM SE '10, pages 1–6, 2010.

[201] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. de Castro Reis, and B. A. Ribeiro-Neto. Efficient search ranking in social networks. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 563–572, 2007.

[202] Open link software. http://vos.openlinksw.com/owiki/wiki/VOS/.

[203] Inference in the semantic web. https://www.w3.org/standards/semanticweb/inference.

[204] H. Wang and C. C. Aggarwal. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*, pages 249–273. 2010.

[205] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *IEEE 30th International Conference on Data Engineering, ICDE*, 2014.

[206] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[207] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[208] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[209] J. Weng, E.-p. Lim, and J. Jiang. TwitterRank : Finding topic-sensitive influential twitterers. In *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM*, pages 261–270, 2010.

[210] J. S. White, J. N. Matthews, and J. L. Stacy. Coalmine: an experience in building a system for social media analytics. In I. V. Ternovskiy and P. Chin, editors, *Proceedings of SPIE*, volume 8408, 2012.

[211] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, Apr. 2012.

[212] S. Wu, J. M. Hofman, W. A. Mason, and D. J. Watts. Who says what to whom on twitter. In *Proceedings of the 20th International Conference on World Wide Web, WWW*, pages 705–714, Mar. 2011.

[213] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *CoRR*, abs/1402.2394, 2014.

[214] Z. Xing and E. Stroulia. Differencing logical uml models. *Automated Software Eng.*, 14(2):215–259, 2007.

[215] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[216] Yahoo! Query Language guide on YDN. https://developer.yahoo.com/yql/.

[217] X. Yan, P. S. Yu, and J. Han. Graph indexing : A frequent structure-based approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 335–346, 2004.

[218] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 766–777, New York, NY, USA, 2005. ACM.

[219] J. Yang, J. J. McAuley, and J. Leskovec. Community detection in networks with node attributes. *CoRR*, abs/1401.7267, 2014.

[220] W. Yang. Identifying syntactic differences between two programs. *Softw., Pract. Exper.*, 21(7):739–755, 1991.

[221] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *Proceedings of the VLDB Endowment*, 21(4):509–534, 2012.

[222] J. Yin, S. Karimi, B. Robinson, and M. Cameron. ESA: emergency situation awareness via microbloggers. In *21st ACM International Conference on Information and Knowledge Management,CIKM 12*, pages 2701–2703, 2012.

[223] P. Yin, W. Lee, and K. C. K. Lee. On top-k social web search. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30,2010*, pages 1313–1316, 2010.

[224] A.-J. N. Yzelman and R. H. Bisseling. *A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve*, pages 627–633. Springer Berlin Heidelberg, 2012.

[225] A. N. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):116–125, 2014.

[226] S. Zhang, S. Li, and J. Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT, pages 192–203, New York, NY, USA, 2009. ACM.

[227] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. P. Amarasinghe. Optimizing cache performance for graph analytics. *CoRR*, abs/1608.01362, 2016.

[228] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1):340–351, 2010.

[229] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proceedings of the VLDB Endowment*, 2(1):718–729, 2009.

Appendix A

List of Abbreviations

Abbreviation	**Description**
ACID	Atomicity, Consistency, Isolation, Durability
API	Application programming interface
AST	Abstract syntax tree
BSP	Bulk synchronous parallel
CSR	Compressed Sparse Row
GDBMS	Graph Database Management Systems
IDE	Integrated Development Environment
IR	Information retrieval
JSON	JavaScript Object Notation
JVM	Java Virtual machine
KLOC	Kilo lines of code
MLDM	Machine learning data mining
MVCC	Multi-version concurrency control

Abbreviation	Description
NER	Named entity recognizers
NLP	Natural language processing
NoSQL	Not only SQL
OIC	Oracle internal codebase
OLAP	Online analytical processing
OLTP	Online transactional processing
PDG	Program dependency graph
POS	Part-of-speech
RDF	Resource description frameworks
SCM	Software Configuration Management
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured query language
URI	Uniform resource identifier