GRAPH PROCESSING IN MAIN-MEMORY COLUMN STORES

Zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.–Ing.)

der Fakultät für Informatik
der Technischen Universität Dresden

vorgelegte

Dissertation

von

MARCUS PARADIES

geboren am 10. Februar 1987 in Ilmenau.

Tag der mündlichen Prüfung:	03. Februar 2017
Erster Gutachter:	Prof. Dr.–Ing. Wolfgang Lehner
Zweiter Gutachter:	Prof. Dr.–Ing. Thomas Neumann

Dedicated to my grandfather.
1936 – 2012

ABSTRACT

Evermore, novel and traditional business applications leverage the advantages of a graph data model, such as the offered schema flexibility and an explicit representation of relationships between entities. As a consequence, companies are confronted with the challenge of storing, manipulating, and querying terabytes of graph data for enterprise-critical applications. Although these business applications operate on graph-structured data, they still require direct access to the relational data and typically rely on an RDBMS to keep a single source of truth and access.

Existing solutions performing graph operations on business-critical data either use a combination of SQL and application logic or employ a graph data management system. For the first approach, relying solely on SQL results in poor execution performance caused by the functional mismatch between typical graph operations and the relational algebra. To the worse, graph algorithms expose a tremendous variety in structure and functionality caused by their often domain-specific implementations and therefore can be hardly integrated into a database management system other than with custom coding. Since the majority of these enterprise-critical applications exclusively run on relational DBMSs, employing a specialized system for storing and processing graph data is typically not sensible. Besides the maintenance overhead for keeping the systems in sync, combining graph and relational operations is hard to realize as it requires data transfer across system boundaries.

A basic ingredient of graph queries and algorithms are traversal operations and are a fundamental component of any database management system that aims at storing, manipulating, and querying graph data. Well-established graph traversal algorithms are standalone implementations relying on optimized data structures. The integration of graph traversals as an operator into a database management system requires a tight integration into the existing database environment and a development of new components, such as a graph topology-aware optimizer and accompanying graph statistics, graph-specific secondary index structures to speedup traversals, and an accompanying graph query language.

In this thesis, we introduce and describe GRAPHITE, a hybrid graph-relational data management system. GRAPHITE is a performance-oriented graph data management system as part of an RDBMS allowing to seamlessly combine processing of graph data with relational data in the same system. We propose a columnar storage representation for graph data to leverage the already existing and mature data management and query processing infrastructure of relational database management systems. At the core of GRAPHITE we propose an execution engine solely based on set operations and graph traversals. Our design is driven by the observation that different graph topologies expose different algorithmic requirements to the design of a graph traversal operator. We derive two graph traversal implementations targeting the most common graph topologies and demonstrate how graph-specific statistics can be leveraged to select the optimal physical traversal operator. To accelerate graph traversals, we devise a set of graph-specific, updateable secondary index structures to improve the performance of vertex neighborhood expansion. Finally, we introduce a domain-specific language with an intuitive programming model to extend graph traversals with custom application logic at runtime. We use the LLVM compiler framework to generate efficient code that tightly integrates the user-specified application logic with our highly optimized built-in graph traversal operators.

Our experimental evaluation shows that GRAPHITE can outperform native graph management systems by several orders of magnitude while providing all the features of an RDBMS, such as transaction support, backup and recovery, security and user management, effectively providing a promising alternative to specialized graph management systems that lack many of these features and require expensive data replication and maintenance processes.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor *Wolfgang Lehner* for his unconditional support throughout the course of the entire PhD project. It was Wolfgang, who sparked my interest in database research during my undergraduate internship at SAP in 2009. Ever since then, Wolfgang has been not only a great technical mentor but also a great motivator and a true source of inspiration.

I would like to express my deepest gratitude to my advisor during the first part of my PhD project at SAP, *Christof Bornhövd*, who perfectly complemented Wolfgang from the production side and gave valuable feedback on applicability and usefulness of my research.

A special thanks goes to *Hannes Voigt*, who co-advised me during the second part of my PhD project, for many insightful discussions on query language design, for proof-reading the thesis document, and finally for introducing me to the art and enjoyment of craft beer.

Besides my advisors, I would like to thank *Prof. Dr. Thomas Neumann* for serving as a reviewer on my PhD committee and for the inspiring discussions and the valuable tips during my visit in Munich.

I thank my PhD fellow *Michael Rudolf*, who started with me the journey of pursuing a doctorate degree on the graph processing project. Although we are quite different in character and in the way we approach challenges and finally find solutions, I always felt inspired by his ideas and I am very thankful for the fruitful discussions we had during the last five years—a real *M+M* team!

I am deeply thankful to my fellow PhD colleagues at the SAP HANA CAMPUS who made my PhD ride a pleasant and not a lonely one. In particular I would like to thank my former office mates *Michael Rudolf*, *Robert Brunel*, *David Kernert*, and *Iraklis Psaroudakis* for accepting me as being terribly grumpy in the mornings, but also my other colleagues *Philipp Große*, *Hannes Rauhe*, *Ingo Müller*, *Jonathan Dees*, *Ismail Oukid*, *Thomas Bach*, *Florian Wolf*, *Matthias Hauck*, *Martin Kaufmann*, *Francesc Trull*, *Elena Vasilyeva*, *Lucas Lersch*, *Frank Tetzel*, *Robin Rehrmann*, and *Georgios Psaropoulos*.

I was lucky to be able to supervise a large number of extremely talented students and PhD students who significantly contributed to the entire graph processing project: *Radwan Deeb*, *Sebastian Rode*, *Luis Brandelli*, *Gaspard Ohlmann*, *Max Wildemann*, *Jan Broß*, *Jens Cram*, *Frank Tetzel*, and *Matthias Hauck*.

I would also like to thank my "second family" at the Database Systems Group in Dresden for always making me feel welcome and well integrated into the group. For many productive discussions I would like to thank *Matthias Böhm*, *Steffen Preißler*, *Benjamin Schlegel*, *Gunnar Fabritius*, *Rihan Hai*, *Frank Rosenthal*, *Lars Dannecker*, *Ulrike Fischer*, *Katrin Braunschweig*, *Julian Eberius*, *Tim Kiefer*, *Robert Ulbricht*, *Ines Funke*, *Ulrike Schöbel*, *Dirk Habich*, *Maik Thiele*, *Martin Hahmann*, *Hannes Voigt*, *Thomas Kissinger*, *Tobias Jäkel*, *Tomas Karnagel*, *Till Kolditz*, *Claudio Hartmann*, *Kai Herrmann*, *Kasun Perera*, *Ahmad Ahmadov*, *Johannes Luong*, *Annett Ungethüm*, *Patrick Damme*, *Lars Kegel*, *Juliana Hildebrandt*, *Alexander Krause*, and *Bernd Keller*.

I am deeply thankful for the offered opportunity of pursuing a PhD at SAP, in particular I would like to thank *Arne Schwarz* for being the good soul of the SAP HANA CAMPUS, helping in all sorts of technical and non-technical questions. Further, I would like to thank *Franz Färber*, *Norman May*, *Alexander Böhm*, and *Stefan Bäuerle* for believing in me and my research vision, and many discussions on various research aspects around graph data management.

Pursuing a PhD is not for the faint-hearted ones: There are many ups and downs during the PhD lifetime, especially during the downs, support and belief from family and friends are really important. I am deeply honored and proud to have a family and friends who motivated and encouraged me during the downs and celebrated with me during the ups.

I would like to thank my parents and my brother for their absolute support and apologize for not letting you visit me in the final year while I was writing the thesis. I also thank my beloved partner *Kerstin Witter* for her support, she certainly suffered the most from evenings that I worked until late at night and weekends that I spent in the office. Finally I would like to thank my grandparents for always believing in me. I dedicate this thesis to my beloved grandfather Helmut who sadly did not live long enough to see me graduate.

Marcus Paradies
Walldorf, April 25, 2017

CONTENTS

1 INTRODUCTION

We are all living in a highly connected world. This connectedness influences almost every part of our daily life and can be found in a large variety of situations—from the phenomenal growth of the World Wide Web to epidemics and financial crises in our modern society. The interrelationships between a set of things can be represented as a *graph (network)*.[1] Graphs can be found in an enormous heterogeneity of domains and applications, ranging from making recommendations in social media platforms to analyzing protein interactions in bioinformatics.

The potential to acquire new business insights from graph-shaped data through graph analytics is increasingly attracting companies from a variety of industries, ranging from web companies to traditional enterprises. Increasingly, companies offer and advertise—in conjunction with their products—social media platforms that allow connecting customers with each other to share ideas, discuss products and features/extensions, and take part in competitions. The recent advent of *connected fitness* is one prominent example, where customers build large communities and report fitness results, compete in monthly challenges, and share common consumption interests. Gartner (Valdes, 2012) classifies the consumer web, a domain with a steeply rising market share, into five graphs: (1) the social graph, (2) the intent graph, (3) the mobile graph, (4) the interest graph, and (5) the consumption/-payment graph.

The main challenge, however, lies not only in storing graph-structured data, but also in performing complex queries efficiently on it to derive new insights that can help to create personalized advertisements and to react faster to the consumption behavior of customers. Enterprise applications, target a wider range of use cases, including *network impact analysis* to find potential multiplicator persons, *route finding* to improve delivery times of transportation companies, *collaborative filtering* to give recommendations according to the buying behavior of users, *supply chain management and logistics* to find bottlenecks in supplier networks, *fraud detection*, and *knowledge graphs* to find domain experts using semantic search technology in a company (digital asset management).

Coherently with the widespread adoption of graph data to model and represent real-world entities and their relationships, the size of graphs is growing to an unforeseen scale. Table 1.1 depicts a graph sizing classification by Burkhardt and Waring (2013). If we compare the presented numbers with the largest freely available real-world graph data set (partial hyperlink graph, common web crawl, 2012)[2], consisting of about $129 \cdot 10^9$ edges, we can reason that—even compared to the smallest predicted scale—the size of graph data is about to exceed current scales by multiple orders of magnitude. Although the table only provides numbers of the graph topology, there is evidence that attributes associated with vertices and edges will multiply the raw size of a graph on disk by several orders of magnitude.

One of the most promising solutions to overcome the sheer flood of data to be stored and processed are NoSQL database systems. These systems target an immense variety of data models and application domains, including key-value stores, document DBMS, graph DBMS (GDBMS), and multi-model DBMS among others. NoSQL systems aim at providing *horizontal scaling* to growing data sizes and achieve this by softening up strong transaction guarantees by employing *eventual consistency* instead of *strong consistency*. Driven by a growing customer adoption, especially GDBMS received a considerable amount of attention in both the research community and the industry. In a recent study Forrester Research (Yuhanna et al., 2014) reported that over 25% of enterprises will be using graph database technology by 2017, resulting in a growth of 500% since 2013.

1 We use the two terms synonymously in the course of this thesis
2 http://webdatacommons.org/hyperlinkgraph/

Table 1.1: Different graph scales and their (predicted) characteristic key numbers, including the number of vertices and edges, and the raw size of the graph topology in a CSV format (Burkhardt and Waring, 2013).

	# of vertices	# of edges	Raw size (CSV format)
Social Scale	$\sim 10^9$	$\sim 10^{11}$	2.92 TB
Web Scale	$\sim 5 \cdot 10^{10}$	$\sim 10^{12}$	29.50 TB
Brain Scale	$\sim 10^{11}$	$\sim 10^{14}$	2.84 PB

Specifically, novel and traditional business applications leverage the advantages of a graph data model, such as schema flexibility and an explicit representation of relationships between data records. Although these business applications mainly operate on graph-structured data, they still require direct access to the relational data. Typically, these industries rely on mature RDBMS technology to keep a single source of truth and access. Although graph structure is already latent in the database schema and inherently represented by foreign key relationships, managing native graph data is moving into the focus as it allows rapid application development due to the absence of an upfront defined database schema. Existing solutions performing graph operations on business-critical data either use a combination of SQL and application logic or employ a graph management system (GMS), such as Neo4J or Sparksee, or distributed graph systems, such as GraphLab or Apache Giraph. For the first approach, relying only on SQL typically results in poor execution performance caused by the functional mismatch between a graph algebra and the relational algebra. Even worse, the relational query optimizer is not *graph-aware*, i.e., it does not keep statistics about the graph topology nor about graph query patterns, and is likely to build a suboptimal execution plan. The other alternative is to process the data in a native GMS to overcome the unsuitability of the relational algebra to express complex graph queries in an RDBMS.

In the following we further elaborate on the characteristics of modern data management system landscapes as can be found in most large companies. In the course of this thesis, we describe the most important problems and challenges that we see for graph data management and outline one possible solution for graph processing in modern data management system landscapes.

1.1 HETEROGENEOUS DATA MANAGEMENT SYSTEM LANDSCAPES

In the era of *Big Data*, companies face tremendous challenges when processing data of different shape, size, and velocity. These challenges are the key drivers that led to the separation of DBMSs into isolated *data silos* and the proliferation of diverse DBMS landscapes. Specifically, these challenges are driven by the following characteristics of typical data management system landscapes: (1) Data is ingested from a large number of different data sources—ranging from structured to unstructured data—and (2) there is a broad range of modern business applications with specific needs in terms of data storage, consistency guarantees, schema flexibility, and scalability to handle larger data volumes. The heterogeneity of the DBMS landscape and its separation into data silos, however, poses the challenges of orchestrating query processing and assuring data consistency across system boundaries. Even a modest-size DBMS landscape of an enterprise can easily consist of several thousands of isolated DBMSs, partially coordinated in a common middleware layer (Brodie and Liu, 2010).

Traditionally, row-oriented, disk-based RDBMSs were designed for transactional business processing with frequent updates to the database and short-running point queries. Although row-oriented RDBMSs are still used for processing transactional data in most business-critical applications, they cannot serve novel business applications demanding a

different set of supported functionality, data models, and query languages. Stonebraker and Çetintemel (2005) raise the question whether the sermonized *"One Size Fits All"* paradigm still holds true for row-oriented RDBMSs and whether they should be considered as the only valid answer to an arbitrary data problem and workload. Specifically, they propose to build specialized DBMSs for these non-traditional workloads, focusing on the specific requirements of the data model and the query workload during the design phase of the DBMS. The recent NoSQL movement is one indicator of this paradigm shift from traditional, row-oriented RDBMSs tailored to business transaction processing towards highly specialized systems for non-traditional workloads, such as graph processing, stream processing, and statistical operations (Mohan, 2013).

Incoming transactional data is usually processed by a few operational systems that rely on mature RDBMS technology to store, manipulate, and query the data. System stability, security, and reliability are of utmost importance for these systems since a database corruption or a security leak can have a tremendous negative business impact. Specialized data management systems are usually not designed to cope with these kinds of non-functional requirements, which is why they are mainly used to run on replicated data only (Mohan, 2013).

A typical business application built on top of an RDBMS interacts with the database through some common relational query interface, such as SQL. If the specialized DBMS should replace the RDBMS, it has to offer an equivalent query interface and other important functionalities required by the business application, such as transaction support or backup & recovery.

Another option would be to replicate the data by filtering and "massaging" the data (ETL) into a dedicated DBMS and to facilitate the query capabilities of the system directly. This approach requires online data replication and demands strong consistency guarantees across both systems, which is difficult to realize for online transaction processing with frequent updates to the data. For batch query processing not requiring an up-to-date view on the data, replicating the data into a specialized DBMS might be practical.

1.2 CROSS-DATA-MODEL QUERY PROCESSING

A graph does not consist of a topology only but also has a rich set of attributes attached to vertices and edges. For example, road networks do not only contain a topology, but also store information about road names, geographic coordinates denoting the intersections of roads, and represent an edge as a line segment using a geospatial data type. Similarly, social networks do not only represent the current state of the graph, but also have a temporal dimension that tracks when certain connections between entities have been established. Or consider a knowledge graph, where all shapes of data are integrated into a single graph instance describing certain facts and entities in the form of documents, videos, or structured data, and their interrelationships.

To issue queries that access data from different data models and types seamlessly, the data should be ideally stored in a single DBMS. Moreover, specialized query processing engines in the DBMS are tailored towards efficient query execution on the data. If one would want to issue the same query against a diverse system landscape, one would need a large number of specialized systems and an additional orchestration layer to merge intermediate results. Further, the graph data has to be accessible from the SQL interface to also allow non-graph applications to query the data via traditional relational queries.

To cope with these issues, there are ongoing efforts driven by industry leaders to consolidate the DBMS landscape where possible and to allow online querying even on the most recent data snapshot that is not necessarily relational. One of the most promising possible solutions is the development of a *data platform* (Färber et al., 2012). A data platform is a multi-engine DBMS accommodating native support for a large variety of data models and query processing capabilities. Such a data platform lowers the total cost of ownership, simplifies the administration of the system landscape, and provides a unified view of the data to the application developer.

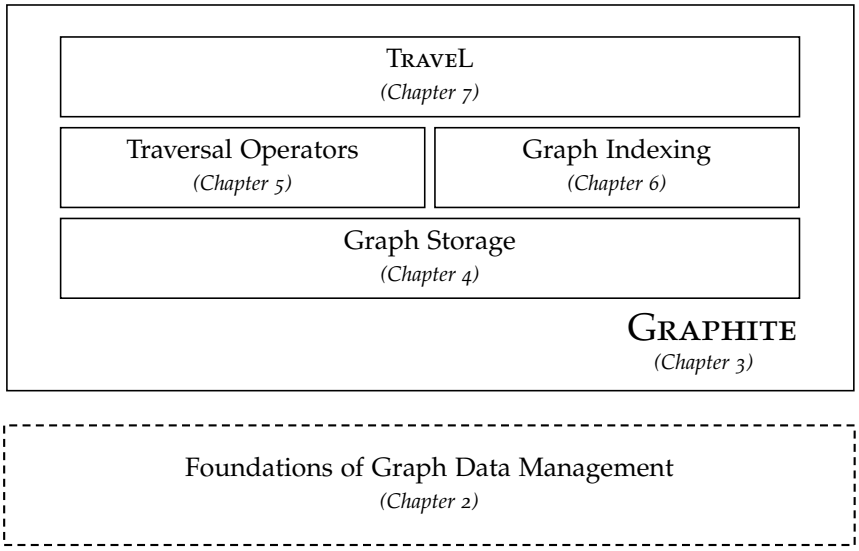


Figure 1.1: General overview of the thesis structure.

1.3 CONTRIBUTIONS

In the course of the thesis, we describe the system architecture of a graph processing runtime as part of an operational RDBMS and discuss its fundamental components. Our main concern is the development of a graph runtime as an extension of an RDBMS that is competitive in terms of execution performance with native graph processing systems while retaining strong guarantees required by enterprise-critical applications, including transaction support, backup & recovery, and security management. To that end, we describe the system architecture and core components of Graphite, a prototypical implementation of a graph runtime, which is integrated into an existing RDBMS kernel. Figure 1.1 illustrates our major contributions in the context of Graphite.

GRAPH STORAGE. The physical graph representation is based on the assumption that graph data originally resides in relational tables and can be transformed into a dedicated tabular graph storage. We extend the simple graph representation based on *universal tables* with strategies to improve the data compression and attribute access of vertices and edges. We discuss the graph representation and the data reorganization techniques applied to vertices and edges in Chapter 4. Parts of the material presented in Chapter 4 have been developed together with Michael Rudolf, Radwan Deeb, and Wolfgang Lehner and have been partly published in Rudolf et al. (2013).

GRAPH TRAVERSAL OPERATORS. In Chapter 5 we introduce our notion of a graph traversal operation, which is inspired by a breadth-first traversal, but allows performing a graph traversal configured by a user-defined traversal specification. A traversal configuration describes the subgraph to traverse through an optional vertex and edge filter. The traversal is specified by the traversal boundaries defining the maximum traversal depth and the traversal iteration from where to start collecting visited vertices. Based on the logical definition of a graph traversal operator, we outline two implementations targeting different traversal configurations and underlying graph properties. We describe a common cost model for both implementations based on collected graph statistics. Parts of the material have been developed together with Wolfgang Lehner and have been published in Paradies et al. (2015).

GRAPH INDEX STRUCTURES. We introduce two graph-specific secondary index structures in Chapter 6, which we use to accelerate the execution of neighborhood queries and

graph traversal operations. One of the major drawbacks of most graph index structures is the lack of update support, which is an essential requirement for an index structure to be used in an operational environment. Therefore, we developed an index structure called GRATIN (graph traversal index), which is a clustered, block-based lightweight index structure and mimics a compressed-sparse-row (CSR) storage format in a relational table. Further, we introduce a high-performance adjacency list as a secondary index structure, which is updatable and maintains logical pointers between the index and the relational base tables. This allows seamlessly combining relational predicate evaluation with graph-oriented topological queries. Parts of the material presented in Chapter 6 have been developed together with Sebastian Rode, Matthias Hauck, and Wolfgang Lehner and have been published in Paradies et al. (2014) and Hauck et al. (2015), respectively.

TRAVEL — A DSL FOR GRAPH ANALYTICS. In Chapter 7 we describe TRAVEL (traversal language), a domain-specific language for traversal-based graph algorithms. To write graph algorithms that are tailored to a specific domain, simple graph traversals are not expressive enough nor do they allow customization to the user's needs. To cope with this issue, graph database vendors provide—in addition to declarative graph query languages—procedural interfaces to write user-defined graph algorithms. Imperative interfaces are a powerful tool, but they also have major drawbacks. A procedural programming interface is cumbersome to use and requires the user to specify the graph algorithm against a low-level graph API in a general-purpose language, such as C++ or JAVA. To the worse, writing graph algorithms in a general-purpose language prevents exploiting data- and domain-dependent optimizations at runtime and certain query optimization and rewriting techniques, such as filter push-down and exploiting intra-query parallelism, are not possible.

We introduce a programming model based on *traversal hooks*, a powerful concept to extend and manipulate graph traversal implementations with domain-specific code provided by the user. Traversal hooks follow an event-based programming model and provide an interface for a variety of traversal events, such as the discovery of a new edge or the visit of an already discovered vertex. Traversal hooks are expressed in TRAVEL and we use the LLVM framework to glue together the traversal code with the traversal hooks at runtime and generate efficient user-defined traversal operators.

Before we start and describe the components of GRAPHITE in detail, we provide a broad overview of graph data management in Chapter 2 and classify therein several alternative graph systems, algorithms, graph index structures, and programming models. In each of the chapters, where we describe core components of GRAPHITE in detail, we will discuss the related work for each component individually. The last chapter summarizes the contributions and findings made in the thesis and discusses several directions for follow-up projects and future research. In Appendix A we give a detailed overview of computed graph properties of real-world and synthetic graph data sets that we use in the course of the thesis.

2 FOUNDATIONS OF GRAPH DATA MANAGEMENT

With the ever-growing availability of graph-structured data from public, freely accessible data sources like WIKIPEDIA and STACKOVERFLOW, there has been a growing interest in mining and understanding the properties of these graphs. Popular repositories, which offer deep insights about the properties and the structure of the contained graphs, are SNAP[1], the WEBGRAPH FRAMEWORK (Boldi and Vigna, 2004), and KONECT[2] (Kunegis, 2013). In addition to the availability of graph data sets, graph libraries, such as SNAP (Leskovec and Sosic, 2016) and the BOOST graph library (BGL) (Gregor and Lumsdaine, 2005), have been developed to gather insights from the data.

2.1 DATA MODELS

There is a zoo of available graph data models to choose from. Graphs appear in many flavors, resulting in a large variety of graph data models, which are usually tailored towards a specific domain or use case (Rodriguez and Neubauer, 2010b). Angles and Gutierrez (2008) give a detailed survey of the evolution of graph data models, with the so-called *property graph data model* described by Rodriguez and Neubauer (2010b) as the latest member of the family of graph data models. In the following we briefly introduce the basic mathematical model of a graph, the RDF data model, which gained quite some popularity in the semantic web and linked data community, and the property graph data model. For a detailed survey we refer the reader to Angles and Gutierrez (2008).

2.1.1 *Mathematical Data Model*

In the following we define fundamental notations and introduce basic concepts used in the course of this thesis.

Definition 1 (Graph) *A directed, multi-relational graph* G *is a pair of sets with* $G := (V, E)$*, where* V *is the set of vertices and* $E \subseteq V \times V$ *is the set of edges. Each vertex is uniquely identified by a vertex identifier. Two vertices* u, v *are equal, iff* $\mathrm{id}(u) = \mathrm{id}(v)$*. An edge* $e \in E$ *is written as* $\langle u, v \rangle$ *with* $u, v \in V$*. We consider each edge as directed, i.e.,* $\forall u, v \in V : \langle u, v \rangle \neq \langle v, u \rangle$ *must hold.*

A vertex v is *incident* with an edge e, if $e = \langle u, v \rangle \in E \vee e = \langle v, u \rangle \in E$. Two vertices u, v are adjacent (neighbors) if $\langle u, v \rangle \in E \vee \langle v, u \rangle \in E$. We call a vertex u the *source vertex* of an edge e, if $\langle u, v \rangle \in E$. Likewise, we call a vertex v the *target vertex* of an edge e, if $\langle u, v \rangle \in E$.

Definition 2 (Path) *A path of length* k *is a non-empty, acyclic graph* $P = (V, E)$ *with* $V = \{v_0, \ldots, v_k\}$ *and* $E = \{\langle v_0, v_1 \rangle, \langle v_1, v_2 \rangle, \ldots, \langle v_{k-1}, v_k \rangle\}$.

2.1.2 *The RDF Data Model*

The Resource Description Framework (RDF) has been initially proposed and designed as a data model for the semantic web, but recently also received a wider adoption in knowledge management applications (RDF, 2014). The fundamental concept of RDF is the ability to make statements about *resources* using so-called *triples*. A triple consists of a *subject*, a *predicate*, and an *object*. Each entity in RDF is identified via an internationalized resource identifier (IRI), where subjects and predicates are always represented by their corresponding IRI, and objects *can* be represented by a IRI or a value. The total of all triples in an RDF data set form a directed, multi-relational labeled graph.

1 Stanford Large Network Dataset Collection — http://snap.stanford.edu/data/ (Last accessed: April 2017)
2 Konect Network Datasets — http://konect.uni-koblenz.de/networks/ (Last accessed: April 2017)

2.1.3 *The Property Graph Data Model*

The property graph data model is based on a directed, multi-relational graph and an arbitrary number of attributes attached to vertices and edges in a key-value pair fashion (Rodriguez and Neubauer, 2010b). It received a wide adoption from the industry, especially from graph database management systems (GDBMS) like NEO4J and SPARKSEE, but also other RDBMS-based systems that support graph processing like SAP HANA and SQLGRAPH. As pointed out by Rodriguez and Neubauer (2010b), the property graph model subsumes simpler graph data models by adding or removing certain constraints imposed on the specific data model. NEO4J and SAP HANA use variations of the property graph data model by extending it with support for *type* (*label*) information that can be attached to vertices and edges. While NEO4J treats vertex/edge labels as normal attributes (except for querying), SAP HANA has a concept of allowing type subsumption over a hierarchy of semantic types, called *terms* (Bornhövd et al., 2012). There is ongoing work trying to reconcile the RDF data model and the property graph data model (Hartig, 2014).

2.2 GRAPH PROPERTIES

Throughout the course of the thesis, we will use fundamental graph properties, such as the number of vertices $|V|$, the number of edges $|E|$, the minimum/maximum/average in/outdegree of vertices, and the (effective) diameter, and define them in the following.

Definition 3 (**Vertex Degree**)
The outdegree deg_{out} of a vertex u is

$$deg_{out}(u) \coloneqq |\{\langle u, v\rangle \in E\}| \tag{2.1}$$

The indegree deg_{in} of a vertex u is

$$deg_{in}(u) \coloneqq |\{\langle v, u\rangle \in E\}| \tag{2.2}$$

The outdegree (indegree) represents the number of vertices that are connected to a given vertex via outgoing (incoming) edges. In addition to the per-vertex degree information, we define the average outdegree deg^{avg}_{out}, the maximum outdegree deg^{max}_{out}, and the minimum outdegree deg^{min}_{out}. The indegree measures are defined equivalently. Another important graph-global property is the diameter of a graph G, which describes the longest shortest path for any given pair of vertices.

Definition 4 (Graph Diameter) *The diameter of a graph G is:*

$$diam(G) \coloneqq max\ \{\ shortestPath(u, v)\ \forall u, v \in V, u \neq v\ \} \tag{2.3}$$

The n-percentile (effective) diameter is often used to disregard long outlier paths:

$$diam_n(G) \coloneqq percentile_n\ \{\ shortestPath(u, v)\ \forall u, v \in V, u \neq v\ \} \tag{2.4}$$

In practice, the *effective diameter* $diam_{0.9}(G)$ is often used to describe the all-but-longest distance between any pair of vertices in a graph.

The availability of graph data sets also spawned research in the area of social network analysis. For example, the *small-world phenomenon*, first observed by Milgram (1967), states that every human being knows every other human being along six hops. In fact, Backstrom et al. (2012) confirmed the general observation of the small-world phenomenon for the Facebook *social graph*, resulting in a diameter of 4.74. In a recent study, Meusel et al. (2014) analyzed a web graph crawl provided by the Common Crawl Foundation containing over 3.5 billion web pages and 128.7 billion links between them. They report an average distance of 12.84. Similar studies have been performed for detecting and characterizing

communities in graphs, i.e., sets of vertices that share more connections to other vertices in the community than to vertices outside of the community (Leskovec et al., 2008).

In contrast to the calculation of graph properties on a static snapshot, Leskovec et al. (2005) investigate the evolution of the degree distribution and the diameter of a graph over time. They report that graphs tend to become *denser* over time, i.e., the average degree increases and the graph diameter *shrinks* over time. Monitoring these properties on a static graph as well as over a period of time is an important tool to improve the quality of graph data generators to capture realistic properties (Chakrabarti and Faloutsos, 2006).

2.2.1 *Graph Statistics*

The collection of graph statistics about global properties of the graph topology, such as the degree distribution (Ribeiro and Towsley, 2012; Zhang et al., 2013), diameter estimation (Kang et al., 2011b; Roditty and Vassilevska Williams, 2013), and neighborhood size estimation (Lipton and Naughton, 1989; Cohen, 1997; Palmer et al., 2002; Boldi et al., 2011; Boldi and Vigna, 2013; Cohen, 2013), has been an active area of research for almost three decades. With the advent of social network analysis and the need to compute graph-global measures, novel estimation and sampling techniques have been developed (Lovász, 1993; Wei et al., 2004). Since graphs do not only consist of a topology, but usually also contain a rich set of attributes, cardinality estimation techniques known from the relational world have been applied to graphs as well. One prominent example is the notion of *characteristic sets*, which estimates the cardinality of a set of vertices exposing certain attributes (Neumann and Moerkotte, 2011).

2.3 QUERY LANGUAGES

Although early proposals of query languages for GDBMS date back to the 1980s, there has been made only little effort lately to develop new or to adapt existing graph query languages. Wood (2012) gives an extensive overview of graph query language proposals over the last 25 years. In contrast to the RDF data model and its corresponding query language SPARQL (SPA, 2013), there are currently no standardization efforts in the field of graph query languages for the property graph data model. In Figure 2.1 we depict the most recent graph query language proposals and classify them in two directions: (1) targeted application area and (2) language paradigm.

2.3.1 *Pattern-Matching-Based Query Languages*

The most prominent graph query languages target the subgraph isomorphism problem and follow a declarative language paradigm (Sakr et al., 2012; Panzarino, 2014; Raman et al., 2014; SPA, 2013). G-SPARQL (Sakr et al., 2012) is an extension of SPARQL with specific language constructs to query property graphs, including the specification of edge attribute filters and unbounded traversals to support reachability queries. The limitation of SPARQL 1.0 of not being able to specify unbounded path expressions has been solved with the SPARQL 1.1 language specification. CYPHER, the declarative query language of NEO4J, is internally transformed into an execution plan and further optimized based on collected graph statistics (Panzarino, 2014). Inspired by SPARQL and CYPHER is PGQL, a declarative graph pattern matching query language for the property graph model that is part of PGX (Raman et al., 2014). PGX exposes both SPARQL and PGQL, and supports the automated conversion of a subset of SPARQL into equivalent PGQL statements.

2.3.2 *Traversal-Based Query Languages*

Traversal-based query languages provide native support for graph traversals as part of the language, examples are GEM (Rudolf et al., 2013), the graph query language of SAP HANA,

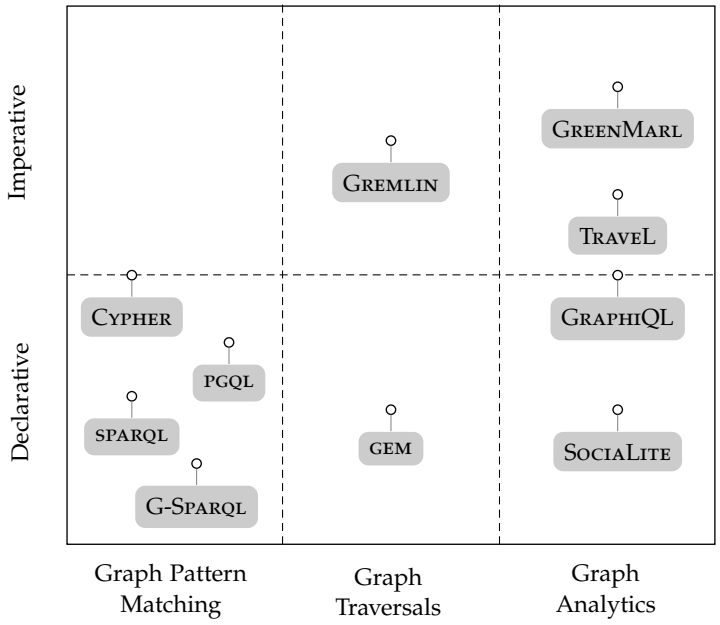


Figure 2.1: Classification of graph query languages according to their main targeted application area and the language paradigm.

and GREMLIN (Tin). GREMLIN is a JAVA-based graph traversal language and is as part of the APACHE TINKERPOP[3] graph computation framework available on top of many graph processing systems, including NEO4J, SPARKSEE, and SQLGRAPH. GREMLIN is one of the few graph query languages that received a wide adoption from the graph community by relying on a vendor-agnostic interface named BLUEPRINTS, the graph API of APACHE TINKERPOP. GREMLIN provides rich query and manipulation capabilities and can be further extended through JAVA and GROOVY interfaces, for example to define domain-specific graph traversal semantics. On the downside, most vendors offering a GREMLIN interface do not perform any query optimizations, with SQLGRAPH being the only exception. SQLGRAPH, however, does not offer a BLUEPRINTS backend, but instead directly translates (read-only) GREMLIN statements into SQL.

2.3.3 *Query Languages for Graph Analytics*

Increasingly, more applications target graph analytics or graph mining, i.e., algorithms that traverse the entire graph, possibly multiple times. While imperative programming models, such as *vertex-centric* or *edge-centric* computation models, are gaining popularity, they are lacking a high-level exposition to the end user. GREENMARL (Hong et al., 2012, 2013b) is among the first domain-specific graph query languages targeting graph analytics and exposes high-level constructs, such as graph traversals and parallelized loops. Similar in that spirit are SOCIALITE, a declarative query language based on DATALOG (Lam et al., 2013), and our language proposal TRAVEL, a traversal-based domain-specific language, and a corresponding TRAVEL code generator that emits executable code (cf. Chapter 7). GRAPHIQL (Jindal and Madden, 2014) is an imperative query language that exposes important primitives, such as loops, recursion, and neighborhood operations, to the end user. GRAPHIQL runs on top of an RDBMS and the corresponding compiler translates GRAPHIQL statements into a series of SQL commands.

3 APACHE TINKERPOP— http://tinkerpop.incubator.apache.org/ (Last accessed: April 2017)

2.4 GRAPH ALGORITHMS

Graph algorithms have witnessed an everlasting interest in the theoretical computer science community with applications in various domains, including routing, electronic circuit design, proteomics, and chemistry. They range from simple graph traversals, such as breadth-first traversals (BFT) and depth-first traversals (DFT) (Cormen et al., 2001), graph summarization algorithms, including the detection of strongly connected components (Tarjan, 1972; Hong et al., 2013a; Slota et al., 2014) and minimum spanning trees (see algorithms by Borkuvka (1926), Prim (1930), and Kruskal (1956)), to complex graph optimization problems, such as traveling salesman or the Chinese postman problem (Christofides, 1973), which are not computable in polynomial time.

Despite the availability of more computing resources and an increased level of available parallelism, graph algorithms can hardly fully benefit from the available hardware resources. Lumsdaine et al. (2007) analyze the root causes of the performance problems of parallel graph processing and make the following observations:

- Graph algorithms are data-driven
- Graph algorithms expose a poor data locality
- Graph algorithms belong to the data-crunching category

Many well-known graph algorithms have been successfully implemented in a multi-core environment by fully leveraging all available compute cores, such as depth-first traversal (Crauser et al., 1998), strongly connected components (Hong et al., 2013a; Slota et al., 2014), and triangle counting (Sevenich et al., 2014). In addition to exploiting the available parallelism, several recent studies explore improvements of graph algorithms on specific hardware aspects, such as exploiting the memory hierarchy and available cache levels by avoiding data cache misses (Cong and Makarychev, 2011), instruction cache misses (Green et al., 2014), or by avoiding remote memory accesses on non-uniform distributed memory (NUMA) machines (Zhang et al., 2015). Several programming models and corresponding abstraction layers have been proposed to provide the basis for writing performance-oriented, hardware-conscious graph algorithms (Nguyen et al., 2013; Shun and Blelloch, 2013; Harshvardhan et al., 2014).

2.4.1 *Graph Traversals*

Graph traversals are a fundamental building block for more complex graph algorithms, such as detecting connected components, bipartite graph matching, finding shortest paths in unweighted graphs, and appear in domain-specific use cases, such as 3D computer graphics (Hanrahan, 1986), image processing (Silvela and Portillo, 2001), and solving boolean satisfiability (SAT) instances (Motter and Markov, 2002).

Increasing graph data sizes and the proliferation of parallelism on different hardware levels as well as heterogeneous processor environments encouraged researchers to revise well-known graph algorithms and to propose novel implementations on high-end computers with a large number of cores and heterogeneous processor types in a single machine.

There are two fundamental traversal strategies to visit all vertices in a graph: *breadth-first* and *depth-first* (Cormen et al., 2001). In a breadth-first traversal, the traversal starts at a root vertex and visits vertices in a level-synchronous manner, i.e., by first visiting all adjacent vertices before the traversal continues with the next level. A depth-first traversal visits recursively one adjacent vertex at a time and continues the traversal from there. If the traversal reaches a vertex with no adjacent vertices, it continues at the unfinished vertex, which is the closest to the root vertex.

In the course of this thesis, we will investigate breadth-first traversals in detail. We chose breadth-first traversals over depth-first traversals since breadth-first traversals can be implemented more efficiently in a multi-core environment. In contrast, depth-first traversals

are inherently sequential and can be hardly parallelized to scale to large server machines with dozens of available computing units. This observation led to the development of parallel graph algorithms, which are originally based on a DFT, but in practice implemented using a parallelized BFT (McLendon III et al., 2005).

Single-Node Breadth-First Traversals

A large body of research has been conducted on efficient parallel graph traversals on a single server (Brodal et al., 2004; Agarwal et al., 2010; Pearce et al., 2010; Chhugani et al., 2012; Cui et al., 2012; Beamer et al., 2012; Yuan et al., 2012; Hong et al., 2011; Xia and Prasanna, 2009; Kernert et al., 2014; Then et al., 2014; Berrendorf and Makulla, 2014; Yasui et al., 2014). State-of-the-art parallel graph traversals use a level-synchronous strategy and parallelize the work to be done at each level. Early work by Xia and Prasanna (2009) adapts the number of worker threads at each level in the traversal according to the working set size. Several implementations optimize for specific hardware characteristics, such as minimizing data cache misses (Brodal et al., 2004; Agarwal et al., 2010; Yuan et al., 2012; Chen et al., 2013) and reducing remote memory accesses (Agarwal et al., 2010; Yasui et al., 2014; Chhugani et al., 2012; Cui et al., 2012). Especially maximizing temporal and spatial memory locality is important for achieving optimal traversal performance (Yuan et al., 2012; Chen et al., 2013).

An effective optimization for scale-free graphs with a low diameter has been proposed by Beamer et al. (2012). They describe a *direction-optimized* traversal, which switches the execution order during the traversal from a *top-down* traversal to a *bottom-up* traversal. While a top-down traversal searches for children of a set of frontiers, a bottom-up traversal searches for a set of unvisited vertices for parents. This optimization leverages the fact that traversals on scale-free graphs tend to expose a large frontier set after the first 2–3 traversal iterations, making it more efficient to switch the execution mode.

A different interpretation of a breadth-first traversal originates from the observation that it can be implemented as a repeated matrix-vector multiplication of the sparse adjacency matrix (Kernert et al., 2014). Hong et al. (2011) propose a breadth-first traversal implementation that leverages CPU and GPU processing resources in a hybrid execution. While most breadth-first traversal implementations focus on the acceleration of a single traversal operation utilizing all available resources, Then et al. (2014) propose a multi-source breadth-first traversal, which shares commonly explored frontier sets between graph traversals to reduce the amount of work to be performed. Such a work sharing between concurrent graph traversals can be exploited in graph algorithms that have to issue a traversal from each vertex in the graph and on scale-free graphs with a low diameter. Berrendorf and Makulla (2014) provide an in-depth experimental analysis of several level-synchronous BFT implementations and discuss several optimization techniques and measure their performance impact.

Breadth-First Traversals on Co-Processors

Graph traversals, specifically breadth-first traversals, have been an active application area for high-performance computing on specialized hardware, such as graphics processing units (GPU) (Harish and Narayanan, 2007; Merrill et al., 2012; Bisson et al., 2014; Gharaibeh et al., 2013; Sallinen et al., 2015; Fu et al., 2014; Slota et al., 2015), INTEL® Many Integrated Core (MIC) (Gao et al., 2014; Slota et al., 2015), and tree-based memory models (St. John et al., 2012). While initial proposals focus on the pure implementation of breadth-first traversals, typically based on CUDA, recent works have focused on providing a higher level abstraction that either guarantees portability to different classes of many-core processors (Slota et al., 2015) or provides a vertex-centric API through a *scatter-gather* interface (Fu et al., 2014; Sundaram et al., 2015).

Distributed Breadth-First Traversals

If the graph size exceeds the memory capacity of a single server, the graph can be either processed from semi-external memory, such as a hard disk or a flash-based storage medium (Pearce et al., 2010), or can be distributed among several servers in a cluster. There are multiple implementations targeting distributed memory graph traversals, which differ mainly in their initial graph partitioning scheme and the strategies to minimize communication across the network (Yoo et al., 2005; Buluç and Madduri, 2011; Lv et al., 2012; Beamer et al., 2013; Shang and Kitsuregawa, 2013; Checconi and Petrini, 2014). One of the first distributed graph traversals is described by Yoo et al. (2005), who implement a breadth-first traversal on a large cluster installation of a BlueGene/L with up to 32,000 processors and achieve up to 100 million traversed edges per second (TEPS). They avoid the expensive all-to-all communication by applying a two-dimensional partitioning on the adjacency matrix. In a related study, Checconi and Petrini (2014) investigate breadth-first traversals on a cluster with 64,000 BlueGene/Q nodes and achieve an impressive number of 15.3 trillion TEPS. Among several algorithmic tricks, they changed the initial two-dimensional partitioning to a one-dimensional vertex partitioning. Beamer et al. (2013) discuss the direction-optimized traversal strategy, which switches from a *top-down* to a *bottom-up* traversal evaluation during execution, on a distributed memory cluster. Shang and Kitsuregawa (2013) propose a novel distributed graph traversal by applying a *degree-based partitioning*, which provides a balanced execution on skewed vertex degree distributions. To reduce the communication overhead in distributed traversals, Lv et al. (2012) propose message compression to minimize the size of communication messages.

2.5 GRAPH DATA GENERATORS AND BENCHMARKING

There have been early discussions on the design of benchmarks for graph processing systems, mainly driven by the graph database community (Dominguez-Sal et al., 2010, 2011) and the high-performance community (Gra). While the high-performance community uses BFT to measure the execution performance of supercomputers, major GDBMS vendors are interested in showing the performance and scalability of their systems for a wider range of use cases and applications. One of the main distinguishing factors of GDBMS is the ability to perform topological queries, such as graph traversals, faster than their RDBMS equivalents using chained self-joins. Therefore, the main focus for GDBMS benchmarking has been on graph traversals (Ciglan et al., 2012). With the more widespread adoption of pattern-matching-based query processing on property graphs with a large number of attributes attached to vertices and edges, new benchmarking initiatives, such as the Linked Data Benchmarking Council (LDBC)[4] (Erling et al., 2015), emerged.

R-MAT Data Generator

The R-MAT data generator proposed by Chakrabarti et al. (2004) creates graphs with a skewed degree distribution (power-law distributed), a community structure, and a small diameter. Figure 2.2 depicts the general data generation model of R-MAT. The data generation works on the adjacency matrix of the graph and recursively subdivides the matrix into four equal-sized partitions a, b, c, and d. Each partition is subsequently divided into four partitions, until a partition size of one is reached. Each edge to be inserted is added with a given probability that is taken from the input parameters a, b, c, and d. Thereby, the precondition $a + b + c + d = 1$ must hold. To generate a power-law degree distribution, a community structure, and a small diameter, the authors recommend choosing the input parameters such that $a \geqslant b \wedge a \geqslant c \wedge a \geqslant d$ holds. For example, the Graph500 benchmark uses a data generator similar to R-MAT (the Kronecker data generator) and relies on the following parameter configuration: $\langle a = 0.57, b = 0.19, c = 0.19, d = 0.05 \rangle$[5]. Although the

4 Linked Data Benchmarking Council — http://ldbcouncil.org/ (Last accessed: April 2017)
5 Graph500 Specification — http://www.graph500.org/specifications#sec-3_2 (Last accessed: April 2017)

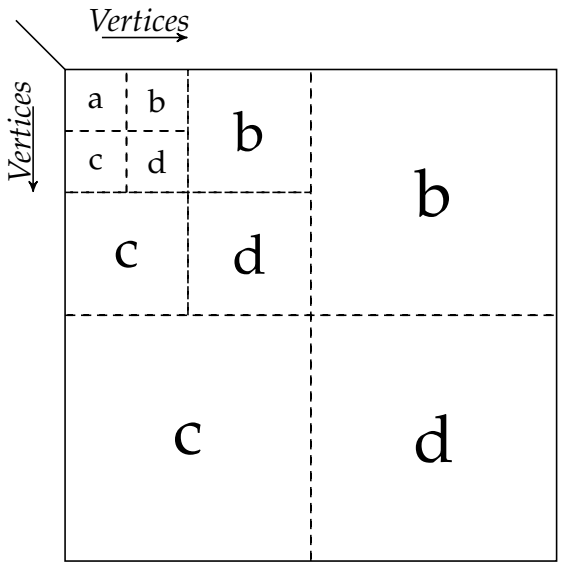


Figure 2.2: R-MAT data generation model (adapted from Chakrabarti et al. (2004)).

R-MAT data generator solves some of the problems of generating large graphs with realistic properties, it does not capture attributes attached to vertices and edges. We use the R-MAT data generator in our experimental evaluations to easily generate realistic graph topologies of different scale.

LDBC Social Network Benchmark

The LDBC social network benchmark provides the first graph data generator that generates a realistic graph topology, but also a set of attributes attached to vertices and edges. It represents a social network application with user activities during a period of time (Erling et al., 2015). The data is used in the interactive, the business-intelligence, and the algorithmic workload (Capotă et al., 2015) of the LDBC social network benchmark. It contains persons, tags, forums, messages, likes, organizations, and places as vertices and about 20 different relations between them. The backbone of the graph is a fully connected component of users and their friendship relationships. The LDBC data generator puts a special emphasis on correlated attributes and structural correlations, such as the correlation between person's first names and their gender or the correlation between the places of study/common interests and the probability of establishing a friendship connection between two users (Pham et al., 2012). The attribute dictionaries are extracted from DBpedia and are assigned using a skewed value distribution. Finally, an implicit time dimension allows representing *trending topics* and the number of messages, comments, and posts are skewed to reflect occurring political and sport events as well as natural disasters.

2.6 GRAPH PROCESSING SYSTEMS

In this section we review, compare, and classify the most popular graph processing systems available that target different graph applications, programming models, and computation environments. We classify them into: (1) single-node graph processing systems, (2) distributed graph processing systems, (3) graph processing systems as extensions of RDBMS, and (4) graph database management systems (GDBMS). We provide a summary of our classification on all reviewed single-node graph processing systems, including graph processing systems on top of RDBMS and graph database management systems, in Table 2.1.

Table 2.1: Comparison of single-node graph processing systems and their supported features.

System	Vertex/Edge Attributes	Updates/ Deletions	Transactions	Language Support	Query/Programming Model	Interaction with Relational
Sparksee (Martínez-Bazan et al., 2012)	✓	✓	✓	—	Blueprints API	—
Ligra (Shun and Blelloch, 2013)	(✓)[a]	—	—	—	Map Functions	—
PGX (Raman et al., 2014)	✓	(✓)[b]	—	✓[c]	Pattern Matching	—
GraphLab (Low et al., 2012)	✓	✓	—	—	Vertex-centric	—
Neo4J (Neo)	✓	✓	✓	✓	Pattern Matching	—
GraphChi (Kyrola et al., 2012)	(✓)[d]	✓	—	—	Vertex-centric	—
Grace (Prabhakaran et al., 2012)	✓	✓	✓	—	Vertex-centric	—
EmptyHeaded (Aberger et al., 2015)	✓	—	—	✓	Boolean Algebra	✓[e]
Galois (Nguyen et al., 2013)	✓	—	—	—	ADP[f]	—
SQLGraph (Sun et al., 2015)	✓	✓	✓	✓[g]	Relational/Pipes	✓
Vertexica (Jindal et al., 2014a)	✓	✓	✓	✓[h]	Relational	✓
TeraData Aster (Simmen et al., 2014)	✓	✓	✓	—	Vertex-centric	✓
System-G (Xia et al., 2014)	✓	(✓)[i]	—	—	Blueprints API	—
LLama (Macko et al., 2015)	✓	✓	—	—	General Purpose API	—
TurboGraph (Han et al., 2013)	✓	✓	—	—	Pin-and-Slide	—
PowerGraph (Gonzalez et al., 2012)	✓	✓	—	—	Vertex-centric	—
Polymer (Zhang et al., 2015)	✓	—	—	—	Map Functions	—

a Only a single numerical edge attribute (e.g, a weight) is supported
b Batch updates
c PGQL
d only a single vertex/edge attribute is supported
e All operations are set-based/based on joins
f Amorphous Data Parallelism Programming Model
g Gremlin
h GraphiQL by Jindal and Madden (2014)
i not all graph representations support mutations

2.6.1 *Single-Node Graph Systems*

Out-of-Core Graph Systems

The initial experimental results of GraphChi (Kyrola et al., 2012), which demonstrate that a carefully implemented disk-based single-node graph processing system can outperform a large cluster installation, triggered an ever-growing interest in processing billion-scale graphs on a single commodity machine (Han et al., 2013; Roy et al., 2013; Cheng et al., 2015; Wang et al., 2015; Zhu et al., 2015; Zheng et al., 2015; McSherry et al., 2015).

GraphChi (Kyrola et al., 2012) is among the first implementations of a disk-based, single-node graph processing system running on commodity hardware and targeting applications dealing with billion-scale graphs. GraphChi avoids random access read operations on the graph stored on hard disk or SSD by transforming them into more disk-friendly, sequential read patterns. The authors propose a parallel sliding window (PSW) approach and distribute the vertex space into disjoint intervals that are subsequently mapped to so-called *shards*. A shard stores all edges whose target vertex is in the corresponding interval and sorts them by their source vertices. GraphChi implements a variation of the bulk-synchronous processing (BSP) model through an asynchronous vertex-centric execution model, but writes updated values directly instead of sending messages to adjacent vertices. The execution model is based on a *one-interval-at-a-time* paradigm—one shard (the memory shard) is fetched completely into memory and other subsequent shards are also read from SSD to fetch the outgoing edges of a given vertex set.

TurboGraph (Han et al., 2013) is comparable to GraphChi and is a disk-based graph engine to process billion-scale graphs and leverages multi-core parallelism and parallel, asynchronous I/O of SSDs to efficiently interleave I/O and CPU processing. TurboGraph proposes a novel execution model, called *pin-and-slide*, which is based on the *column view* of a matrix-vector multiplication. For a given set of vertices, TurboGraph identifies the required pages and *pins* them in the in-memory buffer pool. If some of the pages reside already in the buffer pool, one execution thread per page starts processing the vertex update function. While the first pages are being processed, TurboGraph issues asynchronous I/O calls to the SSD device to fetch the missing pages into the buffer pool. Whenever a page is fetched into the buffer pool, the execution engine is notified and can proceed to process the pages. Once TurboGraph finishes the processing of a page, it *unpins* the page explicitly and the buffer pool eventually evicts it. Thereby, TurboGraph *slides* over the pages of the graph.

In contrast to the vertex-centric systems GraphChi and TurboGraph, X-Stream (Roy et al., 2013) relies on an *edge-centric* programming model. In the *scatter* and *gather* phase, X-Stream iterates over edges and performs updates on edges rather than on vertices. An edge-centric approach avoids expensive random access operations to the edges of the graph. Instead of minimizing random accesses on vertices, X-Stream minimizes random accesses on edges, under the assumption that the number of edges is much larger than the number of vertices in the graph. To overcome the random access pattern on the vertices, X-Stream partitions the set of vertices such that it fits into memory or the CPU cache. While GraphChi requires the entire *shard*, including vertices and incoming/outgoing edges, to be in memory, X-Stream only requires the vertex state to be in memory.

Yuan et al. (2014) propose to use a *path-centric* computation model instead of a *vertex- or edge-centric* model. PathGraph partitions the graph into trees and stores each tree in DFS order to improve the disk and memory locality and uses data compression to further reduce the size of the graph on disk. Further, PathGraph clusters highly correlated paths together and parallelizes the execution at the tree partition level.

Lin et al. (2014) explore an alternative strategy to read graph data efficiently from disk by reusing the *memory mapping* facilities provided by the operating system. This is in contrast to systems like GraphChi and TurboGraph, which manage memory and page tables by themselves.

Zheng et al. (2015) propose FLASHGRAPH, a semi-external memory graph processing system running on an array of SSDs. FLASHGRAPH is built on top of a user-space SSD file system called *set-associative file system*. It stores vertex state in memory and the edge lists on SSDs. Similar to TURBOGRAPH, FLASHGRAPH also overlaps I/O with the actual computation and merges I/O requests where possible.

VENUS (Cheng et al., 2015) exhibits a vertex-centric programming model and loads a graph from disk sequentially in a *streamlined* fashion. Such a streaming approach allows interleaving data loading from disk with the computation by exploiting the large sequential bandwidth of the disk.

GRIDGRAPH (Zhu et al., 2015) stores the vertices of the graph in 1D partitions and the edges in 2D partitions using a level partitioning during the preprocessing phase. Similar to GRAPHCHI, GRIDGRAPH also applies a sliding window technique to stream the edges from disk. In contrast to other vertex- or edge-centric processing models, GRIDGRAPH combines the *scatter* and the *gather* phase into a single *streaming-apply* phase allowing to minimize the number of reads for a block of edges.

GRAPHQ (Wang et al., 2015) proposes an interesting alternative to processing analytic queries on the complete graph: a large graph is represented as multiple levels of abstractions and a query is executed through an iterative refinement across the abstraction levels. The general assumption is that many use cases do not require a complete solution on the entire graph, but instead can also rely on a partial result derived from a small fraction of the input graph. Query processing in GRAPHQ performs repeated lock-step *check-refine* calls, until the query returns the desired result or the given query budget, e.g., memory or CPU, is exhausted. The *check* phase processes the query on each individual graph partition. If no query in the check phase returns successfully, the *refine* phase identifies inter-partition edges and adds them back to the graph. Finally, the procedure continues with the *check* phase in the next iteration.

In-Memory Graph Systems

GALOIS (Nguyen et al., 2013) is a task-based parallelization system that is particularly well-suited for irregular computations, such as graph algorithms. It exposes a sophisticated programming model with *autonomous* and *coordinated* scheduling and with or without specific application priorities to provide correctness guarantees for autonomous scheduling.

LIGRA (Shun and Blelloch, 2013; Shun et al., 2015) is a graph processing library tailored to large multi-core shared-memory machines and for traversal-based graph applications, such as breadth-first search, betweenness centrality, graph radii estimation, graph connectivity, PageRank, and single-source shortest path. A graph in LIGRA can be either a graph topology solely or a weighted graph and stored in two arrays, one for storing the incoming edges and one for storing the outgoing edges for each vertex. Both arrays are partitioned (sorted) by source vertex (for outgoing edges) and target vertex (for incoming edges). Each vertex maintains the positions of its adjacency using an index into the array and degree information for incoming and outgoing edges. For weighted graphs, the edge attribute is interleaved with the target vertices and stored as a tuple in the outgoing adjacency. If additional vertex attributes are required, they have to be stored and processed in the application layer. LIGRA supports two fundamental data types: graphs and sets of vertices. Vertex identifiers have to be continuous, discrete in the range from 0 to $|V| - 1$. Depending on the size, a vertex set can be represented either as an integer array or as a bitset. LIGRA provides a simplistic programming interface to formulate graph algorithms. The core functions are: (1) an edge map function taking as input the graph, a vertex set U, an edge function, and a vertex function C, (2) a vertex map taking as input a vertex set and a vertex function, and (3) a cardinality function that returns the size of a given vertex set.

In an edge map call, LIGRA applies the edge function to all edges with a source vertex in the input vertex set U and target vertex satisfying the vertex condition C. For each active edge (u, v), the edge map returns the target vertex v in the result vertex set. Internally, LIGRA provides two variations of the edge map, one that deals with large vertex sets and

one that deals with small vertex sets. Which version is selected by LIGRA during runtime is determined by a threshold. The threshold is computed as the sum of the size of the vertex set and the sum of the out-degrees of all vertices in the vertex set. For the vertex map, LIGRA applies the vertex function on each vertex in the input vertex set and returns a (possibly reduced) vertex set of all vertices satisfying the vertex function condition. To scale the algorithms to multi-core machines, LIGRA uses CILKPLUS to parallelize loops that call the edge map and the vertex map functions and synchronizes concurrent access to shared data structures using compare-and-swap operations.

POLYMER (Zhang et al., 2015) is a graph analytics engine tailored to large NUMA machines and implements a vertex-centric programming interface. Its main goal is to co-locate graph data and graph computation in memory as much as possible by minimizing the number of random remote memory accesses. POLYMER treats a NUMA machine as a distributed system and partitions vertices and edges across memory nodes under the assumption that the graph topology is immutable. It indexes vertices from 0 to $|V| - 1$ and evenly assigns them to n disjoint vertex sets that are stored on n memory nodes. Further, in/out-edges in the graph are partitioned by their source/target vertex and assigned to memory nodes. POLYMER stores application-defined data along with their owning vertices on the memory nodes. To tackle the mismatch between data allocation threads and computation threads—caused by Linux' first-touch policy to bind virtual pages to physical frames—POLYMER uses local threads to allocate and initialize data with a memory node. To further reduce random remote memory accesses, POLYMER uses a lightweight vertex replication scheme and duplicates vertices, including partial topology information, such as the start of neighboring vertices and degree information, onto multiple memory nodes. To achieve load balancing, the authors propose to partition the graph by edges and to evenly distribute the edges to groups on different memory nodes—in contrast to the general partitioning by vertex.

The combinatorial BLAS library (Buluç and Gilbert, 2011) follows a recent trend to represent simple, traversal-based graph algorithms as a composition of linear algebra operators, such as sparse matrix-matrix multiplication and sparse matrix-vector multiplication. While combinatorial BLAS exposes the linear algebra primitives directly to the end user, GRAPHMAT (Sundaram et al., 2015) proposes a translation layer that maps *vertex-centric* programs to a composition of linear algebra operations. The authors achieve an acceptable slowdown of a factor of 1.2 over the hand-optimized code and demonstrate the feasibility of an additional abstraction layer by bringing together the insights on sparse matrix operations from the high-performance community with the vertex-centric computation model from the graph community.

PGX (Raman et al., 2014) is a parallel subgraph isomorphism and graph analytics engine. It is specifically designed for handling large graphs with billions of edges. PGX stores the graph topology in a CSR data structure consisting of two arrays and separate arrays for properties. Property lookups can be further accelerated by secondary index structures. Supported property types include primitive types, string, string-set, and datetime. PGX has an accompanying declarative graph pattern matching query language (PGQL) that can be used to query RDF and property graph data. To run a PGQL query, the query string is parsed, translated into native C++ code, compiled during runtime, and finally executed. The pattern matching algorithm is based on the conventional backtracking algorithm, but uses a parallelized breadth-first search implementation and matching stages. At each stage, each worker thread stores the generated partial matching result in the thread-local storage. Finally, a merge operation unions all partial matching solutions. Partial matching solutions are not represented as conventional vertex tuples, but due to the fixed matching order of query nodes only stored as data nodes and accessed via position in the partial solution buffer data structures. To gracefully handle a large number of partial solutions, all of them are *inlined* and physically stored in a single large buffer structure. Further, PGX provides a backend for GREENMARL (Hong et al., 2011), a domain-specific language for graph analytics, that is compiled into native C++ code.

LLAMA (Macko et al., 2015) is an in-memory graph processing system for high-performance graph analytics in the presence of vertex and edge updates. It is based on a

mutable CSR data structure and supports batch updates. We discuss the internal storage representation of LLAMA in detail in Chapter 6. To formulate queries, LLAMA provides a general purpose programming model that can be used to write edge- and vertex-centric graph applications.

SOCIALITE (Lam et al., 2013) is a declarative query language and an accompanying compiler based on DATALOG. SOCIALITE uses nested tables to represent the graph topology in an adjacency-like data structure and heavily relies on recursively-defined aggregate functions.

SYSTEM-G (Xia et al., 2014; Tanase et al., 2014) is a graph processing system, including a native graph storage, hardware-conscious graph data structures and algorithms, and graph visualization. Its architecture is tailored towards providing a multi-core, multi-socket-friendly solution focusing on scale-up scenarios, but still providing support for scale-out scenarios via RDMA. Graph data in SYSTEM-G is persisted in a graph key-value store on disk. SYSTEM-G provides a simple, low-level graph API that allows retrieving attributes of vertices and edges, neighboring vertices, and other basic graph operations. Multiple properties on vertices and edges are stored in map data structures. SYSTEM-G provides multiple graph storage representations, depending on the desired graph application, including multi-property graphs and mutable/immutable graphs. To select the most appropriate graph storage representation, SYSTEM-G provides a set of different storage layouts that a user can choose from. For graph traversals on read-only graphs, SYSTEM-G uses a compressed vertex format adapted from a CSR. To allow parallelized graph processing, SYSTEM-G partitions the original graph into nearly independent subgraphs.

EMPTYHEADED (Aberger et al., 2015) is a graph pattern matching engine leveraging worst-case optimal join algorithms to compile graph pattern queries into boolean algebra operations—set intersection and union. The development is driven by two major observations: (1) algorithmic advances, especially the development of worst-case optimal joins and (2) the general availability of wide vector registers for SIMD processing on modern CPUS. We discuss the internal storage representation of EMPTYHEADED in detail in Chapter 6.

GRACE (Prabhakaran et al., 2012) is a graph-aware, in-memory, transactional graph management system that is specifically designed for low-latency graph applications. Its design is tailored to single-node, large-scale multi-core machines with non-uniform memory access. To query a graph stored in GRACE, the user writes a vertex-centric program in the bulk-synchronous model against an imperative programming interface. The authors propose several optimizations to minimize inter-core communication and ensure maximum hardware utilization through update message batching, dynamic thread scheduling, and work stealing. We discuss the internal storage representation of GRACE in detail in Chapter 6.

RINGO (Perez et al., 2015) is an interactive graph exploration system built for big-memory machines that can store *all-but-the-largest* graphs in main memory. Internally, RINGO is based on SNAP (Leskovec and Sosič), a parallelized graph library with over 200 built-in graph functions, and exposes the functionality through a PYTHON frontend. RINGO supports several efficient loading and transformation mechanisms, i.e., to load and process graph data that is initially stored in relational tables.

While most graph processing systems are able to process immutable graphs solely, KINEOGRAPH (Cheng et al., 2012), IMMORTALGRAPH (Miao et al., 2015), and CHRONOS (Han et al., 2014b) are tailored to storing and processing *time-varying* graphs that evolve over time. A time-varying graph captures all changes made to the structure and the attributes of the graph and their time stamps. Such a graph is then a collection of *graph snapshots*, where each snapshot corresponds to the static graph at a given point in time.

Recent advances in hardware/software co-design also have an impact on the development of graph processing systems. Nelson et al. (2011) emulate a multithreading system like the Cray XMT on commodity hardware to hide the memory latency problem prevalent in many graph algorithms through massive parallelism. Other examples that leverage modern hardware to accelerate graph processing include *processing-in-memory* (PIM) as

proposed by Ahn et al. (2015) and *field-programmable gate arrays* (FPGA) (Nurvitadhi et al., 2014).

2.6.2 *Distributed Graph Systems*

The abundance and diversity of massive-scale, graph-structured data and the ever-growing interest of large enterprise companies to analyze them are the key drivers of the recent advances in graph data management research. From a systems perspective, there is a plethora of distributed graph processing systems tailored to different use cases and exposing various programming models to choose from (Kang et al., 2009; Stutz et al., 2010; Malewicz et al., 2010; Gonzalez et al., 2012, 2014; Kang et al., 2011a; Khayyat et al., 2013; Low et al., 2012; Murray et al., 2013; Salihoglu and Widom, 2013; Sarwat et al., 2012; Seo et al., 2013; Shao et al., 2013; Tian et al., 2013; Wang et al., 2013; Bu et al., 2014; Fli; Roy et al., 2013).

Graph Processing based on MapReduce

Two of the first representatives of distributed graph processing systems are PEGASUS (Kang et al., 2009) and GBASE (Kang et al., 2011a), which run on top of the HADOOP ecosystem and are based on the MAPREDUCE programming model to formulate graph queries. PEGASUS and GBASE both rely on *generalized iterative matrix-vector multiplications*. Although the original MAPREDUCE programming paradigm has proven to significantly simplify batch processing tasks, iterative computations, as required by many graph algorithms, are not well-supported due to loops of MAPREDUCE jobs with spilling to disk in between consecutive jobs.

Vertex-Centric Graph Processing Systems

Based on the limitations of the MAPREDUCE programming paradigm, Low et al. (2010) introduced GRAPHLAB, a parallel programming framework for iterative, asynchronous computations, as they are prevalent in machine learning and graph algorithms. There are two main computations in GRAPHLAB, namely *update* and *sync*. While the update function performs stateless, local computations on the neighborhood of a vertex in the data graph, a sync function describes a global aggregation operation. Both, the update and the sync function, can run concurrently in GRAPHLAB, where a task scheduler selects, based on the selected scheduling strategy (synchronous or asynchronous) and the desired data consistency policy, the most suitable scheduling of function calls. The general idea of GRAPHLAB has been extended to run on large cluster setups Low et al. (2012). POWERGRAPH, an extension of GRAPHLAB, specifically targets the skew in the vertex degree distribution and the resulting imbalanced computation and resource utilization (Gonzalez et al., 2012). POWERGRAPH can eliminate the dependency on the degree of the vertex by directly exposing the *gather* and *scatter* operations.

With the advent of the *vertex-centric programming model*, as introduced by Malewicz et al. (2010), numerous systems have been developed that improve distributed graph processing on a large cluster of commodity machines. The vertex-centric programming model performs a graph computation in so-called *supersteps*. In each superstep, each vertex in the graph receives messages from adjacent vertices, performs a local computation, and generates new messages to be sent to adjacent vertices. There is a global synchronization barrier at the end of each superstep guaranteeing that all local computations finished, before the next iteration can start. This computation model is inspired by Valiant's *bulk-synchronous programming model* (Valiant, 1990), which simplifies local parallelization through independent computations on the vertex level. There are multiple efforts to provide an open-source implementation of PREGEL, including APACHE GIRAPH (Gir) and GPS (Salihoglu and Widom, 2013). GPS extends the original proprietary PREGEL implementation with a `master.compute()` function to simplify the development of graph algorithms that require multiple vertex-centric computations and global computations. Further, GPS dynamically

repartitions the graph to achieve an optimal load balancing during the computation. GPS handles graphs with a skewed degree distribution through the partitioning of large adjacencies across multiple compute nodes. To simplify writing programs for GPS, a backend for GREENMARL has been added (Hong et al., 2014).

To cope with the problem of a large number of messages being transferred between computing nodes and the slow convergence of the vertex-centric model, Tian et al. (2013) propose the *think-like-a-graph* programming model. The *graph-centric* approach is in contrast to the *vertex-centric* employed by the majority of distributed graph processing systems. The authors implemented the *graph-centric* programming model in GIRAPH++, an extension of APACHE GIRAPH, allowing the user to fully control and exploit the local structure of a graph partition. GIRAPH++ supports asynchronous, vertex-centric processing as well as graph mutation, which is often required in graph summarization and coarsening algorithms.

Khayyat et al. (2013) introduce MIZAN, an adaptive PREGEL-based system that proposes fine-grained load balancing during supersteps based on collected computation statistics on a vertex level. This is in contrast to an initial static partitioning of the data graph, where graph-specific properties and communication patterns of the graph algorithm have to be known in advance.

Similar to the asynchronous execution mode provided in GRAPHLAB, Wang et al. (2013) introduce asynchronous graph processing in GRACE[6]. In contrast to GRAPHLAB, which exposes the asynchronous execution model directly to the end user—burdening the application developer with the handling of concurrency issues—GRACE exposes a synchronous programming model and executes the program in an asynchronous execution mode. They achieve this by allowing the user to relax certain constraints on the message passing phase. In a follow-up publication, the same authors extend GRACE to improve the execution performance for *computationally light* applications that only perform a few operations per vertex update (Xie et al., 2013). For this kind of graph algorithms, adding more hardware resources, i.e., by providing more hardware threads, does not improve performance, since the algorithm is bounded by the available memory bandwidth. To solve this issue, the authors propose to group vertex updates together into *block updates* and parallelize on the level of a block.

Han and Daudjee (2015) propose a new programming model coined *barrierless asynchronous parallel* (BAP) that improves the *asynchronous* programming model by eliminating the need for distributed locking and by reducing the frequency of global barriers. Their implementation extends APACHE GIRAPH and implements the complete synchronous PREGEL API, but executes the program asynchronously.

Xie et al. (2015) build on the ideas of Khayyat et al. (2013) and rely on collected execution statistics, including the convergence speed and the number of active vertices, using online sampling and offline profiling. In a detailed study they show that *synchronous* and *asynchronous* can supersede each other, depending on the graph algorithm and the input data. Even worse, graph algorithms, such as single-source shortest path, demand an adaptive switching between the two execution modes to achieve the optimal execution performance. The authors extend POWERGRAPH and implement *hsync*, an adaptive vertex-centric graph processing framework that can automatically switch between the two execution modes during runtime.

To cope with the load imbalance of vertex-centric computations on power-law distributed graphs, Chen et al. (2015) propose POWERLYRA, a distributed graph system, which performs a differentiated processing on high-degree and low-degree vertices. Contrary to other distributed graph systems, which process all vertices similarly, POWERLYRA uses a hybrid-cut partitioning algorithm. More specifically, POWERLYRA distributes low-degree vertices and accompanying edges among machines (similar to edge-cut partitioning) and distributes edges of high-degree vertices among machines (similar to vertex-cut partitioning).

6 not to be confused with the single-node system GRACE developed by Microsoft

Graph Processing on Dataflow Systems

Recently, distributed dataflow frameworks, such as SPARK (Zaharia et al., 2010) and NAIAD (Murray et al., 2013), gained popularity not only for relational data processing but also as distributed graph processing systems. GRAPHX is an extension of SPARK and adds graph processing capabilities on top of SPARK's dataflow engine (Xin et al., 2013; Gonzalez et al., 2014). The GRAPHX abstraction employs a variation of the vertex-centric programming model and extends it by the *gather-apply-scatter* decomposition of GRAPHLAB. GRAPHX performs query processing on SPARK *collections* that hold vertices and edges and reuses existing dataflow operators, including *join*, *group-by*, and *map*.

NAIAD (Murray et al., 2013) is a distributed dataflow engine specifically designed for iterative and incremental computations. It employs *timely dataflow*, a computation model that does not only capture the topology of the flow graph, but also permits cyclic flow graphs with structured (nested) loops. The foundation for this are *logical timestamps*, which not only keep the timestamp itself, but also information about the loop progress through lightweight *loop counters*.

PREGELIX (Bu et al., 2014) is a PREGEL-based implementation on top of the HYRACKS general-purpose dataflow execution engine. It reuses the existing data-parallel operators of HYRACKS and implements the full PREGEL interface, including in-memory and out-of-core processing.

Other Graph Processing Systems

While most graph processing systems target offline graph analytics, HORTON (Sarwat et al., 2012) is a distributed graph processing system for interactive graph query processing. HORTON employs a query language to express simple patterns on the data graph and an accompanying graph query optimizer, which selects from an enumeration of possible execution plans the most cost-efficient one.

TRINITY (Shao et al., 2013) is a distributed graph processing system that combines *online query processing* with *offline graph analytics* within the same query engine. At the storage layer, TRINITY keeps the graph in an in-memory distributed key-value store that is shared across all computing nodes in the cluster. TRINITY supports online querying in the form of traversal queries and subgraph pattern matching as well as offline graph analytics by exposing a vertex-centric programming model.

Seo et al. (2013) extend the SOCIALITE system to also run in a distributed environment. SOCIALITE provides DATALOG extensions for graph analytics tasks and relies on *recursive monotone aggregate functions* using *semi-naive* query evaluation. SOCIALITE allows the user to define the data partitioning across the different machines and automatically infers the necessary distributed computation and communication from the partitioning and the given DATALOG program.

Recently, various studies on the scalability and performance of several distributed graph processing systems (Lu et al., 2014; Han et al., 2014a; Satish et al., 2014). Although distributed graph engines show good scalability to large graphs, properly implemented out-of-core graph systems can perform similarly well. Interestingly, even single-core, SSD-based implementations can outperform distributed graph processing systems for certain scenarios, which is an indicator of the system complexity of distributed systems and the added overhead caused by the programming model and communication/synchronization costs (McSherry et al., 2015).

2.6.3 *Graph Processing in RDBMS*

Traditionally, RDBMS have been used extensively to store RDF data either in a single, large triples table with three columns for subjects, predicates, and objects or in so-called *property tables* (Abadi et al., 2007; Sidirourgos et al., 2008; Neumann and Weikum, 2010; Erling, 2012; Bornea et al., 2013). A property table reassembles semantic entities from the RDF data model into database records and groups attributes that are exposed frequently together

in a single table. Abadi et al. (2007) propose to vertically partition the RDF data into two-columnar tables, one for each distinct property in the data set. They use C-STORE, an open-source columnar DBMS prototype, and showcase that a vertically-partitioned storage representation of RDF data can outperform a naive triple table.

VIRTUOSO (Erling, 2012) is a hybrid columnar relational/graph DBMS with a SPARQL interface. Internally, VIRTUOSO translates SPARQL queries into SQL queries with graph-specific extensions, for example the `transitive`[7] keyword enabling transitive closure computations on a tailored operator.

Sakr (2009) investigates how an RDBMS can be leveraged to store and process graph data efficiently. In contrast to other related work, they assume the graph data to contain a large number of small graphs compared to a single graph with a large number of vertices and edges. They employ several query optimization techniques, such as join ordering, pruning to reduce the search space, and filter evaluation accelerated by secondary index structures (Sakr and Al-Naymat, 2010b).

Bornhövd et al. (2012) propose an extension to SAP HANA in the form of a schema-flexible data management system that is based on a graph data model. The graph data model is similar to the property graph data model, but extends it with optional semantic type information, which can be associated with vertices and edges. The graph can be queried using WIPE, a declarative query language with built-in support for graph traversals and set operations. WIPE is internally translated into a set of SQL statements and issued against the underlying RDBMS engine.

SQLGRAPH (Sun et al., 2015) is an extension to a row-oriented RDBMS that provides graph querying and updates without side effects based on GREMLIN, a traversal-based query and manipulation language. SQLGRAPH translates GREMLIN on the basis of pipes and represents each pipe as a table—either a materialized table or a common table expression. A query builder translates GREMLIN queries into a combination of pure SQL functions, user-defined functions, common table expressions, and stored procedures. The latter ones are only used for graph updates without side effects and for recursive traversal queries without a recursion depth boundary. The authors propose several optimization and rewriting techniques, such as predicate rewriting to avoid unselective filter pipes and the invalidation of to-be-deleted vertices (instead of deleting them right away).

TERADATA ASTER (Simmen et al., 2014) is a large-scale analytics platform that supports analytics on multiple types of data, ranging from structured data to unstructured data. The primary focus of the extension for graph processing is on graph analytics through an iterative vertex-oriented JAVA-based programming abstraction that can be used to implement bulk-synchronous graph algorithms. All graph analytics functions (pre-built as well as custom implementations) are modeled as a polymorphic table operator, can be invoked directly from a SQL query, and can operate on tables, files, and other data formats that are accessible from the TERADATA ASTER storage interface. TERADATA ASTER supports the combination of analytics functions, such as the combination of graph analytics with sentiment text analytics, and can ingest graph data via graph projections from a wide variety of data sources.

Although RDBMS have proven to be competitive with specialized GDBMS if properly tuned, there are still use cases where it might be sensible to migrate the relational data into a GDBMS. De Virgilio et al. (2013) investigate techniques to migrate data as well as queries from an RDBMS to a GDBMS. Attributes that are often accessed together are represented as vertex attributes and queries are transformed from SQL into a *traversal-oriented* query language that is inspired by XPATH.

Increasingly, RDBMS functionality, including recursive queries, common table expressions, and secondary index structures, is extensively used to process several graph algorithms. Examples include the evaluation of single-source shortest path (Gao et al., 2011; Welc et al., 2013), subgraph isomorphism queries (Gubichev and Then, 2014), and traversal/reachability queries (Ordonez et al., 2014).

7 http://www.openlinksw.com/dataspace/doc/vdb/weblog/vdb%27s%20BLOG%20%5B136%5D/1435
(Last accessed: April 2017)

A slightly different path is followed by Sakr et al. (2012). They store a graph in a hybrid storage, where the graph topology resides in memory and the attributes of vertices and edges are stored in an RDBMS. While this solution might work for some use cases, transactional guarantees are challenging, since the in-memory representation entirely lacks a transaction context. Query processing is split on top of the RDBMS and relational parts of the query are pushed down to the RDBMS execution engine while topological queries are evaluated on the in-memory representation of the graph topology.

The recent advances in distributed graph processing and the growing interest in providing a high-level abstraction based on the *vertex-centric* programming model triggered the development of extensions to popular RDBMS with support for vertex-centric computations (Jindal et al., 2014a,b; Fan et al., 2015). VERTEXICA is an extension of VERTICA, a columnar RDBMS and enables vertex-centric computation through a mapping onto stored procedures and SQL queries. Similarly, Fan et al. (2015) extended SQL SERVER with support for a vertex-centric computation model. Both solutions rely on materialized, tabular intermediate results and implement message passing through shared message tables.

Recently proposed novel join algorithms, such as LEAPFROG TRIEJOIN and MINESWEEPER, achieve *worst-case optimality* or sometimes even *beyond worst-case optimality*. Nguyen et al. (2015b) investigate whether these new join algorithms can speed up graph pattern matching and close the performance gap to specialized graph processing systems (extended version: Nguyen et al. (2015a)).

Path traversals implemented as recursive queries in RDBMS have been in the focus of research for more than 20 years now (Agrawal, 1988). There have been proposals for extending relational query languages with support for recursion in the past and even the SQL:1999 standard offers recursive common table expressions. However, commercial database vendors often provide their own proprietary functionality, if they do at all. Magic-set transformations are a query rewrite technique for optimizing recursive and non-recursive queries, which was originally devised for DATALOG (Bancilhon et al., 1986) and has been extended for SQL (Mumick and Pirahesh, 1994). Since graph traversals can be expressed as recursive database queries, the magic-sets transformation could also be applied to them.

Trissl (2012) discusses graph processing in the context of RDBMS and employs a cost-based optimizer, graph operators, including a reachability operator, a distance operator, a path operator, and a path-length operator. The implementations of all four operators heavily rely on a secondary index structure, entitled GRIPP, which provides a concise representation of the graph topology.

SAP HANA is the first full-fledged RDBMS that tightly integrates graph processing into the database kernel allowing to seamlessly combine relational and graph operations within the same database engine (Färber et al., 2012; Bornhövd et al., 2012; Rudolf et al., 2013). SAP HANA is extended by a set of graph-specific operators, such as graph traversals, that are not mapped to relational operators but instead implemented as core operators in the RDBMS execution engine (Paradies et al., 2015).

2.6.4 *Graph Database Management Systems*

A different direction is followed by GDBMS, such as NEO4J (Robinson et al., 2015), SPARKSEE (Martínez-Bazan et al., 2007, 2012), INFINITEGRAPH (Inf), TITANDB (Tit), and multi-model DBMS, such as ORIENTDB (Ori) and ARANGODB (Ara). While NEO4J relies on a disk-based storage accelerated by a buffer pool to store recently accessed parts of the graph, SPARKSEE allows manipulating and querying the graph in memory. NEO4J is fully transactional and provides support for the BLUEPRINTS API, GREMLIN, and its own declarative, pattern-matching-based query language CYPHER.

SPARKSEE is a fully transactional GDBMS with out-of-core capabilities. The SPARKSEE internal data structures rely on efficient bitmaps, which represent the set of vertices and edges describing the graph (we discuss the storage representation in detail in Chapter chapter 6) and additional index structures can be specified to accelerate attribute access operations.

Although SPARKSEE does not offer a graph query language, it implements a BLUEPRINTS backend and thereby automatically supports GREMLIN.

Although ARANGODB is mainly a document-oriented DBMS, it allows providing keys to documents and also connecting documents with each other (graph). ARANGODB is fully transactional and provides a dedicated declarative query language AQL, which is derived from SQL and inherits many concepts, such as joins, aggregations, results projection/filtering, sorting, grouping, unions, and intersections. For convenient access to document structure, AQL can traverse documents and the edges between them as well as iterate over lists. ARANGODB can be extended through JAVASCRIPT programs that are executed similarly to stored procedures in the database kernel.

ORIENTDB is in spirit similar to ARANGODB and is also a multi-model DBMS storing documents, key-value pairs, graphs, and objects. In contrast to ARANGODB, ORIENTDB extends SQL with native support for graph traversals. Further, ORIENTDB is fully compliant with APACHE TINKERPOP and provides as a complementary query language for graphs native support for GREMLIN.

TITANDB is a distributed GDBMS with transactional guarantees, either fully ACID or eventual consistency, and runs on a cluster of machines. It supports several storage backends—APACHE CASSANDRA, APACHE HBASE, ORACLE BERKELEYDB —which can be chosen according to the desired properties following the CAP theorem. Similar to the other GDBMS, TITANDB provides a native integration with the APACHE TINKERPOP stack, including the GREMLIN query language and the BLUEPRINTS graph API. Similar to TITANDB is INFINITEGRAPH, another fully ACID-compliant distributed GDBMS. INFINITEGRAPH stores the graph natively and exposes GREMLIN to the end user.

2.7 SUMMARY

In this chapter we gave a broad overview of graph data management, including a discussion on graph data models, graph query languages, and graph algorithms. Moreover, we gave a detailed overview of available graph processing systems, ranging from single-node systems to distributed, cluster-based graph processing systems.

Based on the gained insights from the discussion of related work in this chapter, in the following chapter we emphasize the need for a novel system architecture that tightly integrates graph processing capabilities into an RDBMS, devise requirements that have to be fulfilled by such a system, and finally introduce our system proposal.

3 REQUIREMENTS AND SYSTEM OVERVIEW

In this chapter we propose and discuss the set of requirements that have to be met by a graph processing system (GPS) to be used in a representative enterprise data management landscape as sketched in Chapter 1. In particular, we propose a set of functional and non-functional requirements for a graph storage component and an accompanying low-level graph access layer as part of an artificial GPS. Finally, we introduce GRAPHITE, our prototypical graph processing system that aims at fulfilling all these requirements.

3.1 SYSTEM REQUIREMENTS

3.1.1 *Functional Requirements*

In the following we describe the functional requirements that we anticipate for a graph storage design and define a set of low-level graph access and manipulation operations. We use the *Property Graph Model* as the underlying data model, which we introduced in Chapter 2.1. The property graph model exposes a multi-relational graph model, i.e., multiple edges between the same pair of vertices and of the same edge type are allowed. A vertex can be uniquely identified by its vertex identifier. An edge has no system-provided unique identifier—the user, however, can specify an artificial unique edge identifier.

Query Operations

We distinguish two general classes of low-level read operations on graphs: (1) graph topology queries and (2) vertex/edge attributes queries. In Table 3.1 we list fundamental operations to get access to the graph topology and the attributes of vertices and edges, respectively. To specify the operations, we use a simplified syntax inspired by GEM[1], the graph query language of SAP HANA (Bornhövd et al., 2012). A read operation can expose the following result set types: (1) a single vertex or edge, (2) a set of vertices or edges, (3) a single vertex/edge attribute value, and (4) a set of attribute values. For performance reasons, more complex types of operations, such as graph traversals and shortest path computations, and result sets, such as paths and subgraphs, could be also introduced. In our prototypical implementation, however, they can be easily mimicked and built on top of the described fundamental read operations.

An attribute access operation returns the attribute value for a given vertex/edge and attribute name. A vertex/an edge *exposes* an attribute, if the corresponding attribute value is not `NULL`. Optionally, the user can return the values of all exposed attributes for a single vertex/edge. The result set of attribute values is represented as a set of records with one record per requested vertex/edge.

The evaluation of existential predicates, which probe the graph for the existence of an entity (a vertex or an edge), is a fundamental query type required in many graph applications. The evaluation of an existential predicate returns *true*, if the queried entity is present in the graph, *false* otherwise.

Topological operations, i.e., queries that return the adjacent vertices for a given vertex or a set of vertices, are a fundamental building block of many graph algorithms, including breadth-first and depth-first traversals. A topological query is a 1-hop traversal via incoming (`$v<--`) or outgoing (`$v-->`) edges. Further, given an edge, the attached head and tail vertices can be returned.

Predicate evaluation is an important operation for subgraph isomorphism queries on graphs with attributes associated with vertices and edges. A predicate filter can be either

1 The former graph query language WIPE has been recently renamed to GEM.

Table 3.1: Overview of fundamental read operations that should be supported by a graph storage.

Operation	Description
`$v@a`, `$e@a`	Given a vertex `$v` (an edge `$e`), return the value of attribute `a`
`$v@*`, `$e@*`	Given a vertex `$v` (an edge `$e`), return all attribute values
`EXISTS($v)`, `EXISTS($e)`	Given a vertex `$v` (an edge `$e`), check whether `$v` (`$e`) exists
`EXISTS(EDGE($u,$v))`	Given two vertices, check whether there exists (at least) one edge from `$u` to `$v`
`$v-->`	Given a vertex, find all vertices via outgoing edges
`$v<--`	Given a vertex, find all vertices via incoming edges
`HEAD($e)`, `TAIL($e)`	Given an edge `$e`, return connecting vertices (either head or tail vertex)
`FILTER($VERTICES, p)`	Given a predicate `p`, return all matching vertices
`FILTER($EDGES, p)`	Given a predicate `p`, return all matching edges

evaluated on the complete graph (`$VERTICES,$EDGES`) or on a vertex/edge subset. We define a predicate as a propositional formula consisting of atomic attribute predicates that can be combined by the logical operators $\wedge$, $\vee$, and $\neg$. The result type of a predicate operation is the set of matching vertices or edges.

Manipulation Operations

The ability to handle graph topologies where the structural shape of vertices and edges evolves over time is a fundamental requirement of an operational DBMS. *Structural changes* describe modifications to the structure of a single vertex/edge, i.e., the addition, manipulation, and removal of attributes on vertices and edges. Especially use cases that target social networks and communication networks demand a flexible and scalable graph persistence that can cope with frequent updates to vertex/edge attributes as well as changes to the graph topology. By *topological changes* we refer to modifications of the graph topology, i.e., the addition and removal of vertices and edges. More specifically, we classify graph manipulation operations into two classes: (1) data definition operations (DDO) and (2) data manipulation operations (DMO).

A DDO modifies the structural shape of a vertex or an edge, i.e., by adding, changing, or removing attributes and by adding constraints to vertices or edges. Since a property graph instance does not necessarily expose an upfront rigid database schema definition, a flexible data storage that follows a *data-driven* storage model is desirable. A data-driven storage model does not require an upfront database schema definition, but instead derives required schema information during insert time. This contrasts with a traditional RDBMS following a *schema-first* approach, where a DDO operation is explicitly issued by the user before running any DMO operation. If the necessary data type information is missing or cannot be derived from a given value, a default data type should be used to store the value without losing information (e.g., as a `varchar`).

A DMO operation allows inserting and deleting vertices/edges and accompanying attribute values in the graph. We distinguish between *entity-level* and *attribute-level* graph manipulation operations and list them in Table 3.2. Besides the addition and deletion of vertices/edges, the graph storage should also support changing the attribute value of a specific vertex/edge or unsetting the attribute value. The insertion of a new vertex is composed of a unique identifier chosen by the application and a set of attribute key-value

Table 3.2: Overview of fundamental graph manipulation operations.

Operation	Description
`ADD VERTEX($v)`, `ADD EDGE($e)`	Addition of a vertex `$v` (edge `$e`)
`DELETE VERTEX($v)`, `DELETE EDGE($e)`	Removal of a vertex `$v` (edge `$e`)
`SET $v@a=val`	Addition/change of an attribute
`UNSET $v@a`	Unsetting an attribute

pairs—optionally enriched by information about the data type. An edge is composed of a tuple denoting the source vertex and the target vertex. Similar to vertices, attributes can be assigned to edges on a key-value pair level. Additionally, we define the following global constraints that have to be met during each vertex/edge insertion: (1) the vertex identifier has to be globally unique in the graph and (2) dangling edges (edges with a nonexistent source or target vertex) are not allowed.

Transaction Support

Since the graph storage will be part of an operational DBMS, it also has to provide transactional guarantees comprising atomicity, consistency, isolation, and durability. Specifically, we target *snapshot isolation*, i.e., a guarantee that all query operations see a consistent snapshot taken at the beginning of the transaction. The most popular concurrency control method is *multiversion concurrency control*, which provides *point-in-time* consistent views and is implemented in most commercial RDBMS products.

Interplay with Data Types and Models

We anticipate a tight integration into the existing database kernel of an RDBMS to leverage available data types, such as ordinary data types, geospatial, JSON, and text data types. The graph storage shall be able to store attributes along with vertices/edges and thereby ideally reuse already existing storage containers. For example, geospatial data is usually stored in a binary, interpreted format. From a data model perspective, we wish to integrate the property graph data model with the relational and the temporal relational/graph data model.

The graph storage shall be able to process all available data types and seamlessly combine operations on different data types within a single graph query. For example, a graph traversal operation on a road network might be restricted by a geospatial bounding box predicate limiting the traversal to a specific geospatial region in the graph. All graph operators should support different output formats, including graph-oriented and relational output formats. A relational output format guarantees that the result of a graph operator can be consumed by the subsequent (possibly relational) plan operator.

Figure 3.1 depicts a physical execution plan that combines relational operators with graph operators in a single execution plan. The query fetches data from the vertex and edge column groups, performs an initial predicate evaluation on vertices and edges based on some selection criteria, feeds the intermediate results—a subgraph—to the graph traversal operator, which then performs a breadth-first search returning visited vertices and their level of discovery. The output of the traversal operator can be joined with the vertex table and finally projected to some selected attributes, such as first name and birthday.

Deep Integration

The graph storage should not be an isolated component in the RDBMS, but instead be integrated and interoperable with core database components on different layers in the database architecture. From a query language perspective, we want to be able to inject

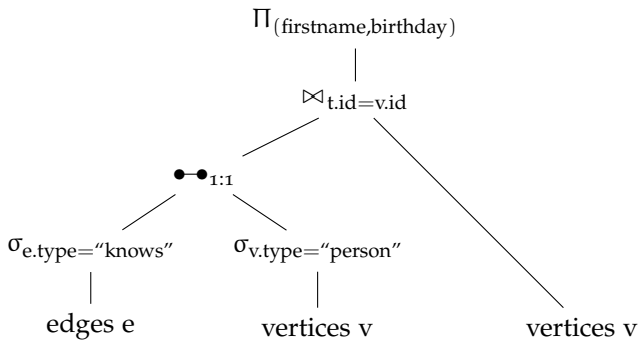


Figure 3.1: Heterogeneous execution plan with relational and graph operators.

graph queries as subqueries into a relational SQL query and vice versa. Further, metadata information, including catalog information and granted privileges, should be shared between the RDBMS and the graph storage. We aim at a holistic solution to offer a deep integration of graph processing with other data models and data types, such as relational, spatial, temporal, and text. Each data type and data model have own plan operators that should be composable within unified, heterogeneous logical and physical query execution plans.

On the query optimization layer, we aim at extending the query optimizer of the RDBMS to become *graph-aware*, i.e., to treat graph operators as first class citizens in the query optimization process and to use graph-specific statistics, such as degree distributions, the graph diameter, and centrality measures, for cardinality estimation of graph operators. If a graph query consists of operations that are not graph-specific—for example, order by, top-k, and selections—all graph operators have to be configurable to return a relational, tabular output structure to be consumed by the subsequent operator in the execution plan. We also envision basic operator push-down into graph operators, such as the push-down of a relational predicate into a graph traversal operator.

3.1.2 *Non-Functional Requirements*

Besides functional requirements, we pose a set of non-functional requirements to a graph storage and discuss them in the following.

Resource Efficiency

Based on the assumption that the graph data already resides in relational tables, the graph storage should be space-efficient, i.e., it should reuse the primary storage representation of the graph where possible and avoid replicating data into specialized data structures. Since our system should reside on a single machine, an efficient graph representation is essential to be able to store even large graphs of multiple hundreds of giga bytes in the memory of a single machine. If the uncompressed graph does not fit into memory, the graph storage should offer lightweight compression techniques, such as dictionary encoding or delta encoding, to compact reoccurring attribute values and low dimensional attributes.

Saving memory bandwidth is an important design goal for memory-resident data structures. We aim at providing CPU-friendly algorithms that make efficient use of the memory hierarchy, vectorization, and efficient processing on numerical values. Further, the data should be organized in such a way that processing it allows efficient memory prefetching into the caches of the CPU, avoid branch mispredictions in code where possible to improve out-of-order execution, and increasing spatial and temporal data locality of memory accesses.

Extensibility

The graph storage should be extensible such that specific components can be exchanged with alternative implementations, such as enhanced adjacency list handling based on the neighborhood degree and distribution. There should be a minimal and well-defined programming interface that all graph storage instances have to implement. Further, the graph storage has to be extensible for novel data types, such as text, geospatial, and other complex, composed data types. Ideally, the graph storage should not only support the property graph data model but also simplified variations thereof, such as a graph topology without additional vertex/edge attributes. In general, the graph storage should not assume a specific shape of graph topology nor assume specific graph characteristics and optimize for them. Our aim is to offer a holistic solution targeting a large variety of graph applications. We envision, however, that graph-specific secondary index structures can be tailored to specific graph topologies and can be added using a plugin mechanism.

Scalability

We target large multi-core server machines with possibly several terabytes of available RAM and therefore see scalability to larger graph instances (in terms of the number of vertices/edges and attributes) as well as scalability to more available hardware resources as key requirements. Concerning graph size, we aim at providing a graph storage that can keep the largest freely available real-world graph data sets in the memory of a single large server machine. Further, all write-oriented data structures have to be multi-thread safe and accessible from multiple concurrent writers. Regarding multi-socket machines, we aim at minimizing cross-socket communication to avoid NUMA effects as much as possible.

Stability

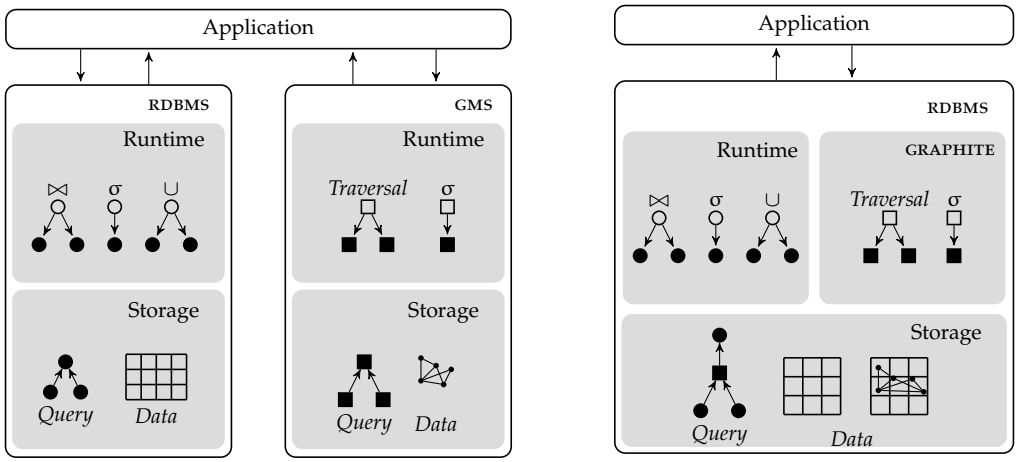
To rely on the mature components of an operational RDBMS and the expected higher system stability has several advantages over a green-field approach, especially for business-critical applications. First of all, a native operational graph processing system will likely duplicate the coding of a transaction manager of an RDBMS and consequently will result in a significant development and maintenance overhead. Our goal is to reuse existing components of an RDBMS where possible, especially for infrastructure-related tasks, such as transaction handling, logging, locking, recovery, and privilege granting and authorization.

In some places, however, we extend relational components with customized implementations, especially the graph processing stack and additional secondary index structures. This contrasts with related work, which either follows the green-field approach by implementing a native graph processing system—possibly with limited functionality compared to a mature RDBMS—or translates graph queries into their relational counterparts, resulting in poor performance compared to hand-crafted graph queries in specialized systems.

3.2 GRAPHITE SYSTEM OVERVIEW

There are two main approaches for adding graph processing capabilities to the operational data management landscape, namely by adding a native GMS next to an RDBMS or by syntactically extending an RDBMS with a graph abstraction.

While a GMS offers a native graph abstraction and data structures and algorithms tailored to graph processing, it is logically and physically separated from the RDBMS and can only run on a snapshot of the operational data (cf. Figure 3.2a). To close the functional gap between the operational RDBMS and a native graph processing system, recent works propose to add a query translation layer on top of the RDBMS (Sun et al., 2015; Jindal et al., 2014b). These extended RDBMS expose a graph programming model—based on pattern matching or on vertex-centric programming, which is accessible through an imperative API or a graph query language. Subsequently, the RDBMS translates the graph program into a relational execution plan, typically involving chained self-joins, recursive common



(a) Separated RDBMS and graph management system (GMS) with specialized storage engines and processing via application.

(b) RDBMS with integrated graph processing capabilities and a common storage engine.

Figure 3.2: Architecture alternatives for graph processing.

table expressions, and user-defined functions. Although in general these graph programs can be translated into SQL or a lower-level counterpart, they typically suffer from poor execution performance due to a suboptimal graph storage representation and inefficient representation of intermediate results in relational structures.

To overcome the disadvantages of the two main alternatives, we developed GRAPHITE, a hybrid relational/graph solution that combines the best of both worlds. GRAPHITE is a high-performance columnar graph runtime that is embedded into an operational RDBMS. The general architecture and the integration of GRAPHITE into an operational RDBMS are depicted in Figure 3.2b. One of the major design goals of GRAPHITE is to be embeddable into an RDBMS by selectively extending the database kernel with graph-specific components. GRAPHITE inherits many of the concepts that can be found in columnar in-memory RDBMS, such as dictionary encoding, column compression, vectorized execution, and late materialization. In contrast to a graph processing system residing next to an RDBMS, GRAPHITE is integrated as a runtime component into an RDBMS. Located next to a relational runtime stack, a graph runtime allows querying graph data on top of a common relational storage engine. Besides graph-specific operators, such as traversals, GRAPHITE facilitates graph index structures and extends the relational query optimizer by graph statistics and graph operator cost models. One of the advantages of this tight integration is the ability to express *cross-data-model operations* that combine data from different data models and types, such as the combination of relational, text, geospatial, temporal, and graph (Abadi et al., 2014) within a single query.

For example, a clinical information system might store patient records in an RDBMS. Providing graph analytics on the knowledge graph of patient records and their relationships helps physicians to improve diagnostics and identify complex co-morbidity conditions. Such a medical knowledge graph does not only contain information about the relationships between diagnoses and patients, but also unstructured data, such as text, from patient records and temporal information about prescriptions.

3.3 SUMMARY

In this chapter, we discussed a list of functional and non-functional requirements that have to be fulfilled by a GMS in a database management landscape as described in Chapter 1. We defined a minimal set of read and write operations as a low-level programming interface

to a graph storage and discussed several non-functional requirements, which arise from a tight integration into a heterogeneous system landscape.

Finally, we introduced GRAPHITE, a hybrid relational/graph solution that is embeddable into an operational RDBMS while still providing execution performance guarantees comparable to native graph processing systems. In the next chapters, we will look at several important building blocks of GRAPHITE, such as the graph storage, graph operators, index structures, and the query interface.

4 GRAPH STORAGE

In this chapter we propose a general graph data storage representation and an accompanying low-level programming interface. Specifically, we discuss several data reorganization techniques to improve the query performance and to lower the overall memory consumption. We concentrate our optimization efforts on the following performance goals: (1) maximizing spatial and temporal memory locality for data accesses to minimize data cache misses, (2) leveraging available multi-core parallelization on large server machines, and (3) minimizing the memory footprint of the graph, i.e., by applying data reorganization and compression techniques. In the experimental evaluation, we compare two in-memory graph representations with respect to query performance and memory consumption and evaluate our in-memory graph representation for read and write operations on several real-world graph and generated graph data sets.

4.1 RELATED WORK

In this section we extend the discussion of related work on graph processing in RDBMSs, in particular we review several storage alternatives for representing graph data in relational tables, and also discuss related compression techniques that are similar to the ones we propose in this chapter. For a detailed review of related work on native storage layouts for graphs, such as adjacency lists and matrices, we refer the reader to Chapter 6.1.

4.1.1 *Graph Processing in RDBMS*

Traditionally, RDBMS have been used extensively for storing RDF data either in a single, large triples tables with three columns for subjects, predicates, and objects or in so-called *property tables*. A property table reassembles semantic entities from the RDF data model into database records and groups attributes that are exposed frequently together in a single table. Abadi et al. (2007) propose to vertically partition the RDF data into two-columnar tables, one for each distinct property in the entire data set. They use C-STORE, an open-source columnar DBMS prototype, and demonstrate that a vertically-partitioned storage representation of RDF data can outperform a naive triple table implementation.

RDF-3X is a RISC-style DBMS designed and optimized for efficient RDF processing and provides a generic solution for storing RDF data, thereby eliminating the need for a physical design tuning (Neumann and Weikum, 2010). RDF triples are stored in a single, large triple table with three columns representing *subject*, *predicate*, and *object*.RDF-3X maintains an extensive set of index structures—implemented as clustered B+-trees—to index all possible permutations of subject, predicate, and object. Further, RDF-3X stores *aggregated indexes* storing only a composition of two of the three columns and for each entry the value occurrence of the pair in the full set of triples. Finally, RDF-3X maintains another three indexes for storing for each possible value of subject, predicate, and object the value count. All triples are dictionary encoded and stored in the leafs of the B+-trees. The dictionary encoding is implemented as a B+-tree to retrieve the value code for a value and as a direct mapping index to retrieve the value for a given value code. Triples on a single leaf page are further byte-compressed and delta encoded. Storing the triples sorted on a leaf page allows RDF-3X converting SPARQL patterns into a set of merge joins and simple range scans. RDF-3X maintains six differential indexes for all possible permutations of subject, predicate, and object. During the query processing, RDF-3X injects additional merge join operations into the execution plan to unite the partial results from the main indexes and the differential indexes.

RDF-3X uses staging to allow fast insertions of triples and buffers incoming triples in a session-private in-memory heap. When the operation finishes, the heap is written into globally shared differential indexes. A differential index is implemented as an uncompressed clustered B+-tree that is periodically merged into the main indexes. In-place updates in RDF-3X are implemented as a pair of a deletion and an insertion operation. To delete a triple, RDF-3X inserts the corresponding triple into the workspace of the query session and the differential indexes with a special deletion flag. During query processing, triples tagged with the deletion flag are ignored.

Bornea et al. (2013) propose a different storage representation of RDF data in a RDBMS by relying on an *entity-based approach*. They store RDF data in a *direct primary hash* table, in which each logical entity (a subject and all its associated predicates and objects) are stored in a single database record. Specifically, it contains a series of $\langle \mathtt{pred}_i, \mathtt{val}_i \rangle$ column pairs storing the predicate and the corresponding object. If a row is full, a *spill record* is created which contains the remaining predicate-object pairs. Multi-valued predicates are stored in a separate table, called *direct secondary hash*, and map an system-generated key representing the multi-value to a list of database records. For a multi-valued predicate, the direct primary hash table only contains a foreign key to the direct secondary hash table. The authors propose two techniques to minimize the number of required columns: (1) hashing predicates to columns, or (2) using graph coloring to assign predicates to columns. The overall optimization goal is to ensure that predicates that co-occur together are never assigned to the same database column. Similarly, they also store the reverse relationship—the incoming edges to a subject—by providing a *reverse direct primary hash* and a *reverse direct secondary hash*.

SQLGraph (Sun et al., 2015) is an extension to an RDBMS that provides graph querying and updates without side effects based on Gremlin, a traversal-based query and manipulation language. A graph is stored in a composite of relational and non-relational data types. For the adjacency list, they propose a relational storage layout and combine that with a storage layout for vertex and edge attributes that is based on JSON (cf. Figure 4.1). The adjacency list is conceptually derived from the RDF representation proposed by Bornea et al. (2013) and hashes multiple edge labels into a smaller set of columns to avoid sparsely populated columns. If there is more than one incoming/outgoing edge of a specific type, a secondary adjacency list is created and referenced in the primary adjacency list via a foreign key relationship. Consequently, each neighborhood lookup of a vertex requires a join of the primary and the secondary adjacency list. If there are conflicts during the hashing phase, SQLGraph creates a spill record, resulting in multiple rows representing the adjacency list of a single vertex. Vertex attributes are stored in the vertex table, which contains a unique primary key denoting the vertex identifier and a JSON column containing all attributes of that vertex. The edge table duplicates the information about the graph topology by representing each edge by its identifier, its source vertex, its target vertex, its edge label, and a set of attributes that is represented as a JSON object. To efficiently support Gremlin queries on this shredded graph representation, SQLGraph uses heavy indexing on the queried attribute keys, the vertex id primary key, the key of the secondary adjacency list, and the source and target vertex columns of the edge table.

TeraData Aster (Simmen et al., 2014) is a large-scale analytics platform that supports analytics on multiple types of data, ranging from structured data to unstructured data. A graph is hash-partitioned by vertex identifier and stored in graph partitions of vertex and edge tables. Each graph partition stores the vertex and all of its outgoing edges. In the case of an out-of-memory situation, TeraData Aster can spill graph data structures into a disk-based key-value store (LevelDB).

The AIS system proposed by Bornhövd et al. (2012) represents a graph as a set of *InfoItems*—the vertices—and a set of *Associations*—the edges. Each InfoItem can have an arbitrary number of attributes assigned and a semantic type, called *Term*. Data in AIS is stored in an RDBMS in a set of relational tables. In contrast to our approach, the authors rely on a schema-fixed storage representation and avoid explicit `NULL` value representations where possible. They propose a vertical schema layout, where attribute values are stored in so-

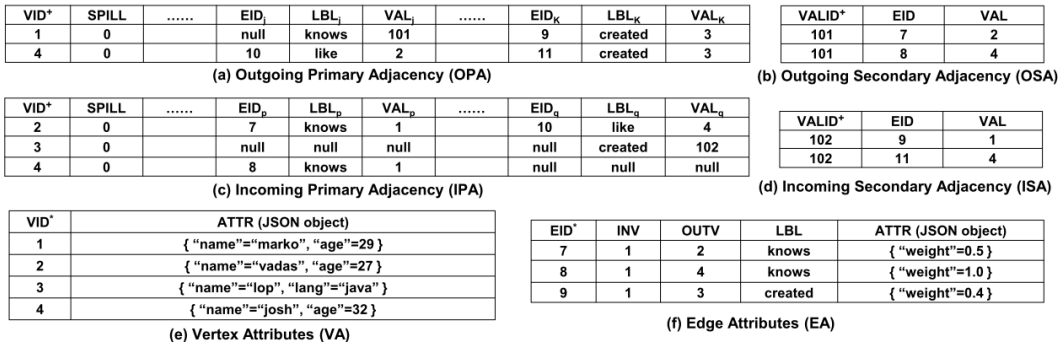


Figure 4.1: Example graph representation in SQLGRAPH (Sun et al., 2015).

called *value tables* that store triples of object identifier, attribute name, and value. Each technical data type has its own value table. In addition, Associations are stored in a second triple table consisting of source object, target object, and semantic type. In contrast to the general property graph model, Bornhövd et al. (2012) do not consider attributes on Associations.

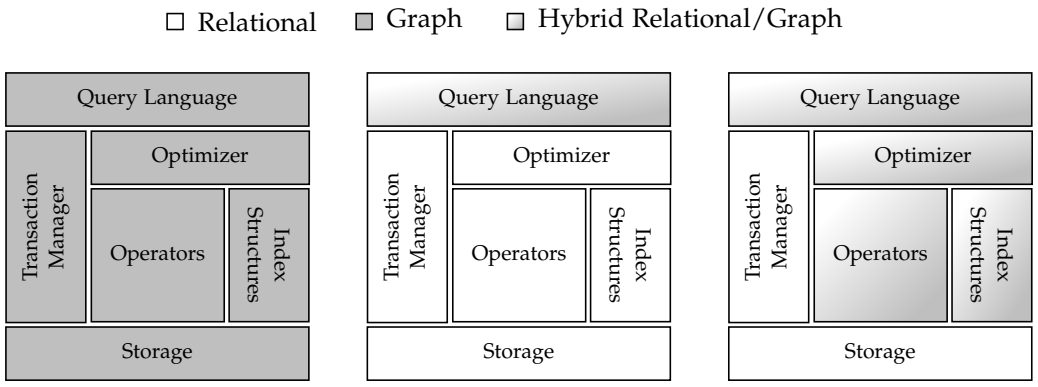
4.1.2 *Compression Techniques*

Several light-weight compression techniques for column stores have been proposed in recent years. Lemke et al. (2009) propose several compression techniques, including dictionary encoding, run-length encoding, sparse encoding, cluster encoding, and indirect encoding for the in-memory column store SAP HANA. Important database operators, such as scans and aggregations, can operate directly on the compressed data using SIMD instructions (Lemke et al., 2010). They apply column-wise compression and reorder rows in a table to improve the overall compression rate. Identifying the optimal row order that minimizes the memory consumption of the table is known to be an NP-complete problem (Alsberg, 1975). SAP HANA uses several heuristics to identify promising column candidates for reordering, and sorts rows so that blocks contain only few distinct values. The heuristics are based on collected statistics about the value distribution in the column—most common value, average value occurrence, number of distinct values—and select candidates one after the other. Thereby, the most promising candidate is selected first, a global row reordering is applied, and then the next iteration continues with the unsorted remainder in the second column. The benefit of compression therefore decreases with every selected column. Such a column-wise greedy strategy and reordering rows lexicographically makes this approach not well suited for wide tables with a large number of columns.

Abadi et al. (2006) propose complementary light-weight compression techniques to the work of Lemke et al. (2009) but do not discuss how row reordering can improve the compression ratio. Rather, they assume that the data is already sorted by the system to increase query performance. An active row reordering, however, is not provided.

Lemire and Kaser (2009) investigate the effect of sorting on the total number of runs of repeating values to improve RLE compression in columns. They show that for tables with uniformly distributed data sorting the columns in the order of increasing column cardinality is asymptotically optimal. Further, the authors evaluate other sorting criteria, such as ordering by gray codes and ordering by Hilbert space-filling curves. In the experimental evaluation, they show that a sorting criterion based on Hilbert space-filling curves is suboptimal for minimizing the number of runs within a column.

MULTIPLE-LISTS is another heuristic proposed by Lemire et al. (2012), which is based on the NEAREST NEIGHBOR heuristic. The NEAREST NEIGHBOR heuristic, which runs in $\mathcal{O}(n^2)$, chooses the next tuple to append to the table by computing a nearest neighbor function value. The core idea of MULTIPLE-LISTS is to solve the scalability problem of the NEAREST NEIGHBOR heuristic by reducing the number of potential neighbors for each tuple.



(a) Native graph system. (b) Graph on top of an RDBMS. (c) GRAPHITE.

Figure 4.2: Graph processing and different levels of integration into an RDBMS.

The authors propose to construct a graph representation from the table, where each row represents a vertex. An edge is constructed by sorting the table by several columns and subsequently connecting vertices (rows) that represent consecutive rows in each sort. The number of different orders k is an input parameter to the algorithm; If $k = c!$, where c is the number of columns, the MULTIPLE-LISTS heuristic is functionally equivalent to the NEAREST NEIGHBOR heuristic. The second heuristic proposed by Lemire et al. (2012) is VORTEX, which aims at forming long runs of repeating elements. The evaluation indicates that VORTEX shows promising results by improving RLE compression by a factor of 3.

Herrmann et al. (2014) tackle the problem of handling wide and sparsely populated tables from a different angle. Instead of relying on light-weight compression techniques, they propose to horizontally partition the table instead. Such a set of partitioned tables does not require a dedicated compression handling and query processing can be applied only to those partitions that exhibit the queried attributes. Incoming records are probed against all available partitions and a rating is computed for each partition. The record is inserted into the partition that received the highest rating. When even the highest rating is to low (negative), a new partition is created. If a partition is full, it is split into two new disjoint partitions. Computing these ratings is conceptually similar to our proposed reorganization technique, except we compute the partitioning only once upon creation of the read-optimized data structures and do not need to handle incremental updates to the database.

4.2 PHYSICAL GRAPH REPRESENTATION

In this section we discuss the physical storage representation of graph data in GRAPHITE. Figure 4.2 depicts the different levels of integration of graph processing capabilities into an RDBMS. Native GDBMS, such as NEO4J or SPARKSEE, provide their own storage representations that are tailored to storing graphs efficiently, own transaction management, graph-specific index structures, and a graph-aware runtime system (cf. Figure 4.2 (a)). In contrast to a native GDBMS, we discussed several approaches to perform graph processing on top of an RDBMS as shown in Figure 4.2 (b). For these approaches, all components except for the query language component are shared with the relational stack and the graph query language is translated into a sequence of SQL statements. Our approach in GRAPHITE (cf. Figure 4.2 (c)) combines both worlds by reusing transaction management and storage representation of an RDBMS and extending the system by graph-specific operators, index structures, and query languages.

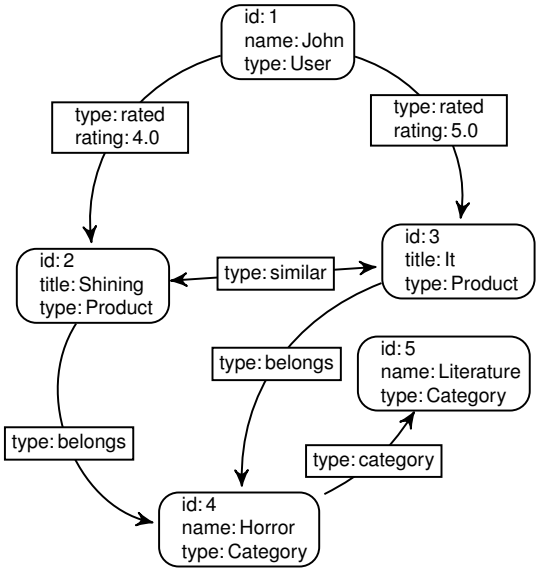


Figure 4.3: Example of a property graph instantiation representing products, users, and product ratings.

4.2.1 *General Storage Layout*

Graphite supports the property graph model, which has emerged as the de-facto standard data model for general purpose graph processing in enterprise environments (Rodriguez and Neubauer, 2010a). The property graph model describes a multi-relational, directed graph by a set of vertices and a set of edges. Both, vertices and edges, can have an arbitrary number of attributes assigned as key/value pairs. Figure 4.3 depicts an example graph representing data from a e-commerce platform, such as Amazon or Ebay. We represent users, products, and product categories as vertices and their relationships as directed and typed edges.

Since graph data usually has no upfront defined and rigid database schema, finding a suitable normalized database design for it is a nontrivial and tedious task. One approach to overcome this is to store vertices and edges in a single physical *universal table*, respectively. A universal table maps each distinct attribute to a separate column and each object—a vertex or an edge—is represented as a single record in the table. This allows reconstructing a vertex/an edge through a selection and a subsequent attribute projection. Such a join-free approach dramatically simplifies query processing as it completely eliminates join operations for constructing the final result set of vertices/edges. Besides for storing graph data, wide universal tables are commonly used for storing product classifications and product catalogs from online shopping portals as well as semistructured data, such as json, xml, or rdf data.

In Graphite we use a common storage infrastructure that is shared with the relational runtime stack and store a graph in two physical column groups, one for the vertices and one for the edges, respectively. A column group is a vertically partitioned physical universal table, where a new attribute can be added by appending a new column to the column group (Abadi, 2007). Figure 4.4 depicts an example representation of the vertices and edges from the example graph shown in Figure 4.3. We map each vertex and edge to a single entry in the column group and each attribute to a separate column. Each vertex has a unique identifier as the only mandatory attribute. An edge is drawn from the columns V_s and V_t that represent the *source vertex* and the *target vertex* of an edge, respectively. The edge direction is implicitly given by the assignment to the source and the target vertex column, respectively. We use a foreign-key constraint to guarantee that all source and target vertices already exist in the vertex column group to avoid dangling edges.

id	type	name	title	...
1	User	John	–	...
2	Product	–	Shining	
3	Product	–	It	...
4	Category	Horror	–	
5	Category	Literature	–	...

(a) Vertex column group.

V_s	V_t	type	rating	...
2	3	similar	–	...
2	4	belongs	–	
3	4	belongs	–	...
1	3	rated	5.0	
1	2	rated	4.0	...
4	5	category	–	

(b) Edge column group.

Figure 4.4: Mapping of a property graph to column groups.

Representing each distinct attribute as a separate column allows the application to use all available basic and composite data types in the RDBMS. Further, by reusing the relational storage infrastructure, we can directly benefit from available data compression techniques, transaction management, crash recovery mechanisms, and efficient predicate evaluation.

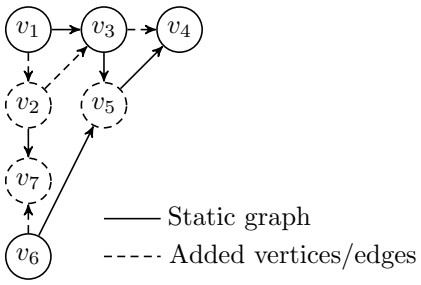
For heterogeneous data with a polymorphic set of entities, however, a column group can become quite wide as it represents each attribute as a separate column and most columns are sparsely populated. Even worse, a wide table is generally considered to be problematic for vertically partitioned tables due to the random access behavior for single-record selections and insertions. Further, a large number of sparsely populated columns sacrifices memory consumption for schema flexibility. We address both challenges, the wideness and the sparseness of column groups, in Section 4.3.1.

4.2.2 *Read- and Write-Optimized Storage*

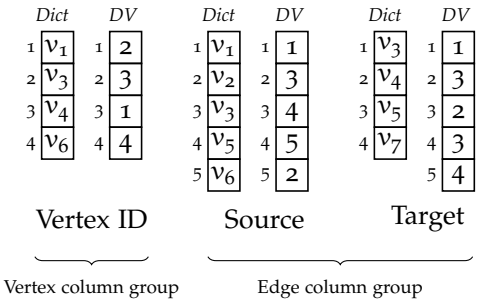
We divide the graph storage into a read-optimized and a write-optimized data container. Such a separation is commonly used in columnar RDBMS to allow fast read operations on the compressed read-optimized store while retaining a high data ingestion rate on the write-optimized store (Erling, 2012; Sikka et al., 2012; Raman et al., 2013; Larson et al., 2013). Therefore, we store a dynamic graph in GRAPHITE in two read-optimized column groups for vertices and edges, respectively, and two write-optimized column groups for vertices and edges, respectively.

Figure 4.5 depicts an example graph topology and the internal representation in GRAPHITE. The vertex column group consists of one column storing the vertex identifier. The edge column group stores the graph topology in the source and target vertex columns. In the example graph, solid lines indicate that the corresponding vertex/edge resides in the read-optimized column group, a dashed line indicates that the vertex/edge resides in the write-optimized column group. Since we separate the storage of vertices and edges, it is possible that although two vertices reside in the write-optimized column group, their connecting edge can reside in the read-optimized column group.

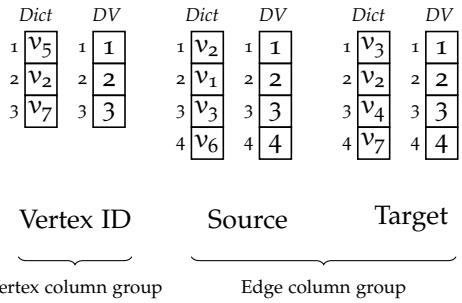
We employ two levels of data compression, the first level is dictionary encoding, the second level uses lightweight compression techniques to compact reoccurring values. We apply dictionary encoding on each column and map each distinct value in the column to a fixed-length numerical value code. Consequently, each column is a composite structure of a dictionary providing mappings between values and their corresponding value codes and a *data vector* only containing the value codes instead of the actual values. In a read-optimized column group, the dictionary is sorted to allow the efficient evaluation of range predicates and to enable binary search to locate the value code for a given value. To retrieve the value for a value code, we use direct addressing to fetch the corresponding value code. A dictionary creates a dense domain by guaranteeing that all value codes are drawn from $[1, |D|]$, where $|D|$ refers to the number of distinct values in the column. Since the data vectors contain only fixed-length numerical values, we can exploit vectorization capabili-



(a) Example graph partitioned across read- and write-optimized store.



(b) Read-optimized data store.



(c) Write-optimized data store.

Figure 4.5: Example dynamic graph and the data representation in GRAPHITE.

ties of modern processors, i.e., SIMD, to facilitate fast column scans (Willhalm et al., 2009). On the second compression level, we apply lightweight compression on the data vector, for example to compact reoccurring values through run-length encoding (cf. Section 4.3.1).

The read-optimized column group is immutable, i.e., all data insertions and updates are redirected to the write-optimized column group. We handle deletions through a validity vector, which indicates whether a record is visible and accessible in the specific transaction context. To guarantee fast query processing, we periodically merge the write-optimized column group into the read-optimized column group. During this so-called *delta merge*, all dictionaries and data vectors are recreated. The dictionary of the write-optimized column group does not rely on a sorted data organization, but instead provides an append-only interface and an additional B+-tree to accelerate the lookup of a value code for a given value.

4.3 GRAPH DATA REORGANIZATION TECHNIQUES

Our proposed columnar graph storage has two major shortcomings: (1) through the explicit representation of `NULL` values in sparsely populated columns, the memory consumption is higher than for an equivalent normalized database schema and (2) the materialization of a complete row is more expensive since more columns have to be accessed although most of them only contain `NULL` values.

4.3.1 *Graph Compression*

Lightweight compression algorithms, such as run-length encoding, sparse encoding, indirect encoding, and prefix encoding, are a key component of in-memory columnar RDBMS to lower the overall memory footprint and fully utilize the available memory bandwidth by compacting the data stored in columns. Typically, a columnar RDBMS compresses columns independently from each other by applying—depending on the data distribution and characteristics—the optimal compression algorithm. It is well-known that reordering rows within a table can improve the overall compression ratio significantly (Lemire et al., 2012).

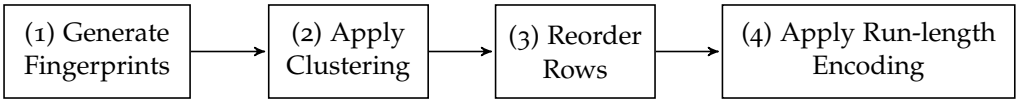


Figure 4.6: TETRIS workflow.

For wide and sparsely populated column store tables, we consider two major problems to solve: (1) lowering the memory footprint by compressing `NULL` values and (2) providing efficient access to all exposed attributes of an entity.

In this section we introduce TETRIS, a row reordering technique specifically designed for wide and sparse vertically partitioned tables. TETRIS runs as a preprocessing step before the actual compression and reorders rows for minimizing the memory footprint. We address both challenges by following the general idea of automatically detecting and deriving the logical semantic types of each entity in the table and by subsequently grouping records of the same type physically together. Compared to previous solutions, we do not analyze columns independently to find an optimal row reordering, but instead rely on fingerprinting to characterize each row in the table. Based on the collected fingerprints, we apply row clustering to group records with similar fingerprints together and to reorder the rows based on their cluster membership. Our experimental results demonstrate that this reordering approach achieves a significantly better compression ratio than conventional column-based approaches.

4.3.1.1 TETRIS *Workings*

TETRIS is based on a novel criterion for reordering rows to improve the overall data compression ratio on wide and sparsely populated tables. Examples for this type of table include graph applications, product catalog classifications, and shredding on non-relational data models, such as XML, JSON, and RDF. Since `NULL` values are the most frequent value in such scenarios, TETRIS compresses the table by producing long runs of `NULL` values within a column and by subsequently applying run-length compression on each column.

In contrast to traditional row reordering approaches based on lexicographical sorting and heuristics for candidate column selections, TETRIS relies on clustering techniques applied on row level. To find the optimal row reordering that enables a run-length based compression algorithm to produce maximal length runs of repeating values is known to be NP-hard.

Although our main goal is to improve the overall compression ratio, there are positive side effects that the RDBMS execution engine can leverage to improve the overall query performance. Naturally, TETRIS groups records with a similar set of exposed attributes close to each other in the table, resulting in a logical partitioning of records semantically belonging to the same type. This observation can be used to implement advanced scan routines that restrict the scan range to certain groups of records in the table. Further, materializing attribute values—for example for `SELECT * FROM MYTABLE` queries—the collected row fingerprints can be used to only materialize values that are different from `NULL`.

Figure 4.6 depicts the overall workflow of TETRIS. First, we generate for all records in the table a representative fingerprint, which is used in a subsequent step to apply the clustering algorithm by assigning each row to a cluster. In the reordering phase, we sort the table by cluster identifier and apply further sort order optimizations. Finally, we apply run-length compression on the sorted columns.

FINGERPRINT GENERATION. We generate for each record in the table a fingerprint that represents the set of exposed attributes. We use fingerprints to compute a normalized distance measure between any two records in the table. Naturally, a record fingerprint can be represented as a bitset, where each bit represents an attribute and the status of the bit b_i indicates whether the attribute is exposed ($b_i = 0$) or is `NULL` ($b_i = 1$). We use here the logical representation of a bitset and discuss the physical implementation

Row	Data	Initial values	Scan col. #1	Scan col. #2	Scan col. #3	Scan col. #4
1	a b - -	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0	0 0 1 1
2	- b c d	0 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0
3	- - a b	0 0 0 0	1 0 0 0	1 1 0 0	1 1 0 0	1 1 0 0
4	d - - -	0 0 0 0	0 0 0 0	0 1 0 0	0 1 1 0	0 1 1 1
5	f a - -	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0	0 0 1 1
6	- f d b	0 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0

Table 4.1: Column-wise scanning and construction of fingerprints.

alternatives in Section 4.3.1.3. Table 4.1 depicts the process of generating fingerprints for a table with six rows and four possible attributes. To compute the fingerprints, we scan the table column by column for each attribute A_i with a filter predicate "A_i `IS NULL`" and update the corresponding fingerprints after each scan. From the generated fingerprints, we derive information about the similarity of records in terms of exposed attributes. For example, records 2 and 6 expose the same set of attributes, records 4 and 5 differ in one attribute, and records 1 and 3 are dissimilar. The overall time complexity is $\mathcal{O}(|R| \cdot |A|)$ and the space complexity is $\mathcal{O}(|R| \cdot |A|)$ for $|R|$ referring to the number of rows and $|A|$ referring to the number of attributes.

ROW CLUSTERING. We apply the clustering on all generated row fingerprints and assign each record to exactly one cluster. As a result, all entities belonging to the same cluster expose a similar set of attributes and are more likely to represent entities of the same semantic type. Before we describe the applied clustering technique in detail, we discuss properties that the clustering algorithm in TETRIS has to fulfill. The clustering should be *partitional* and not *hierarchical*, i.e., no relationship between clusters and no containment information between clusters has to be maintained. We assign each record to exactly one cluster, resulting in an *exclusive* clustering. We define the clustering as a binary relationship between the rows and the clusters, i.e., a single row either belongs to the cluster or it does not belong to the cluster. The clustering is *complete*—we assign each record to exactly one cluster, even if the entity exposes all attributes. Finally, the clustering has to be able to generate groups with a potentially varying but fixed number of elements.

Based on the required clustering properties, we select two algorithms that are generally suitable for TETRIS: DBSCAN and k-means clustering. DBSCAN, however, has a time complexity of $\mathcal{O}(n^2)$, which makes it less suitable for tables with a large number of rows n. Contrary, the time complexity for k-means is $\mathcal{O}(qdnk)$, where q is the number of iterations, k is the number of clusters, n is the number of rows, and d is the number of columns. Further, DBSCAN relies on the input parameter ϵ, which is drawn from the range $[1, d]$. For epsilon $\epsilon = 1$, two fingerprints within the same cluster can differ by at most one position. For small values of ϵ, DBSCAN considers almost all entities as noise points and for a large value of ϵ, all entities are assigned to the same cluster. Due to the sensitivity of DBSCAN to the input parameter ϵ, it is difficult to achieve consistently good results. To this end, we choose k-means clustering to group similar records into clusters and describe in the following specific modifications that we did on the original algorithm to adapt it to TETRIS.

We use a *hamming distance function* $d(x, y)$ between two vectors x and y, where the distance is defined by the number of positions in which they differ. In TETRIS, the hamming distance is a measure to describe how similar two records are. In an information theoretical interpretation, the hamming distance describes the minimum number of substitutions required to change one vector into the other. For example, for two vectors $x =$ `001101`

Algorithm 1: Selection of initial center μ for cluster C_μ

```
Input  : X = {x_1, ..., x_n} : input samples
Input  : μ_1, ..., μ_{q-1} : existing cluster centers
Input  : q : cluster index
Output : μ_q : cluster center
1  begin
2      maxDist ← 0;
3      for i ← 1 to n do
4          minDist ← Dist(x_i, μ_1);
5          idx ← 1;
6          for j ← 2 to q − 1 do
7              if Dist(x_i, μ_j) < minDist then
8                  minDist ← Dist(x_i, μ_j);
9                  idx ← j;
10         if Dist(x_i, μ_index) > maxDist then
11             μ_q ← μ_index;
```

and $y = \texttt{010101}$, the hamming distance $d(x, y)$ is 2. We implement the computation of the hamming distance by computing the bit-wise `XOR` between the two arguments, followed by a count of the number of bits set in the result of the `XOR` operation.

The basic variant of the k-means algorithm selects the initial centroids randomly, which can result in a suboptimal clustering result as the quality of the clustering depends on the selection of the initial centers. In Tetris we select the initial centroids methodically by choosing k samples such that the total distance between them is maximized. More specifically, we select the value which maximizes the minimum distance to all other centers and repeat this procedure k times. Algorithm 1 illustrates our procedure for selecting initial centroids for k clusters.

In contrast to the basic k-means algorithm, Tetris does not require an upfront defined number of target clusters k, but instead adjusts the number of clusters automatically during the clustering phase. We use Bayesian statistics to dynamically determine the value of k. Bayesian statistics measures the relation—the distance—between two objects as a continuous value instead of a binary value. By the two objects, we refer to the entity to associate to a cluster and the candidate cluster. When the distance is larger than a configurable, system-internal threshold λ, the entity is considered unrelated to the cluster and the algorithm continues with the next cluster. If no cluster satisfies the cluster membership criterion, we create a new cluster and add the entity to the cluster.

In the basic k-means algorithm, the final centroids do not necessarily represent actual entities but instead correspond to a mean value. Tetris guarantees that the final centroids represent an existing entity in the table by choosing the entity that is closest to the centroid. Algorithm 2 illustrates the modified k-means algorithm used in Tetris.

REORDERING. After the clustering phase, each entity is assigned to a cluster of similar entities. We construct an artificial, temporary column denoting the cluster membership of each record and generate a mapping vector that assigns each old position to an new position in the table. In a final step, we apply the mapping to all columns in the table and sort the columns accordingly.

4.3.1.2 *Optimizations*

INTER-CLUSTER REORDERING. So far we only considered to reorder rows in ascending lexicographical order by their corresponding cluster identifier. We can, however, reduce the memory footprint even further by applying a row reordering based on gray codes.

Algorithm 2: K-means algorithm with bayesian inference

Input : $X = \{x_1, \ldots, x_n\}$: input samples
Input : λ : Cluster penalty parameter
Output: k : number of clusters
Output: $C = \{c_1, \ldots, c_k\}$: output clusters
Output: $L = l_1, \ldots, l_n\}$: cluster labels

```
1  begin
2      k ← 1;
3      c_1 ← {x_1, ..., x_n};
4      μ_1 ← globalMean();
5      forall x_i ∈ X do
6          l_i ← 1;
7      while not converged do
8          forall x_i ∈ X do
9              minDist ← minDistance(x_i, μ_j) : j ∈ [1, k];
10             if minDist > λ then
11                 k ← k + 1;
12                 l_i ← k;
13                 μ_k ← x_i;
14             else
15                 l_i ← arg_j minDistance(x_i, μ_j) : j ∈ [1, k];
16         generate clusters c_1, ..., c_k based on l_1, ..., l_n : c_j = {x_i ∈ X ∨ l_i = j}
           updateCenters();
```

(a) Lexicographical ordering (14 blocks).

(b) Gray ordering (10 blocks).

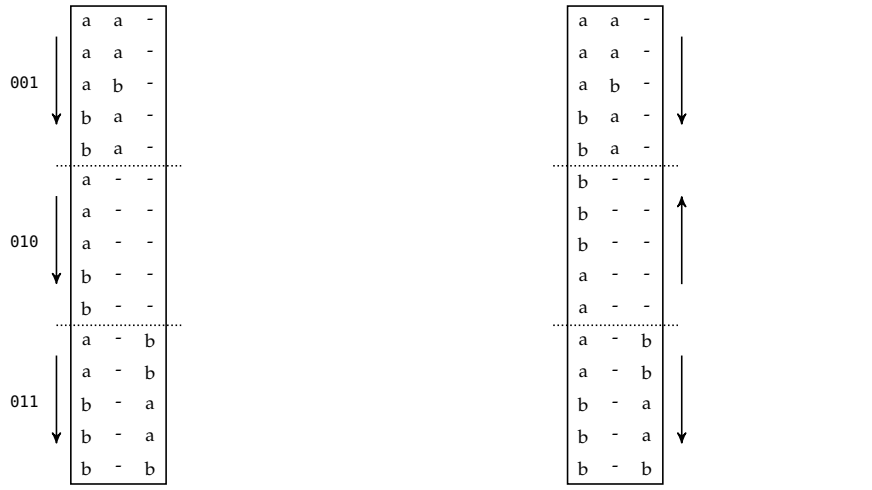
Figure 4.7: Cluster reordering by gray code.

Reordering rows by gray code is a commonly used technique to minimize the total number of runs of repetitive values (Faloutsos, 1986; Lemire et al., 2012). A gray code, also known as reflected binary code, is a binary numeral system where two consecutive values differ by a single bit. For example, for two bit values, the ascending order of gray codes is $[00, 01, 11, 10]$ while the lexicographic ordering is $[00, 01, 10, 11]$. We apply this technique on the cluster level by treating each cluster as a single block and by reordering those blocks by the gray code of their row fingerprint centroids. Figure 4.7 depicts two example row orderings, one based on lexicographical ordering and the other based on gray ordering. By applying a reordering based on the gray code of the center fingerprints, we can reduce the number of blocks of repetitive values from 14 to 10 blocks. This reduction of almost 30% is caused by the increased size of runs of repeated values. Algorithm 3 depicts the pseudo code for applying gray code ordering on a table. It receives as input a set of k clusters $C_1, \ldots, C_k$ and a set of center fingerprints $f_1, \ldots, f_k$. For each center fingerprint, we convert the bit sequence into the corresponding gray code, collect all computed gray codes in F', and finally sort the clusters by their gray codes. The algorithm outputs the set of clusters in ascending order sorted by gray code of their corresponding fingerprint centers.

Algorithm 3: Inter-cluster reordering by gray code.

Input : $C = \{C_1, \ldots, C_k\}$: clusters, $F = \{f_1, \ldots, f_k\}$: center fingerprints
Output: C : Clusters sorted by gray code

```
1 begin
2     F' ← ∅;
3     forall f ∈ F do
4         f' ← f ⊕ (f ≫ 1);  // Convert to gray code
5         F' ← F' ⋃ {f'};
6     sort C by F';
```



(a) Ascending sorting.

(b) Alternate ascending/descending sorting.

Figure 4.8: Alternating the sort order in consecutive clusters.

INTRA-CLUSTER REORDERING. To further decrease the total number of runs of non-NULLvalues within a column, we use lexicographical sorting within each cluster. A cluster is bounded by a range of start row and end row. For each cluster, we determine its boundaries and subsequently apply lexicographical sorting within the cluster.

Since sorting each cluster independently might result in larger value gaps between cluster, we sort clusters alternating ascending and descending. This allows forming even larger runs of consecutive values across cluster boundaries. Figure 4.8 depicts an alternate sorting between different clusters. In the example, this techniques reduces the total number of runs from six to four.

4.3.1.3 *Implementation Details*

We implemented TETRIS as an alternative row reordering heuristic in GRAPHITE. In the following we provide details on the implementation, specifically we discuss various data structures to store row fingerprints and the implementation of the k-means clustering algorithm.

A row fingerprint is a compact representation of all the exposed attributes of a single entity in a table. Figure 4.9 depicts an example row fingerprint of a table with 12 columns, where columns 1, 2, 5, 7, 8, 10 and 11 expose values different from NULL and columns 3, 4, 6, 9 and 12 denote columns with a NULLvalue. The resulting fingerprint is 001101001001.

We investigate alternative data structures to store row fingerprints, namely integer numbers, uncompressed bitsets, compressed bitsets, and bloom filters, and evaluate them in terms of memory consumption and query performance (elapsed time to compute the hamming distance between two row fingerprints). For tables with less than or equal to 64 columns, we use a numerical value with the minimum number of required bits (8, 16, 32

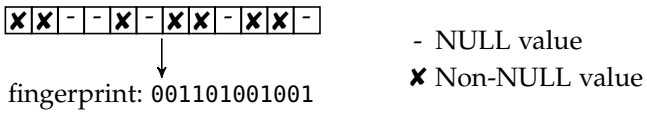


Figure 4.9: Row fingerprint represented as a bitset.

and 64). For wider tables we rely on sparse data structures, such as bitsets and bloom filters.

In the most general form, a bitset is an array of bits allowing to set, unset, and test a single bit as well as to perform set operations based on the boolean algebra to intersect and union two bitsets. For a table of 30,000 columns and 1 million rows, each row fingerprint occupies about 30 kbit, accumulating to about 3.5 GB for all row fingerprints. If the number of bits set is small, the memory overhead of the bitset compared to a dense array-based representation can be significant.

A compressed bitset has a lower memory footprint for sparsely populated bitsets, but has an increased processing overhead for testing whether a bit is set. Further, our implementation is based on *EWAH-bitsets* and does only allow setting bits in increasing order (Lemire et al., 2010). This limits the potential of parallelization during the construction of the row fingerprints since columns cannot be processed in parallel.

A bloom filter is a probabilistic data structure with a fixed width, a set of hash functions, and operations to test whether a certain element is in the set. All elements in the set are hashed and stored at the position of the hashed value. Due to the randomness of the hash function and the limited width of the bloom filter, more than one element can be hashed to the same position in the data structure possibly resulting in *false positives* during membership testing. To check, however, whether an element is *not* in the set is guaranteed to deliver a correct result.

We distinguish two variations of computing and storing the pairwise distances between row fingerprints: (1) to pass all row fingerprints to Tetris and compute the hamming distance between any pair of fingerprints on demand or (2) to cache already computed distances in a *dissimilarity matrix* for later usage. Caching the hamming distances is advantageous, if the distance between two row fingerprints is frequently requested and can be directly fetched from the cache. On the downside, the dissimilarity matrix has a quadratic space overhead of $\mathcal{O}(n^2)$ for n rows in the table.

Determining the cluster threshold λ is critical to achieve a good clustering result. We use data statistics and the standard deviation of the data distribution as initial value for the clustering threshold λ. If the input elements are close to each other in terms of the utilized distance function, the standard deviation value is small and therefore Tetris will output clusters with small diameters.

4.3.2 *Edge Ordering*

The basic implementation of topological queries does not rely on a particular ordering of the edges in the edge column group. To fully leverage the benefits of an in-memory graph storage, however, we can use data reorganization techniques that lead to a more memory-friendly data access pattern. A physical reorganization of records is a common optimization strategy to reduce data access costs (Abadi et al., 2006). In the following, we describe two strategies to further reduce the overall execution time of topological queries by maximizing the spatial locality of memory accesses and by reducing the total number of records to scan.

4.3.2.1 *Edge Type Clustering*

Typically, real-world graph data sets are modeled with a widespread and diverse set of edge types that connect the vertices in the graph. Conceptually, an edge type describes a subgraph and can be interpreted as a separate layer or view on top of the original data

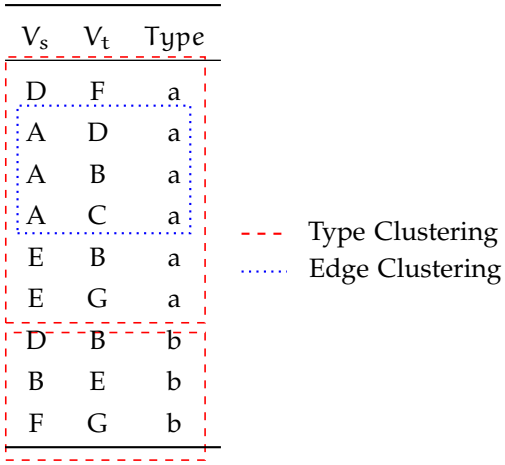


Figure 4.10: Clustering by source vertex and edge type.

graph. Such multi-relational graphs with multiple edge types are common in a variety of scenarios, such as product batch traceability, social network applications, or material flows graphs. For example, a product rating website might store different relationships between entity types *rating*, *user*, and *product*, such as rating relationships, product hierarchies, and user fellowships. To that end, topological queries are specific with regard to which parts of the graph they refer to. We propose to arrange edges sharing the same type physically together, allowing a topological query to operate directly on the subgraph instead of the entire original graph. Thus, a graph that comprises n different edge types results in n different subgraphs. A subgraph is associated with an area in the column that contains all edges forming the subgraph. Figure 4.10 illustrates an edge column group with two different edge types. Here, a topological query that refers to edges of type b would only have to scan the corresponding subgraph. The portion of the column for edge type b is indicated by the dashed lower rectangle. If the edge predicate contains a disjunctive condition, for example to traverse only over edges of type a or b, the execution engine spawns two parallel scan operation and unions the partial scan results thereafter.

4.3.2.2 *Edge Clustering*

The most fundamental component of any complex graph algorithm is to retrieve the set of adjacent vertices for a given vertex. Therefore, an efficient graph algorithm implementation should provide efficient access to adjacent vertices located in memory. To achieve this, we introduce the notion of *topological locality* in a graph. Topological locality describes a concept for accessing all vertices adjacent to a given vertex $v \in V$. If a neighboring vertex of a vertex v is accessed, it is likely that all other vertices adjacent of v are also accessed.

We translate topological locality in a graph directly into spatial locality in memory by grouping edges based on their source vertex. Such an edge clustering increases spatial locality, i.e., all edges sharing the same source vertex are stored consecutively in memory. Maximizing spatial locality for memory accesses results in a better last-level cache utilization and minimizes the amount of data to be loaded from memory into the last-level cache of the processor (Manegold et al., 2000). Figure 4.10 sketches an example for edge clustering on vertex A. All edges with vertex A as source vertex are stored consecutively in the edge column group. To that end, applying first clustering by type and then by source vertex extends the physical reorganization on a second level. Especially the materialization step of a topological query benefits from an increased spatial locality while fetching adjacent vertices from the V_t column.

Besides the spatial locality, *column decompression* plays an important role in materializing adjacent vertices. Major in-memory database vendors rely on a two-level compression strategy. The first level is dictionary encoding, where a value is represented by its numerical

value code from the dictionary and stored in a bit-packed, space-efficient data container. Here, a lightweight, but still notable decompression routine is used to reconstruct the actual value code. If adjacent vertices are not stored in a consecutive chunk of memory, the decompression routine might decompress unnecessary value codes. A similar behavior can be observed on the second level of compression, the value-based block compression. Edge clustering allows retrieving blocks of value codes that can be reconstructed efficiently by leveraging SIMD instructions.

4.4 EXPERIMENTAL EVALUATION

We evaluate the columnar graph representation of GRAPHITE on a variety of real-world and generated data sets and demonstrate the effectiveness of our data reorganization techniques—edge clustering and TETRIS—on graph query and manipulation operations. Further, we conduct experiments to quantify the memory consumption of the graph topology and their vertex/edge attributes in GRAPHITE for different scale factors of the LDBC data set. We evaluate several lightweight compression schemes, including dictionary encoding, run-length encoding based on TETRIS, bit-compression, and elias-delta encoding. Finally, we evaluate the graph storage of GRAPHITE in terms of single-user read queries, including topological access, attribute access, and predicate evaluation.

4.4.1 *Setup and Data Sets*

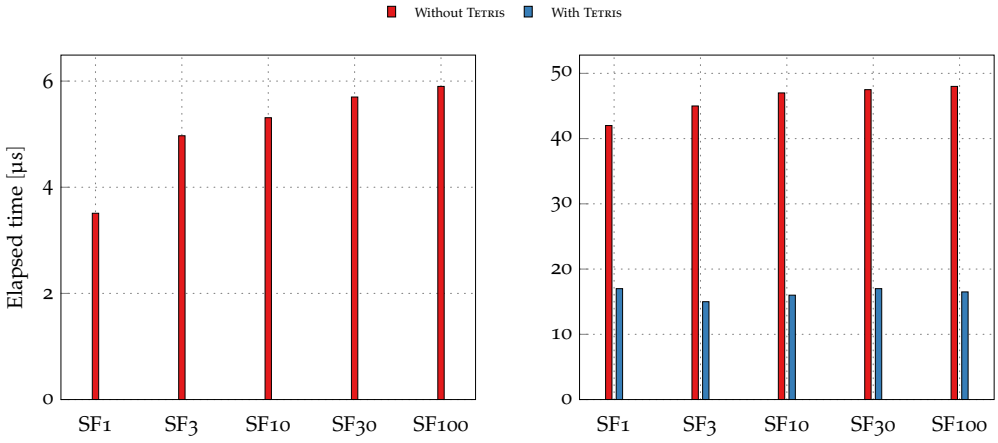
We conducted all experiments on an INTEL® XEON® E5-2660v3 machine with 2 sockets, 10 cores per socket, each core running at 2.6 GHz. The machine runs on SLES 12 SP1 and is equipped with 128 GB of DDR4 RAM and 25 MB last level cache.

We use an extensive number of real-world and generated data sets for our experimental evaluation. Table A.1 summarizes the evaluated data sets—whenever we refer to one of the data sets, we either use the long identifier for mentions in the text or the short identifier for referencing in plots. In addition to the real-world data sets, we use generated data sets from the WIDETABLE data generator.

WideTable Data Generator

The WIDETABLE data generator is specifically designed for producing data that is heterogeneous, but still contains a configurable number of implicit semantic types, which could be stored in a single wide and sparsely populated table. Our main objective for creating our own data generator is two-fold: (1) real data sets are difficult to collect, or if freely available, are too small (both in terms of total number of entities and number of distinct properties) and (2) we are particularly interested in evaluating the effect of specific parameter configurations, such as changing the number of distinct attributes. The WIDETABLE generator outputs a comma-separated file with data of five possible data types: *Boolean, Integer, Float, Char, and String*. Specifically, the WIDETABLE data generator can be configured by the following parameters:

- **Total number of entities.** Controls the total number of rows in the table.
- **Total number of distinct attributes.** Sets the total number of columns in the table.
- **Total number of groups.** Controls the total number of input groups to be generated. An input group loosely corresponds to a semantic type and represents entities that expose similar attributes.
- **Group size distribution.** Modifies the size distribution of the input groups. Allowed configurations are *uniform distribution* (all groups contain the same number of entities) and *normal distribution* (the group size distribution follows a Gaussian curve and can be further configured by the standard deviation σ).



(a) Single-attribute access.

(b) All-attribute access.

Figure 4.11: Results for the evaluation of vertex attribute access operations in Graphite on ldbc data sets for scale factors 1 to 100. We report the elapsed time of the full materialization of all exposed attributes of a vertex with (▮) and without (▮) Tetris.

- **Attribute sparsity.** Controls the fuzziness of the input groups by specifying the sparsity within an input group to model optional attributes within a semantic type. For example, although cars share in general most attributes, electro cars exhibit additional attributes, such as the charge time of the battery.

4.4.2 *Read Operations*

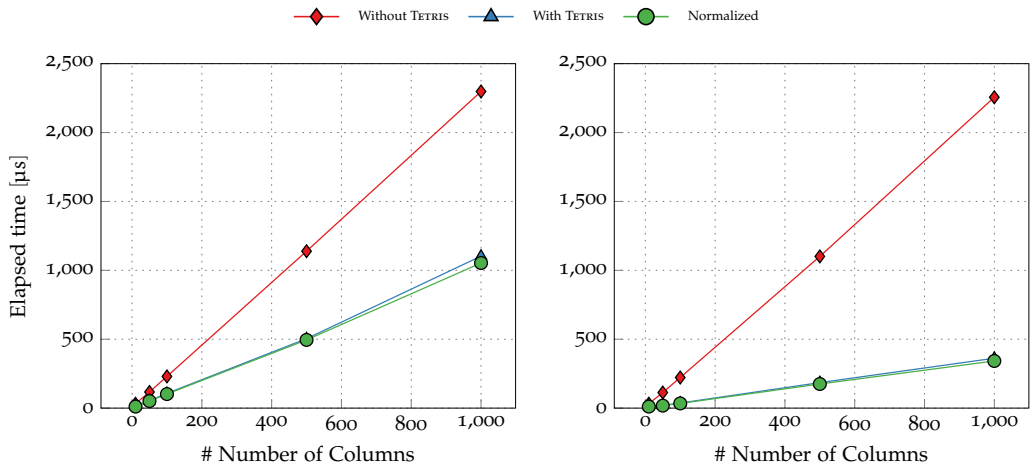
In this section we evaluate the graph storage of Graphite for the read operations that we defined in Chapter 3.1.1. Specifically, we report experimental results for attribute access operations with single and group attribute access, topological queries to retrieve the adjacent vertices for a given vertex, and predicate evaluation queries that return a set of vertices/edges matching a filter condition.

4.4.2.1 *Attribute Access*

In this experiment we evaluate the overhead of the graph storage of Graphite for attribute access operations that either return a single or multiple attribute values for a given vertex. We conducted all experiments on the ldbc data set for scale factors 1 to 100 and on generated data sets using the WideTable data generator. Specifically, we evaluate the following two dimensions: (1) the impact of the total data size, and (2) the performance implications of the wideness and sparseness of the column group.

In Figure 4.12 we present the results for single-attribute access and all-attribute access, where we compare Graphite using Tetris against Graphite without using Tetris. For the single-attribute access experiments, we generated 1,000 queries containing a randomly selected vertex identifier and a randomly selected attribute. Figure 4.12 (a) depicts the mean elapsed time for single-attribute access operations on the vertex column group.

A single-attribute lookup consists of a positional lookup in the data vector, followed by a positional dictionary lookup and the retrieval of the actual value. Both operations require constant time and are therefore independent of the overall data set size and the number of attributes present in the column group. Since the dictionary size is fixed for different scale factors, the variation in the execution time between different scale factors can be explained by caching effects on the data vector. For a smaller scale factor it is more likely that we access the attribute of the same vertex or some other vertex that is spatially co-located again than for larger scale factors.



(a) Attribute access with $s = 0.25$.

(b) Attribute access with $s = 0.75$.

Figure 4.12: Results for the evaluation of attribute access operations in Graphite on WideTable data sets with a sparsity $s = 0.25$ and $s = 0.75$, respectively. We report the elapsed time on the full materialization of all exposed attributes of a record with (—▲—) and without (—◆—) using Tetris and compare our approach with a naive, normalized database schema (—●—).

Figure 4.12 (b) depicts the mean elapsed time for all-attribute access operations on vertices with (▮) and without (▮) using Tetris on the ldbc data set. A vertex in the ldbc data sets exposes about 4 attributes, which is about 20 % of the total number of attributes on vertices. Without using Tetris, all attribute values have to be fetched from the column group and eventually discarded, if the attribute value is `NULL`. When using Tetris, we only read those attributes that are likely to have attribute values different from `NULL`. With Tetris, we achieve an performance improvement of factor 3 compared to the approach without Tetris, which is caused by the limited number of column accesses.

To investigate the effect of the sparseness and the wideness of a column group on the attribute projection operation, we generated data sets using the WideTable data generator with 1 million rows each, a varying number of columns ranging from 10 to 1,000, and a column sparsity of 0.25 and 0.75, respectively. Our comparison includes operations leveraging Tetris (—▲—), an implementation without Tetris iterating over all columns (—◆—), and a normalized database schema with one column group per vertex type (—●—).

Figures 4.12 (a) and (b) depict the experimental results for all-attribute access operations. For both plots, we can see that attribute access operations perform best on a normalized schema since a column group only contains attributes, for which the vertex exposes attribute values. In contrast, the time required to materialize a single vertex in a wide and sparsely populated column group grows linearly with the number of columns. Tetris shows a similar query performance as a normalized schema by utilizing additional data structures derived from semantic information about the vertex types and their exposed attributes. Ideally, an all-attribute access operation with Tetris performs equally well to a normalized database schema and significantly outperforms a naive implementation that has to read all columns of a single row. For wider column groups, the benefit of Tetris becomes even more noticeable, resulting in an performance improvement of a factor of 2 for the denser data set with a sparsity of 25 % and an improvement of a factor of 8 for the sparser data set with a sparsity of 75 %. Compared to a normalized database schema, Tetris adds a moderate overhead of less than 5 % on average to the overall elapsed time, which is due to the additional lookup to fetch the attributes to be read from the Tetris clusters.

Table 4.2: Run time distribution (in ms) for simple neighborhood queries of pattern `$v-->`.

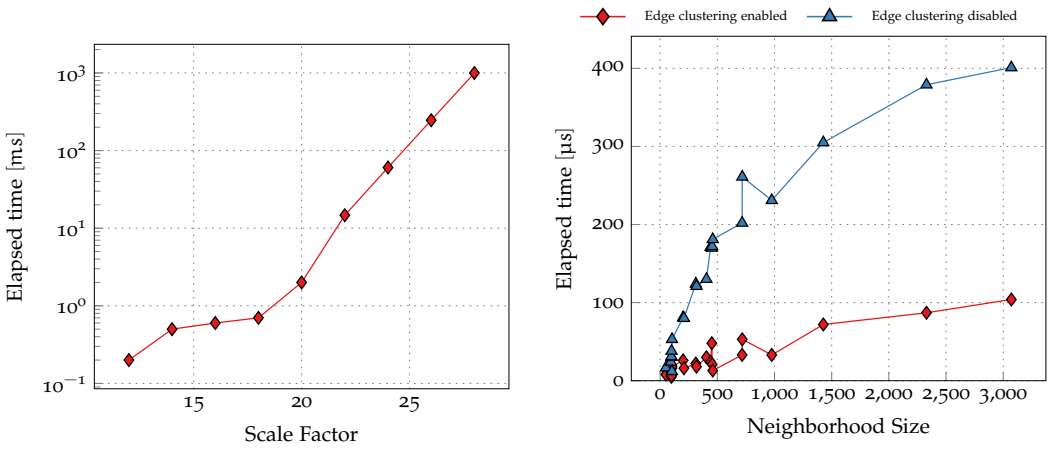
Data Set	1st Quartile	Median	3rd Quartile	99th Percentile
CALI	0.45	0.63	0.81	1.43
SKITTER	0.84	0.95	1.11	1.46
PATENTS	1.99	2.12	2.25	2.64
POKEC	5.37	5.47	5.60	5.90
LIVEJOURNAL	14.61	14.76	14.923	15.51
ORKUT	26.15	26.31	26.45	26.77
WIKIPEDIA	135.66	135.97	136.31	137.32
TWITTER	321.19	321.71	322.94	331.58
HYPERLINK	458.18	458.83	459.42	460.95
FRIENDSTER	569.92	583.73	584.85	586.79

4.4.2.2 *Topological Queries*

We evaluate the performance of GRAPHITE for neighborhood queries on real-world and generated graphs using the R-MAT data generator. For each experiment we generate 10,000 queries with random start vertices and return the fully materialized set of adjacent vertices. To illustrate the behavior of GRAPHITE for various graph topologies and output cardinalities, we present the experimental results of the run time distribution for real-world graphs in Table 4.2. In general, we observe that the elapsed time increases linearly with the total number of edges in the graph, leading to a time complexity of $\mathcal{O}(|E|)$. GRAPHITE uses column scans to identify incoming/outgoing edges for a given vertex, followed by a positional lookup to fetch adjacent vertices. Although a column scan is parallelized by partitioning the column into equally-sized logical chunks and by scanning each chunk with a different worker thread, the overall elapsed time increases drastically for large graphs. Here, we restrict ourselves to column scans without the utilization of additional secondary index structures. We will introduce and discuss secondary graph index structures in Chapter 6. GRAPHITE handles large output cardinalities caused by supernodes and small output cardinalities gracefully through adaptive output data structures that are selected depending on the estimated output cardinality. For large outputs, GRAPHITE uses a bitset data structure, while for small output cardinalities a dense array structure is used.

Figure 4.13 depicts the results on data sets generated using the R-MAT data generator for various scale factors with (—♦—) or without (—▲—) edge clustering enabled. Figure 4.13 (a) shows the scalability of GRAPHITE on neighborhood queries for increasing graph sizes. The number of edges grows linearly with the scale factor by a factor of 2. Overall, GRAPHITE can handle even very large graphs gracefully—scale factor 28 has about 4 billion edges—and returns adjacent vertices in less than a second.

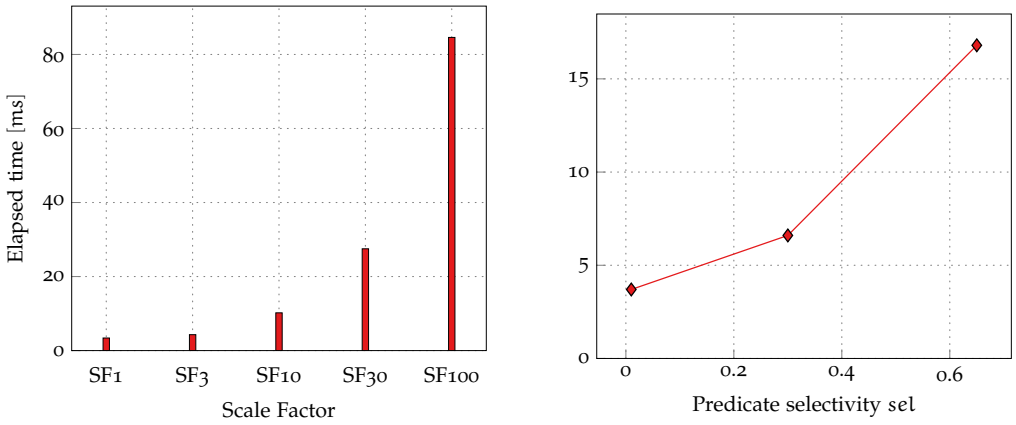
In Figure 4.13 (b) we evaluate the effect of different output cardinalities and plot the elapsed times of the fetch operation to retrieve adjacent vertices. Compared to the overall elapsed time of the column scan, the fetch operation only consumes a small fraction of the total elapsed time. If an additional secondary index structure is used, however, the elapsed time of the fetch operation becomes considerable again. With edge clustering enabled, GRAPHITE can fetch adjacent vertices from a continuous chunk of memory, thereby utilizing the CPU cache more efficiently since less data is transferred to the CPU. Storing adjacent vertices also spatially adjacent to each other avoids random memory accesses and results in a 4× speedup over a randomly ordered edge list. For larger output cardinalities, the difference becomes even more noticeable.



(a) Mean elapsed time.

(b) Run time distribution of fetch operation.

Figure 4.13: Absolute run time for neighborhood queries `$v-->` and runtime distribution of fetch operations with (—◆—) and without (—▲—) edge clustering enabled. We generated data sets using the R-MAT data generator with scale factors $sf \in \{12, 14, \ldots, 28\}$ and matrix coefficient configuration $\langle a = 0.57, b = 0.19, c = 0.19, d = 0.05 \rangle$.



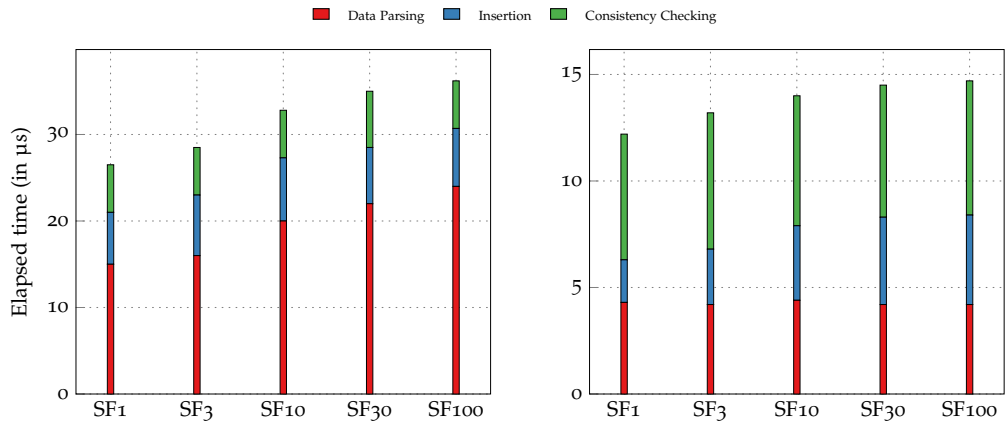
(a) Mean elapsed time.

(b) Runtime distribution on SF1.

Figure 4.14: Total runtime and runtime distribution for varying predicate selectivities for predicate evaluation queries.

4.4.2.3 *Predicate Evaluation*

We evaluate the query performance of GRAPHITE on predicate filter operations for point and range queries. A predicate filter receives a conjunctive predicate, evaluates the predicate for each vertex or edge in the graph, and returns a set of matching vertices or edges. We conduct our evaluation on the vertex set of the LDBC data set and present the results in Figure 4.14. To evaluate the performance for point queries and queries with a high selectivity ($sel < 1\%$), we generated 1,000 queries with random predicates and executed them on LDBC scale factors 1 to 100—the results are shown in Figure 4.14 (a). GRAPHITE scales linearly with increasing data set sizes ranging from 3.4 ms on scale factor 1 to 84.6 ms on scale factor 100. For a decreasing selectivity (larger result sets), GRAPHITE scales gracefully from 3.4 ms for a point query to 3.4 ms up to 16.8 ms for a predicate selectivity of 60%.



(a) Vertex insertions.

(b) Edge insertions.

Figure 4.15: Insertion performance of GRAPHITE on LDBC data sets for scale factors 1 to 100.

4.4.3 *Write Operations*

In this section we evaluate the performance of GRAPHITE for data manipulation operations, i.e., for adding and removing vertices/edges, updating attributes of vertices/edges, and deleting vertices/edges. We use generated data sets from the LDBC data generator. For the LDBC data sets, we generate scale factors 1 to 100 including update streams. An update stream is a separate portion of the entire data set consisting of a number of records to be inserted. Specifically, LDBC generates update streams to support the insertion of new vertices, such as the addition of persons, posts, and comments, and the insertion of edges, such as new friendship relationships between persons.

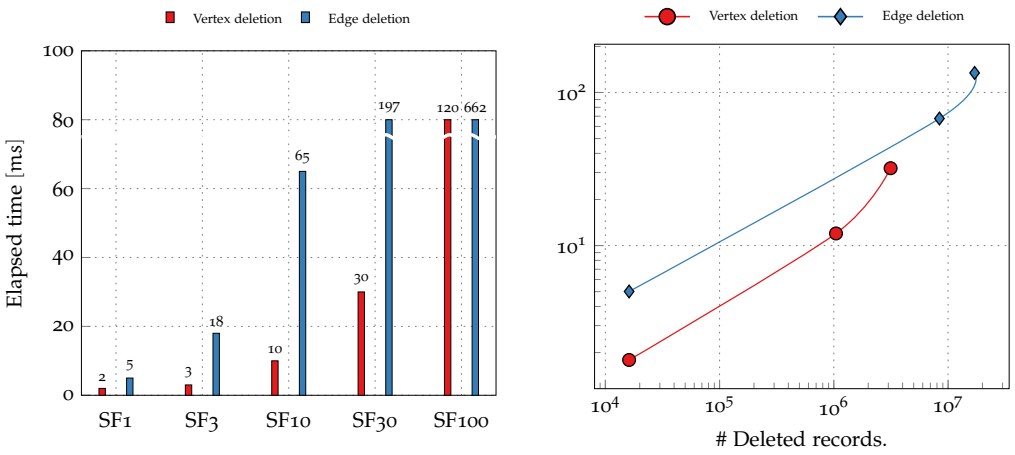
Insertions

We populate the database with the static graph that has been generated by the LDBC data generator. Next, we issue a set of insert operations against GRAPHITE and measure the time of the data manipulation operation, including data parsing, consistency checking, and data insertion. Each insertion operation is executed sequentially and without leveraging batch insertions or prepared insert statements. GRAPHITE handles insertion operations by redirecting them to the delta storage. In the delta storage, new values are appended at the end of the column and the unsorted dictionaries are updated accordingly by appending the value at the end of the dictionary array and by inserting the value into an additional tree-based secondary index structure.

Figure 4.15 depicts the average elapsed time for vertex and edge insertions for LDBC scale factors 1 to 100. In general, insertion operations in GRAPHITE are mainly agnostic to the database size, since new values are only appended at the end of the column. The data parsing overhead is for vertices considerably larger than for edges since vertex insertions tend to have more attributes associated than edge insertions. Further, the consistency checking for vertices consults the secondary index structure on the primary key to guarantee uniqueness. For edge insertions, the overall elapsed time is dominated by the consistency checking, which performs dictionary lookups for both, source and target vertex to disallow dangling edges in the graph.

Updates

An update operation in GRAPHITE is effectively a deletion operation followed by an insertion operation. Since a deletion operation has a time complexity of $\mathcal{O}(n)$, where n is the number of rows in the column group, the update operation is dominated by the task to identify the row(s) to invalidate. GRAPHITE also supports in-place updates that modify



(a) Vertex/edge deletions.

(b) Deletion scalability.

Figure 4.16: Performance evaluation of deletion operations—separated into vertex and edge deletions—in GRAPHITE on LDBC data sets for scale factors 1 to 100 (single record deletion) and for different numbers of deleted records on scale factor 1.

the read-optimized column group directly, if the new attribute value is already present in the data dictionary. Otherwise, the record has to be invalidated and inserted into the write-optimized column group.

Although unsetting an attribute value (setting to `NULL`) can be always done in-place, the update operation is still dominated by the time to find the corresponding entities to update.

Deletions

Figure 4.16 depicts the experimental evaluation of deletion operations in GRAPHITE—specifically, the removal of vertices and edges based on some predicate condition. We evaluate GRAPHITE in terms of increasing data volumes and working sets, i.e., the number of records to be deleted. We handle deletions in GRAPHITE through two visibility vectors, one for the read-oriented data store and one for the write-oriented data store. If a vertex/an edge is about to be deleted, we identify the corresponding row(s) in the table and mark the matching positions in the visibility vector as invalid. Any subsequent read operation on the graph will consult, before reading any data, the visibility vectors that indicate which vertices/edges are valid and can be read by the query.

Figure 4.16 (a) shows the average elapsed time for single record deletions on vertices ( ) and edges ( ), respectively. We randomly selected vertices and edges by their primary key—for the vertices the vertex identifier, for the edge an artificial primary key—and generated a set of 100 deletion operations.

To delete a vertex/an edge in GRAPHITE, we first identify the matching rows through a column scan on the corresponding column group and secondly, we invalidate the entries in the visibility vectors. If there are no additional secondary index structures on the attributes defined, the overall elapsed time of a deletion operation is dominated by the identification of the rows to be deleted. From the results we conclude that the total elapsed time increases linearly with growing data volumes and that vertex deletions are about 6 times faster than edge deletions. This can be explained by the sparsity of the graph ($|V| \ll |E|$, cf. Table A.1).

Figure 4.16 (b) shows the scalability of GRAPHITE on the LDBC data set for scale factor 1 and for different working set sizes, i.e., the number of deleted entities per deletion operation. Therefore, we generated deletion operations with predicates resulting in different working sets, ranging from selective predicates—a predicate on the primary key—to unselective predicates—a predicate that removes all vertices that have a specific type. The elapsed time increases linearly with the size of the working set, for both vertex and edge

Table 4.3: Memory consumption (in MB) of various graph topologies from Table A.1.

Data Set	Raw Size (CSV format)	Uncompressed		Compressed	
		Dictionaries	Data Vectors	Data Vectors (`bit-compressed`)	Data Vectors (`elias-delta`)
CALI	79	15	42	24	1.6
SKITTER	143	10	85	54	3.1
PATENTS	252	20	126	85	4.7
POKEC	405	11	234	153	9
LIVEJOURNAL	955	33	522	318	20
ORKUT	1687	22	894	614	34
WIKIPEDIA	9,714	141	4,586	3,439	177
TWITTER	24,240	289	11,203	8,752	435
HYPERLINK	34,453	549	15,588	10,230	605

deletions. This is caused on one side by the increased amount of work performed during the scan operation—more write operations and larger output cardinality—and by the increased number of operations on the visibility vector.

To summarize, GRAPHITE scales deletion operations gracefully to growing data volumes and working sets. The actual record invalidation contributes only a small fraction of the total elapsed time (in our experiments less than 5%). The identification of records to be deleted could be accelerated by additional secondary index structures.

4.4.4 *Memory Consumption*

In this section we evaluate the overall memory consumption of the graph storage in GRAPHITE for a variety of real-world and generated graph data sets. We apply several lightweight compression techniques, such as bit compression, dictionary compression, and elias-delta compression to the data and report our results in Tables 4.3 and 4.4, respectively. For the real-world data sets, we populate the graph data into a column group representing the edge list in two columns, one for the source vertex and one for the target vertex. All input vertex ids are represented by 32 bit numerical values which are dictionary-encoded upon data loading to create a dense value domain.

In Table 4.3 we present the memory consumption of several real-world graph data sets stored in GRAPHITE and compare their uncompressed, dictionary-encoded memory footprint with two optional compression techniques, namely `bit-compression` and `elias-delta compression`. Compared to the raw data set stored in a two-columnar CSV file on disk, GRAPHITE stores the graph topology in a column group with two columns, one for source vertices and one for target vertices. Thereby, GRAPHITE reduces the memory consumption by about 50% compared to the data size on disk.

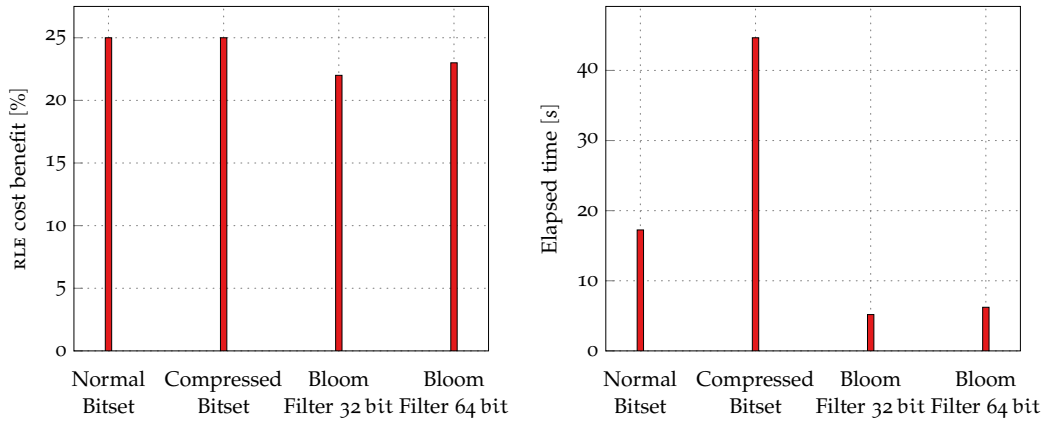
If we apply bit-level compression and compute the number of required bits to represent each distinct value in the value domain, GRAPHITE allows reducing the memory footprint of the data vectors by 35% on average.

We also experimented with `elias-delta` compression achieving superior compression ratios reducing the memory footprint by almost 90% on average. A data vector encoded using `elias-delta`, however, has the disadvantage that decompressing the data vector during query execution is rather expensive.

Table 4.4 depicts the memory consumption for data sets from the LDBC project for scale factors 1 to 100. The data sets are composed of two column groups, one for vertices with 18 attributes and one for edges with 7 attributes. For all data sets, we report the memory consumption of two CSV files on disk, the uncompressed column groups, and the compressed data vectors using `bit-compression` and `elias-delta compression`. In contrast to the real-world graphs without additional attributes, storing a large variety of attributes

Table 4.4: Memory consumption (in GB) in GRAPHITE for LDBC data sets with scale factors 1 to 100. The graph properties are listed in Table A.1.

Data Set	Raw Size (CSV format)	Uncompressed		Compressed	
		Dictionaries	Data Vectors	Data Vectors (`bit-compressed`)	Data Vectors (`elias-delta`)
LDBC-SF1	1.3	0.6	0.7	0.4	0.2
LDBC-SF3	3.9	2.1	1.9	1.1	0.5
LDBC-SF10	13.2	6.9	6.6	5.5	2.9
LDBC-SF30	40.8	20.8	19.8	14.3	7.8
LDBC-SF100	130.0	69.5	62.5	45.1	21.4



(a) RLE Cost Benefit.

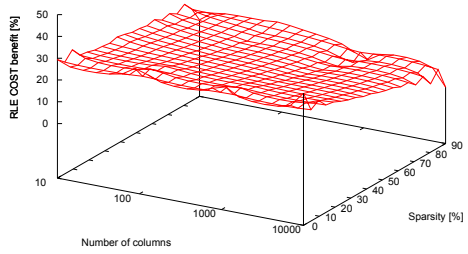
(b) Construction time.

Figure 4.17: Effect of different fingerprint implementations on the RLE cost for a data set generated by the WIDETABLE data generator with 10,000 rows, 100 columns, 20% sparsity, and 10 generated input groups.

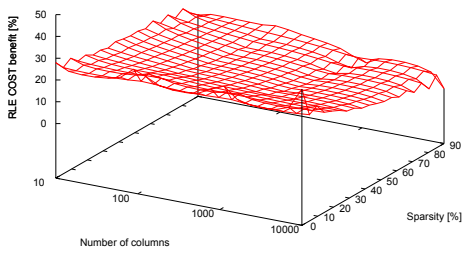
on vertices and edges results in a larger memory consumption that is similar to the memory consumption on disk. This is caused by the requirement of LDBC to represent most of the attributes as expensive character sequences. A more sophisticated handling of different data types and data compression on the data dictionaries can mitigate this overhead. When we apply `bit-compression` on the data vectors, we can lower the memory footprint on average to 20% of the uncompressed data vectors. If we apply even more aggressive `elias-delta compression`, we can reduce the memory consumption of the data vectors to about 50% of the uncompressed size.

In the following we evaluate TETRIS on a variety of data sets and present our findings for different data set characteristics, row fingerprint implementations, and the effect of the proposed optimizations. If not stated otherwise, we enable all optimizations to TETRIS in the experiments, i.e., intra cluster reordering, inter clustering reordering, and the combination of both.

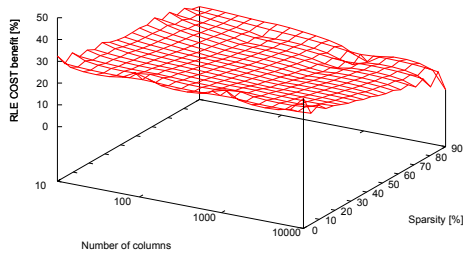
To quantify the RLE cost benefit, we use a simplistic compression model (cf. Equation 4.1) that abstracts from the actual memory consumption. We accumulate the cost of storing each cell in a table, where a cell has either cost zero or cost one. The cell cost is 0, if the value stored in the cell is equal to the value stored in the previous cell in the same column, 1 otherwise. The intuition behind this simplistic metric is that repeating values do not impose a storage overhead since they would be compressed using run-length encoding.



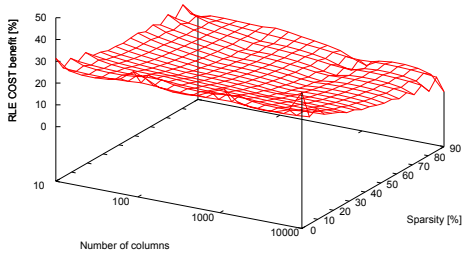
(a) 10 input groups
(uniform distribution)



(b) 100 input groups
(uniform distribution)



(c) 10 input groups
(normal distribution)



(d) 100 input groups
(normal distribution)

Figure 4.18: Effect of the number of input groups and the group size distribution on the RLE cost benefit metric.

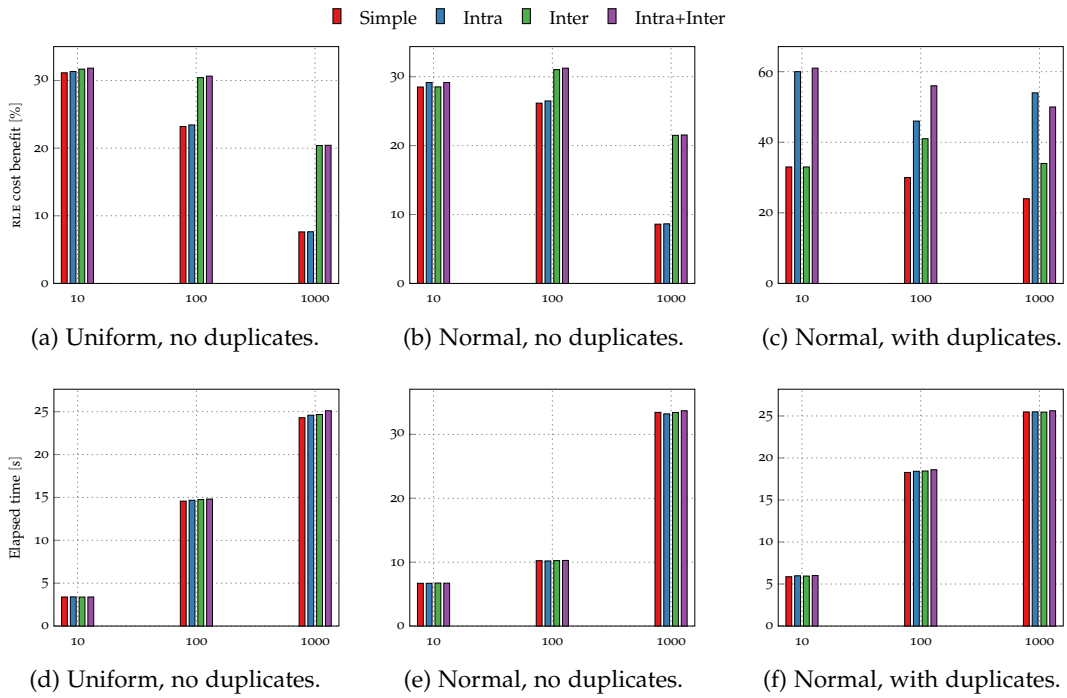
$$\text{RLE cost} = \sum_{c=1}^{C} \sum_{r=1}^{R} \sigma_{r,c}$$

$$\sigma_{r,c} = \begin{cases} 1 & : \quad v_{r,c} \neq v_{r-1,c} \vee r = 1 \\ 0 & : \quad v_{r,c} = v_{r-1,c} \end{cases} \tag{4.1}$$

$$\begin{array}{lll} R & : & \textit{Number of rows} \\ C & : & \textit{Number of columns} \\ v_{r,c} & : & \textit{Value of row r at column c} \end{array}$$

For the first experiment, we evaluate TETRIS for three different row fingerprint implementations, namely uncompressed bitsets, compressed bitsets, and bloom filters (cf. Figure 4.17). We use the WIDETABLE data generator and generate a data set with 10,000 rows, 100 columns, 20% sparsity, and 10 generated input groups. For the bloom filter implementation, we evaluate two configurations: 32 bit and 64 bit. For all experiments, we ran TETRIS using different fingerprint implementations and measure the RLE cost benefit percentage, i.e., the memory savings compared to a simple RLE implementation.

Figure 4.17 (a) shows the results for the evaluation of TETRIS for different fingerprint implementations. The fingerprint implementations show comparable RLE cost benefits between 22% and 25%. Since both bitset implementations provide a lossless representation of the fingerprint, they both result in the same RLE cost benefit. The bloom filter (32 bit) shows the lowest benefit as its highly compressed representation discards important fin-



(a) Uniform, no duplicates. (b) Normal, no duplicates. (c) Normal, with duplicates.

(d) Uniform, no duplicates. (e) Normal, no duplicates. (f) Normal, with duplicates.

Figure 4.19: Results for data set with 10,000 rows, 100 columns, and a sparsity of 10%. We evaluate four different reordering phases of Tetris to quantify the effectiveness of each single optimization. Specifically, we evaluate simple cluster-based reordering (■), intra cluster reordering (■), inter cluster reordering (■), and inter+intra cluster reordering (■).

gerprint information. For wide tables with potentially thousands of columns, the bloom filter approach performs even worse.

Figure 4.17 (b) depicts the effect of the fingerprint implementation on the construction time of the Tetris algorithm. Since the overall performance is mainly dominated by the evaluation of the pair-wise distance function, compact bit representations of the exposed attributes of a row tend to perform significantly better. Based on this observation, the bloom filter (32 bit) shows the lowest construction time with about 5.2 s compared to the compressed bitset with about 44.7 s.

In the next experiment we present our findings for different data configurations and report the achieved RLE cost benefit. We used the WideTable data generator to generate random data sets with 10,000 rows. To evaluate the effect of different data set characteristics, we varied the number of columns between 10 and 10,000, the sparsity between 0% and 90%, and the number of generated input groups between 10 and 100. Further, we generated two different input group size distributions following a uniform and a normal distribution, respectively. For the generated normal distribution, the standard deviation of the group size is 2, 20, and 200 for the number of generated groups 10, 100, and 1,000, respectively. Figure 4.18 depicts the RLE cost benefit of Tetris for normally and uniformly distributed input group sizes for 10 and 100 input groups. Tetris shows a stable compression ratio improvement over a naive RLE implementation and is not sensitive to changing data set characteristics, such as the number of columns, the sparsity within a group, and the group size distribution. For data sets with a high sparsity, i.e., fuzzy cluster boundaries, the benefit of applying Tetris on the table decreased slightly and a naive lexicographical reordering tends to work better on very sparse columns.

In Figure 4.19 we depict the results of Tetris for a data set with 100 columns, a sparsity of 10%, and varying group size distributions. The WideTable data generator generates by default unique values for numeric and varchar columns—we experiment in addition to

that with duplicates in columns triggering run-length encoding not only on NULL values but also on other sequences of repeating values.

The SIMPLE and the INTRA compression steps show similar results for the RLE cost benefit, since the INTRA technique reorders values within a cluster. Since we generate unique values for the first two experiments in Figures 4.19 (a)–(b), there is no compression gain when using INTRA as it does not produce longer runs of repeating values. Most beneficial in our experiments is INTER, which reorders clusters to form longer runs of repeating NULL values across cluster boundaries. For a larger number of input groups, INTER achieves a larger RLE cost benefit as longer runs of NULL values can be formed between clusters.

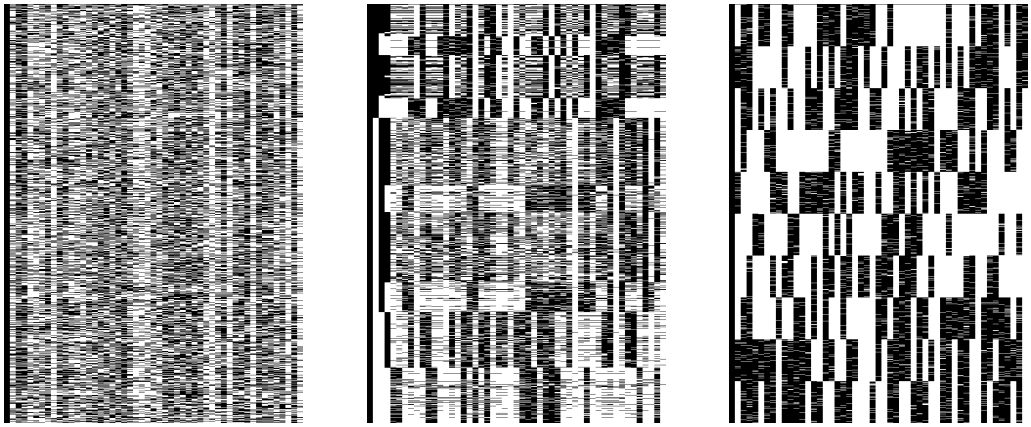
Figures 4.19 (d)–(e) illustrate the time to apply the reordering of the rows. The reordering time increases linear with an increasing number of input groups. Further, we can see that no considerable extra time is spent on the additional optimization steps. This is because we tightly integrated the optimization steps into the reordering phase, effectively allowing to reorder the entire table in a single pass.

In Figure 4.19 (c) we present the results for data sets that contain duplicate values, which is quite common in realistic scenarios. Duplicate values are beneficial for the run-length encoding algorithm as the general routine only compresses large runs of repeating values well, independent whether the value is NULL or any other value. For columns that contain duplicate values other than NULL, INTRA cluster reordering increases the runs of repeating values, effectively increasing the RLE cost benefit to up to 60%.

4.5 SUMMARY

Based on a basic set of read and write operations, we designed, implemented, and evaluated a graph storage in GRAPHITE solely relying on mature column store technology. Further, we discussed several data reorganization techniques to improve the spatial memory locality for accessing the neighbors of a vertex, to efficiently retrieve all exposed attributes of a vertex/an edges, and how we can apply light-weight compression techniques on our columnar graph storage to eliminate NULL values. We introduce TETRIS, a data reorganization technique that reorders rows in a wide and sparsely populated table such that run-length encoding applied afterwards significantly outperforms a naive column-by-column reordering approach (cf. Figure 4.20).

Based on an extensive experimental evaluation we conclude that for attribute-centric graph queries that also require fast access to the attributes of vertices and edges, a columnar graph representation is superior to a native graph representation where attributes are only considered as a "payload" of a vertex/an edge. Instead, we propose to store the attributes in wide, sparsely populated column groups and represent the graph topology as a simple edge list stored in two columns. To further improve the query performance of neighborhood queries, (relational) secondary index structures as well as specialized graph index structures can be added. We discuss the use of additional index structures to accelerate neighborhood queries in Chapter 6.



(a) Random row order. (b) Lexicographical row order. (c) TETRIS row order.

Figure 4.20: Visualization of a wide and sparsely populated table; white corresponds to `NULL`, black to a value different from `NULL` (the first column on the left side is the primary key column).

5 GRAPH TRAVERSAL OPERATORS

Graph traversals are a fundamental building block in many graph algorithms, such as graph pattern matching, detecting (strongly) connected components, and shortest path computation. We introduce an abstract graph traversal operator and define the traversal semantics formally. Based on the abstract notion of a graph traversal operator, we propose two traversal strategies on a columnar graph storage, a *level-synchronous* and *fragmented-incremental* graph traversal. As a result of the system architecture of Graphite, we propose several techniques to extend the graph traversal to distributed graphs that are spread across a read-optimized and a write-optimized graph storage. In the experimental evaluation, we demonstrate the performance and efficiency of our traversal implementations and showcase that they are competitive to those available in native graph processing systems.

5.1 RELATED WORK

In this section we review related work on single-node and distributed graph traversals. While we only gave a broad overview of graph traversals in Section 2.4, in the following we provide details on the specific graph traversal implementations and techniques applied to achieve superior performance.

5.1.1 *Single-Node Graph Traversals*

Early efficient parallel BFT (breadth-first traversal) algorithms date back to the theoretical investigations by Gazit and Miller (1988), who reduce the BFT problem to a matrix multiplication problem and derive a time complexity for single-source BFT in the *exclusive-read, exclusive-write model* of $\mathcal{O}(\log^2 n)$ using $M(n)$ processors and n being the dimensionality of the adjacency matrix.

Bader and Madduri (2006) implement a parallel BFT on a multi-threaded, thread-centric Cray MTA-2 system. The Cray MTA-2 employs no memory hierarchy but instead hides memory latency through hardware multi-threading. The authors leverage the hardware-assisted fine-granular parallelism and the zero-overhead synchronization primitives offered by the Cray MTA-2 and implement a level-synchronous BFT algorithm. Parallelization is applied on two levels, on the queue level and on the adjacency level. Concurrent access to the queue and the output data structures are protected using lock-free synchronization primitives provided by the Cray MTA-2. To avoid thread underutilization at the queue level, the authors propose to identify high-degree vertices upfront and to process them separately to improve load balancing.

Xia and Prasanna (2009) investigate the impact of the graph topology on the scalability to the number of available threads. They propose a parallel BFT implementation, which estimates the scalability during each traversal iteration and also provide an accompanying cost model to predict the expected elapsed time. Based on the observation that BFT implementations on graph topologies with a small working set per iteration scale poorly, they propose to adaptively increase or decrease the level of parallelization (active number of threads) per iteration, depending on the estimated working set size.

Agarwal et al. (2010) investigate scalable BFT algorithms on multicore processors and specifically targeted Intel® Nehalem EP and EX consisting of 2 sockets (4 cores/socket) and 4 sockets (8 cores/socket), respectively. Based on a naive BFT implementation, they propose several optimizations that deal with the management of intermediate vertex sets and that enable scaling across multiple sockets. The naive BFT implementation uses two queues, one for storing the vertices to be explored on the current level and one for storing

the vertices to be explored on the next level. At the end of each traversal iteration, the contents of both queues are swapped. To enable a parallel BFT running on a single socket, the authors propose to add atomic instructions instead of locks to protect the queue operations against race conditions and to use a bitmap to check whether a vertex has been already visited. In contrast to the parent map, which can be used to check whether the vertex already has a parent assigned, a bitmap is more memory-efficient and is more likely to entirely fit into the last level data cache of the socket. To avoid cache line invalidation and locking due to random access atomic writes, the authors introduce a communication mechanism between groups of cores from different sockets. They leverage a *FastForward* data structure implementing single producer/consumer lock-free queue. If the vertex resides on the local socket, it is inserted into a local queue, otherwise the remote queue of the corresponding socket is used. Instead of updating the queues on a fine-granular vertex-level, a batching mechanism groups writes into batches and updates the queues batch-wise.

Pearce et al. (2010) propose an asynchronous approach to graph traversals and implement this mechanism for BFT, single-source shortest path, and connected components, for in-memory and semi-external memory computation. By relying on an asynchronous implementation, the authors mitigate costly synchronizations that are required on concurrently accessed data structures in multi-threaded environments. Surprisingly, they implement BFT as a specialized SSSP implementation with edge weights equal to one. The idea is based on the *vertex visitor* pattern, which is commonly used to describe a traversal-based graph algorithm in an abstract way. The central component of the approach is a *visitor queue*, which queues all vertex visitor requests. During the execution, the requests can be concurrently removed from the visitor queue and processed in parallel. Although each thread maintains a private visitor queue, it remains unclear how to synchronize between the thread-private queues.

Leiserson and Schardl (2010) present a parallel BFT algorithm, which is *work-efficient*[1] and achieves linear scalability with the number of processors by replacing the standard FIFO queue by a *bag* data structure.

Hong et al. (2011) build on the ideas of Agarwal et al. (2010) and propose a parallel BFT algorithm that makes efficient use of the available memory bandwidth. Instead of implementing the *current-level set* and the *next-level set* as lock-protected queues, they encode both sets into a single array structure with sufficient capacity to store level information for all vertices in the graph. The advantage of storing the level information in a single array-based data structure is that no locks are required to protect the elements in the array from concurrent write access. Since the array stores level information in increasing node identifier order, the memory access to the underlying graph structure, if also ordered by node identifier, can be processed sequentially and thus minimizes the number of random memory accesses. The main disadvantage is that the entire array has to be read at every traversal iteration. Additionally, for large graphs it is unlikely that the entire array fits into the last level data cache and access to it will result in data cache misses. For graphs with a large diameter, the authors propose a hybrid approach, which combines the queue-based traversal for traversal iterations with a small frontier set and the *read-based traversal* for large frontier sets. This hybrid approach has been also generalized to process the first traversal iterations on the CPU and switch for the traversal iterations with a large frontier set size to the GPU.

Chhugani et al. (2012) propose a parallel BFT implementation, which aims at balancing load and locality-aware computation. The authors tackle several issues, which cause a negative performance impact, including low cache hit ratio, remote memory accesses, and load balancing. A graph is represented as a two-dimensional adjacency array, where the adjacent vertices of a given vertex are stored contiguously, and the adjacencies are distributed evenly across all sockets. To check, whether a vertex has been already visited, the authors propose a bitset data structure, which is—in contrast to previous approaches—tailored to reside mainly in the last-level cache. Since for large graph the bitset of visited

1 A parallel algorithm is *work efficient*, if the total number of operations performed is within a constant factor of the best serial algorithm.

vertices does not entirely fit into the last-level cache of a socket, the authors partition the bitset evenly across all sockets. For the bitset, they maintain an atomic-free and lock-free update mechanism, as even atomic instructions impose a considerable performance overhead as they behave as memory fences. Frontiers are partitioned across sockets and kept thread-local. To reduce the number of TLB misses, the authors rearrange the vertices of the frontier set according to collected memory access statistics.

Beamer et al. (2012) propose a BFT algorithm that combines traditional top-down traversal with bottom-up traversal on low diameter, scale-free graphs, such as social networks. The top-down searches for unvisited children given a vertex from the frontier set, while the bottom-up approach searches for parents of unvisited vertices. Once a parent is found, the bottom-up traversal can continue the traversal with the next vertex. In the top-down approach, *all* adjacent vertices have to be examined, while for the bottom-up approach, the discovery of *any* parent vertex suffices. Especially when the frontier set is considerably large, a bottom-up traversal can significantly reduce the number of edges to traverse. The authors propose a hybrid algorithm that switches between top-down and bottom-up traversal dynamically, depending on the size of the frontier set. The first iterations during the traversal are performed top-down as the frontier set size is typically small. For the following iterations, the algorithm switches the processing strategy to bottom-up traversal, also including a transformation of the frontier set from a queue implementation to a bitset implementation. For the final iterations, where the frontier set size typically shrinks again, the algorithm switches back to a top-down traversal. The two switching points are determined dynamically considering graph properties, such as the number of edges from unexplored vertices, the number of vertices in the frontier set, and the number of edges to check from the frontier set. Further, they introduce two tuning parameters, which have to be determined experimentally in advance.

Yasui et al. (2013) build on the ideas of Beamer et al. (2012) and propose a NUMA-aware BFT implementation. The authors use the general hybrid implementation provided by Beamer et al. (2012) and extend it by partitioning the vertex set, i.e., all outgoing edges of a vertex reside in the local memory of a socket. Further, each socket maintains a local variable to store visited vertices, discovered vertices, and the predecessor map.

Berrendorf and Makulla (2014) classify parallel BFT implementations and perform an experimental evaluation on multi-core shared-memory NUMA machines. They use several graph topologies with varying properties, including road networks, Delaunay graphs, and social networks, and showcase that no BFT implementations supersedes the others on all evaluated data sets. Similar to our observations in the context of GRAPHITE, there is evidence that the graph topology (and in our case the traversal query) has severe implications on the overall execution time of the graph traversal. The authors classify the examined implementations into *container-centric* and *vertex-centric* approaches. While container-centric implementations use a set of central data structures to keep track of visited and unexplored vertices, they suffer from NUMA effects due to possible remote memory access on memory associated to other sockets. In contrast, a vertex-centric implementation assigns a parallel thread to each vertex in the graph. After each iteration, a global barrier allows synching the state between adjacent vertices. On the downside, a vertex-centric approach suffers from graphs with a large diameter and lacks possibilities for explicit load balancing.

Then et al. (2014) explore how to efficiently run a set of BFTs in parallel on a single machine by sharing computation work between BFT runs. This is in contrast to previous related work that aim at improving the performance and scalability of a *single* BFT run by leveraging and saturating all available compute units of the machine. The authors argue that running a large number of BFTs from different source vertices in parallel is a common pattern seen in graph centrality measures, such as betweenness centrality and closeness centrality. They leverage the properties of social networks, i.e., a power-law vertex degree distribution and a small graph diameter, to share commonly explored vertices between multiple BFT runs. In their MS-BFS implementation, they use well-known algorithmic tricks, such as switching between top-down and bottom-up traversal (cf. Beamer et al. (2012)), leveraging SIMD instructions to speed up set operations, and software-assisted memory

prefetching. Although *scale-free* graph topologies are rather common in many realistic use cases, sharing work between graph algorithms in general requires a certain amount of *sharable* work between single algorithm instances. For example, hypersparse graphs, such as road networks, are unlikely to benefit from work sharing between BFT runs. Further, MS-BFS requires all BFT instances to run in sync, i.e., only work is shared between BFT instances if the vertex is explored on the same BFT level. A more flexible work sharing approach, as can be found in related work on *cooperative scans*, however, would be desirable.

Buluç and Gilbert (2011) and Kernert et al. (2014) discuss the implementation of BFT based on iterative matrix-vector multiplications, where the transpose of the adjacency matrix is repeatedly multiplied with a vector containing the frontier vertices. Kernert et al. (2014) stores the underlying matrix in a CSR-like data structure in a column-oriented RDBMS and returns as a result of the BFT vertices discovered at a certain depth. The BFT implementation does not perform cycle detection and therefore does not guarantee to produce a valid BFT tree.

5.1.2 *Distributed Graph Traversals*

Yoo et al. (2005) describe a parallel, level-synchronous BFT on undirected graphs and use a 2D partitioning to distributed the edges across the computing nodes of a massively parallel BLUEGENE/L system. The authors employ a 2D edge partitioning over a 1D partitioning to reduce communication overhead and to effectively reduce the number of computing nodes involved in the collective communication phase from $\mathcal{O}(P)$ to $\mathcal{O}(\sqrt{P})$. The 2D partitioning splits the edge set such that each vertex and each edge is owned by a single computing node, and not all edges incident to a vertex are stored in the same partition.

The major challenge of distributed graph traversals is the high communication cost to distribute the frontier set to all computing nodes. Lv et al. (2012) propose two techniques to reduce the message sizes in the all-to-all communication phase by leveraging lossless compression techniques and lightweight vertex sieving. The frontier set is represented as a compressed bitset using a word-aligned compression scheme. To avoid sending the entire frontier set to each computing node, the authors propose a sieving of frontier sets such that vertices are only distributed to the corresponding computing nodes, when they contain a partial edge list to continue the traversal. Therefore, a *cross directory* is introduced, which contains information about which frontier vertex will be processed at which computing node. The sieving is a preprocessing step of the actual message compression and improves the compression ratio even more.

Buluç and Madduri (2011) tackle the problem of running graph traversals on distributed memory systems. They propose two complementary approaches to distributed BFT, namely an implementation based on 1D partitioning of the edge set and another implementation based on linear algebra operations using a CSR representation in combination with a 2D partitioning on the adjacency matrix. To quantify the memory access costs (local and remote), they present a simple linear memory-reference performance model. For the 2D partitioning, however, the authors argue that a pure CSR representation wastes memory space and propose to use a doubly-compressed sparse columns (DCSC) instead. In the 1D implementation, they use a stack data structure, in the 2D implementation frontier vertices are represented in a sparse array format, by only storing indices of non-zero vertices. The 1D implementation leverages thread-local stacks to avoid synchronization overhead; the local stacks are eventually merged at the end of each traversal iteration. The 2D implementation relies on the linear algebra framework of the Combinatorial BLAS. The authors achieve a "... reasonable load-balanced graph traversal..." , by randomly shuffling the vertices before the actual data partitioning.

While there has been conducted an extensive body of research on graph partitioning for distributed BFT, Shang and Kitsuregawa (2013) were the first to propose an adaptive partitioning strategy based on the degree distribution of the vertices. The authors argue that high-degree vertices cause a large communication overhead and propose a hybrid partitioning approach, where the incident outgoing edges of low-degree vertices are par-

titioned by the source vertex and for high-outdegree vertices partitioned by their target vertex. To store the information which vertex belongs to which group, the authors use a bitset structure encoding the binary information for each vertex in the graph. The bitset is replicated to each computing node.

Checconi and Petrini (2014) present a parallel distributed-memory BFT implementation on a cluster of 64,000 BlueGene/Q nodes. They achieve an impressive traversal ratio of 15.3 trillion edges per second on an R-MAT graph at scale factor 40 (1 trillion vertices and 32 trillion edges) on 4 million threads in total. The graph is stored in a 1D decomposition, where all edges incident to a vertex are stored on the same node in the cluster. The internal graph representation is based on a compressed adjacency list and a coarse-grained index structure. The adjacency list is stored as a contiguous array with a source vertex immediately followed by its adjacent vertices. Additionally, the adjacency list maintains shortcut links to jump from one source vertex to the next. The coarse-grained index stores one entry per 64 source vertices, allowing to check a full word of 64 bit in the vertex bitmap data structure and skip all the corresponding vertices if none of the bits is set. They apply direction-optimized BFT, as introduced by Beamer et al. (2012), and alternate top-down and bottom-up traversal based on a simple formula considering the number of unexplored vertices and the number of reachable edges (vertices). The authors leverage *collective operations* available in BlueGene/Q for coordination. Load balancing is achieved by further partitioning high-outdegree vertices across nodes in the cluster. To lower the pressure on the network bandwidth and to use computation and network resource efficiently, they use message batch/compression and overlap computation and communication using an asynchronous communication layer. Vertex sets are represented as bitmaps and partitioned across nodes; the parent array storing parent information for each visited vertex are similarly partitioned.

5.2 ABSTRACT GRAPH TRAVERSAL OPERATOR

In this section we introduce the notion of a graph traversal operator and describe the operator interface, i.e., the specification of the input and the output parameters. The graph traversal operator is one representative from a family of *graph-specific* plan operators, which can be used in combination with other relational plan operators in a heterogeneous execution plan in an RDBMS.

5.2.1 *Traversal Query Specification*

The graph traversal operator receives as input a *traversal query* and returns a set of discovered vertices. In the following we define the traversal query parameters and semantics.

Definition 5 (Traversal Query) *Let* $G := (V, E)$ *be a graph according to Definition 1. A traversal query is defined as a tuple* $\rho := (S, \varphi, c, r, d)$ *composed of a set of start vertices* $S \subseteq V$*, an edge predicate* φ*, a collection boundary* c*, a recursion boundary* r*, and a traversal direction* d*. A graph traversal* $\tau_G(\rho)$ *is a unary operation on* G *and returns a set of discovered vertices* $R \subseteq V$*.*

The vertex set S consists of all the vertices from where the traversal operators starts the graph exploration. In most cases the vertex set S consists of a single vertex, but multiple root vertices are also supported.

The edge predicate φ defines a propositional formula consisting of atomic attribute predicates that can be combined with the logical operators `AND` ($\wedge$), `OR` ($\vee$), and `NOT` ($\neg$). To specify or change the predicate evaluation precedence, atomic attribute predicates can be arbitrarily nested using additional parentheses. The evaluation of the edge predicate φ returns a set of *active edges* $E_{\text{active}} \subseteq E$ satisfying the filter condition and restricts the graph traversal to a subgraph $G' := (V, E_{\text{active}})$.

The recursion boundary $r \in \mathbb{N}^+$ denotes the maximum number of traversal levels to process and refers to the maximum traversal depth. The upper bound of the recursion

boundary is defined by the diameter of the graph, i.e., the longest shortest path between any pair of vertices in the graph. To support unrestrictive traversals or transitive closure computations, the recursion boundary can be set to infinite (∞).

The collection boundary $c \in \mathbb{N}$ denotes the traversal depth from where the operators starts collecting discovered vertices for the final result set. For $c = 0$, the operator adds all vertices from the vertex set S—the root vertices—to the result set. For any traversal configuration, the condition $c \leqslant r$ must hold.

The traversal direction $d \in \{\rightarrow, \leftarrow\}$ can be either *forward* or *backward*. A forward traversal ($\rightarrow$) traverses edges from the source vertex to the target vertex, a backward traversal ($\leftarrow$) traverses edges in the opposite direction. This is an important distinction from other traversal implementations, which either assume that the graph is *undirected* or only support forward traversals. The traversal operation outputs a set of vertices R that have been discovered in the range defined by the collection and the recursion boundary.

5.2.2 *Formal Description*

We define a graph traversal as a totally ordered set P of path steps, where a path step describes the transition between two consecutive traversal iterations and is evaluated sequentially according to the total ordering in P. The maximum number of path steps is bounded by the minimum of the recursion boundary and the graph diameter.

We define a graph traversal based on the mathematical notion of sets and their basic operations *union* and *complement*. Each path step $p_i \in P$ with $1 \leqslant i \leqslant \min\{r, \text{diam}(G)\}$ receives a set of vertices D_{i-1} with $D_{i-1} \subseteq V$ discovered at level $i-1$ and returns a set of adjacent vertices D_i with $D_i \subseteq V$. Initially we assign the set of start vertices to the set of discovered vertices ($D_0 = S$). In the following we define the transformation rules for a path step p_i with $i > 0$ for forward ($\rightarrow$) and backward ($\leftarrow$) traversals, respectively.

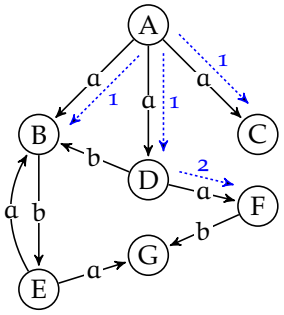
$$D_i^{\rightarrow} = \left\{ v \;\middle|\; \exists u \in D_{i-1} : e = \langle u, v \rangle \in E_{\text{active}} \wedge \underset{0 \leqslant k \leqslant i-1}{\forall} D_k : v \notin D_k \right\} \tag{5.1}$$

$$D_i^{\leftarrow} = \left\{ u \;\middle|\; \exists v \in D_{i-1} : e = \langle u, v \rangle \in E_{\text{active}} \wedge \underset{0 \leqslant k \leqslant i-1}{\forall} D_k : u \notin D_k \right\} \tag{5.2}$$

$$R = \underbrace{\left(\bigcup_{i=c}^{r} D_i \right)}_{\text{target vertices}} \setminus \underbrace{\left(\bigcup_{i=0}^{c-1} D_i \right)}_{\text{visited vertices}} \tag{5.3}$$

In path step p_i, we construct the set of vertices D_i by traversing from each vertex in D_{i-1} over all outgoing/incoming edges in E_{active} satisfying the edge predicate. We handle cycles in the graph by marking each discovered vertex as *visited* and add a vertex only to the vertex set D_i if it has not been discovered in a previous traversal iteration and is therefore not member of one of the other vertex sets $D_0, \ldots, D_{i-1}$. After the completion of the path step, the vertex set D_i contains all the reachable vertices within one hop from the vertices in D_{i-1} via edges from the set of active edges E_{active}.

We define the resulting vertex set R in Equation 5.3. The collection boundary c and the recursion boundary r divide the discovered vertices into two disjunct vertex sets. The set of *visited vertices* contains all vertices that have been discovered before the traversal reached the collection boundary c. Vertices from the set of visited vertices are not considered for the final result, but are required to complete the traversal operation. The set of *target vertices* refers to the set of vertices that are potentially relevant for the final result set. We build the set of visited vertices by computing the union of all vertex sets $\{D_0, D_1, \ldots, D_{c-1}\}$ from path steps p_1 to p_{c-1}. To produce the set of target vertices, we union all vertex sets $\{D_c, \ldots, D_r\}$ from path steps p_c to p_r. To retrieve the final result, we compute the complement between the set of target vertices and the set of visited vertices.



Traversal configuration	Result
$(\{A\}, \text{“type} = a\text{”}, 0, 1, \rightarrow)$	$\{A, B, C, D\}$
$(\{A\}, \text{“type} = a\text{”}, 1, 1, \rightarrow)$	$\{B, C, D\}$
$(\{A\}, \text{“type} = a\text{”}, 2, 2, \rightarrow)$	$\{F\}$
$(\{A\}, \text{“type} = a\text{”}, 1, \infty, \rightarrow)$	$\{B, C, D, F\}$
$(\{E\}, \text{“type} = b\text{”}, 2, 2, \leftarrow)$	$\{D\}$
$(\{A\}, \text{“type} = a \text{ OR type} = b\text{”}, 2, 2, \rightarrow)$	$\{E, F\}$

Figure 5.1: Example traversal queries and their corresponding results.

Figure 5.1 depicts a set of traversal queries and their corresponding results on the given example graph. For example, traversal query $(\{A\}, \text{“type=a”}, 2, 2, \rightarrow)$ performs a 2-hop traversal starting from vertex A and traverses on edges of type a. Here, we only collect discovered vertices in the last path step p_2 that have not been already discovered at path step p_1. Dashed arrows with numbers illustrate the traversed edges and the path step they were discovered in. First, path step p_1 transforms the vertex set $D_0 = \{A\}$ into the vertex set $D_1 = \{B, C, D\}$. Next, path step p_2 transforms vertex set D_1 into vertex set $D_2 = \{F\}$. Finally, the output of the traversal query is a vertex set containing vertex F only. In contrast, traversal query $(\{A\}, \text{“type=a OR type=b”}, 2, 2, \rightarrow)$ returns vertices E and F only, although vertex B can be reached in 2 hops as well. Since vertex B has been discovered after the first hop, it becomes a member of the visited vertices set and is therefore removed from the final result set.

5.3 GRAPH TRAVERSAL OPERATOR IMPLEMENTATIONS

We now discuss the implementation of the abstract specification of the graph traversal operator. We subdivide the processing of a traversal operation into three processing phases—a *preparation phase*, a *traversal phase*, and a *decoding phase*—as depicted in Figure 5.2. Although the separation of the graph traversal operator into three different processing phases might seem artificial in the first place, our aim is to provide a unified graph traversal operator interface while still allowing to extend or replace the *traversal kernel* with other implementations.

Preparation Phase

Initially, the preparation phase of the traversal operator receives a traversal query ρ from some external interface, i.e., a graph query language. We transform the set of start vertices S into a set of internal numerical value codes that stem from the dictionary encoding of the source/target vertex column of the edge column group. In addition, we evaluate the edge predicate φ on the edge column group and retrieve a set of *active edges* E_{active}. Depending on the selectivity of the edge predicate, we choose between a dense, bitset-based representation for low selectivities or a sparse array-based representation for high selectivities, respectively. Finally, we pass the set of active edges to the traversal phase for further processing.

Traversal Phase

We propose two traversal operator implementations, a *level-synchronous* (cf. Section 5.3.2) and a *fragmented-incremental* (cf. Section 5.3.3) traversal strategy. We select the optimal traversal operator implementation based on collected graph statistics and the properties of the traversal query. In an offline process, we collect information about the graph diameter,

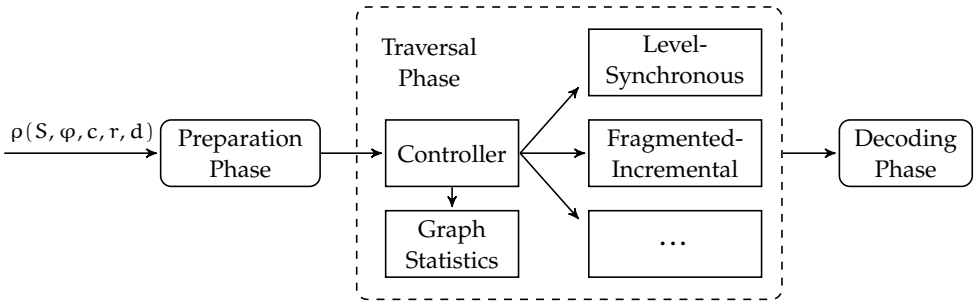


Figure 5.2: Graph traversal processing phases.

neighborhood size distribution, and degree distribution. In addition to graph statistics, we also take the query parameters, such as the root vertex, the edge predicate, and the traversal depth, into account. We use a *traversal controller* to estimate the execution cost for each traversal implementation and select the implementation with the lowest cost.

Initially, we pass a collection boundary c, a recursion boundary r, a traversal direction d, a set of active edges E_{active}, and the encoded vertex set S to the traversal phase. The result of the traversal phase is a set of discovered vertices in an internal, numerical set representation of value codes.

Decoding Phase

To translate the value codes of discovered vertices into the external vertex identifiers, we consult the dictionary of the source/target vertex column for each value code and add the external vertex identifier to the final output set. If the storage engine does not rely on dictionary encoding, the decoding phase can be omitted and the result set can be returned directly without additional translation overhead.

5.3.1 *Traversal Strategies*

Traversal algorithms appear in many variations favoring different graph topologies and types of traversal queries. While a dense graph with a high average degree and a skewed degree distribution benefits from a skew-resilient, level-synchronous traversal algorithm, a sparse graph with a low average degree takes advantage from a more fine-granular traversal strategy. There is a large body of recent work on the analysis of the degree distribution of real-world graphs, including detailed studies of so-called *scale-free* graphs that expose a skewed degree distribution (Leskovec et al., 2005, 2008; Meusel et al., 2014). Although there is a broad consensus stating that graphs from a large variety of application domains follow a power-law degree distribution, this observation only applies when the complete graph is considered. If we look at a property graph with a rich set of attributes attached to vertices and edges and a query workflow that permits initial selections of subgraphs based on some vertex/edge predicates, the general degree measures collected on the entire graph topology do not necessarily reflect the actual topology anymore. The subgraph $G' := (V, E_{active})$, which is the result of a predicate filter evaluation, can (and likely will) expose a different degree distribution than the original graph topology. This observation and the availability of other, not power-law degree distributions, which can be found in road networks and Delaunay graphs, demands a more distinctive analysis of the implementation of graph traversal algorithms for a variety of different graph topologies.

We propose two functionally equivalent traversal strategies, namely a *level-synchronous* (LS) traversal and a *fragmented-incremental* (FI) traversal, which target different graph properties and traversal queries. Table 5.1 summarizes the characteristics of both traversal strategies. The level-synchronous traversal works particularly well on graphs with a small graph diameter, a power-law degree distribution, and for long-running traversal queries with a

	Graph Topology			Traversal Query	
	Degree Distribution	Average Degree	Diameter	Depth	Predicate
LS-traversal	*power-law*	*large*	*small*	*small*	*unselective*
FI-traversal	*uniform*	*small*	*large*	*large*	*selective*

Table 5.1: Overview of traversal strategies and their targeted graph topologies and query characteristics.

large traversal depth. In contrast, the fragmented-incremental traversal favors a large graph diameter, a very sparse graph with a small average vertex degree, and short-running traversal queries with a small traversal depth.

5.3.2 *Level-Synchronous Traversal*

The LS-traversal operates in a level-synchronous manner and discovers vertices in a strict breadth-first ordering. By relying on a breadth-first ordering, reachable vertices are always discovered on the shortest path. The LS-traversal operates on an edge list represented by the columns V_s and V_t, which store source and target vertices of edges, respectively. For each traversal iteration, the LS-traversal scans the complete edge list to retrieve neighboring vertices and returns a set of vertices adjacent to the vertices of the input set. Hence, each edge is scanned possibly multiple times although each edge is only traversed at most once.

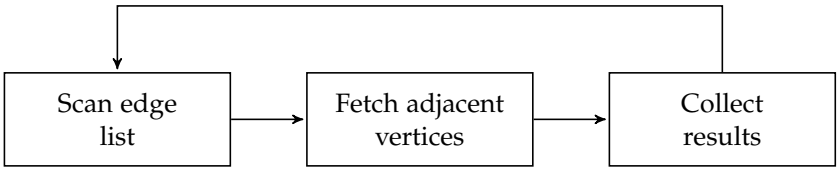


Figure 5.3: General processing steps of the LS-traversal.

We use the set-based formalization of the graph traversal from Section 5.2.2 and implement the LS-traversal based on set operations. The motivations for a set-based implementation are two-fold: (1) set operations can be easily mapped to vectorized operations through SIMD instructions and (2) the traversal formalization does not require to track the traversal path, i.e., the parent relation, but instead only returns a set of vertices discovered at a specific traversal depth and can therefore be implemented with set operations. Aberger et al. (2015) build on a similar idea and formulate graph pattern matching queries as *boolean algebra* expressions. Boolean algebra expressions can be efficiently implemented as set operations by exploiting SIMD processing routines.
In Figure 5.3 we sketch the execution phases of the LS-traversal. We use two important building blocks from the columnar storage engine: *full column scans* and *positional value fetching*. Each traversal iteration is composed of a full column scan, followed by a positional value fetch operation to retrieve adjacent vertices for a given set of vertices. The final *result collection* phase performs cycle handling and merges intermediate results.

Parallelization

We employ data parallelization in the LS-traversal on the *vertex level* and on the *adjacency level* by splitting the source vertex (V_s) and target vertex (V_t) columns into n equal-sized logical edge partitions, respectively. Figure 5.4 depicts a high-level overview of our parallelization scheme with $s_1, \ldots, s_n$ denoting partitions of column V_s and $t_1, \ldots, t_n$ corresponding to partitions of column V_t. We chose a uniform partitioning over a partitioning by source vertex, since a workload imbalance caused by a skewed degree distribution can

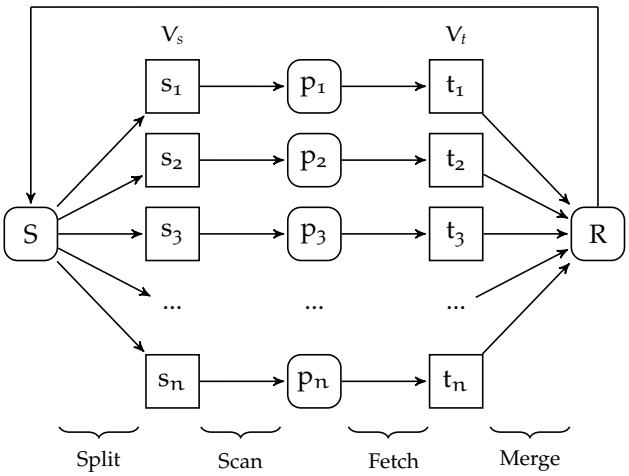


Figure 5.4: LS-traversal algorithm.

have a tremendous negative impact on the execution time of a level-synchronous traversal implementation. We use a *split-join* parallelization strategy, where we assign each partition to a worker thread. Each worker thread receives the vertices to expand from as input and produces a thread-local list of discovered adjacent vertices.

In the *scan* phase each worker thread searches for vertices from the input working set in the local partition s_i with $1 \leqslant i \leqslant n$ and stores hits as record identifiers in a thread-local position array p_i. Since the scan routine operates sequentially within a partition, we can guarantee that the position array is sorted.

Once the scan routines terminates, the same worker thread proceeds with the fetch routine using the local position array p_i to retrieve the adjacent vertices from the local partition t_i. We use arrays to collect thread-local results and bitset data structures to store the final set of discovered vertices.

In the final stage the merge routine collects and combines discovered adjacent vertices from all worker threads and merges them into the vertex set R. A part of this merge routine performs cycle detection, duplicate elimination, and the removal of vertices, which have been already discovered in an earlier traversal iteration. The traversal algorithm either terminates if the recursion boundary is reached or no more vertices have been discovered and forwards its output to the decoding phase, or continues with the next traversal iteration.

Implementation Details

We sketch the implementation of the LS-traversal in Algorithm 4. Initially we pass the traversal query $\rho = (S_m, E_{active}, c, r, d)$ to the traversal kernel, which is derived from the initial traversal query with the following two modifications: (1) the vertex set S_m contains the value-encoded root vertices and (2) the set of *active edges* E_{active} represents the materialized result of the evaluation of the edge predicate φ. The output of an LS-traversal execution is a set of discovered vertices R.

SET-ORIENTED SUBROUTINES. The LS-traversal implementation relies on two subroutines, namely `scan` and `fetch`, which are general data access routines available in most columnar RDBMS. The `scan` procedure receives as input a set of vertices represented by their corresponding value-encoded vertex identifiers (F for *Frontiers*, i.e., the vertices to expand from) and a set of *active edges* E_{active}. The set of vertices is treated as an *in-list* with an equality predicate and the `scan` returns a set of matching positions P, which are represented as an array. Further, matching records, which represent edges in the graph interpretation, are removed from the set of active edges E_{active} to mark the edge as already traversed. The `fetch` subroutine receives a list of positions and extracts the values at the given positions and appends them to the result set F. The input set is sorted to improve the

Algorithm 4: LS-traversal

```
1  Procedure scan(F, E_active, P)
2      forall v ∈ V_s do
3          if v ∈ F ∧ v.rid ∈ E_active then
4              P.append(v.rid);
5              E_active[v.rid] = false;
6  Procedure fetch(P, F)
7      forall p ∈ P do
8          F.append(V_t[p]);
   Input  : Traversal configuration κ = (S, E_active, c, r, d).
   Output: Set of discovered vertices R.
9  begin
10     if d is backward then
11         swap(V_s,V_t) ;                          // Adjust column handles
12     if c = 0 then
13         R ← S ;                                  // Add start vertices to result
14     p ← 1;
15     F ← S;
16     V_vis ← ∅ ;                                  // Visited vertices before collection boundary
17     V_tar ← ∅ ;                                  // Target vertices after collection boudary
18     V_dis ← ∅ ;                                  // Discovered vertices
19     while p ⩽ r do
20         if F = ∅ then
21             return ;                             // No more vertices to discover
22         P ← ∅;
23         V_s.scan(F, E_active, P) ;               // Parallel scan for F
24         V_dis ← V_dis ∪ F ;                      // Mark expanded vertices as discovered
25         F ← ∅ ;                                  // Reset vertex working set
26         V_t.fetch(P, F) ;                        // Fetch adjacent vertices from V_t
27         F ← F \ V_dis ;                          // Remove already discovered vertices
28         if p ⩾ c then
29             V_tar ← V_tar ∪ F ;                  // Add vertices from F to V_tar
30         else
31             V_vis ← V_vis ∪ F ;                  // Add vertices from F to V_vis
32         p ← p + 1;
33     R ← R ∪ (V_tar \ V_vis) ;                    // Compute final result
34     return R;
```

memory locality while extracting values from the column. The `fetch` routine benefits from *edge clustering*, which we introduced in Section 4.3.2.2, through a higher memory locality while accessing values from the column.

ALGORITHM DESCRIPTION. We use two set implementations to store intermediate and final vertex sets, such as F, V_{tar}, V_{vis}, and V_{dis}, that are based on sorted arrays and bitsets, respectively. Both implementations can be seamlessly processed in combination and adjust their internal representation according to the type of the set operation and the cardinalities of both participating sets.

LS-traversal uses internally several data structures to keep track of collected vertices, such as the vertex sets before (V_{vis}) and after (V_{tar}) reaching the collection boundary, the vertex set F to store the frontier vertices of the current traversal iteration, and the set of already discovered vertices V_{dis}.

The LS-traversal can either traverse the graph along the edge direction or opposite to the edge direction. Since we read the edges directly from the columns V_s and V_t respectively, we can easily exchange the handles to both columns without having to change the underlying traversal algorithm. If the collection boundary c is zero, we add all vertices in S to the final result R (Line 13). For the first traversal iteration, we use the vertices from the vertex set S as *frontiers* and assign them to the vertex set F.

The core of the LS-traversal algorithm issues a single traversal iteration and is executed at most r times (Lines 19–32). At the beginning of each iteration we check whether the vertex set contains *frontier vertices*, i.e., vertices from which we can continue the traversal. If the vertex set F is empty, no more vertices can be discovered and the execution of the LS-traversal terminates. In each traversal iteration the LS-traversal scans the source vertex column V_s in parallel and emits matching edges into a position list P. After the scan routine finished, LS-traversal marks all vertices from the frontier set as discovered and stores them in V_{dis}. Next, the LS-traversal algorithm materializes adjacent vertices into the vertex set F. The vertex set F contains potential vertices, which have to be cleaned from already discovered vertices from a previous traversal iteration (Line 27). LS-traversal collects two result sets of vertices, one for the traversal range $\{1, \ldots, c-1\}$ referring to the *visited vertices* (V_{vis}) and one for the *target vertices* (V_{vis}) in the range $\{c, \ldots, r\}$. If the traversal reached the recursion boundary or terminated the traversal earlier, the final result set R is computed as the subtraction of the two vertex sets V_{tar} and V_{vis}.

Cost Estimation

The execution time of the LS-traversal is dominated by the total number of edges in the graph and the number of processed traversal iterations. The LS-traversal has a worst case time complexity of $\mathcal{O}(r \cdot |E|)$, where r denotes the recursion boundary and $|E|$ refers to the total number of edges in the graph. For each traversal iteration, the LS-traversal scans the complete edge list for adjacent vertices from a given vertex set. In Equation 5.4 we provide a cost function to describe the execution time behavior of the LS-traversal. The cost of an LS-traversal can be derived from the number of edges to read and the number of traversal iterations to perform.

$$\mathcal{C}_{LS} = \min\{r, \tilde{\delta}\} \cdot |E| \cdot \mathcal{C}_e \tag{5.4}$$

We define the cost $\mathcal{C}_{LS}$ as the composite product of the minimum of the recursion boundary r and the estimated diameter $\tilde{\delta}$ of the graph, the number of edges $|E|$, and a constant factor $\mathcal{C}_e$ to access a single edge.

Discussion

The LS-traversal operates level-synchronously and scans the complete edge list during each traversal iteration to retrieve the adjacent vertices for a given set of vertices. If the performed number of traversal iterations or the diameter of the graph is small and all

available hardware resources can be utilized, a scan-based graph traversal can provide a reasonable execution performance. In this case we can diminish the computational overhead imposed by the LS-traversal for reading edges multiple times through parallelized scan operations on the edge list. If, however, a single traversal query cannot leverage all available parallelization capabilities of the DBMS—caused by a high query workload with possibly hundreds of traversal queries running in parallel—, the LS-traversal suffers from the *work inefficiency* of the algorithm.

Although the LS-traversal is predestinated for applying *cooperative query processing* techniques to improve the throughput of traversal queries, our focus in the scope of this thesis is on improving single query performance. A recent work by Then et al. (2014) demonstrated the application of cooperative query processing to graph traversals. This technique is, however, limited to power-law distributed graphs with a small diameter and the synchronized execution of a batch of traversal queries. Since GRAPHITE is designed as a general-purpose graph system with support for arbitrary graph topologies and no restrictions on query task scheduling, these techniques cannot be directly applied. Instead we propose a lightweight secondary index structure and a *level-asynchronous* traversal algorithm to limit the number of accessed edges during the traversal.

5.3.3 *Fragmented-Incremental Traversal*

In this section we propose an alternative traversal strategy, which reduces the number of accessed edges compared to the LS-traversal significantly. We build on the general observation that the size of the frontier set in each traversal iteration is not uniformly distributed across all traversal iterations, but instead grows until the traversal reaches a certain traversal depth—the traversal iteration where most of the vertices are discovered—and then shrinks again afterwards (Beamer et al., 2012). While for scale-free graphs with a skewed degree distribution and a small graph diameter the increase of the size of the frontier set is steeper, for very sparse graphs, such as road networks, the increase of the size of the frontier set is smoother. For a large frontier set, the LS-traversal is beneficial; for a small frontier set, a more fine-granular, incremental graph traversal is advantageous. In the following we sketch the general idea of the *fragmented-incremental* traversal strategy.

5.3.3.1 *General Idea*

The FI-traversal divides the edge list represented in columns V_s and V_t into non-overlapping, disjoint *column fragments* and executes the traversal *fragment-wise* instead of *column-wise*. A column fragment contains a subset of the edges of the graph and can be seen as a logical partition of the edge list. The FI-traversal accesses only those column fragments that are relevant for the traversal and skips all other column fragments. Based on the frontier set, the FI-traversal determines the next column fragments to read. In contrast to the LS-traversal, which operates *level-synchronously*, the FI-traversal runs *level-asynchronously* and reads in each traversal iteration only a small portion of the graph instead of the complete edge list. More specifically, the LS-traversal collects frontiers from a single traversal iteration during the scan of the complete edge list; the FI-traversal collects frontiers from multiple traversal iterations during a single read of a column fragment. To determine the set of candidate fragments, the FI-traversal leverages a secondary data structure, the *transition graph*, which stores transitions between column fragments.

5.3.3.2 *Transition Graph*

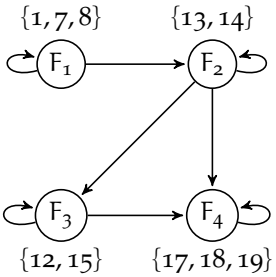
A transition graph $T := (V_F, E_F)$, where $V_F := \{v_{F_1}, v_{F_2}, \ldots, v_{F_n}\}$ represents the set of column fragments and the edge set $E_F \subseteq V_F \times V_F$ represents transitions between column fragments, is a single-relational, directed graph. The transition graph is derived from the graph topology of the data graph $G_D := (V_D, E_D)$ and the edge ordering in the edge column group. Each column fragment F_i with $1 \leqslant i \leqslant n$ consists of an edge set $E_{F_i} \subseteq E_D$

	V_s	V_t
F_1	7	8
	1	7
	8	1
	8	13
	8	14
F_2	13	12
	14	19
	14	15
	14	13
F_3	12	15
	15	19
	15	17
	15	18
F_4	19	18
	18	17
	17	16

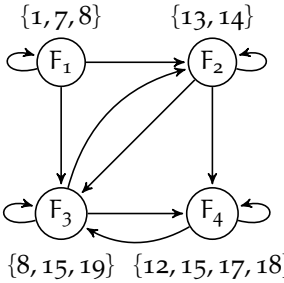
(a) Edge table *(clustered)*

	V_s	V_t
F_1	1	7
	8	1
	7	8
	8	13
F_2	13	12
	14	19
	14	15
	14	13
F_3	8	14
	15	19
	19	18
	15	18
F_4	12	15
	15	17
	18	17
	17	16

(b) Edge table *(unclustered)*



(c) Transition graph *(clustered)*



(d) Transition graph *(unclustered)*

Figure 5.5: Example edge tables and corresponding transition graphs with column fragment size 4.

such that $E_D \coloneqq E_{F_1} \cup E_{F_2} \cup \ldots \cup E_{F_n}$. We add a transition between two column fragments, if the following implication holds:

$$\langle v_{F_1}, v_{F_2} \rangle \in E_F \implies \exists v \in V_D : \langle u, v \rangle \in E_{F_1} \wedge \langle v, w \rangle \in E_{F_2} \tag{5.5}$$

Figure 5.5 depicts two example edge tables *with* and *without* edge clustering enabled and their corresponding transition graphs. A transition between two column fragments indicates the existence of (at least) one path of length two with one edge $e_1 \coloneqq \langle u, v \rangle \in E_{F_1}$ and one edge $e_2 \coloneqq \langle v, w \rangle \in E_{F_2}$. For example, in Figure 5.5 (c) there is a transition between F_2 and F_3 since there is a path $13 \rightsquigarrow 12 \rightsquigarrow 15$ with $\langle 13, 12 \rangle \in E_{F_2}$ and $\langle 12, 15 \rangle \in E_{F_3}$.

Additionally, we store a *column fragment synopsis* attached to each column fragment in the transition graph. A column fragment synopsis $S_{F_i} \coloneqq \{ u \mid \langle u, v \rangle \in E_{F_i} \}$ is a concise representation of the distinct source vertices in the edge set E_{F_i}. For example, the column fragment synopsis of the column fragment F_2 in the transition graph depicted in Figure 5.5 (c) is the set $\{13, 14\}$.

GRAPH DENSITY. When we compare the topologies of the transition graphs in Figure 5.5 (c) and Figure 5.5 (d), we can see that the edge ordering in the edge column group directly affects the shape of the topology and the density of the transition graph. The graph density d is defined as the ratio between edges and vertices and can be computed as $d \coloneqq \frac{|E|}{|V|^2}$. In the context of the FI-traversal, our goal is to exclude as many unrelated column fragments as possible during the traversal. We achieve this by minimizing the density of the transition graph, i.e., by minimizing the total number of edges. While the graph density for the clustered edge column group (cf. Figure 5.5 (c)) is 0.5, the graph density for the unclustered edge column group (cf. Figure 5.5 (d)) is 0.69. Consequently, by applying the edge clustering on the edge column group, we can achieve a significantly lower density of the transition graph. A second important observation is that for a clustered edge column group the column fragment synopses are smaller. In a clustered setup, each distinct source vertex is only member of exactly one column fragment and all outgoing edges are stored in the same column fragment. A column fragment has at most one fragment transition

to any other fragment, including to itself. Based on these observations, we focus for the remainder of this chapter on the FI-traversal on *clustered* edge column groups.

FRAGMENT SIZE. The fragment size ξ is a configuration parameter of the FI-traversal algorithm and denotes the *minimal* size of a column fragment in the transition graph. We choose a lenient fragmentation approach to allow fitting the outgoing edges of high-degree vertices into a single column fragment. In the construction phase, we adapt and increase the fragment size appropriately to store all the outgoing edges of a vertex in a single column fragment. In general, however, a column fragment contains the outgoing edges of multiple vertices. There is a tradeoff between the memory consumption of the transition graph and the size of the individual column fragments. A larger fragment size leads to a transition graph with fewer vertices but a higher density. In contrast, a smaller fragment size leads to a larger transition graph with a higher memory consumption but exposing a lower graph density. We analyze in the evaluation the effect of different fragment sizes on the memory consumption of the transition graph and the query performance of the FI-traversal.

5.3.3.3 *Algorithm Description*

The FI-traversal operates in a *one-fragment-at-a-time* manner and traverses the graph in a series of fragment-level traversals. A fragment-level traversal applies the *scan-and-fetch* technique introduced in the LS-traversal on a column fragment instead of the complete column. In contrast to the LS-traversal, which maintains a single frontier set, the FI-traversal maintains a list of frontier sets, one for each traversal level, and expands all of them while processing a fragment. In contrast to a *level-synchronous* traversal strategy, the FI-traversal does not necessarily discover all the vertices on level i before continuing with the traversal on level $i + 1$.

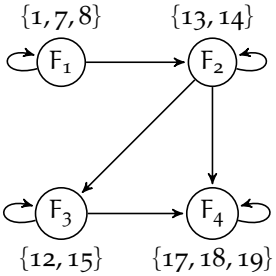
After the adjacent vertices for each frontier set have been computed, the FI-traversal queries the transition graph with the latest discovered vertices and generates new candidate column fragments from where to continue the traversal. The intuition behind this traversal strategy is to only access those column fragments, which contribute to the final result of the traversal operation. Thereby, the transition graph serves as a *navigational index* by providing connectivity information between column fragments.

Since the FI-traversal operates on lists of frontier sets, processing a single fragment usually generates multiple new candidate fragments. In addition, each discovery of a vertex can trigger the generation of an individual candidate column fragment. We only choose and process one column fragment at a time and enqueue all the other generated candidate column fragments in a query-specific priority queue, the *fragment queue*. We set the initial priority of a candidate column fragment to 1, but increase the priority for every subsequent candidate generation of the same column fragment. We store already processed column fragments in a *fragment list* and append the last processed column fragment.
Once the traversal finished processing the column fragment, theoretically all adjacent column fragments in the transition graph can become new candidate column fragments. For each discovered frontier vertex we probe all column fragments, which are adjacent to the last processed column fragment—the tail of the fragment list—and probe their column fragment synopses. If an adjacent column fragment matches, i.e., there is a transition between the tail and the column fragment and the column fragment synopsis contains one of the frontier vertices, we add it to the fragment queue. If the column fragment is already enqueued, we increase its priority. If there are no frontier vertices, we remove the column fragment with the highest priority from the fragment queue and continue the traversal. The traversal terminates, when there are no more column fragments to process, i.e., the fragment queue is empty.

EXAMPLE. Figure 5.6 depicts an exemplary edge table and the corresponding transition graph. Let us consider a traversal query $\rho := (\{8\}, \text{“*”}, 2, 2, \rightarrow)$ that starts at vertex 8 and

	V_s	V_t
F_1	7	8
	1	7
	8	1
	8	13
	8	14
F_2	13	12
	14	19
	14	15
	14	13
F_3	12	15
	15	19
	15	17
	15	18
F_4	19	18
	18	17
	17	16

(a) Edge table.



(b) Transition graph.

Step	*Fragment List*	*Frontiers*	*Depths*	*Fragment Queue*
1	$[F_1]$	$\{1, 13, 14\}$	$1 \to \{1, 13, 14\}$	$\{(1, F_1), (1, F_2)\}$
2	$[F_1, F_1]$	$\{7\}$	$1 \to \{1, 13, 14\}$, $2 \to \{7\}$	$\{(1, F_2), (1, F_1)\}$
3	$[F_1, F_1, F_1]$	$\varnothing$	$1 \to \{1, 13, 14\}$, $2 \to \{7\}$	$\{(1, F_2)\}$
4	$[F_1, F_1, F_1, F_2]$	$\{12, 13, 15, 19\}$	$1 \to \{1, 13, 14\}$, $2 \to \{7, 12, 13, 15, 19\}$	$\varnothing$

(c) Exemplary step-by-step processing.

Figure 5.6: Stepwise processing in FI-traversal for traversal configuration $(\{8\}, \text{“*”}, 2, 2, \to)$.

performs a 2-hop traversal. In Figure 5.6 (c) we show a step-by-step processing with the states of the most important auxiliary data structures. Each row in the table represents the state of the FI-traversal after processing a column fragment and after updating the fragment queue. Since the outgoing edges of vertex 8 all reside in fragment F_1, we start the traversal in this fragment. We perform the fragment-level traversal and generate the set of frontiers, i.e., the vertex set $\{1, 13, 14\}$, and store the discovered vertices and their corresponding traversal depths in *Depths*. Column fragment F_1 has two adjacent fragments, F_1 and F_2, which are probed with the frontiers $\{1, 13, 14\}$. Since vertices 13 and 14 are both in the fragment synopsis of F_2 and vertex 1 is in the fragment synopsis of F_1, we add both candidate fragments to the fragment queue. The second step removes column fragment F_1 from the fragment queue, processes F_1, and generates a frontier set $\{7\}$. Since vertex 7 has not been discovered before, we add it to the *Depth* list and generate a new candidate column fragment F_1. In step 3 we scan column fragment F_1 again for adjacent vertices of 7. Although vertex 8 is adjacent to vertex 7, we omit it from the result, since vertex 8 is the root vertex and therefore has been already discovered. In step 4 we remove the column fragment F_2 from the fragment queue and generate a frontier set $\{12, 13, 15, 19\}$. Since the traversal reached the maximum traversal depth and there are no more fragments to process from the fragment queue, we can terminate the traversal and produce the final result set $\{7, 12, 15, 19\}$. Since vertex 13 has been already discovered in the first traversal iteration, we remove it from the final result.

Algorithm 5: FI-traversal

```
1  Procedure nWayScan(begin, end, F[], E_active, sFactor, P[])
2      forall v ∈ V_s[begin : end] do
3          forall i ∈ {0, 1, ..., sFactor − 1} do
4              if v ∈ F[i] ∧ v.rid ∈ E_active then
5                  P[i].append(v.rid);
6                  E_active[v.rid] = false;

7  Procedure nWayFetch(P[], mFactor, F[])
8      forall i ∈ {0, 1, ..., mFactor − 1} do
9          forall p ∈ P[i] do
10             F[i + 1].append(V_t[p]);

   Input  : Traversal configuration κ = (S_m, E_active, c, r, d).
   Output: Set of discovered vertices R.
11 begin
12     if d is backward then
13         swap(V_s,V_t) ;                          // Adjust Column Handles
14     D_w[0] ← S_m;
15     Frontiers ← S_m;
16     sFactor ← 1;
17     mFactor ← 1;
18     while getNextFragment(Frontiers, F) do
19         V_s.nWayScan(F.begin, F.end, D_w, E_active, sFactor, P);
20         V_t.nWayFetch(P, mFactor, D_w, Frontiers);
21         if sFactor ⩽ r then ++sFactor;
22         if mFactor < r then ++mFactor;
23     R ← generateResult(c, r, D_w);
```

5.3.3.4 *Implementation Details*

After introducing and describing the general idea of the FI-traversal, we sketch the algorithm in Algorithm 5. We pass a traversal configuration ρ with a vertex set S_m, an edge set E_{active}, a collection boundary c, a recursion boundary r, and a direction d to the FI-traversal algorithm. The FI-traversal outputs a set R of vertices that have been discovered between the collection boundary c and the recursion boundary r.

ALGORITHM DESCRIPTION. Since the execution of an FI-traversal is based on sequential reads of fragments, we parallelize the execution of the scan if necessary and materialize operations within a single column fragment. Further, we use compact bloom filters with bits set for all distinct values present in the fragment as fragment synopsis.

An FI-traversal runs in a series of steps, where we process one fragment per step. At the beginning of each step, the algorithm `getNextFragment` receives a set of frontier vertices and returns the next fragment F to read. A fragment contains the start and end position in the column and limits the scan to that range. Initially, we pass the set of start vertices as frontiers to `getNextFragment`. The body of the main loop performs a scan operation and a fetch operation. The scan takes the first *sFactor* working sets from the traversal iterations and returns matching edges in the corresponding position lists from the vector of position lists P. For example, an n-way scan with *sFactor*=2 probes the column with two vertex sets from two different traversal iterations and returns matching edges into two position lists. Subsequently, newly discovered adjacent vertices are materialized in a similar multi-way

Algorithm 6: Procedure `getNextFragment(`*Frontiers*`,`F`)`

Input : Set of frontier vertices *Frontiers*.
Output: Candidate fragment F.

```
1  begin
2      F_last ← m_chain.getLast();
3      foreach outgoing edge e = (F_last, F_cand) from F do
4          foreach v ∈ F do
5              if F_cand.matches(v) ∧ ¬I.hasKey(F_cand, v) then
6                  I.insert(F_cand, v);
7                  if PQ.hasKey(F_cand) then
8                      PQ.increasePrio(F_cand)
9                  else PQ.insert(F);
10     if ¬PQ.empty() then
11         F ← PQ.extractMin();
12         m_chain.add(F);
13         return true;
14     else return false;
```

manner as in the scan operation. Depending on the *mFactor*, we read the collected position lists and add adjacent vertices to the working sets from D_w. In addition, newly discovered vertices are added to the set of frontier vertices *Frontiers*. Once the recursion boundary is reached, the traversal reads and processes all remaining fragments from the fragment queue. If `getNextFragment` does not return any more fragments, the traversal terminates and generates the final result according to the given collection and recursion boundaries.

CANDIDATE FRAGMENT SELECTION. Algorithm 6 describes how to find the next fragment given a set of frontier vertices. It starts with the last processed fragment and probes adjacent fragments for matching vertices. For each adjacent fragment, we consult its fragment synopsis and compare the frontiers against it. If a frontier matches, we update the fragment queue accordingly. If the fragment is already in the queue, we increase its priority, otherwise we insert it. Further, we invalidate vertices in the synopses that triggered the candidate fragment selection. We keep invalidated vertices and their corresponding fragments in an invalidation list I (Line 6). Finally, we return the fragment with the highest priority from the fragment queue and append it to the execution chain. Since the fragment synopses are implemented as compact bloom filters, false positive fragments can occur. However, a false positive does not harm the traversal correctness. There is a tradeoff between space consumption and execution time for the fragment synopses. We evaluate the effect of the size of a bloom filter in the experimental evaluation. Since the value distribution in fragments might vary, each bloom filter can have a different size depending on the number of distinct values present in the fragment.

Cost Estimation

The cost estimation of the FI-traversal is slightly more complex than for the LS-traversal since the calculation of the costs depends on a larger set of input parameters. The costs of an FI-traversal can be directly related to the number and the size of the accessed fragments. Hence, we can use the chain of read fragments F_p to derive the cost of the FI-traversal. The overall cost is the accumulated cost of the reads for all accessed fragments in F_p. Consequently, the traversal cost is not directly dependent on the number of traversal iterations anymore. We define the cost C_{FI} of an FI-traversal in Equation 5.6 as follows.

$$\mathcal{C}_{\text{FI}} = \sum_{i=0}^{\min\{r,\tilde{\delta}\}} (1+p)(\bar{d}_{\text{out}})^i \cdot \xi \cdot \mathcal{C}_e \tag{5.6}$$

The cost depends on the average false positive rate p of the fragment synopses, the average vertex outdegree $\bar{d}_{\text{out}}$, and the fragment size ξ. The FI-traversal is bounded by the minimum of the recursion boundary r and the estimated effective graph diameter $\tilde{\delta}$. The most important factors effecting the memory consumption of the transition graph index are the size and number of fragments. We can minimize the memory consumption of the transition graph index by grouping edges by source vertex (cf. Section 4.3.2.2). Then, each vertex with incoming and outgoing edges contributes exactly once to a single fragment transition. We discuss these configuration parameters and their performance implications for the FI-traversal in the evaluation in Section 5.5.

5.3.4 *Customization by User Code*

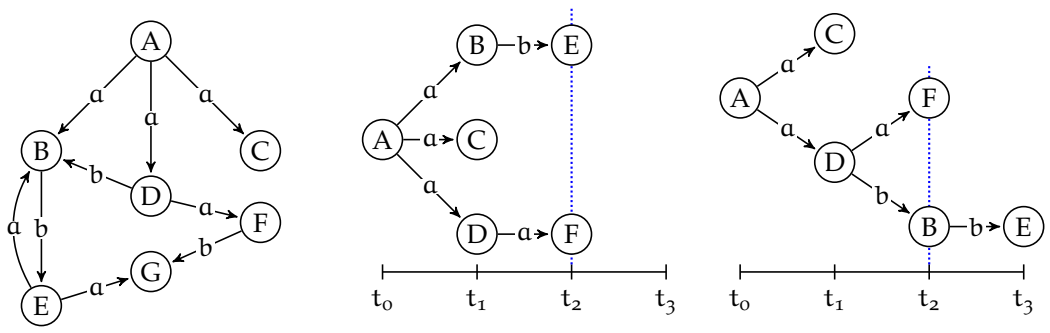
Up to this point, we used the traversal operator in a traditional way as a plan operator inside an RDBMS and it like a black box that receives a well-defined input and produces a single output. This perfectly matches the requirements of a declarative language, where a query is translated into an execution plan with a set of operators. Nevertheless, graph-related applications often demand a more flexible and extensible implementation providing an interface to plug in custom code. For example, an application aggregating vertex attributes at each traversal iteration requires a strict ordering of the discovered vertices during the execution of the traversal operator. At certain checkpoints, the extended traversal implementation relies on the *completeness* and *correctness* of the intermediate traversal result.

Figure 5.7 illustrates two different intermediate result snapshots for two traversal operator implementations. We represent the intermediate results as *traversal trees*, including the discovered vertices and the paths on which they were visited. Both traversal trees depict the intermediate results for the traversal query $(\{A\}, \text{'}type{=}a \vee type{=}b\text{'}, 3, 3, \rightarrow)$ at checkpoint t_2. The left traversal tree shows an intermediate result of a level-synchronous implementation, the right traversal tree sketches an intermediate result of a fragmented-incremental implementation. The FI-traversal shows a different output of the intermediate results compared to the LS-traversal and does not follow a strict breadth-first ordering. The final output for both traversal implementations will be the same, but the intermediate results can be different. Based on this observation, we provide the following definitions to describe these characteristics.

Definition 6 (Intermediate Completeness) *A graph traversal implementation is intermediate complete, if it can provide a complete intermediate result for a running traversal query at well-defined checkpoints. An intermediate result is complete, if all vertices with a distance of exactly* d *have been discovered at each checkpoint* t_d.

Definition 7 (Intermediate Correctness) *A graph traversal implementation is intermediate correct, if it can provide a correct intermediate result for a running traversal query at well-defined checkpoints. An intermediate result is correct, if it is intermediate complete at distance* d *and no other vertices have been discovered at each checkpoint* t_d.

Following these definitions, we conclude that a level-synchronous traversal implementation is both *intermediate complete* and *intermediate correct* and could be used to execute user code at certain checkpoints. In contrast, a fragmented-incremental traversal implementation is not guaranteed to be *intermediate complete* and therefore also not guaranteed to be *intermediate correct*. Consequently, a fragmented-incremental traversal implementation can be used only in combination with custom code, if the code does not rely on *intermediate completeness* nor on *intermediate correctness*. We discuss further challenges for the integration of user code into traversal operators in Chapter 7.



(a) Example graph. (b) LS-traversal. (c) FI-traversal.

Figure 5.7: Traversal trees for LS-traversal and FI-traversal for traversal configuration $(\{A\}, 'type{=}a \vee type{=}b', 3, 3, \rightarrow)$.

5.4 DISTRIBUTED TRAVERSALS IN SHARED-MEMORY

So far we were only concerned with graph traversals on graphs that reside in a single storage container and are not partitioned either on the same machine or across multiple machines. We distinguish between *shared-memory* graph traversals, where the graph resides on a single machine but is split into graph partitions, and *shared-nothing* graph traversals, where the graph is partitioned across multiple hosts. Since our aim to provide a single-node graph processing solution, we focus our attention in this section on distributed, shared-memory graph traversals. The reason for separating the graph into partitions on a single machine can be many-fold: (1) For example, the graph can be partitioned across NUMA nodes to improve load balancing of parallel graph algorithms across several threads. (2) Since our graph storage is based on column groups, the graph storage can be horizontally partitioned based on a round robin or a hash-/range-based partitioning strategy to improve query performance or to circumvent table size restrictions. (3) Vertices and edges are not directly inserted into the (possibly compressed) graph representation, but instead temporarily stored in a write-optimized buffer storage. Query processing on a graph storage, which separates the graph into a static read-optimized storage and a dynamic write-optimized storage, automatically leads to distributed graph queries on multiple graph partitions.

	Partitioning Strategy	Size Distribution	Number of Partitions
Read/Write Separation	*update-driven*	*read: large* *write: small*	*up to 2*
Column Group Partitioning	*round-robin/hash*	*uniform*	*arbitrary*

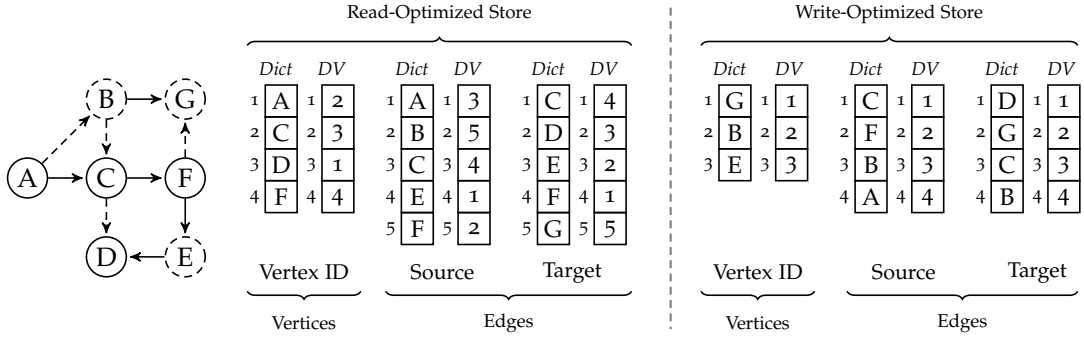
Table 5.2: Overview of graph partitioning and separation strategies.

Table 5.2 summarizes the employed graph partitioning and separation techniques in GRAPHITE. For the graph separation into a read-optimized and a write-optimized graph partition, the partitioning scheme is driven by the query workload, i.e., the order of graph manipulation operations. In contrast to a graph partitioning, which aims at minimizing a certain cost function—such as minimizing cross-partition edges—, the graph separation leads to an non-optimal graph partitioning. We refer to this as *update-driven* as the DBMS cannot influence the partitioning criteria. Since the write-optimized graph partition is only a buffering data structure, the inserted vertices and edges are periodically merged into the read-optimized graph partition. Therefore, we assume that the read-optimized graph storage contains a large fraction of the entire graph, i.e., more than 90% of all vertices and edges, and the write-optimized graph storage contains less than 10% of the entire

graph. The column group partitioning instead generates a uniform partition size distribution and a well-defined partition criterion, such as partitioning by vertex identifier and edges by source/target vertex. We note here that the two partitioning schemes are nested, i.e., a graph *can* be partitioned into multiple vertex and edge column groups and each vertex/edge column group *is* separated into a read-optimized and a (possibly empty) write-optimized graph storage. Therefore, we consider distributed graph traversals rather as the normal case than as the exception.

5.4.1 Impact of Dictionary Encoding

GRAPHITE employs dictionary encoding on each column individually and maps each distinct value in the column to a 32 bit value code. Dictionary encoding is particularly beneficial on variable-length values, such as elements of type VARCHAR, and for repeating values, i.e., when the value domain is small compared to the number of entries in the column group. Although vertex identifiers are mostly represented by 64 bit unsigned integers, graph data models, such as RDF or the extended property graph data model of SAP HANA represent vertex identifiers as string-based URIS.

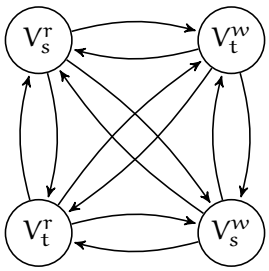


(a) Example graph. (b) Internal graph representation.

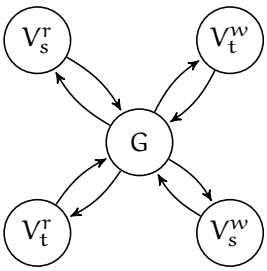
Figure 5.8: Example graph with vertices and edges from read-optimized store (solid) and write-optimized store (dashed).

Figure 5.8 depicts an example graph, where solid lines represent vertices and edges residing in the read-optimized graph store and dashed lines correspond to vertices and edges residing in the write-optimized graph store. The dictionary data structures (*Dict*) represent the mappings of distinct values to value codes for the vertex identifier, the source vertex, and the target vertex, respectively. The data vectors (*DV*) contain the actual data solely using the value code to reference the actual values. The dictionaries of the read-optimized graph store are sorted, the dictionaries of the write-optimized graph storage are not sorted to allow fast insertions. Since a dictionary only captures the value domain of the corresponding column, the same vertex identifier can be mapped to a different value code in each column. For example, vertex C is represented as value code 3 in the read-optimized graph store source vertex dictionary, as value code 1 in the read-optimized graph store target vertex dictionary, as value code 1 in the write-optimized graph store source vertex dictionary, and as value code 3 in the read-optimized graph store target vertex dictionary. If we consider a single vertex/edge column group, each vertex identifier can have 4 different value codes; if we consider multiple vertex/edge column groups as a result of a partitioning, the problem becomes more severe.

To tackle the value coding problem, there are essentially three solutions: (1) Use a global dictionary for all columns sharing the same value domain, (2) operate on values instead of value codes, which results in a higher memory footprint of runtime data structures and increased processing time, and (3) use mapping tables to transform the value codes from one column into the corresponding value codes of the other column.



(a) Mapping between local value codes.



(b) Mapping between global and local value codes.

Figure 5.9: Value code translation schemes.

Although a global dictionary would significantly simplify query processing on encoded columns, it also has major drawbacks, such as more complicated data loading and delta merge routines. Operating directly on values instead of value codes also simplifies query processing but has the disadvantage that it suffers from poor query performance for string-based columns and by giving up the properties of a complete range of possible values. Since value codes are generated strictly sequential and without gaps in between them, compact data structures, such as bitsets can be used to represent sets of value codes. Therefore, we focus in the remainder of this section on the development of light-weight, maintainable mapping structures, which cannot only provide a mapping between columns in the read- and write-optimized graph storage, but also across multiple vertex/edge column groups.

5.4.2 *Data Structures*

We employ so-called *translation tables*, i.e., array data structures, which map each value code from a column C_1 to the corresponding value code in column C_2, such that the corresponding external values are equal. Figure 5.9 (a) depicts the classical mapping scheme between the four columns that encode the graph topology—two columns V_s^r and V_t^r in the read-optimized graph storage and two columns V_s^w and V_t^w in the write-optimized graph storage. The resulting mapping graph is fully connected, i.e., each column maintains a *local mapping table* to every other column. For k columns, the total number of local mappings is $k \cdot (k-1)$. This approach has three major drawbacks, namely a high memory consumption, a high maintenance overhead in the presence of updates, and a poor scalability for a large number of multiple column group partitions.

Instead of relying on a *local-to-local* mapping, we propose a *global-to-local* mapping, which leverages the fact that we can easily build a virtual, global dictionary G across all vertices present in the graph. Since the vertex identifier column is a primary key on the vertex column group and uniquely identifies a vertex in the graph, we can generate globally unique value codes for all vertices. This is based on the fact that a vertex can either reside in the read-optimized graph storage *or* in the write-optimized graph storage, but never in both at the same time. Figure 5.9 (b) depicts the translation mapping scheme for the *global-to-local* mapping, where the virtual, global dictionary G is at the center and consists of k *global-to-local* mappings. Additionally, each column maintains a *local-to-global* mapping. This effectively reduces the number of translation tables for k columns from $k \cdot (k-1)$ down to $2 \cdot k$.

In Figure 5.10 we depict an example of the graph shown in Figure 5.8 (a) and the corresponding translation tables (global-to-local and local-to-global). The global virtual dictionary contains all vertices in the graph and uses the value codes of the vertices from the read-optimized graph store and adds an offset to the value codes from the write-optimized graph store. The global-to-local mapping transforms a global value code into the corresponding local value code of the column. This mapping is sparsely populated since each column only exposes a subset of the entire vertex set. The global-to-local mapping can be

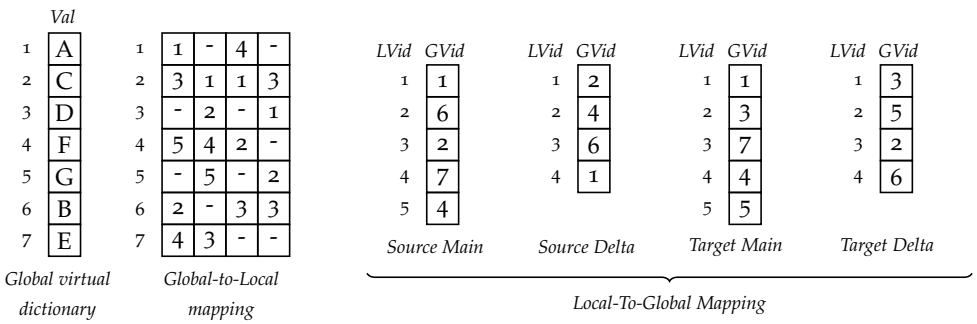


Figure 5.10: Value code mapping for vertex id, source, and target vertex columns.

represented as a set of hash maps with the global value code as key and the local value code as value. Alternatively, each mapping vector could be compressed using a sparse encoding which eliminates all empty entries from the mapping. We keep one local-to-global mapping per column, where we store the mapping as an array of the size of the number of distinct values in the column. Instead of relying on a direct translation infrastructure, we add an indirection, which allows scaling to a large number of column partitions.

5.4.2.1 *Translation Table Maintenance*

After describing the *translation tables* in general, we now sketch the maintenance of translation tables in the presence of edge insertions. Since newly inserted vertices and edges are stored in the write-optimized graph store, we need to consider maintaining the local-to-global mappings and the global-to-local mappings accordingly. We can distinguish the following two cases:

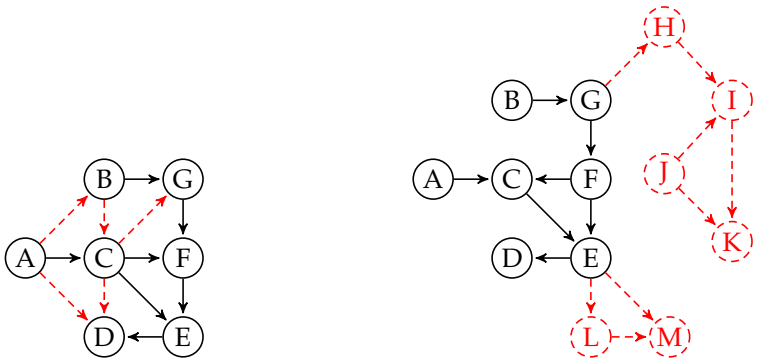
1. Both source and target vertex already reside in the write-optimized graph store.
2. Source or target vertex do not reside in the write-optimized graph store.

If both source and target vertex already reside in the dictionaries of the write-optimized graph store, the translation tables do not need to be updated. In contrast, if any of the two vertices incident to an edge is not present in the dictionaries, it is appended to the dictionary and receives a new local value code. New entries in the local-to-global mapping can be added at the end of the array structure or inserted into the hash map for the global-to-local mappings. If a new vertex is inserted, a new global value code is generated by appending the vertex at the end of the virtual global dictionary. By that, all updates to the translation tables can be performed in constant time. The number of updates is limited to at most four, where two updates are append-only operations to the corresponding local-to-global mappings and two updates are insertions into the global-to-local mappings.

5.4.3 Graph Update Patterns

To be able to efficiently process graph traversal across read- and write-optimized graph store, it is crucial to know how the graph is spread across the two graph partitions. Especially since the partitioning criterion is not driven by the DBMS, but instead by the incoming graph manipulation requests. We distinguish two major graph growth patterns, *graph topology densification* and *graph topology extension*. Figure 5.11 depicts both update patterns where solid black lines denote vertices and edges residing in the static partition of the graph, i.e., the read-optimized graph store; red dashed lines correspond to vertices and edges that are stored in the dynamic partition of the graph, i.e., the write-optimized graph store.

The topology densification is characterized by a growing graph density, i.e., the number of edges grows much faster than the number of vertices. Additionally, the graph becomes



(a) Topology densification. (b) Topology extension.

Figure 5.11: Graph update patterns.

more connected, the average number of adjacent vertices grows, and the overall graph diameter shrinks.

Contrary to the topology densification is the topology extension, which describes the growth of the graph into certain directions by forming new clusters and expanding the topology at the borders of the graph. Topology extension can be characterized by a large number of vertex and edge insertions, where edges are primarily added between recently added vertices and not by connecting old vertices.

Although collecting statistical information about the graph growth is useful to efficiently process graph traversals on multiple graph partitions, the growth type and rate might change over time.

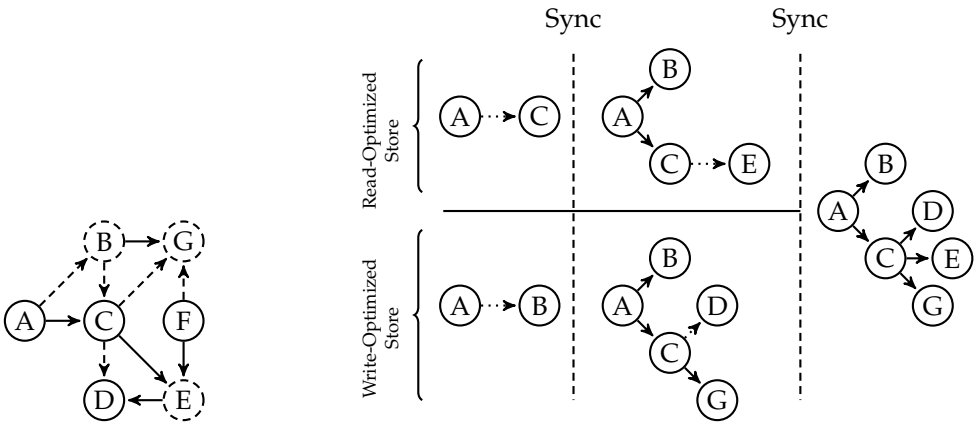
5.4.4 *Query Processing*

In the following we describe two graph traversal strategies on distributed graphs that are partitioned across a read-optimized and a write-optimized graph store.

5.4.4.1 *Distributed Level-Synchronous Traversal*

The distributed level-synchronous traversal works similar to the non-distributed algorithm described in Section 5.3.2. During each traversal iteration, we perform a single-hop traversal individually on the read-optimized and the write-optimized graph store. After each traversal iteration, a global synchronization barrier ensures that all local traversals have been completed before the next traversal iteration starts. Figure 5.12 depicts an exemplary traversal for the traversal configuration $(\{A\}, \text{“*”}, 2, 2, \rightarrow)$, where dotted edges represent expanded edges in the current traversal iteration. The first iteration expands the edge $\langle A, C\rangle$ in the read-optimized graph store and the edge $\langle A, B\rangle$ in the write-optimized graph store in parallel. At the synchronization barrier, both local results are translated into a global vertex set using the translation mappings described in Section 5.4.2, and perform the same checks to handle cycles and for building the frontier vertex set for the next traversal iteration.

Although the distributed level-synchronous traversal has several advantages, such as algorithmic simplicity and good parallelization opportunities, the skewed partition size distribution of the read-optimized and the write-optimized graph store poses an additional overhead and results in an underutilization of available resources. If the edges present in the write-optimized graph store are relevant for the traversal, we have to access both read-optimized and write-optimized graph store. The distributed level-synchronous traversal works particularly well when the graph update pattern describes a topology densification, i.e., new edges are added between already existing vertices. If the write-optimized graph store is very small ($< 5\%$ of all edges), we can materialize the value code mapping by



(a) Example graph. (b) Traversal iterations.

Figure 5.12: Exemplary partitioned level-synchronous traversal for configuration ({ A }, "*", 2, 2, →).

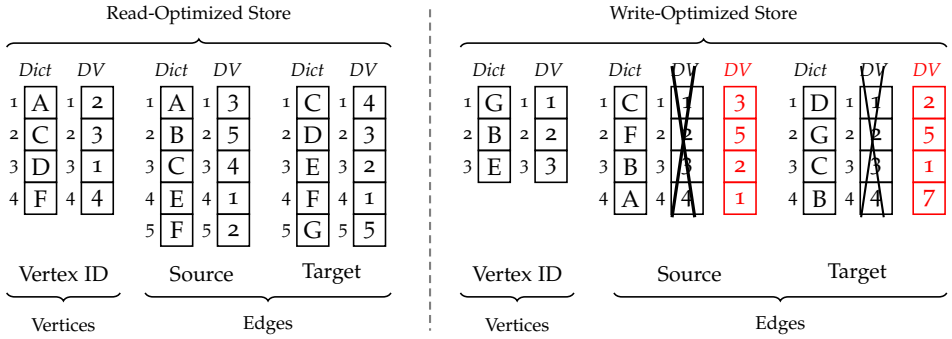


Figure 5.13: On-the-fly value code reencoding.

encoding the data vectors of the write-optimized graph store *on-the-fly* as a preprocessing step.

Internal Reencode

We refer to this strategy as *internal value reencoding*, which transforms the value codes of the write-optimized graph store into the value codes of the read-optimized graph store. By doing so we perform a *simulated delta merge*, where we reencode the data vectors but do not persist the changes. Figure 5.13 depicts an example for a value code reencoding. If the corresponding value code exists, the local value code of the corresponding column, otherwise we compute an offset encoding, where the new value code v_n in a column C is computed as $v_n = v_w + |C|$, where v_w corresponds to the value code in the write-optimized graph store.

5.4.4.2 *Successive Graph Traversal*

When the write-optimized graph store only represents a small fraction of the entire data graph and the graph topology grows by extending the graph at the boundaries, we can use an optimistic approach assuming that we can serve the traversal from the read-optimized graph store only. The algorithm performs the traversal first on the read-optimized graph store only and keeps intermediate frontier vertex sets for validation purposes. When the traversal finishes, the kept intermediate vertex sets are used to validate that all reachable vertices have been visited in the correct order. Figure 5.14 depicts an example traversal, where the traversal on the read-optimized graph store can be processed independently from the processing of the traversal on the write-optimized graph store. Since the visitation

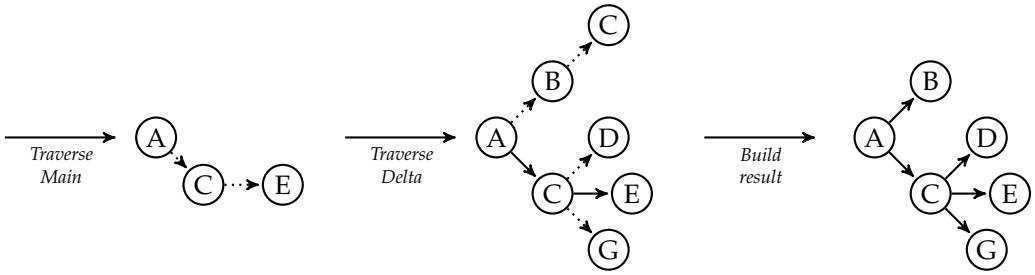


Figure 5.14: Example of mixed topology updates for (2,2) traversal.

order might vary, we have to validate the result and eliminate all vertices from the final result, which have not been discovered on the shortest path (for example vertex C has been first discovered at depth 1 in the read-optimized graph store, but again at depth 2 in the write-optimized graph store). If the validation phase reveals that relevant edges exist in the write-optimized graph store, then we have to probe the read-optimized graph store again afterwards. We refer to this as *traversal ping ponging*, where paths across the two partitions lead to an increased synchronization overhead.

To summarize, we described two alternatives for processing distributed graphs, where the graph topology is spread across two graph stores, a read-optimized one and a write-optimized one. A distributed level-synchronous graph traversal performs well on graphs, where the graph topology becomes denser over time, i.e., edges are mainly inserted between already existing vertices. In contrast, the successive traversal first operates on one graph store, the read-optimized store, and then performs a validation of the partial result against the write-optimized graph store. This approach performs well, when the update pattern extends the graph topology, i.e., adds new edges by adding new vertices. We can categorize the level-synchronous traversal as *pessimistic* approach, where we assume that all graph partitions contribute to the traversal, and the successive traversal as *optimistic*, where we assume that the complete traversal can be processed from a single graph partition in most cases.

5.5 EXPERIMENTAL EVALUATION

We evaluate the LS-traversal and the FI-traversal on a diverse set of real-world and generated graphs and for different types of graph traversal queries. In the following, we describe the environmental setup and the evaluated data sets. We evaluated different aspects of both traversal implementations, including the memory consumption, the execution time, and a system-level comparison with a native disk-based GDBMS and two main-memory, columnar RDBMS.

5.5.1 *Setup and Data Sets*

We implemented the LS-traversal and the FI-traversal algorithm in GRAPHITE as traversal kernels in our traversal framework that we introduced in Section 5.3. Initially, we load the data sets into their corresponding vertex and edge column groups in GRAPHITE, and populate the transition graph index for the FI-traversal .

We conducted all experiments on an INTEL® XEON® E5-2660 machine with 2 sockets, 10 cores per socket, 2 threads per core, each core running at 2.6 GHz. The machine runs on SLES 12 SP1 and is equipped with 128 GB of DDR4 RAM and 25 MB last level cache. We compiled GRAPHITE using GCC 4.9.3 with compiler options `-O3` and `-march=native`. For the LS-traversal we use all available threads of the machine, for the FI-traversal we use 1 thread to scan a single fragment.

To evaluate our approach on a wide range of different graph topologies, we selected six real-world graph data sets from the domains: social networks (ORKUT, TWITTER, LIVE-

JOURNAL), citation networks (PATENTS), autonomous system networks (SKITTER), and road networks (CALI). Additionally, we evaluated LS-traversal for generated data sets using the R-MAT data generator. Table A.1 summarizes the most relevant topology properties of the evaluated data sets.

All evaluated queries are of the form $\{\{ s \}, '*', k, k, \rightarrow\}$, where s is a randomly selected start vertex, '*' refers to a nonselective edge filter, and k denotes the traversal depth. Without losing generality, we focus in the evaluation on traversal queries where the collection boundary is equal to the recursion boundary. Such traversal queries only return vertices first discovered in traversal iteration k. For the runtime analysis, we randomly selected start vertices for the traversal and report the median execution time over 50 runs. We decided to report the median since the execution highly varies for different start vertices.

We compare the LS-traversal and FI-traversal implementations against two join-based approaches (with and without secondary index support) in SAP HANA and the community edition of the native GDBMS NEO4J 3.0.3 (Robinson et al., 2015). Before we ran the experiments, we prepared and configured the evaluated systems as follows:

NEO4J

We use the NEO4J 3.0.3 Community Edition and CYPHER, NEO4J's declarative query language, to run the experiments. Initially, we load the graphs into the system, where each graph consists of a vertex set with attribute *id* and an edge set with attributes *start node*, *end node*, and *label*. We followed the instructions provided by NEO4J and configured the page cache and the heap size of the JVM such that the entire graph fits into the main memory of the machine. We warmed up the caches by running randomly 100 traversal queries against the database instance, prior running the experiments.

Listing 5.1: General structure of the evaluated CYPHER queries.

```
// 2-hop traversal
 MATCH (a)-->(b)-->(c)
 WHERE id(a) = (?)
   AND a <> b AND a <> c AND b <> c
RETURN COUNT(DISTINCT(c));
```

Listing 5.1 shows an example query in CYPHER, which implements our traversal semantics. Since CYPHER follows a *homomorphic* pattern-matching query paradigm with edge uniqueness, we have to avoid the repeated matching of already discovered vertices. We do this by adding additional constraints to the query, i.e., by pairwise adding inequality constraints on all defined vertex variables. Additionally, we found out that the result materialization of discovered vertices does not scale well to growing result set sizes. To overcome this limitation, we only return the number of discovered vertices instead of materializing the complete result set.

SAP HANA

We loaded the graph topology consisting of two columns, one for source vertices and one for target vertices, into an in-memory, columnar table. For the table, we used the following schema:

```
CREATE COLUMN TABLE EDGES(SOURCE INTEGER NOT NULL, TARGET INTEGER NOT NULL);
```

After the initial loading phase, we performed a delta merge on the edge table and loaded the entire table into memory. For the index-assisted experiments, we generated an additional secondary index on the `SOURCE` column. For the generation of the SQL statements that perform the graph traversal, we have to take two important considerations into account: (1) cycles in the graph should be handled gracefully and no cyclic paths should be evaluated during the traversal, and (2) vertices are only returned if they have not been discovered already in an earlier traversal iteration.

Listing 5.2: SQL implementation of a $\{\{1\}, '*', 3, 3, \rightarrow\}$ traversal starting from vertex 1.

```
SELECT DISTINCT(E3.TARGET) FROM EDGES E1, EDGES E2, EDGES E3
 WHERE E1.SOURCE = 1
   AND E1.TARGET = E2.SOURCE
   AND E2.TARGET = E3.SOURCE
   AND E1.SOURCE <> E2.SOURCE
   AND E1.SOURCE <> E3.SOURCE
   AND E1.SOURCE <> E3.TARGET
   AND E2.SOURCE <> E3.SOURCE
   AND E2.SOURCE <> E3.TARGET
MINUS
SELECT DISTINCT(E2.TARGET) FROM EDGES E1, EDGES E2
 WHERE E1.SOURCE = 1
   AND E1.TARGET = E2.SOURCE
   AND E1.SOURCE <> E2.SOURCE
   AND E1.SOURCE <> E2.TARGET
```

Listing 5.2 depicts an implementation of a 3-hop traversal in SQL for a traversal configuration $\{\{1\}, '*', 3, 3, \rightarrow\}$. The structure of the SQL statement follows a similar structure as our set-based formal notation of the traversal, which separates the discovered set of vertices into two different sets, the *visited vertices* and the *target vertices*. We implement this notation using two queries to compute the two vertex sets and removing vertices, which have been already discovered at an earlier traversal iteration from the final result set. To avoid the evaluation of cyclic paths, we add pair-wise inequality constraints on all joined edge tables along the traversed paths. This example serves as our canonical implementation for general traversal implementations for traversal configurations of the shape $\{\{s\}, '*', k, k, \rightarrow\}$ for some root vertex s and some traversal depth k. We note that although this query pattern looks quite repetitive and inefficient, most modern query optimizers can detect that most of the intermediate results can be shared between the two subqueries.

5.5.2 *Memory Consumption*

In this experiment we study the impact of different parameter configurations for the FI-traversal on the memory consumption of the transition graph index. We created transition graph indexes for six different data sets on a clustered edge column group and varied the fragment size ξ and the desired false positive rate p of each fragment synopsis. The results are depicted in Figure 5.15. To evaluate the impact of the fragment size ξ, we construct the transition graph index for different fragment sizes from 2^6 to 2^{16} and a fixed average false positive rate of 1%. Further, we analyze the effect of the average false positive rate for a representative fragment size $\xi = 512$ and construct fragment synopses based on an average false positive rate selected from $\{1\%, 5\%, 10\%, 20\%\}$.

For fragment size $\xi = 1024$, the transition graph index consumes on average only about 10% of the size of the input graph for all evaluated data sets. For a fragment size $\xi = 1024$, the transition graph index of the TWITTER data set has the highest memory consumption with about 2.09 GB (about 8.4% of the raw size of the graph). Similarly, the CALI data set has the lowest memory consumption (about 8.8% of the raw size of the graph). If the edge clustering on the edge column group is disabled, i.e., edges are placed in random order, the memory consumption of the transition graph index grows up to a factor of 10 of the original graph, effectively making it impractical for realistic scenarios.

IMPACT OF FRAGMENT SIZE. For all evaluated data sets, the memory footprint of the transition graph index decreases for increasing fragment sizes. A larger fragment size leads to a smaller number of vertices in the corresponding transition graph and to fewer

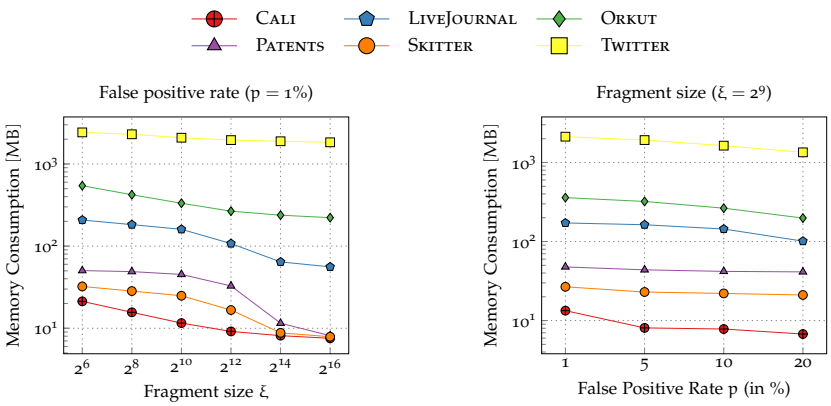


Figure 5.15: Memory consumption of the transition graph index for different false positive rates and fragment sizes.

possible transitions between them. Although larger fragments in general result in a denser transition graph, the total number of fragment transitions is much lower. For very sparse graphs, such as Cali or Skitter, the transition graph index consumes for a fragment size $\xi = 2^{16}$ only up to 37% of the memory compared to a fragment size $\xi = 2^6$.

IMPACT OF FALSE POSITIVE RATE. We store fragment synopses in space-efficient bloom filter structures, where each fragment synopsis occupies as much memory as needed to fulfill the predefined false positive rate. A smaller false positive rate causes the FI-traversal to access more fragments, but reduces the memory footprint of the transition graph index. We show the memory overhead of the transition graph index for different false positive rates in Figure 5.15. For the Patents data set, a false positive rate of 20% leads to a memory footprint decrease of 13% compared to a false positive rate of 1%. In contrast, the Cali data set reaches a decrease in the memory footprint of almost 50% for a false positive rate of 20% compared to a false positive rate of 1%.

5.5.3 *Runtime Analysis*

In Figure 5.16 we present the performance results of the LS-traversal for all data sets and different traversal queries. We report median execution times of the three traversal phases *preparation*, *traversal*, and *decoding* as well as average output sizes. In general, we can see that the traversal phase dominates the overall execution time of the traversal operator and consumes up to 95% of the total runtime. The preparation phase only accounts for about 5% of the overall execution time and is independent from the number of traversal iterations as it only evaluates the edge predicate and processes the start vertices. The execution time of the decoding phase depends on the output set size as it translates for each vertex the internal value code back into the corresponding external vertex identifier. For the Skitter data set, the effect of the output size on the execution time of the decoding phase becomes noticeable, where the output size steadily grows until it reaches its maximum output size after 6 iterations. The LS-traversal scales linearly with an increasing number of traversal iterations since the full column scan takes nearly the same time for each iteration to complete. For traversal iterations with a large frontier set, the scan operation takes slightly longer since more search hits have to be written to the intermediate scan buffer data structures.

In Figure 5.17 we present a comparison of the LS-traversal and the FI-traversal on all evaluated data sets. We evaluate the FI-traversal with a false positive rate of 1% and for fragment sizes ranging from 2^7 to 2^{10}. We also evaluated larger fragment sizes, which resulted in higher execution times and are therefore omitted in the results. For all data sets, LS-

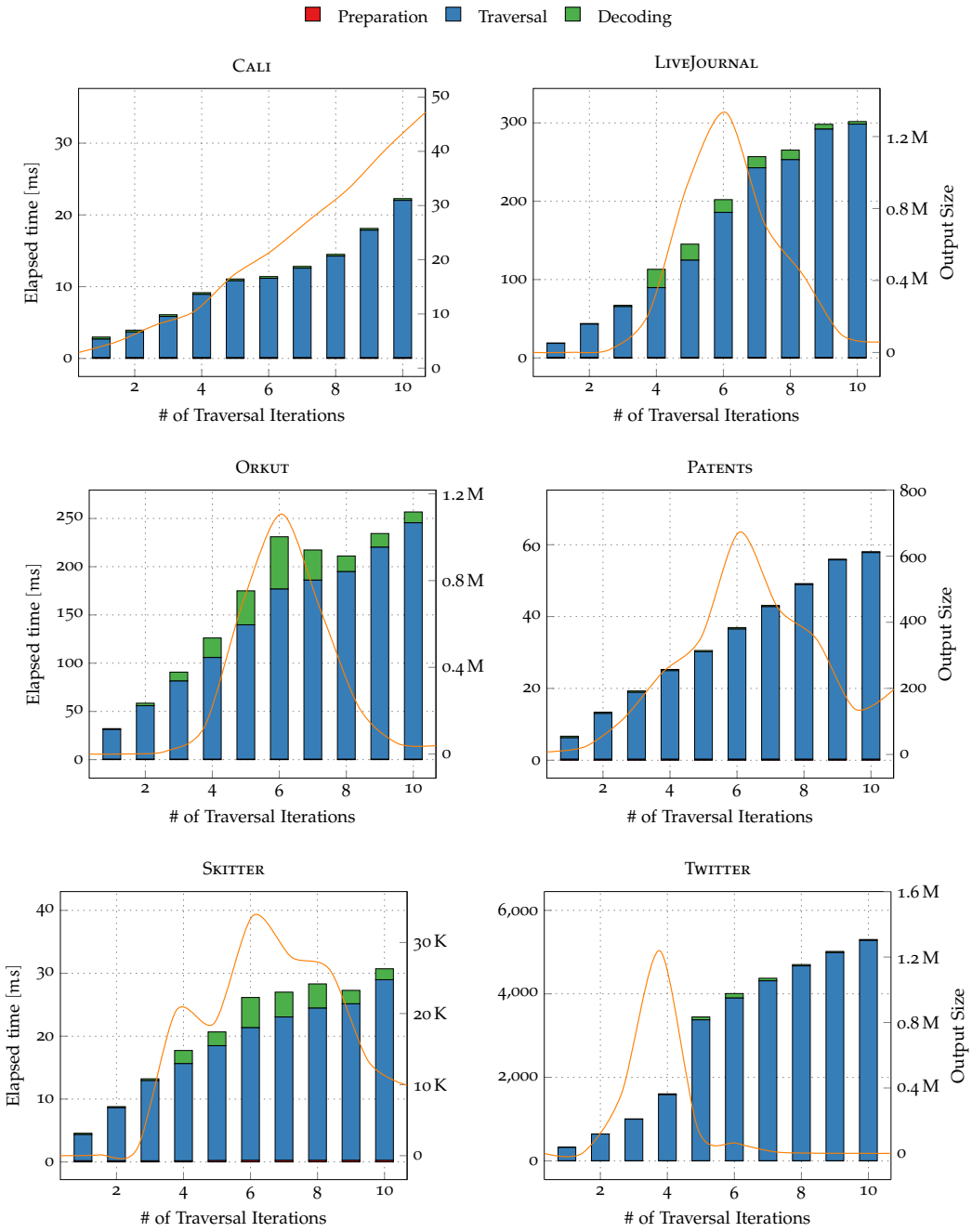


Figure 5.16: Execution time and output size of LS-traversal for different queries and data sets.

traversal exhibits a linear execution time behavior for an increasing number of traversal iterations until the queries reach the effective diameter. After a traversal query reached the effective diameter, the plot flattens for longer traversal queries as most vertices have been already discovered. In comparison, the data plots of the FI-traversal grow much faster for an increasing number of traversal iterations. For short traversals with a small number of traversal iterations, the FI-traversal outperforms the LS-traversal by up to two orders of magnitude. This can be explained with the fine-granular graph access pattern of the FI-traversal. For the first 1 to 3 traversal iterations, only a small fraction of the entire graph is traversed and a more fine-granular access on a fragment level outperforms a full-column scan. For a large frontier set, potentially many fragments have to be accessed which causes a large number of cache misses since new fragments have to be fetched from memory. If a

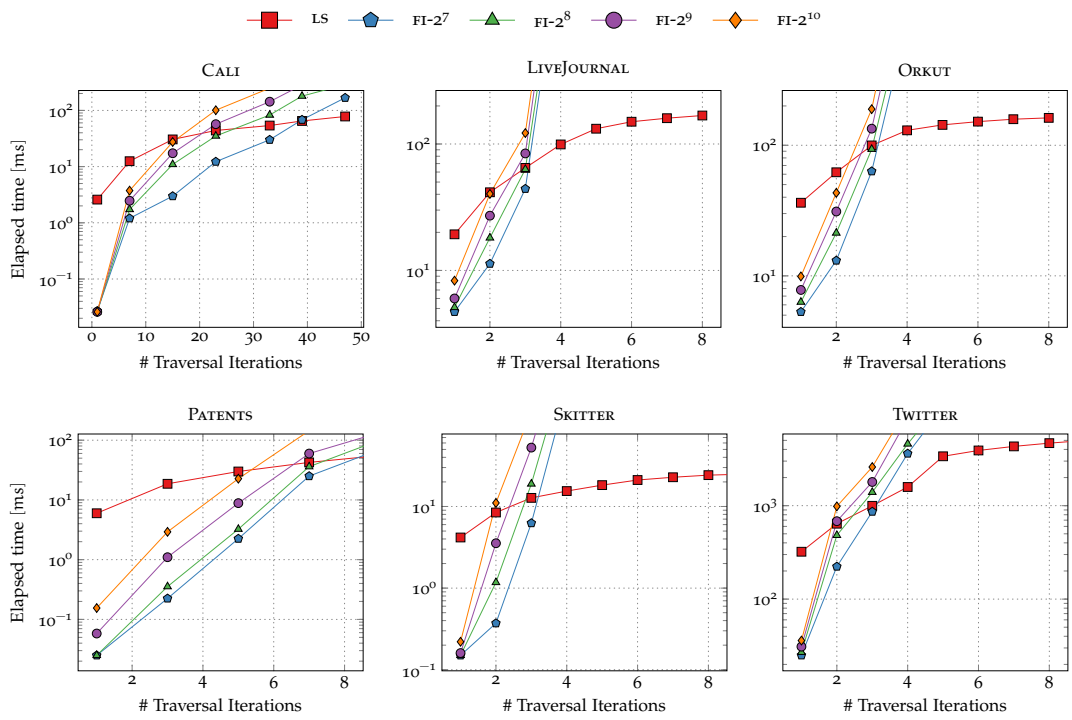


Figure 5.17: Comparison of LS-traversal and FI-traversal for different queries and data sets.

large fraction of the graph is accessed in a traversal iteration, a full-column scan is faster and more cache-friendly compared to a large number of small fragment scans.

The break-even point when the FI-traversal outperforms the LS-traversal depends on the graph topology and the given traversal query. From the experimental results, we observe that for short traversal queries, the FI-traversal outperforms the LS-traversal by up to two orders of magnitude. For four out of six data sets, the FI-traversal outperforms the LS-traversal for traversal queries with $r \leqslant 5$. Another observation that we made is that the fragment size has a severe impact on the overall execution performance of the FI-traversal. For the CALI data set, the fragment size does not only effect the total execution time, but can also increase the range of traversal queries, where the FI-traversal outperforms the LS-traversal. For example, a traversal query with traversal depth 14 on the CALI data set consumes for a fragment size $\xi = 2^7$ only about 26% of the execution time than for a fragment size $\xi = 2^{10}$. In general, the FI-traversal outperforms the LS-traversal for short traversals with traversal depth up to 3 or on very sparse graphs for even deeper traversals. This is caused by the frontier set sizes, which have to be handled during the traversal. For small frontier set sizes, using the transition graph index provides efficient access to a small number of fragments. If the frontier set is large, the LS-traversal outperforms the FI-traversal due to better cache locality during the scan operation.

Figure 5.18 depicts the slowdown factor of the FI-traversal for different fragment sizes ranging from 2^6 to 2^{16}. Further, we analyze the effect of the false positive rate on the query execution time. To compute the slowdown factor, we use the data point for the smallest fragment size and the smallest false positive rate as baseline and relate all other results to this baseline. Without losing generality, we conduct all experiments on a representative query with the following configuration: $\{\{\,s\,\}, '*', 3, 3, \rightarrow\}$.

For all evaluated data sets we observe that smaller fragments whose size is close to the expected average vertex outdegree exhibit lower execution times than larger ones. Although we could theoretically use a fragment size that is very small or even close to 1, the memory overhead would be prohibitively expensive. Thus, we use as lower boundary for the fragment size the average vertex outdegree. A smaller false positive rate increases the

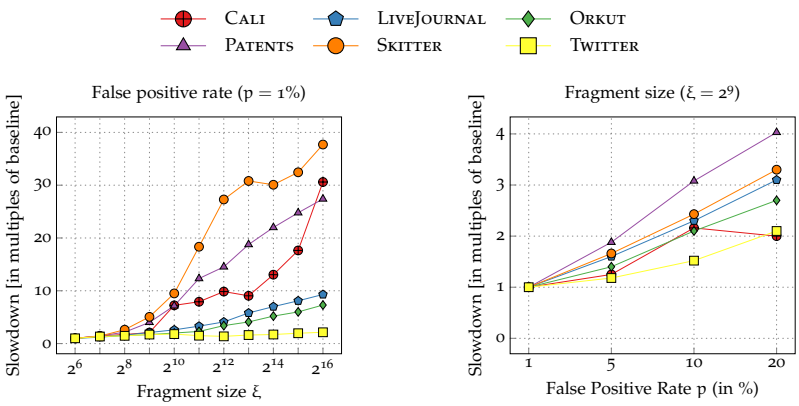


Figure 5.18: Execution time of FI-traversal for {{ s }, '*', 3, 3, →}.

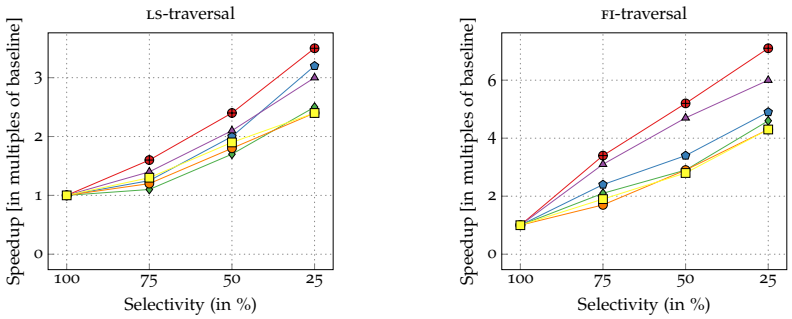


Figure 5.19: Speedup in multiples of baseline for different edge predicate selectivities with query {{ s }, '*', 3, 3, →}. The baseline is a traversal query without edge predicate, i.e., a traversal on the entire graph.

memory consumption of the transition graph index, but also speeds up the execution of the FI-traversal by up to 4×. If the false positive rate is too large, many fragments have to be accessed although they do not contribute to the traversal query result.

5.5.3.1 *Impact of Edge Predicates*

In the next experiment, we study the effect of edge predicates with different selectivities on the query performance of the LS-traversal and the FI-traversal and present our results in Figure 5.19. An edge predicate evaluation produces a subgraph from the complete graph and effectively reduces the traversal to a subset of the edges. We generated random edge weights following a zipfian distribution with $s = 2$ and assigned them to the edges.

For a selectivity of 25%, i.e., an edge predicate that selects only 25% of all edges results in a 2.5× to 3× speedup for the LS-traversal . We observed that an edge predicate with a high selectivity dramatically reduces the size of intermediate results, which subsequently leads to smaller execution times. Since the LS-traversal is a scan-based traversal algorithm, it still has to scan the entire column for each traversal iteration, which results in a lower speedup than for the FI-traversal. In contrast, the FI-traversal reaches a speedup of up to 6× for a selectivity of 25%. If the selectivity is high, more fragments can be pruned during the traversal and cause a larger speedup compared to the LS-traversal.

5.5.3.2 *System-Level Benchmarks*

We compared our two traversal implementations, LS-traversal (—■—) and FI-traversal (—⬟—), against a join-based approach implemented in SAP HANA with (—◆—) and without (—▲—) secondary index support, and the native GDBMS Neo4J (—●—). Figure 5.20 depicts the results.

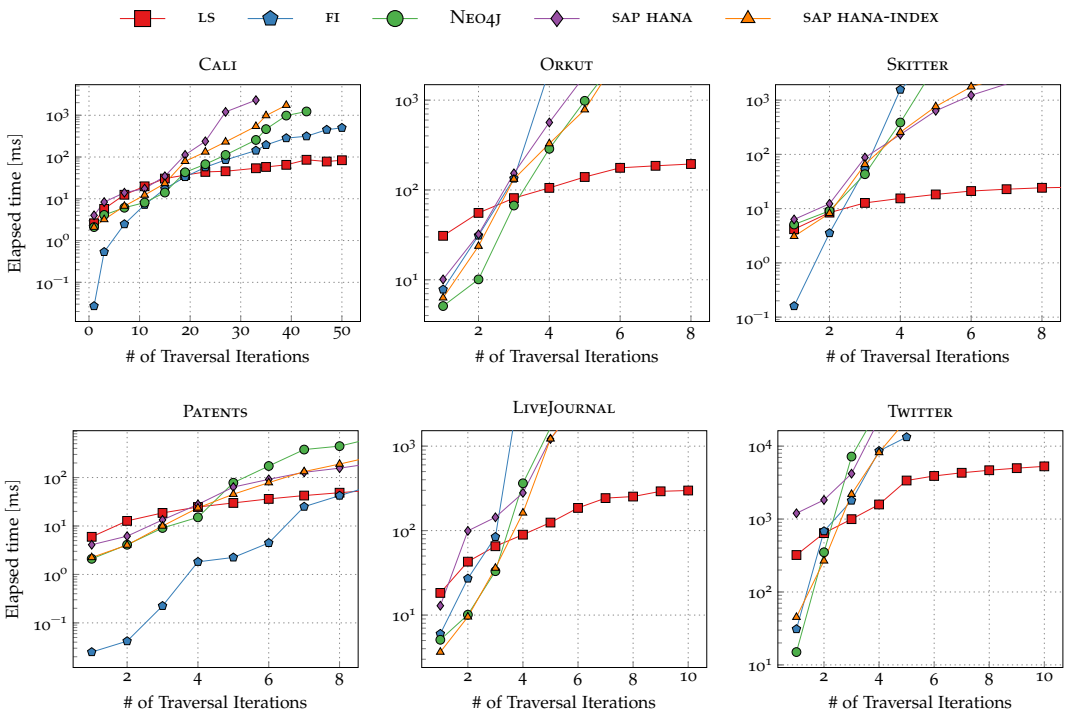


Figure 5.20: Comparison of LS-traversal (—■—), FI-traversal (—⬟—), NEO4J (—●—), and a self-join-based approach in SAP HANA with (—◆—) and without (—▲—) secondary index support.

For graphs that expose a power-law distribution, we see similar execution times for short traversals of 1 to 2 hops for SAP HANA and NEO4J. With increasing traversal depth, the sizes of intermediate results grow and are directly reflected in a steep rising execution time for the evaluated systems. Surprisingly, even SAP HANA does not handle large intermediate results of repeated self-joins well. The FI-traversal shows a comparable performance for short traversals with up to traversal depth 3 and significantly outperforms the join-based approach by up to an order of magnitude and NEO4J by up to a factor of 4. While the FI-traversal fails to scale well for longer traversals with large intermediate results, the LS-traversal shows a good scalability for longer traversals and outperforms the other systems by up to an order of magnitude. This can be explained by the careful handling of intermediate results in compact bitset data structures.

5.6 SUMMARY

In this chapter we proposed a graph traversal operator that is based on a BFT, but extends it with support for edge predicates and a more fine-granular steering of the traversal depth and the result set construction. Based on a formal description of the abstract traversal operator, we proposed two traversal implementations—LS-traversal and FI-traversal —to support a wide range of different graph topologies and graph traversal queries efficiently. We implemented the LS-traversal as a level-synchronous traversal based on repeated full-column scans over the edge column group. The FI-traversal avoids to scan the entire edge column group for each traversal iteration by consulting during each traversal iteration a light-weight secondary index structure—the transition graph index. The transition graph index provides detailed information about column fragments and potential transitions between them during the traversal. Thereby, it employs a *fragment-at-a-time* processing model and processes the traversal asynchronously. The FI-traversal outperforms the LS-traversal for graphs with a low density and short traversal queries by up to two orders of magnitude. In contrast, the LS-traversal performs significantly better than the FI-traversal, if the graph is dense or the query traverses a large fraction of the whole graph.

6 SECONDARY INDEX STRUCTURES FOR GRAPHS

We refer to *secondary graph indices* as a general class of index structures that consist of a condensed graph representation derived from the primary graph storage. In GRAPHITE, the primary graph storage is organized in column groups—so we consider *adjacency lists* as secondary index structures. For other native graph managements systems (GMS), however, an adjacency list might be the primary storage of the graph topology.

Graph index structures have been an active area of research for several decades, resulting in a plethora of different graph indexing approaches tailored to specific graph query classes, graph topologies, and hardware constraints. Fundamental graph indexing types include *reachability indexing*, *shortest path indexing*, and *subgraph pattern indexing*. Commonly, the proposed approaches balance the tradeoffs for the three major quality measures, namely *construction time*, *memory consumption*, and *query performance*. Although graph indices provide superior query performance compared to their BFT or DFT-based counterparts, they typically suffer from high initial construction time and a prohibitively large memory footprint that can exceed the original graph size by up to multiple orders of magnitude.

Although most of these graph queries could be also expressed using a BFT or DFT-based algorithm, the linear time complexity of $\mathcal{O}(|V| + |E|)$ of a BFT/DFT algorithm—if stored in an adjacency list—becomes impracticable on large graphs. Although the exploitation of parallelism can reduce the performance gap between BFT/DFT-based solutions and efficient graph index structures with constant lookup time, it cannot fully hide the linear time complexity overhead. For applications, such as route planning, it can be even desirable to guarantee a constant query time complexity, i.e., $\mathcal{O}(k)$ for a small constant k.

Only a few graph indices can handle evolving graphs, which are becoming increasingly the predominant graph workload pattern. An evolving graph changes its topology over time, with frequent edge insertions and deletions. Due to the nature of the mentioned graph indices—they provide a condensed view of the underlying graph topology, often compacting entire subgraphs—their update performance is up to two orders of magnitude slower than the respective query performance. Even worse, the overhead for maintaining the graph index is data-dependent, resulting in unpredictable update costs for edge insertions/deletions.

For the tight integration into a DBMS, graph index structures tend to appear in too diverse flavors and require specialized handling in the database engine, and are therefore usually not considered for productive implementations. Moreover, the unpredictable update performance and the large memory footprint make specialized graph index structures unattractive for a general-purpose DBMS.

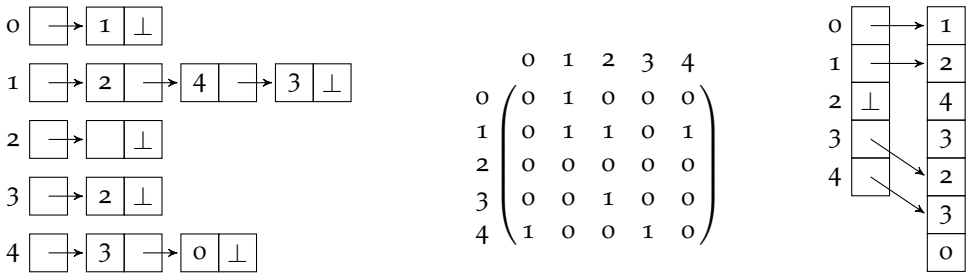
In this chapter we discuss related state-of-the-art graph index approaches and propose two updatable, lightweight graph index structures with a low memory footprint that can be seamlessly integrated into a DBMS while offering superior performance for neighborhood queries compared to purely scan-based approaches.

6.1 RELATED WORK

6.1.1 *Primary Graph Index Structures*

There are two prevalent data structures to store a graph topology in memory: (a) as adjacency list and (b) as adjacency matrix (Cormen et al., 2001).

An adjacency list stores a linked list of adjacent vertices for each vertex (cf. Figure 6.1 (a)). An adjacency matrix stores the topology as a binary matrix representing each edge in the graph by an individual bit stored at coordinate $\langle u, v \rangle$ for vertices $u, v \in V$ (cf. Figure 6.1 (b)).



(a) Adjacency list. (b) Adjacency matrix. (c) CSR.

Figure 6.1: Fundamental graph data structures.

The adjacency list has a space complexity of $\mathcal{O}(|V| + |E|)$ compared to the adjacency matrix with a space complexity of $\mathcal{O}(|V|^2)$. For sparse graphs with $|V| << |E|$, an adjacency list is more space-efficient since only existing edges are stored—for denser graphs with $|V| < |E|$, an adjacency matrix can be more memory-efficient since the existence of an edge can be represented in a single bit. Technically, an adjacency list can be implemented either by a pointer array to represent vertices and a set of linked lists to represent adjacent vertices or by an array of arrays. While a list-based (sorted) adjacency data structure exhibits higher in-place update rates, an array-based adjacency list allows a better CPU cache utilization for retrieving the adjacency of a vertex through a strictly sequential and contiguous memory access pattern.

A variation of the adjacency list has been developed in the context of sparse matrix multiplications and is widely used in graph processing systems (Gustavson, 1978). Caused by the duality between matrices and graphs—a graph topology can be represented by a matrix and vice versa—the authors developed an immutable *compressed sparse row* (CSR) data structure to compactly store sparse matrices (cf. Figure 6.1 (c)). The CSR data structure represents the graph topology as an *edge array* containing the target vertices of all edges in sequential order. To retrieve the adjacency of a single vertex, the *offset array* is used to compute the boundary values of the adjacency. In comparison to the adjacency list and the adjacency matrix, a CSR is more compact since the entire topology is represented in a single large, continuous chunk of memory. On the downside, adding and removing vertices/edges is expensive and results in a partial reconstruction of the CSR to maintain the ordering. For a detailed experimental analysis on several graph representations, we refer the reader to Blandford et al. (2004).

Brisaboa et al. (2009) propose a compact storage layout for simple, directed graphs based on k^2-ary tree structures as depicted in Figure 6.2. Specifically, they exploit the sparseness of the adjacency matrix and represent it through a compact tree structure of height $\lceil \log_k n \rceil$. The tree consists of nodes having assigned a single bit indicating whether there is at least one edge in the sub matrix. A node assigned zero means that is has no child nodes and all elements in the sub matrix are zero. The authors use the parameter k to modify the height of the tree at the cost of larger internal nodes with more child nodes. Internally, a k^2-tree is represented by two bit sets, one for the internal nodes (except leaves) and one for the leaves only. Bits are assigned according to a level-wise traversal of the tree, i.e., first all bits of level one, then all bits of level two, and so on. For the internal nodes, auxiliary data structures are added to the bit sets to support *rank* and *select* operations in constant time. Although the data structure provides a concise representation of the graph topology, multi-relational graphs, i.e., multiple edges between a pair of vertices, are not supported. Additionally, the k^2-ary tree representation is not designed for dynamic graphs with frequent edge insertions/deletions as each of these operations might trigger a rewriting of multiple internal nodes in the tree.

Álvarez et al. (2010) extend the idea to represent the adjacency matrix as a k^2-ary tree to also support multi-relational graphs with attributes associated to vertices and edges (cf. Figure 6.3). Specifically, they propose to represent a graph as three k^2-trees—one for

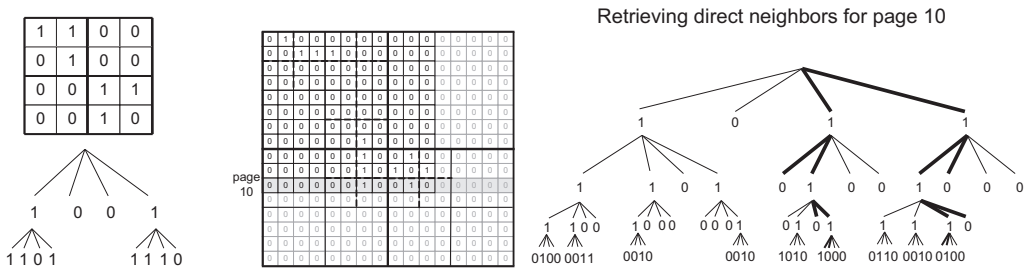


Figure 6.2: k²-trees example for k = 2 (Brisaboa et al., 2009).

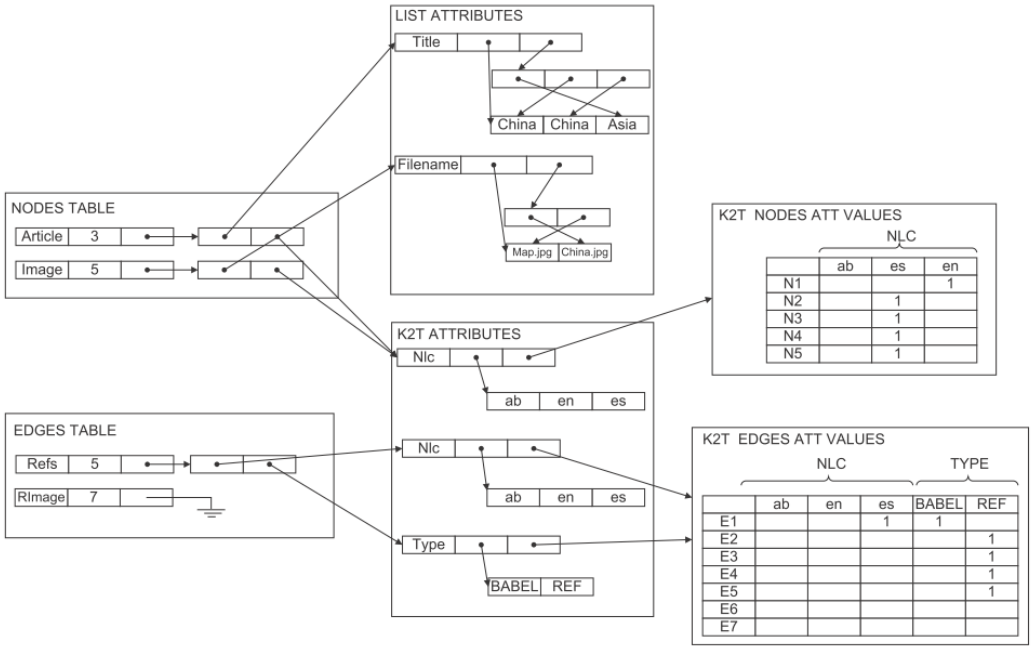


Figure 6.3: Example graph representation using k² trees (we omit the k² tree representing the graph topology here for brevity) (Álvarez et al., 2010).

vertex attributes, one for edge attributes, and one for vertex relationships. The *node table* is sorted by vertex type and contains one record for each type. Each record contains the number of vertices of that type and pointers to the data structures of attributes with a high cardinality (many distinct values). To allow fast lookups on the attribute values, a separate sorted array is used, which maps a value to object identifiers. Vertex identifiers are drawn from a statically assigned range of possible identifiers to allow the fast determination of the vertex type. The storage layout of the *edge table* is conceptually equivalent to the node table. Vertex and edge attributes with a low cardinality (few distinct values) are represented by two k²-tree structures depicting a bit set if the vertex/edge exposes the specific value. Multiple relationships between a pair of vertices is represented by a k²-tree and three auxiliary data structures. A bit set of the size of the number of ones in the relationship matrix stores a one, if there are multiple edges between a pair of vertices, zero otherwise. Additional edges are represented by their edge identifiers in an offset map. The complete data structure supports trivial operations through a programming interface, including the retrieval of vertex/edge types, vertex/edge filtering based on the type, vertex/edge filtering based on attribute values, and simple neighborhood queries. As in the initial proposal to use k²-trees to compactly represent a graph topology in memory, such a succinct representation does not support edge insertions/deletions and attribute value updates.

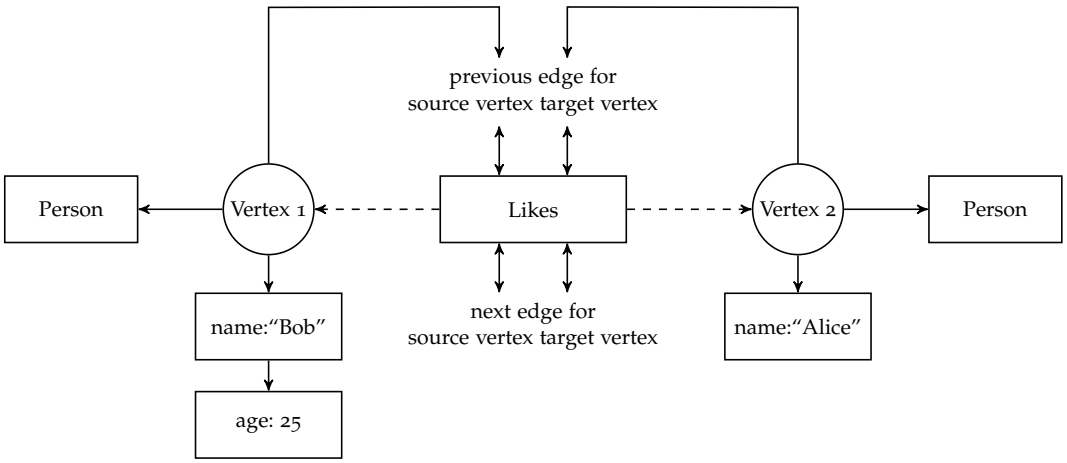


Figure 6.4: Example graph representation in Neo4j (adapted from Robinson et al. (2015)).

Neo4j stores a graph in a series of files on disk, partitioned into vertices, edges, labels, and properties (cf. Figure 6.4). The *node store* contains a set of fixed-length records—each 15 byte in size—and consists of a visibility flag (1 byte), a pointer to the first edge connected to the vertex (4 byte), a pointer to the first property (4 byte), a pointer to the vertex label store (5 byte), and a final byte for flags. In the flags field, Neo4j currently supports tagging a vertex as a *densely connected vertex*. The *edge store* maintains the edges connecting vertices. Each fixed-length edge record occupies 34 byte and constrains pointers to the start and end vertex. Further, it contains a pointer to the edge type, pointers to the next and previous edges for start and end vertex, respectively, and a pointer to the first property associated with the edge. Neo4j stores the properties of a single vertex or edge as a single-linked list. Each property consists of three entries, a pointer to the property data type, a pointer to the property index (to resolve the property name), and the property value. If the property value is fixed-length, i.e., an integer or a float value, Neo4j inlines the property value into the property store. Otherwise, if the property value is a variable-length string or array, Neo4j stores a pointer to a separate dynamic store record. Since Neo4j stores for each edge the predecessor and successor edges of both start and end vertex, these pointers form a double-linked list, which is used to perform graph traversals using pointer chasing in both traversal directions. To accelerate query processing, Neo4j utilizes an in-memory cache which acts similar to a buffer pool in traditional, disk-based RDBMS and holds discrete regions of the files in main memory. Given sufficient heap space, Neo4j can run entirely in memory—the internal data structures are the same as for the disk-based storage representation.

LLAMA is based on a mutable CSR data structure and supports batch updates (Macko et al., 2015). A graph is stored in a series of snapshots, where the adjacency list of single vertex might span multiple snapshots. The graph topology is stored in multiple *edge tables*—one per snapshot—and consists of a consecutive representation of adjacency list fragments (cf. Figure 6.5). LLAMA keeps a global vertex table that is shared across all snapshots and maps vertex identifiers to per-vertex structures. The vertex table is implemented as a mutable array with snapshotting support and uses a software-based copy-on-write mechanism. The vertex table array is partitioned into equally sized data pages and one indirection array per snapshot. The indirection array maps vertex identifiers to the corresponding data pages.

In LLAMA a new snapshot can be created by copying the indirection array. A data page can be modified by copying the data page first, then updating the corresponding indirection array to point to the new page, and to finally modify the page. A vertex record consists of the following elements: a snapshot identifier, an edge table offset, and an adjacency fragment list length. The snapshot identifier contains a reference to the first adjacency list fragment that belongs to the vertex. The edge table offset corresponds to the entry in-

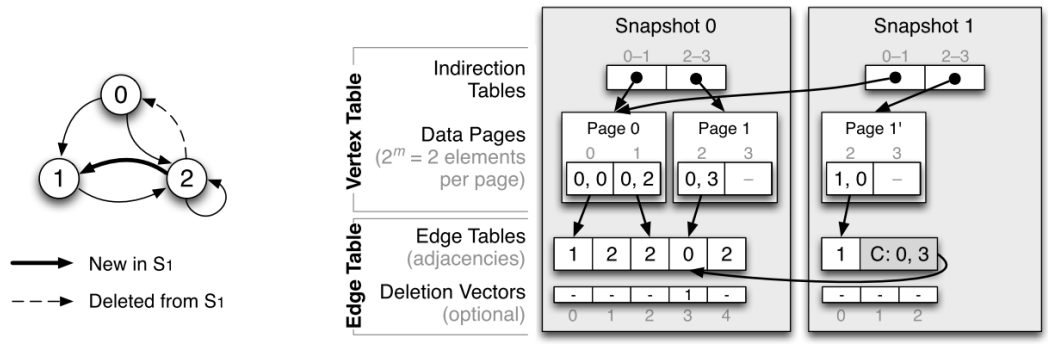


Figure 6.5: Example graph representation in LLAMA (Macko et al., 2015).

dex into the edge tables that stores the adjacency list fragments consecutively. To add a new vertex to a snapshot, LLAMA increases the size of the vertex table; to delete a vertex, LLAMA replaces the entry in the vertex table by a *nil record*. An edge table is represented by a memory-mapped array data structure of consecutive adjacency list fragments—an edge is stored by the target vertex identifier of the edge. If the adjacency list of a vertex contains multiple fragments, each fragment ends with a 16-byte *continuation record* pointing to the next fragment. LLAMA supports two kinds of edge deletions: (1) rewriting the entire adjacency list (without the deleted edge) of the source vertex into a new snapshot or (2) invalidating the edge in a *deletion vector* that encodes in which snapshot the edge was deleted.

LLAMA allows multiple attributes to be stored with vertices and edges through arrays that can be accessed via a positional index. Each property is stored in its own mutable array structure—variable-sized attributes have to be stored in a separate in-memory key/value store. By default, LLAMA buffers incoming updates in a high-throughput key/value store where each modified vertex is represented as an object containing the list of newly inserted and deleted edges. This write-optimized delta map, however, is excluded from the query execution for performance reasons. If the query requires access to the latest version of the data, a new snapshot has to be created from the delta map first.

GRACE (Prabhakaran et al., 2012) partitions a graph into subgraphs based on different partitioning schemes, including hash-based partitioning and heuristic partitioning. This can be further improved by placing proximity vertices close to each other in memory by using spectral partitioning. GRACE maintains two dynamic arrays (vertex/edge log) per partition, one for vertices and one for edges (cf. Figure 6.6). In conjunction with the *Edge Pointer Array*, the edge log is effectively a CSR data structure with a clustering on the source vertices. Attributes are stored in separate array structures, where an attribute value can be retrieved via a positional lookup. GRACE supports transactional modifications to the graph through snapshot isolation and collects graph updates—vertex and edge insertions/deletions and edge weight modifications—in temporary in-memory buffers. Deleted vertices and edges are invalidated in a validity map.

EMPTYHEADED represents vertices as integers that stem from dictionary encoding and the edge relation by a collection of neighborhood sets. All neighborhood sets together form a CSR data structure, where the vertex array consists of pointers that point to their corresponding (possibly different) neighborhood set representations. EMPTYHEADED distinguishes between unsigned integer representations (`uint`) and bit set representations (`bit set`) of neighborhood sets at three different levels: (1) the graph level, (2) the set level, and (3) the block level. The `bit set` representation merges the sparse representation of the `uint` with the dense representation of a `bit set` by combining both into a single data structure. The data structure consists of a set of blocks, where each block contains a `bit set` that is sized to the hardware characteristics, i.e., the size of a single cache line. A `bit set` is organized as a set of pairs, where each pair consists of an offset and a bit set. The offset corresponds to the index of the smallest bit set in the bit set. The size of the bit set—the

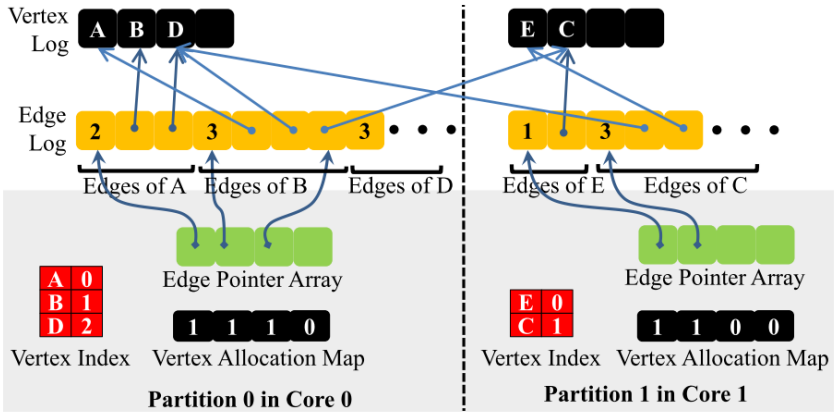


Figure 6.6: In-memory graph representation in GRACE (Prabhakaran et al., 2012).

block size (the default is 128)—is a power of two and tailored to fit onto a single cache line. Although the logical store layout foresees a set of pairs, EMPTYHEADED physically stores first all offsets, followed by all bit sets. The authors define the *density* of a neighborhood set as the cardinality of the set divided by the value range of the set. Real-world graphs tend to have a large skew in their neighborhood density distribution. EMPTYHEADED allows storing a single neighborhood set either in a dense representation (`bit set`) or in a sparse representation (`uint`), the actual representation is selected by the optimizer based on collected statistics about the density and the cardinality of the neighborhood set.

GRAPHCHI (Kyrola et al., 2012) distributes the vertex identifier space into disjoint intervals that are subsequently mapped to so-called *shards*. A shard stores all edges whose *target vertex* is in the corresponding vertex identifier interval and sorts the edges within a shard by their source vertex. GRAPHCHI supports accompanying attributes attached to vertices and edges and stores them in separate arrays. GRAPHCHI uses several auxiliary data structures to keep graph statistics—such as degree information for each vertex, which is used for memory allocation.

GRAPHCHI-DB extends the GRAPHCHI system and adds support for edge insertions, updates, and deletions (Kyrola and Guestrin, 2014). Like GRAPHCHI, GRAPHCHI-DB partitions the set of vertices into shards, where each shard has an associated *edge partition* storing all the edges such that the target vertex of each edge is stored in the same shard. The set of edge partitions can be interpreted as a partitioned adjacency list, where each adjacency list partition is stored in a CSR data structure. An edge partition consists of three parts, an *edge array* storing the edges, a *pointer array* providing efficient access to the outgoing edges of a vertex, and an *reverse pointer array/in-start-array* storing references to the first incoming edge of each vertex (cf. Figure 6.7). Vertex and edge attributes are stored in a columnar storage, providing positional access to the attributes of a single vertex or edge.

Vertices in both pointer arrays are sorted in ascending order, thereby directly applying the idea of a CSR data structure. The edge array stores 64 bit values, where 36 bit are used for the destination vertex, 4 bit for the edge type, and 24 bit are used to point to the next entry with the same destination vertex. Since a binary search can perform badly when the pointer array does not entirely fit into main memory, the authors propose to either add a secondary sparse index on top of the pointer array or to leverage delta-compression techniques to lower the memory footprint.

GRAPHCHI-DB uses a staging architecture for graph updates by redirecting edge insertions into an in-memory buffer (optionally persisted in a disk-based log), which is periodically merged into the existing edge partitions. To scale this approach to bulk insertions on large graphs, the authors propose to use log-structured merge trees. To update an attribute of an edge, GRAPHCHI-DB implements in-place updates, and to delete an edge, the record in the edge array is invalidated. Finally, GRAPHCHI-DB only provides rudimentary sup-

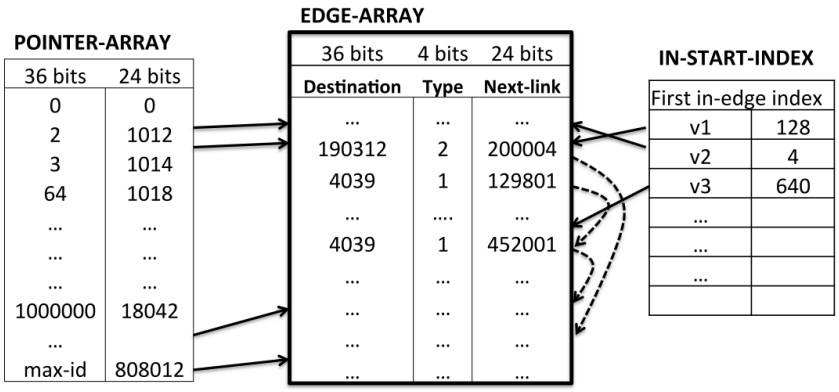


Figure 6.7: File structure of an edge partition in GraphChi-DB (Kyrola and Guestrin, 2014).

port for transactions and implements *fire-and-forget* transaction semantics, which prevent transactions from running in isolation from each other.

Formally, Sparksee (Martínez-Bazan et al., 2012) specifies a multi-relational, attributed graph G as a tuple $G := (V, E, L, T, H, A)$, where V refers to the set of vertices, E refers to the set of edges, and L refers to the set of labels, i.e., $L = \{(o, l) | o \in (V \cup E), l \in \text{string}\}$. Each vertex and edge can have exactly one assigned label. The set T refers to the set of tail pairs, i.e., $T = \{(e_1, t_1), \ldots, (e_m, t_m)\}$ with $t_i \in V$ and $e_i \in E$. Similarly, the set H refers to the set of head pairs, i.e., $H = \{(e_1, h_1), \ldots, (e_m, h_m)\}$ with $h_i \in V$ and $e_i \in E$. Finally, the set A refers to the set of attributes with $A_i = \{(o_1, c_1), \ldots, (o_r, c_r)\}$, where o_i is either a vertex or an edge and c_i is a literal value.

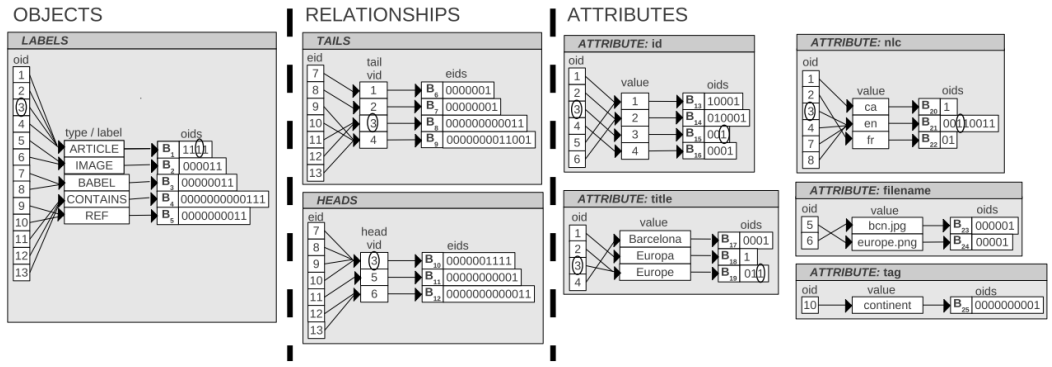


Figure 6.8: Example graph representation in Sparksee (Martínez-Bazan et al., 2012).

Sparksee stores a graph based on *value sets*, which groups all pairs of the original set with the same value as a pair between value and the set of objects—vertices or edges—with this value. Internally, a value set is represented by two maps—implemented as B+-Trees—and a collection of compressed bit sets to represent the corresponding vertex/edge sets. The first map stores assignments of object identifiers to a corresponding value. The second map assigns to each value a set of object identifiers, where the set is represented as a compressed bit set. Sparksee applies word-aligned compression scheme on each bit set to compact long sequences of zeroes.

An example for a graph representation in Sparksee is depicted in Figure 6.8. The graph storage layout is divided into three main parts: (1) a mapping between object identifiers (vertices/edges) and their corresponding labels, (2) a mapping between edge identifiers and their corresponding head/tail vertices, and (3) a mapping of object identifiers and their corresponding values.

The generic graph storage layout of Sparksee requires storing each attribute value only once, effectively eliminating the need of dictionary-encoding. On the downside, evaluating

complex predicate across multiple attributes requires costly join operations of intermediate results. For large graphs, the memory overhead of many sparse bit sets for large value domains is unavoidable and leads to a high memory footprint. For dynamic graphs with frequent edge insertions/deletions and attribute value updates, it remains unclear how SPARKSEE maintains the compressed bit sets without the need of a full recompression of the entire bit set.

6.1.2 *Secondary Graph Index Structures*

Corneil and Krueger (2005) apply an ordering to the vertices based on the initial idea by Rose et al. (1976). They try to determine an ordering based on the detection level of the vertices and create a path-based index structure, which may consume too much memory to be applicable. A similar idea is to use an index on reachability as proposed by Trißl and Leser (2007). Their index structure is based on pre- and post-ordering values, which are determined during an initial traversal process that calculates the transitive closure. Besides the fact that their index is not capable of storing distances, the only guaranteed update time is a complete recreation of the index.

All those data structures may be invalidated by insertion or deletion, which may require a complete index recreation. As a direct consequence, their approach cannot guarantee an acceptable upper boundary for inserts. Also, further performance improvements, which were proposed by Wang et al. (2012) and use edit distances between similar substructures, cannot solve the problem of a too high boundary for manipulation time.

As established by Sakr and Al-Naymat (2010a) and as explained above, the existing graph indices are not applicable to evolving graphs. Therefore, most graph databases measure their performance only on static data without guaranteed update performance, as discussed by Ciglan et al. (2012) and Dominguez-Sal et al. (2011). This contradicts our goal to be capable of handling evolving graphs and provide strict upper boundaries for manipulation time.

A more suitable approach is discussed by Faust et al. (2013). They propose a *Paged Index* to reduce the amount of data to be processed during column scans in main-memory column stores. We follow the same principle of splitting the entire column into smaller parts for the *block-based topology index*. But while their index structure consists of a consecutive bit set that stores the relevant pages for certain entries and relies on clustered data, we apply the idea to fit unclustered data and typical graph traversal requests.

Reachability Index Structures

Reachability queries are one of the most fundamental graph query classes and ask for a query with vertices u and v, whether there exists a path $u \rightsquigarrow v$ in the graph. Although reachability queries can be answered in linear time of $\mathcal{O}(|V| + |E|)$ using a DFT, this brute-force approach is usually too slow for large graphs with tens of millions of vertices and hundreds of millions of edges. The other extreme approach is to fully compute and store the transitive closure using a variation of the Floyd-Warshall algorithm, which has a cubic time complexity of $\mathcal{O}(|V|^3)$ and quadratic space complexity of $\mathcal{O}(|V|^2)$. As reachability queries are not in the focus of our work, we only briefly discuss here the most important approaches—a recent and detailed overview of reachability index structures has been collected by Jin et al. (2012).

Reachability index structures try to achieve a balance between three important quality criteria, namely *construction time*, *memory consumption*, and *query performance*. To cope with the quadratic space complexity of the full transitive closure, several approaches compress the transitive closure by either applying *interval labelings* (cf. the seminal work of Agrawal et al. (1989)), word-aligned bit set compression (van Schaik and de Moor, 2011), or tree-covering approaches (Jin et al., 2008). A popular example for tree cover-based transitive

closure compression is PATH-TREE (Jin et al., 2008), which extracts disjoint paths from the data graph and creates a condensed tree structure from it.

A second fundamental approach is based on a *2-hop cover* (Cohen et al., 2002), where the main idea is to maintain a subset of ancestors and descendants for each vertex in the graph. Then, a reachability query can be answered by intersection both subsets. Jin et al. (2009) describe with 3-HOP a similar approach, which is based on the idea of *highway routes* in the graph using a *chain decomposition*.

The third category uses labeling information to accelerate online search, i.e., DFT, by pruning the search space during the graph exploration. In contrast to full transitive closure compression and hop labeling approaches, a refined online search does not require an expensive index construction phase and has a considerably lower memory footprint, effectively making it scalable to large graphs with millions of edges. The best-performing representative of this category is GRAIL by Yildirim et al. (2010).

SCARAB (Jin et al., 2012) is a reachability querying framework that aims at overcoming the scalability issues for construction time and index size of other reachability index structures. The authors propose a *reachability backbone graph*, i.e., a condensed graph representation of the original graph, which carries the general reachability information. For a reachability query $Q = (u, v)$ for vertices u, v the query starts in the backbone graph by accessing the outgoing and incoming backbone vertices of u and v, respectively. Then, a forward (backward) BFT traversal is started at vertex u (v) to access the underlying reachability backbone. The output of the forward (backward) traversal is the set of ϵ-hop reachable backbone vertices B^{ϵ}_{out} (B^{ϵ}_{in}). In the second step, the *reachability join test* determines, whether there is a path between any vertex in B^{ϵ}_{out} and B^{ϵ}_{in}.

All previously discussed index structures handle static, immutable graphs, but do not consider edge insertions and deletions. Caused by the nature of these index structures, updates to the index can result in expensive maintenance operations. Depending on the location of the inserted/deleted edge in the graph, the update can affect the complete index structure. Yildirim et al. (2013) propose DAGGER, an extension of GRAIL that can handle edge insertions and deletions. Although they support edge insertions/deletions, the performance of update operations is still several orders of magnitude higher than respective query operations—a conclusion, which has been also drawn by Zhu et al. (2014).

Despite some initial work on reachability index structures for dynamic graphs, there has been only little interest yet in the research community to integrate them into a GDBMS. We conclude from this discussion of related reachability index structures that they cannot be easily integrated into a dynamic database environment with potentially limited hardware resources, contradicting the high demands of these index structures on initial construction time and memory consumption.

Graph Pattern Matching Index Structures

A second important class of graph queries deals with graph pattern matching, commonly also referred to as *subgraph isomorphism* problem. A pattern matching query returns for a given query graph all matching subgraphs from a database of graphs. Since the subgraph isomorphism problem is NP-complete (Garey and Johnson, 1979), the research community proposed approximate pattern matching and indexing techniques. We categorize index structures for graph pattern matching into three main categories: *path-based*, *tree-based*, and *graph-based* approaches.

One of the seminal works on graph indexing for exact graph pattern matching was proposed by Giugno and Shasha (2002). They developed GRAPHGREP, an application-independent, path-based index structure to accelerate subgraph queries on a database of graphs. The general idea of GRAPHGREP is to build up an index of paths up to a maximum length and to encode in which graphs they occur. The actual graph query filters the database using GRAPHGREP to generate potentially matching graphs and to process them in a final validation phase. GRAPHGREP has the advantage that the memory con-

sumption of the index is bounded by the maximum length of the paths. On the downside, by breaking up a graph into a set of paths, important structural information can be lost.

To overcome the limitations of GraphGrep, Yan et al. (2004) propose `gIndex`, an index structure, which uses frequent subgraphs instead of paths as basic indexing feature. In contrast to GraphGrep, `gIndex` does not split the query and graphs into paths, where important structural information can be lost, but instead identifies frequent subgraphs. The authors facilitate frequent graph mining algorithms to identify these discriminative subgraphs in the database. By relying on information-preserving features in the index, fewer false positives candidate matches are generated and the index is more stable to database updates. Although `gIndex` outperforms GraphGrep in terms of index size and processing speed, it exhibits a time-consuming graph mining process to identify the frequent subgraphs and construct the index.

In a similar spirit to `gIndex` is `C-Tree`, a graph-based indexing technique that is assembled as a tree, where nodes represent so-called *graph closures* and capture discriminative information about descendant tree nodes (He and Singh, 2006). `C-Tree` accelerates subgraph and graph similarity queries and supports dynamic insertions/deletions to the underlying database.

Zhao et al. (2007) and Zhang et al. (2007) investigate, whether *tree-based* features can be used for indexing a large graph database without the need to identify discriminative subgraphs using expensive graph mining algorithms. They observe that in practice, frequent subgraphs are often tree-structured in nature. The authors propose to select frequent tree features using tree mining techniques, which are known to be computationally less expensive than graph mining algorithms, and to also store a small number of discriminative graph features on-demand.

In contrast to previous pattern matching index structures, which dealt with a large number of relatively small graphs, Zhang et al. (2009) propose indexing techniques to accelerate subgraph queries on a single, but large data graph. They introduce Gaddi, which relies on a *neighboring discriminating substructure distance*. In contrast to previous approaches, they do not index subgraphs, but instead index the distance between neighboring pairs of vertices and thereby achieving a considerably higher pruning rate while scaling better to larger graphs. In an extensive experimental evaluation, Han et al. (2010) compare several indexing techniques for graph pattern matching against each other.

6.2 GENERAL REQUIREMENTS

Graph index structures exist in various configurations and are typically tailored to a specific class of graph queries, such as reachability queries, shortest path queries, or subgraph isomorphism queries. Graph queries that rely on specialized graph index structures can outperform their naive counterpart implementations by several orders of magnitude. A general-purpose graph processing system, however, would have to support a large variety of graph index structures to accelerate different types of graph queries. This does not only pose severe performance drawbacks for write-intensive workloads with frequent updates to the graph (and consequently frequent index updates), but also causes an increased development and maintenance overhead for the code base.

Specifically, neighborhood queries, i.e., graph queries that access vertices adjacent to a given vertex or a set of vertices, are inefficient in the columnar graph storage of Graphite as the time complexity is linked to the total number of edges in the graph. Therefore, we focus on the integration of graph index structures to accelerate neighborhood queries, which in turn are a basic building block for more complex graph queries. In the following we describe the main design goals, which such a general graph index structure for efficiently supporting neighborhood queries has to fulfill.

MAINTAINABILITY. Graph processing systems that support evolving graphs with frequent updates to the graph topology demand efficient maintenance routines to keep the index structures in a consistent state. Additionally, the maintenance routine

should have a guaranteed maximum execution time to refresh the index structure. The insertion/deletion of an edge should only have a limited effect on the overall index structure and trigger the recreation of only a small portion of the complete index.

APPLICABILITY. The index structure has to be integrated into a DBMS and provide a session-specific, transactional view of the data. Further, the result of an initial predicate evaluation, for example the filtering on a specific vertex or edge type, should be usable on the index structure to filter out non-matching vertices/edges. Similarly, the output of an index lookup should be usable as an input for a predicate evaluation.

SCALABILITY. The index structure should be scalable in terms of construction time and memory consumption. Especially graph index structures with a super-linear construction time and space complexity are not applicable to large graphs with billions of vertices and edges. The memory footprint of the graph index should not exceed the size of the original edge-list-based representation.

Based on the design goals we devise two graph index structures, namely a *block-based* and an *adjacency-based* graph index structure. Both index structures can be used interchangeably, but expose different advantages and disadvantages in terms of construction time, index maintenance, lookup time, and memory footprint.

6.3 BLOCK-BASED TOPOLOGY INDEX

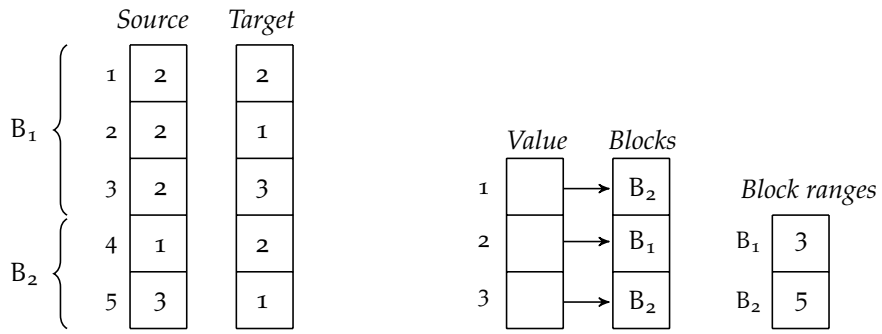
The block-based topology index extends the idea of an immutable CSR data structure with the ability to efficiently perform edge insertions at runtime. In general, a basic CSR data structure could be updated as well, but it would require a considerable and typically not practicable processing overhead to keep the target vertex array sorted. To insert a new edge into a CSR data structure, the corresponding insert location in the target vertex array has to be calculated using the offset array. In a subsequent step, the insert operation has to move all subsequent vertices in the target vertex array by one position, potentially leading to a large number of copy operations. Finally, the offset array has to be rewritten starting from the insert location.

Instead of devising an immutable, sorted primary index, such as a CSR data structure, we propose a lightweight, mutable, secondary index, which operates solely on the edge column group representation. Using an integrated secondary index instead of a standalone primary graph index has several advantages: (1) Predicates can be evaluated on the column groups and combined directly with index lookups, (2) the memory overhead is in the range of $\mathcal{O}(|V|)$ instead of $\mathcal{O}(|V| + |E|)$, and (3) complex transaction and visibility handling can be reused from the relational backend.

To construct a block-based topology index, we divide the clustered source vertex column into non-overlapping, contiguous blocks of potentially varying size. Conceptually, we represent a block as a tuple $\langle id, start, end \rangle$, where *id* corresponds to a unique identifier, *start* corresponds to the start position and *end* to the end position of the block in the edge column group, respectively. In a subsequent step, we store for each distinct source vertex a set of blocks. The initial assignment of source vertices to blocks is a 1-1 mapping, which might degenerate into a 1-N mapping for evolving graphs. This is in contrast to a simple CSR data structure, which always provides a 1-1 mapping of vertices to blocks.

We use the notion of a *minimal block size* (b_{min}) to refer to the smallest allowed block size. A block cannot be smaller than the minimal block size and can be increased dynamically to assure that all outgoing edges of a vertex fit into a single block.

Figure 6.9 (a) depicts an edge column group clustered by source vertex and a logical partitioning into two blocks with a minimal block size $b_{min} = 2$. Figure 6.9 (b) shows the corresponding index representation consisting of two core data structures, one for mapping values to blocks and one for encoding the ranges of the blocks. If the source vertex column is clustered, each value points to exactly one block. Since all blocks are



(a) Edge column group representation. (b) Block-based topology index.

Figure 6.9: Block-based topology index with minimal block size = 2.

contiguous and non-overlapping, we only store the end position of the block. The block ranges of a block can be reconstructed from the end position of the previous block and the end position of the current block. For example, block B_2 spans the column range $[4, 5]$.

6.3.1 *Construction and Maintenance*

INDEX CONSTRUCTION. Before we construct the block-based topology index, we cluster the edge column group by source vertex. Clustering by source vertex has the advantage that all adjacent vertices for a vertex can be pulled from a contiguous chunk of memory, resulting in the initial 1-1 mapping of the block-based topology index.

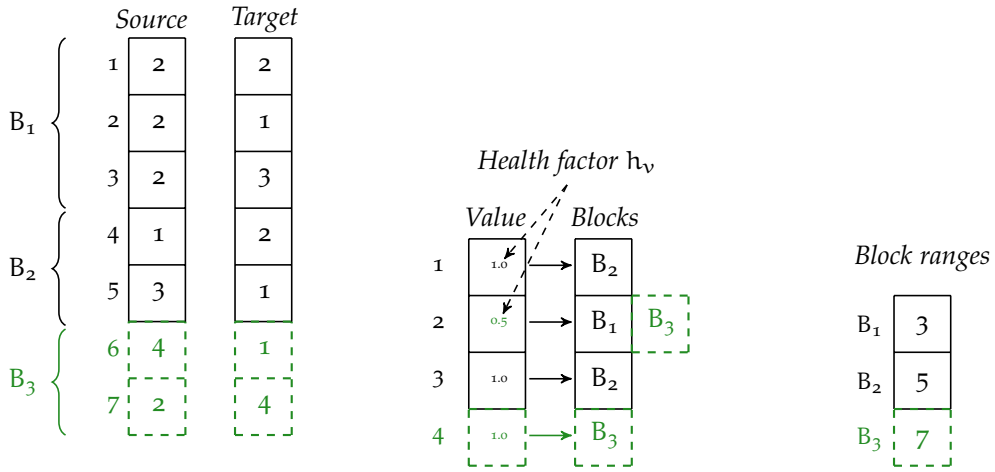
We construct the index in two steps: the first step scans the source vertex column and computes the block ranges, i.e., the begin and end positions of each block according to the blocking criteria using a prefix sum scan. We define two blocking criteria: (1) a block contains at least b_{min} elements, where b_{min} refers to the configurable, minimal block size and (2) all adjacent vertices of a vertex are stored in the same block.

We provide an adaptive mechanism that allows handling low outdegree and high outdegree vertices—as they appear in scale-free graphs—equally well. If the outdegree of a vertex is larger than the minimal block size b_{min}, we extend the block accordingly to store all outgoing edges of a vertex in a single block. If the outdegree of a vertex is smaller than the minimal block size, we fill the block with other vertices until the minimal block size is reached.

The second step iterates over all computed block ranges in parallel and identifies the set of distinct source vertices in each block. We use thread-local hash sets to determine the set of distinct source vertices. For each source vertex, we concurrently add an entry to the block-based topology index, which maps the vertex identifiers to block identifiers. Since all the outgoing edges of a vertex appear in a single block, we can directly add the mapping entry to the topology index without the need of further synchronization.

INDEX UPDATES. In many realistic scenarios the graph evolves over time and therefore efficient mechanisms to deal with topology manipulations, i.e., insertions, updates, and deletions of edges, are required. Thus, a graph index structure for evolving graphs should also support efficient topology manipulation operations. We designed the block-based topology index to support both static and evolving graphs with frequent changes to the graph topology.

For the rest of the discussion, we focus on the support of edge insertions and deletions as they occur more frequently than in-place edge updates (Leskovec et al., 2005). An in-place edge update, however, could be easily modeled as an edge deletion followed by an edge insertion.



(a) Edge column group representation.

(b) Block-based topology index.

Figure 6.10: Updating the block-based topology index with minimal block size = 2 (modifications colored in green).

Edge deletions: We handle edge deletions by marking invalid edges in a lightweight invalidation data structure, which is part of the visibility and access control component of GRAPHITE. The invalidation data structure uses a single bit per edge to track whether the edge has been marked for deletion. By delaying the physical edge deletion to a later point in time, we avoid the overhead of reorganizing the index structure immediately after the deletion operation, i.e., to remove the element from the target vertex array and to update the offset array. To lower the memory footprint, the index structure can be rebuilt on a regular basis and deleted edges can be removed permanently.

Edge insertions: To insert an edge, we add it to the end of the edge column group. Appending a new entry at the end of a column is a common update strategy in column stores to preserve the compression and the sort order of the static fraction while still offering acceptable performance for insertions (Abadi et al., 2006). By appending an edge to the edge column group, however, we likely break the source vertex clustering criterion. Although the block-based topology index does not rely on a strict edge clustering, it shows the best performance for an optimal clustering, as we can map each vertex in the column to exactly one block. If the column is not optimally clustered, each vertex (and its outgoing edges) can appear in multiple blocks and therefore points to a set of blocks.

Figure 6.10 illustrates the insertion of two edges $\langle 4, 1\rangle$ and $\langle 2, 4\rangle$ at the end of the edge column group and the corresponding updates to the index. The first insertion triggers the creation of a new block B_3, which is increased until the *minimal block size* is reached. While blocks residing in the static fraction of the edge column group can vary in size, all blocks in the dynamic part have a fixed, but configurable size. After inserting the edges into the edge group, we update the mapping of vertices to blocks. We distinguish two possible cases: (1) The edge is the first outgoing edge of the vertex or (2) the vertex already has outgoing edges. In the first case, we append the new mapping entry at the end of the mapping array. In the second case, we add a new block to the set of blocks. In a final step, we update the last element in the block ranges array and set it to the new end position of the block.

The second case increases for a single vertex the number of blocks to read. If the source vertex column is perfectly clustered or the outgoing edges for each vertex can be fetched from a single block, we refer to the index structure as being in a good *health state*. An index lookup achieves the best possible query performance as for each vertex only a single block needs to be scanned. Frequent edge insertions, however, *pollute* the index and degrade

Algorithm 7: Index lookup

```
Input  : Set of values F
Output : Set of blocks C with range information
1 Procedure lookupBlocks(F, C)
2     forall v ∈ F do
3         B ← bi[v];
4         forall k ∈ B do
5             C ← C ∪ ⟨k, brv[k − 1] + 1, brv[k]⟩;
```

the index health. At some point, the index health degradation is so severe that even a full-column scan outperforms an index lookup.

We use a measure—the *health factor*—to quantify the overall quality of the index structure with respect to query performance. For each vertex v we define a local health factor $h_v = \frac{1}{|B_v|}$. The health factor h_v reaches its maximum ($h_v = 1.0$), when all adjacent vertices of a vertex can be pulled from a single block. For sets of blocks, the health factor decreases inversely proportional to the number of blocks to read. The health factor of the complete index structure is defined as follows:

$$h = \frac{1}{|V|} \sum_{i=1}^{|V|} h_i \tag{6.1}$$

If the health factor h of the index is below a threshold τ, we consider the index as not beneficial anymore to considerably speed up neighborhood queries. When the index becomes impractical, we merge the dynamic fraction of the edge column group into the static fraction and rebuild the index.

6.3.2 *Index Lookups*

The block-based topology index accelerates neighborhood queries, which return the adjacent vertices for a given set of vertices. In Section 5.3.2 we described a naive implementation of neighborhood queries based on full-column scans. The time complexity of a full-column scan is $\mathcal{O}(n)$, where n refers to the number of elements in the column. Although a full-column scan can be implemented efficiently using a parallelized scan routine, queries with a high selectivity, such as neighborhood queries, can exploit index scans and thereby significantly outperform a full-column scan.

We support two types of index lookups: single value lookups and batch lookups. A single value lookup receives a value and returns all blocks, which contain at least one occurrence of the value. A batch lookup receives a set of values and performs a grouped execution of single value lookups. An index lookup returns a set of block descriptors, each consisting of a tuple $\langle b, e \rangle$, where b refers to the begin position of the block and e to the end position of the block, respectively. We use the set of block descriptors to limit the column scan to range scans, thus allowing to potentially skip large column parts during the scan.

Algorithm 7 depicts the index lookup routine, which operates in two steps: (1) for each value from the input set, we determine the corresponding block identifiers and (2) for each block identifier, we compute the block boundaries from the block range vector. Since both steps rely on positional data accesses to dynamic arrays, their performance is limited by data cache misses. To increase the data cache hit ratio, we perform an in-place sort of the input data to guarantee a sequential access pattern to the block index and also sort the block identifiers to achieve a sequential access pattern on the block range vector.

If the result set contains consecutive blocks, i.e., there are two blocks b_1 and b_2 with block descriptors $\langle s_1, t_1 \rangle$ and $\langle s_2, t_2 \rangle$ and $t_1 + 1 = s_2$, we adaptively merge them to form

Algorithm 8: Level-synchronous index-based graph traversal

```
Input  : F = Set of frontier vertices
         E_active = Set of valid edges
Output : P = Set of matching records
1 Procedure scan(F, E_active, P)
2     B ← ∅;
3     lookupBlocks(F, B) ;                                  // See Algorithm 7
4     forall b ∈ B do
5         P_loc ← ∅;
6         scanRange(b, F, E_active, P_loc) ;                // Limit scan to block boundaries
7         P ← P ∪ P_loc;
```

blocks of maximal possible size. This decreases the total number of blocks to process and increases spatial and temporal locality when performing the range scans.

6.3.3 *Memory Consumption*

The memory consumption of the block-based topology index depends on the number of distinct source vertices $\mathcal{V}_s = |\{ u \mid (u, v) \in E \}|$ in the graph and the total number of blocks $\mathcal{B}$. We represent the mapping of values to blocks as an array of size $\mathcal{V}_s$, whereby each entry might contain multiple block identifiers of 4 bytes each. In a perfectly clustered column, the value-block mapping occupies $4 \cdot \mathcal{V}_s$ bytes. Additionally, we maintain for each block the block range. Since block ranges are consecutive and non-overlapping, we only store the end position of each block range, resulting in a total memory consumption of $4 \cdot \mathcal{B}$. The total memory consumption of the block-based topology index can be written as $4 \cdot (\mathcal{B} + \mathcal{V}_s)$.

6.3.4 *Graph Traversal Implementations*

In the following we describe how the block-based topology index can be used to accelerate graph traversals. We discuss two traversal implementations, a purely index-based graph traversal and a hybrid scan/index-based graph traversal.

INDEX-BASED GRAPH TRAVERSAL. We use the algorithm skeleton of the level-synchronous, scan-based graph traversal introduced in Section 5.3.2 and replace full column scans by index scans using the block-based topology index. In Algorithm 8 we sketch the revised level-synchronous graph traversal routine. Since the scan-based and the index-based traversal expose the same programming interface, we can easily exchange their respective implementations.

The index-based routine receives as input a set of frontier vertices F and a set of active edges E_{active}, and returns a set of positions P in the edge column group. In the first step, we retrieve for all frontier vertices their corresponding blocks using the function `lookupBlocks`, which is shown in Algorithm 7. After computing the blocks to scan, we perform for each block a range scan in the block boundaries $\langle b, e \rangle$ and collect the matching edges in a position list P. Depending on the number of blocks to process, we distribute the single scan operations across multiple worker threads and merge their partial position lists in a subsequent merge operation.

HYBRID SCAN/INDEX-BASED GRAPH TRAVERSAL. The scan-based graph traversal has a linear time complexity to the number of edges in the column group, rendering it a work-inefficient solution for small frontier sets. In contrast, the index-based graph traversal provides superior query performance for small frontier sets, but is inefficient for relatively

large frontier sets. Instead of treating both approaches as separate strategies, we propose a *hybrid scan/index-based graph traversal* that automatically switches between a scan-based and an index-based execution based on the frontier set size. To determine the right traversal strategy for each traversal iteration we use a threshold τ compute the fraction of the frontiers to the overall number of source vertices in the graph. If the computed fraction exceeds the threshold τ, we use the scan-based traversal routine, otherwise the index-based traversal routine. In the experimental evaluation we demonstrate the performance implications for different values of the threshold τ.

6.4 ADJACENCY-BASED TOPOLOGY INDEX

The adjacency-based topology index is a secondary index structure, which allows answering neighborhood queries directly and without the need to consult the primary copy of the data. This is in contrast to the block-based topology index, where neighborhood queries cannot be answered by index lookups only. In addition to the general requirements detailed in Section 6.2, we pose the following supplementary requirements on the index structure:

BI-DIRECTIONAL GRAPH TRAVERSAL. The index structure should support both forward and backward traversals without sacrificing the query performance of neither traversal direction.

REFERENCES TO ATTRIBUTE COLUMN GROUPS. The index structure should allow the conjunctive evaluation of relational predicates and neighborhood queries, i.e., the output of a relational predicate evaluation should be usable as an additional pre-evaluated filter condition in a neighborhood query and a neighborhood query should be able to probe relational predicates during execution.

INDEX MUTABILITY. The index should be mutable and support arbitrary vertex/edge insertions and deletions. Further, in contrast to the block-based topology index, the index lookup performance should be only dependent on the actual size of the adjacency for each single vertex and not degrade significantly for update-heavy workloads.

HIGH-PERFORMANCE NEIGHBORHOOD QUERIES. The index lookup should be fast and while accessing the graph topology impose as few indirections as possible. Ideally, the query performance is close to a memory copy operation.

DELTA MERGE STABILITY. The index structure should be stable between two delta merge, i.e., the periodic incorporation of the dynamic graph storage into the static graph storage, operations and omit having to recreate the complete adjacency structure after a delta merge has been executed.

To address the additional requirements mentioned above, we introduce the adjacency-based topology index, a high-performance, mutable adjacency list with direct references to the corresponding vertex and edge column groups. The index structure can be directly constructed from the underlying vertex and edge column groups and store both traversal directions, i.e., one adjacency list organized by outgoing edges and one by incoming edges. The adjacency-based topology index holds internal mapping structures to allow accessing attribute values of vertices and edges from the adjacency list and to access the graph topology based on a predicate evaluation on the vertex/edge column groups. Supported operations on the graph topology are the retrieval of neighbors of a given vertex (via incoming or outgoing edges) and the bulk retrieval of adjacent vertices for a given set of vertices. Further, the index structure can be also used to access the connecting edges via their corresponding identifier. In the following we describe an efficient initial loading mechanism to construct the adjacency-based topology index and also describe the internal mapping data structures.

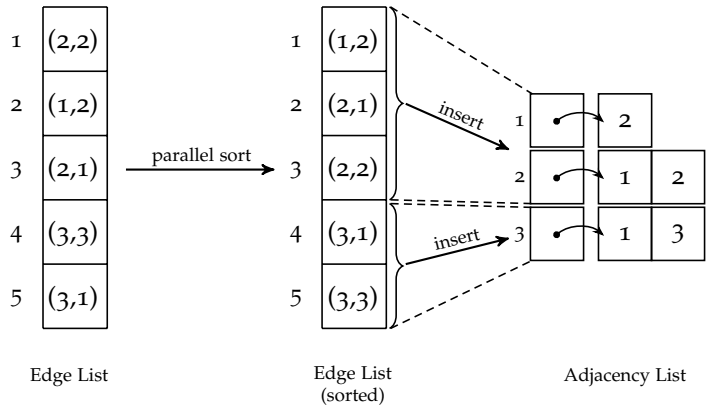


Figure 6.11: Parallelized construction of the adjacency list from an edge list.

6.4.1 *Index Construction*

The core adjacency list consists of a nested-array structure, where the *outer array* stores pointers to the corresponding *inner arrays*. The outer array keeps one entry per vertex in the graph. The inner array stores an unordered set of vertex IDs—the adjacent vertices. We leverage the dictionary encoding capabilities of GRAPHITE to store only densely-packed value codes in the adjacency list. Thus, we can use positional indexing in the outer array to retrieve the adjacent vertices for a given vertex.

We build the core adjacency list from a projection of the edge column group to the edge ID, the source vertex, and the target vertex column. The algorithm is based on three passes: a *statistics gathering pass*, a *sorting pass*, and an *insertion pass*. We use a *multi-pass* algorithm because a parallelized *single-pass* algorithm increases the construction time considerably due to increased synchronization overhead between worker threads. In a single-pass algorithm, each insertion into the data structure has to acquire a lock to avoid data races on the inner array of the corresponding vertex. Even worse, for adjacencies of *super nodes*, memory reallocations during the adjacency list construction lead to unnecessary copy operations. If multiple threads write to the same cache line by inserting new vertices into the same inner array, CPU *false sharing* effects limit the multi-core scalability of the algorithm and slow down the construction routine even further.

Figure 6.11 illustrates the steps of the parallelized adjacency list construction routine. The *statistics gathering pass* collects information about the average vertex outdegree d_{out}^{avg} and the largest used vertex ID, which could be also gathered from the vertex ID dictionary. Based on the largest vertex ID, we allocate memory for the outer array and for each of the inner arrays to hold at least d_{out}^{avg} vertices. The *sorting pass* reorders the edge list first by *source vertex*, then by *target vertex*. In a subsequent step, we determine the partition boundaries, i.e., the start and end positions in the edge list that should be handled by a single thread. By applying a sorting on the *source vertex* and a block assignment, we can ensure that each inner array is only operated by a single thread. This effectively eliminates the need to lock the inner array and avoids CPU false sharing. In the final *insertion phase*, we perform the parallelized edge insertions, where each thread handles a subset of edges and inserts new vertices by appending them to the corresponding inner array.

6.4.2 *Mapping Tables*

To combine graph with relational processing, such as filtering and aggregation on the vertex and edge column groups, we add mappings between the adjacency list and the corresponding vertex and edge column groups. GRAPHITE keeps bidirectional, light-weight, and updateable mapping tables between the adjacency list and the corresponding column groups. We maintain both combinations, i.e., to continue the execution on the intermediate

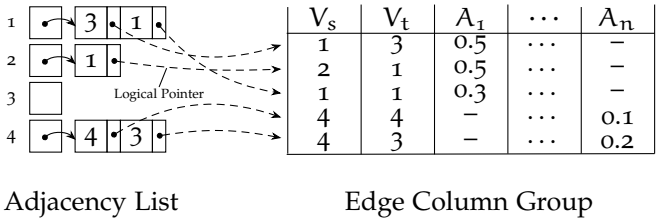


Figure 6.12: Mapping of entries in the adjacency list to rows in the edge column group through logical pointers.

result from a relational operation on the adjacency list or to access vertex/edge attributes from within the adjacency list. In the following we describe the initial construction of the mapping tables to and from the edge column group. The vertex mapping tables can be trivially constructed since the insertion of new vertices does not require to propagate them to the adjacency list.

MAPPING TO EDGE COLUMN GROUP. We use the mapping to the edge column group to support attribute access operations on intermediate results of graph operations without having to scan the complete edge column group. For example, a set of adjacent vertices might be further processed by looking up and aggregating some vertex attribute value. Effectively, we maintain *logical pointers* to the corresponding physical positions in the edge column group, as depicted in Figure 6.12. Technically, we map adjacent vertices (or implicitly the edge over which the vertex has been reached) to their corresponding row IDs in the edge column group and maintain a second nested-array data structure, where we store the corresponding row IDs.

MAPPING FROM EDGE COLUMN GROUP. The mapping from the edge column group to the corresponding entries in the adjacency list requires two positional indices to locate a single entry in the adjacency list. One index is required to access the outer array and one to access the inner array, respectively.

We use two array-based data structures to derive the pair of indices to locate an edge in the adjacency list: *a prefix array* and a *mapping array*. The *prefix array* provides global entry points into the adjacency list and stores a prefix sum of all neighborhood sizes, one for each entry in the outer array. The *mapping array* stores local entry points into the adjacency list through mapping values that provide local information of the vertices within a single neighborhood and how they can be accessed. In combination with the *prefix array*, a mapping value allows computing the position of the target vertex for the corresponding edge in the inner array.

We construct the prefix array by computing a parallel prefix scan over the core adjacency list. Subsequently, we iterate over the mapping to the edge table–the mirrored adjacency list with row identifiers—and compute for each entry the mapping value using the computed sums from the prefix scan.

Figure 6.13 depicts an exemplary mapping between the adjacency list and the corresponding edge column group. For example, to access the adjacency list entry for the edge with ID 2, we first retrieve the corresponding mapping array entry at position 2 (in this case the mapping value is 2). Next, we use the prefix array and perform a search to retrieve the largest element, which is lower or equal to the mapping value (in this case 0_2). The index for the outer array (the source vertex) is the position 0_2 in the prefix array. The index for the inner array—the index within the adjacency—can be computed as the subtraction of the entry in the prefix array from the mapping value. For the edge with ID 2, we retrieve the tuple $\langle 2, 0 \rangle$.

Algorithm 9 sketches the routine to compute the adjacency list indices for a given edge. In Line 2, we derive the mapping value `m` for edge `e` via positional index. We use the mapping value `m` to determine the outer array index by performing a binary search on the

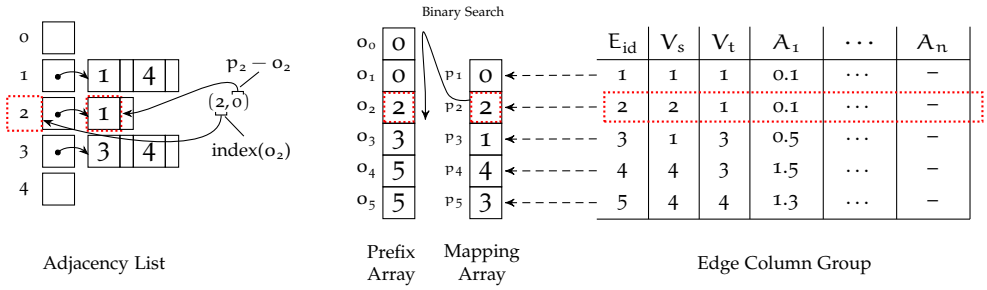


Figure 6.13: Mapping of rows in the edge column group to entries in the adjacency list.

sorted prefix array. Next, we store the position of the largest value, which compares lower-or-equal to `m` in `p` (Line 3). Finally, we subtract the prefix sum value found at position `p` in the prefix array from the mapping value `m`.

Algorithm 9: Adjacency list index computation

```
   Input  : e = Edge
   Output : i_o = Index for accessing the outer array
   Output : i_i = Index for accessing the inner array
1  Procedure computeIndices(e, i_o, i_i)
2      m ← mapping[e] ;                          // Compute mapping value
3      p ← upper_bound[m] − 1 ;   // Compute position for lower-equal (binary search)
4      i_o ← p;
5      i_i ← m − prefix[p];
```

6.4.3 *Index Maintenance*

We use a light-weight update routine that propagates insertions performed to the edge column group to the adjacency list. Upon insertion of a new edge into the edge column group, we use an internal message bus in GRAPHITE to notify the registered adjacency list index about the newly inserted edge. Updating the core adjacency list is as simple as appending the target vertex of the edge to the corresponding adjacency of the source vertex. Similarly, we update the structure, which maps entries in the adjacency list to entries in the edge column group.

Since updating the mapping table, which translates edges in the edge column group into their corresponding positions in the adjacency list would be to expensive to maintain in-place, we use a delta buffer to keep the mapping table up-to-date. The delta buffer is a dynamic array with offset-encoded positional access to newly inserted edges. As value, the delta buffer stores a tuple of indices, where the first index encodes the position in the outer array and the second index to the inner array, respectively.

6.5 EXPERIMENTAL EVALUATION

In this section we conduct an experimental evaluation of the *block-based topology index* and the *adjacency-based topology index*. We compare both index structures and their respective traversal implementations as well as the scan-based traversal against each other. We implemented both index structures in GRAPHITE on top of a dictionary-encoded edge column group, which effectively enables the traversal algorithm to operate directly on value codes.

6.5.1 *Experimental Setup*

We conducted all experiments on an INTEL® XEON® E5-2660v3 machine with 2 sockets, 10 cores per socket, each core running at 2.6 GHz, and 2 threads per core. The machine runs on SLES 12 SP1 and is equipped with 128 GB of DDR4 RAM and 25 MB last level cache.

We use the following data sets—we refer the reader to Appendix A for detailed graph topology characteristics—in our experiments: social graphs (AMAZON, LIVEJOURNAL, ORKUT, POKEC, TWITTER), road graphs (CALI), web graphs (WIKIPEDIA), citation graphs (PATENTS), computer network graphs (SKITTER), and generated graphs using the R-MAT data generator (RMAT-SF20, RMAT-SF22, RMAT-SF24, RMAT-SF26).

Initially, we load the data sets into the corresponding vertex and edge column groups. For each performance experiment, we randomly select 100 start vertices and perform traversals with up to depth 10 and report the median elapsed time. In the following we present our experimental results—for each evaluated index structure we report the construction time and memory footprint as well as their query performance for traversal queries on static and dynamic graphs.

6.5.2 *Index Construction Time and Memory Consumption*

We investigate the impact of the graph topology on the construction time and the memory footprint of the block-based topology index and the adjacency-based topology index, respectively. Our experimental results are summarized in Table 6.1. For the block-based topology index, we omit the results for varying block sizes, since the block size has only marginal impact on the overall memory consumption. This can be explained by the fact that the index structure consists of two main parts, a *block index*, which maps value codes to blocks and a *block range vector*, which stores the block boundaries. Since the number of vertices in the graph is usually larger than the number of blocks, we can assume that the overall index memory footprint is dominated by the number of vertices in the graph. For all subsequent experiments, we use a fixed block size of 512. For the adjacency-based topology index, we report the construction time and the memory footprint for the creation of the core adjacency list and the mapping tables to the corresponding edge column group.

The block-based topology index is a memory-efficient secondary index structure and on average consumes less than 5% of the CSV-based edge list graph representation on disk. This can be explained by the fact that the block-based topology index has an worst-case space complexity of $\mathcal{O}(V + |B|)$, where $|B|$ refers to the total number of blocks, and for all evaluated graphs $|V| << |E|$ holds. Since the block-based topology index does not replicate the complete graph topology, the memory consumption is considerably lower than similar primary, adjacency-based index structures.

We observe a similar performance trend for the construction of the block-based topology index. We construct the index structure in two passes, one for determining the block ranges and one for inserting the block assignments into the index We parallelize each step by partitioning the edge column group into equally sized chunks and assigning each chunk of edges to a worker thread. For the majority of the evaluated data sets, it took considerably less than a second to construct the block-based topology index. Exceptions are the large data sets WIKIPEDIA, RMAT-SF26, and TWITTER, which take up to 37 s for the index construction.

In contrast to the block-based topology index, the adjacency-based topology index replicates the complete graph topology into a dedicated data structures, thereby exhibiting a considerably higher memory footprint. The space complexity of the core adjacency list is $\mathcal{O}(E)$ as it stores one entry for every edge in the graph. The space complexity of the mapping table to the edge column group is $\mathcal{O}(E)$, as it points for each edge in the graph to the corresponding position in the edge column group. The mapping table from the edge column group to the adjacency list has a space complexity of $\mathcal{O}(V + E)$, with a space complexity of $\mathcal{O}(V)$ for the prefix array and a space complexity of $\mathcal{O}(E)$ for the mapping array. Compared to the size of the corresponding edge list on disk, the adjacency-based topology

Table 6.1: Memory consumption and construction time for the block-based topology index (minimal block size = 512) and the adjacency-based topology index.

Data Set	Block-Based Topology Index		Adjacency-Based Topology Index	
	Memory Footprint [MB]	Construction Time [s]	Memory Footprint [MB]	Construction Time [s]
AMAZON	1.5	0.15	80.61	0.24
RMAT-SF20	2.3	0.35	389.16	0.67
SKITTER	3.7	0.36	266.89	1.08
POKEC	5.7	0.5	713.35	1.37
PATENTS	8.1	0.64	406.89	2.14
RMAT-SF22	8.2	0.81	1,555	2.47
ORKUT	11.1	0.99	2,706	3.36
LIVEJOURNAL	16.9	1.24	1,604	3.90
RMAT-SF24	30.6	2.07	6,214	9.03
WIKIPEDIA	101.4	25.38	13,954	21.87
RMAT-SF26	113.7	23.51	24,840	63.02
TWITTER	158.2	37.11	33,926	58.87

index consumes up to 1.2× the memory and up to 3.5× the memory of the in-memory edge list representation. Compared to the construction time of the block-based topology index, the adjacency-based topology index is up to a factor of three slower, which is mainly caused by the creation of the mapping tables to the edge column group and the creation of own dictionary structures.

6.5.3 *Traversal Performance*

In the next experiment we compare four different traversal implementations against each other. For all experiments across different traversal implementations, we use the same query configuration—same start vertex and traversal depth—and only vary the traversal strategy to obtain comparable results. Specifically, we evaluated the following traversal implementations:

- a scan-based traversal
- an index-based traversal (using the block-based topology index)
- a hybrid scan-/index-based traversal (using the block-based topology index)

6.5.3.1 *Static Graphs*

In the first experiment, we evaluate the block-based index and the corresponding index-based traversal for different block sizes $\{2^9, 2^{12}, 2^{15}\}$ and a scan-based traversal. We present the results in Figure 6.14. For all evaluated data sets, the index-based traversal outperforms the scan-based traversal by up to two orders of magnitude for short traversals with a traversal depth of up to four. The break-even point, where the scan-based traversal starts to outperform the index-based traversal, depends on the underlying graph topology. For example, for social graphs, such as POKEC, LIVEJOURNAL, and the data sets generated by R-MAT, this break-even point is reached at a traversal depth of around 3 to 4. One exception is ORKUT, which is also a social network graph, but has not such an extremely power-law

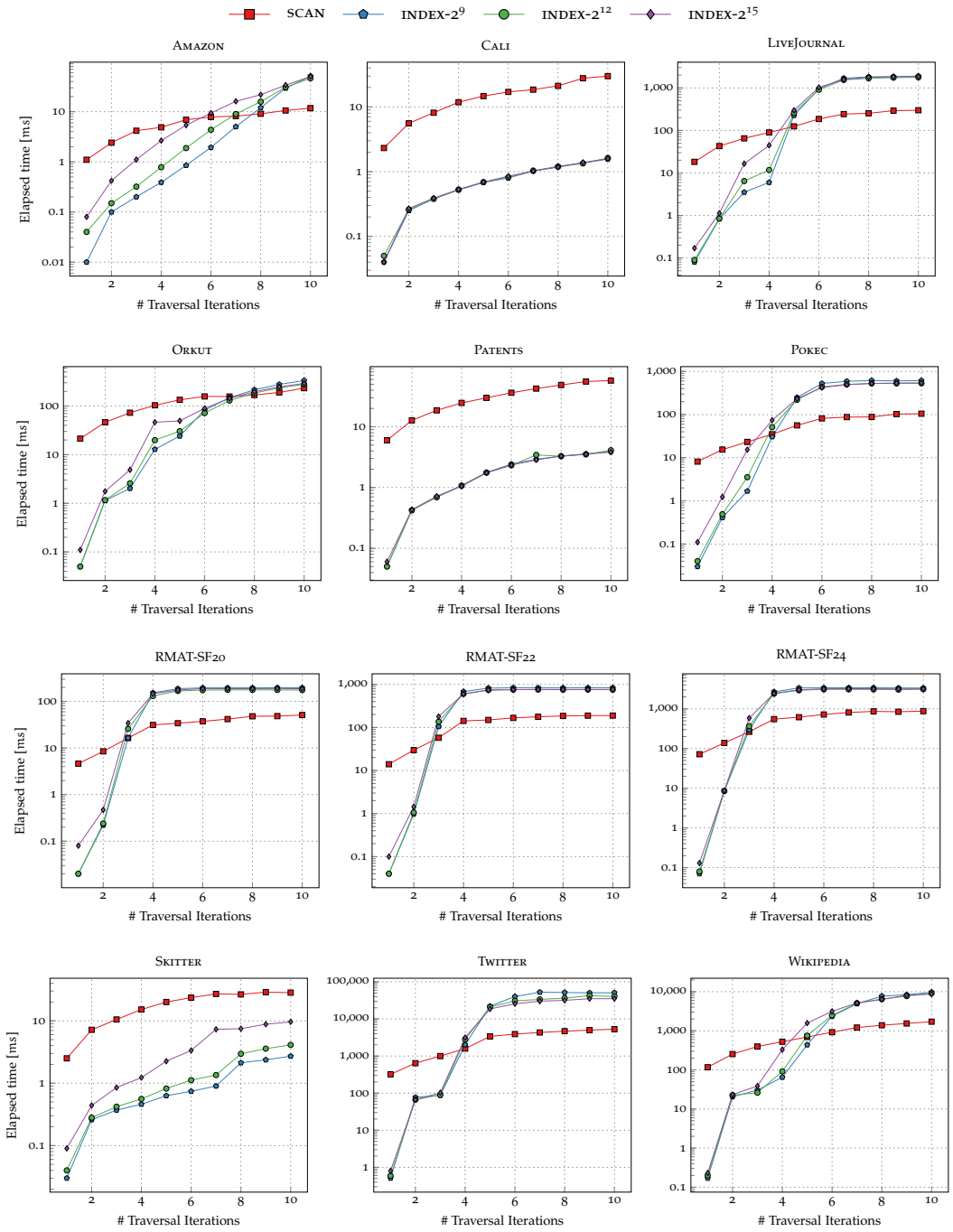


Figure 6.14: Performance evaluation of the scan-based traversal implementation (—■—) and the index-based traversal implementation (—⬟—, —●—, —◆—) with varying block sizes drawn from $\{2^9, 2^{12}, 2^{15}\}$.

degree distribution leading to a better performance for the index-based traversal. In contrast, for extremely sparse graphs, such as SKITTER, PATENTS, and CALI, the scan-based traversal never outperforms the index-based counterpart. While analyzing the size of intermediate frontier sets, we observed very large frontier sets for social graphs, leading to a performance degradation for the index-based traversal caused by a large number of index lookups and consequently a large number of range scans. This is analogous to the classical decision problem in an RDBMS to perform—depending on the query selectivity—a full-column scan or an index scan on the column. If the number of index lookups to perform is large, a brute-force full-column scan is preferable while for a small number of

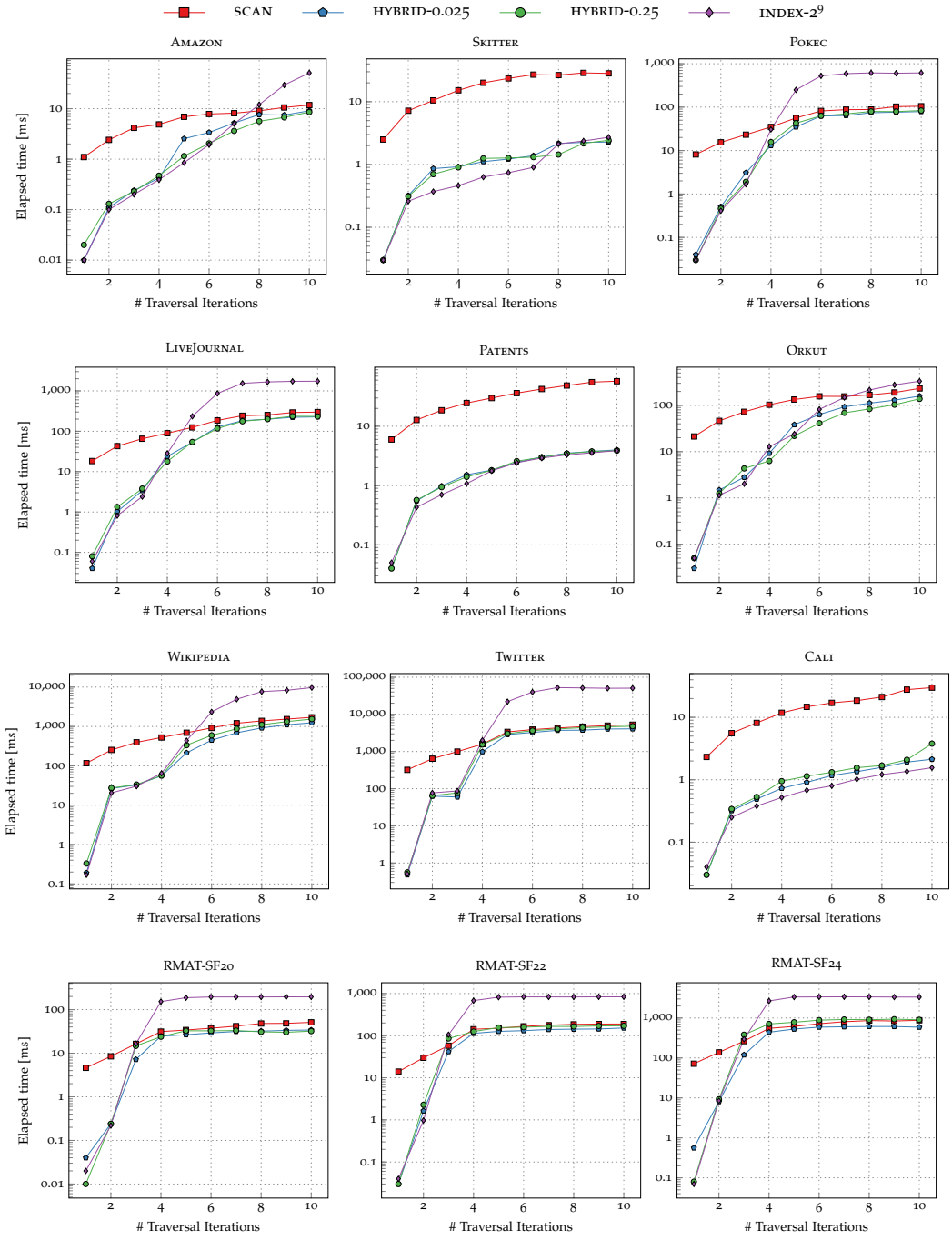


Figure 6.15: Performance evaluation of hybrid scan-based/index-based graph traversal.

index lookups, an index scan is advantageous. For social graphs, the size of the frontier set grows fast with increasing traversal depth, reaching a peak size after 3 to 5 traversal hops. For large frontier sets, the overhead of the index-based traversal is significant and the scan-based traversal outperforms the index-based counterpart by up to two orders of magnitude for these queries.

Figure 6.15 summarizes the results of our next experiment, where we evaluate and compare the hybrid traversal operator against the purely scan-based and index-based traversal implementations. For the hybrid traversal, which switches automatically between an index-based and a scan-based traversal depending on the size of the frontier set, we use two exemplary threshold values—2.5% and 25%—to switch between the two modes. The

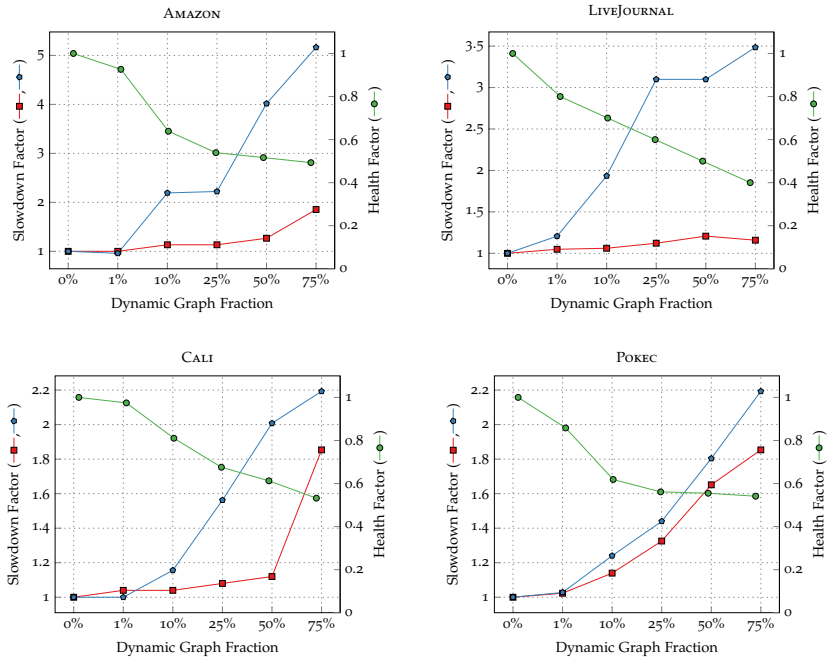


Figure 6.16: Slowdown (in multiples of execution time on the respective static graph) factor and health factor (—●—) of block-based topology index on dynamic graphs with varying dynamic part fractions drawn from {0%, 1%, 10%, 25%, 50%, 75%}. We evaluate two different traversal queries: 2-hop traversals (—■—) and 5-hop traversals (—⬟—).

threshold defines the fraction of the frontier set compared to the total number of vertices in the graph. For the index-based traversal implementation, we used a block size of 512.

For all evaluated data sets, the hybrid traversal provides comparable query performance for short traversals to the index-based traversal, but switches to a scan-based traversal for larger frontier sets. Thereby, the hybrid traversal avoids the execution time explosion of the index-based traversal and stays below the execution time of the scan-based traversal for deep traversals due to the performance gain in the first iterations. For extremely sparse graphs, such as Skitter, Patents, and Cali, the hybrid traversal does not switch to the scan-based traversal at all. This is because the size of the frontier set never reaches our defined threshold for switching between the two modes. Interestingly, the configuration of the threshold is not very sensitive to different input values. We experimentally increased the threshold even further to up to 75% and found that the optimal switching criterion, i.e., when the scan-based traversal starts outperforming the index-based traversal is at around 40%.

6.5.3.2 *Dynamic Graphs*

In the following experiment we evaluate the impact of edge insertions on the block-based topology index and present our results in Figure 6.16.

For each experiment, we divide the data set into a *static part*, which we import at program start, and a *dynamic part*, which we insert batchwise as part of the workload. In detail, we conduct the experiments as follows: (1) we load the static part of the graph and generate the read-optimized storage format, (2) we insert the dynamic part of the graph and generate the write-optimized storage format, (3) we generate the block-based topology index, and (4) we run the traversal queries.

We generate workloads with varying sizes of the dynamic part (in percentage p of the original graph) drawn from {0%, 1%, 10%, 25%, 50%, 75%} to simulate different graph workloads, ranging from read-only to write-heavy graph scenarios. In a preprocessing step, we randomly remove edges with probability p from the original data set and generate in-

sert statements for them. By loading always the entire original data set, we avoid query processing artifacts that stem from varying result set sizes due to cut paths in the graph.

We evaluate the index-based traversal on dynamic graphs for three representative data sets: AMAZON, CALI, POKEC, and LIVEJOURNAL. For each data set, we report the *slowdown factor*, i.e., the quotient of the current traversal execution time and the corresponding base line on a static graph. On the second y-axis, we report the corresponding *health factor* of the block-based topology index. For all data sets, we can see that the health factor decreases with an increasing dynamic graph fraction, i.e., neighborhoods are more cluttered over multiple blocks, resulting in a higher execution time because multiple blocks have to be accessed to retrieve the adjacent vertices for a given vertex. Longer traversals are more affected by a larger dynamic graph fraction as they tend to discover more vertices and perform more index lookups to retrieve adjacent vertices, consequently resulting in a larger slowdown factor compared to an equivalent traversal on a static graph. Depending on the graph topology, the size of the dynamic fraction of the graph can a different impact on the overall execution time. For example, for a 5-hop traversal and a 10% dynamic graph fraction, a query exhibits a slowdown of 1.2× for CALI, while it is about factor 2 slower for AMAZON. If the application has tight query performance constraints, only a moderate slowdown—if at all—can be tolerated. In such a case, the index should be rebuilt once the health factor of the index goes below 0.95, otherwise the slowdown factor might not be tolerable any more.

6.6 SUMMARY

In this chapter we presented two graph index structures—a block-based topology index and an adjacency-based topology index—to accelerate neighborhood queries. In contrast to a scan-based approach with a complexity of $\mathcal{O}(|E|)$, a neighborhood query on the proposed index structures has an overall runtime complexity of $\mathcal{O}(c)$ for a small constant c.

We proposed two secondary index structures with a low memory footprint and fast construction methods that can be even used on large graphs with billions of edges. The block-based topology index is an auxiliary data structures that indexes the edge column group and allows to limit the range to scan to a block level. If the edge column group is perfectly clustered, all adjacent vertices for a given vertex can be retrieved from a single block. The index supports an efficient update routine that allows to maintain the index in constant time at the expense of giving up the cluster criterion on the index. Our experiments show that for read-mostly graph workloads the performance penalty compared to a read-only workload and a perfect clustering is less than 10% on average.

The adjacency-based topology index is at its core a native adjacency list with additional mapping structures to seamlessly allow hybrid query processing on the column groups and the adjacency list. This comes with a considerably higher memory footprint compared to the block-based index of about 100×, but also achieves significantly higher query performance speed as no scan operations are necessary to fetch a set of neighbors for a given vertex.

Finally, we proposed a hybrid traversal implementation that combines scan-based and index-based execution with the operator. At each traversal iteration, the operator decides based on the size of the frontier set, whether to process the next iteration scan-based or index-based. In the experiments we show that a hybrid solution can preserve the performance of the index-based traversal for the first traversal iterations and switches to the scan-based traversal once the frontier set becomes too large.

7 TRAVEL — A DSL FOR GRAPH ANALYSIS

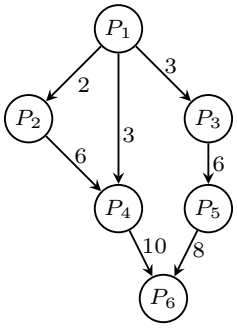
The abundance and diversity of massive-scale graph-structured data and the ever-growing interest of large enterprise companies to analyze them are the key drivers of the recent advances in graph data management research. From a systems perspective, there is a plethora of graph processing systems to choose from—all tailored to different use cases and programming models. On the other side of the spectrum there is a tremendous amount of efficient graph algorithms designed for domain-specific scenarios. Especially graph traversals—one of the most fundamental building blocks for graph processing—have been studied extensively, with contributions for various system architectures, ranging from commodity notebook machines to many-core, distributed server clusters.

Real-world graph applications are typically domain-specific and model complex business processes in property graphs. To implement a domain-specific graph algorithm in the context of such a graph application, simple graph traversals are not expressive enough nor do they allow customization to the user's needs. For example, a summarized bill-of-materials (BOM) explosion does not only traverse the part hierarchy, but also accumulates part quantities to build the final result containing all subparts and their corresponding quantities. Standard traversal algorithms are not only limited in their extensibility and expressiveness, but they also provide a fixed traversal semantics, i.e., a repeated visit of specific vertices or a data-dependent traversal restriction to certain paths is not possible.

To cope with these issues, graph database vendors provide—in addition to their declarative graph query languages—procedural interfaces to write user-defined graph algorithms (Neo; Spa; Ori; Inf). Such imperative interfaces offer a powerful abstraction to write user-defined, domain-specific graph algorithms, but they also have major drawbacks. A procedural programming interface is cumbersome to use and requires the user to specify the graph algorithm against a low-level graph API in a general-purpose programming language, such as C++ or JAVA. Additionally, if the user code runs in the same operating system process as the database server, software bugs or misuse of available resources can pose the danger of harming the overall database system stability significantly. If the user code runs in a separate process, the algorithm might suffer from network communication delays between client and server process due to excessive data transfers. To the worse, writing graph algorithms in a general-purpose language prevents exploiting data- and domain-dependent optimizations at runtime and certain query optimization and rewriting techniques, such as selection push-down and leveraging intra-query parallelism cannot be applied.

In this chapter, we introduce *traversal hooks*, a powerful concept to extend and manipulate graph traversals with domain-specific code provided by the user. Traversal hooks follow an event-based programming model and provide an interface for a variety of traversal events, such as the discovery of a new edge or the visit of an already discovered vertex. Although they act similar to database triggers and execute a piece of code provided by the user at certain traversal events, traversal hooks only react to query operations. We provide two built-in traversal strategies—breadth-first (BFT) and depth-first (DFT)—which can be extended by traversal hooks and a high-level, domain-specific language called TRAVEL. We use the LLVM framework to glue together the traversal code with the traversal hooks at runtime and generate efficient user-defined traversal operators on the fly. We leverage static program analysis to rewrite and embed the traversal hooks into traversal operators and compile them for later use. Our contributions can be summarized as follows:

- We describe the programming model and the concepts behind traversal hooks and introduce a novel domain-specific language TRAVEL for traversal-based graph algorithms.



Part	Subpart	Total Qty
P1	P2	2
P1	P3	3
P1	P4	15
P1	P5	18
P1	P6	294

Figure 7.1: Summarized BOM explosion for part P1.

Listing 7.1: Summarized BOM explosion.

```
-- Preamble
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> sum = 0;

-- hook definition
HOOK EDGE "H" (CONTEXT $e) {
  $h = HEAD($e);
  $t = TAIL($e);
  UPDATE $t { SET sum = $t@sum + ($e@quantity * $h@sum); }
}

-- Traversal definition
TRAVEL "BOM" (VERTEX $root) GRAPH "G" {
  UPDATE $root { SET sum = 1; }
  TRAVERSE BFS $root-->(*) HOOK "H";
}
```

- We show the generation of efficient executable code from TRAVEL scripts and describe in detail how we perform code optimizations using static program analysis.
- We evaluate our implementation of traversal hooks and TRAVEL based on series of real-world applications, show how traversal hooks can be expressed in TRAVEL, and perform an experimental evaluation for a large variety of realistic graphs and applications.

Introductory Example

Query support for bill-of-materials (BOM) applications is a common requirement in business environments. A BOM hierarchy is represented as an acyclic, single-rooted graph and describes the relationships between product parts as depicted in Figure 7.1. The graph contains an edge attribute `quantity`, which describes how many instances of the *subpart* are required to manufacture the *part*. To illustrate the use of traversal hooks, we revisit a fundamental operation on BOM hierarchies—summarized BOM explosion. It provides an answer to the question "What is the total quantity of each part required to build part P1?".

Listing 7.14 depicts a summarized BOM explosion expressed in TRAVEL. A TRAVEL script consists of an optional preamble, followed by a set of traversal hook definitions, and a main clause. We use a temporary vertex attribute `qnty` to collect intermediate results and initialize all values to zero, except for the root vertex P1. For this example, we define a traversal hook to react on newly discovered edges. For each traversal hook invocation, we extract the head and the tail vertex from the context edge `$e` and store them in two temporary variables. We update the temporary vertex attribute `qnty` with the sum of the `qnty` of the tail vertex `$t` and the multiplication of the edge weight `quantity` and the vertex

attribute `qnty` of the head vertex `$h`. The final result of the summarized BOM explosion starting from root vertex `P1` is shown in the table in Figure 7.1. For a more detailed description of TRAVEL, we refer the reader to Section 7.3.

The remainder of this chapter is structured as follows. We discuss related work in Section 7.1, where we compare several high-level graph programming models and interfaces to TRAVEL. In Section 7.2 we introduce the TRAVEL programming model and describe its language features in Section 7.3. We outline the main components of TRAVEL and the code generation process in Section 7.4 and present several optimizations thereof in Section 7.5. Section 7.6 describes a set of realistic use cases where we show the generality and applicability of our programming model and TRAVEL. In Section 7.7 we conduct an experimental evaluation, including low-level micro-benchmarks and complex TRAVEL scripts, before we summarize our findings in Section 7.8.

7.1 RELATED WORK

We divide the review of related work into three categories, namely programming models, high-level graph abstractions, and code generation techniques.

7.1.1 *Programming Models*

A related concept to traversal hooks is the *visitor concept* implemented in the Boost Graph Library (BGL) (Siek et al., 2002). The BGL provides a set of built-in graph algorithms, such as breadth-first and depth-first traversals, and allows the user to pass additional logic to the graph algorithms through functors. Each graph algorithm can be optionally extended by multiple visitor objects, which are invoked at specific event points. We follow a similar programming model, but restrict the visitor types to the most general ones, i.e., the discovery of an edge or a vertex. The BGL is a compile-time library and uses C++ templates extensively. Thus, the BGL achieves a similar query performance compared to TRAVEL, but lacks a mechanism to generate and instantiate graph algorithms dynamically at runtime. Additionally, all optimizations performed are merely low-level compiler optimizations, but do not exploit any high-level knowledge about the restrictive semantics of the visitor concept.

The *Gather-Apply-Scatter* (GAS) model is a vertex-centric computation model initially coined by Low et al. (2010). The GAS model is conceptually divided into three phases: *gather*, *apply*, and *scatter*. The gather phase collects information from adjacent vertices/edges and passes the result to the apply phase, where a user-defined function is applied on all vertices in parallel. Finally, the scatter phase propagates computed values to adjacent vertices.

A large variety of graph processing systems—such as GRAPHLAB and PREGEL—implement the GAS model with specific modifications. For example, PREGEL (Malewicz et al., 2010), a distributed graph processing framework based on the bulk-synchronous parallel model (BSP), uses *data pushing* through message passing in the scatter phase, whereas GRAPHLAB relies on a *data pulling* in the gather phase. Both strategies are primarily relevant for distributed graph processing, where the overall execution time is limited by the network delay and the available network bandwidth.

A program written for PREGEL runs in a sequence of iterations—so-called *supersteps*—where the framework invokes for each vertex a user-defined function. This in spirit similar to traversal hooks in TRAVEL with the exception that in PREGEL there is no invocation ordering within a superstep. In contrast, TRAVEL guarantees that the invocation ordering is steered by the traversal semantics. In PREGEL, the user-defined function is executed for all vertices in the graph, except if the corresponding vertex already voted to halt. In TRAVEL, the traversal hook is only executed for discovered vertices in the traversal operation. State between supersteps is passed by messages in PREGEL, TRAVEL keeps computation state between hook invocations in temporary attributes. PREGEL treats only vertices as first-class citizen of the programming model, user-defined functions cannot be applied on edges.

Algorithm 10: LIGRA breadth-first traversal implementation.

```
1  Procedure COND(i)
2      return (Parents[i] == −1);

3  Procedure UPDATE(s, d)
4      return (CAS(Parents[d], −1, s));

   Input  : Graph G, root vertex v_0.
   Output: Parent vector Parents.
5  Algorithm BFS(G, v_0)
6      forall v ∈ V do
7          Parents[v] ← −1;
8      Parents[v_0] ← v_0;
9      Frontier ← {v_0} ;
10     while Frontier ≠ ∅ do
11         Frontier ← EDGEMAP(G, Frontier, UPDATE, COND);
```

TRAVEL treats both, vertices and edges, as first-class citizen of the programming model and allows traversal hooks on vertices and edges, respectively.

GREMLIN (Rodriguez, 2015), a traversal-based graph query language allows the user to specify an arbitrary number of so-called *traversal steps*, which transform an input into an output. Examples for traversal steps include steps with side effects (addEdge, addVertex) and aggregating steps (count, max). Although the language by itself is extremely powerful and can be extended by a JVM-based host programming language and can be optimized using simple rewriting techniques, GREMLIN and its implementation is not targeting performance-critical graph algorithm problems.

7.1.2 *High-Level Graph Abstractions*

LIGRA Shun and Blelloch (2013) employs a parallel, level-synchronous BFT; we depict a simple implementation of a BFT in LIGRA in Listing 10 (adapted from Shun and Blelloch (2013)). It uses the `EdgeMap` function and user-defined functions to check if a vertex has been already discovered (Line 1) and to update the parent array (Line 3). In contrast to TRAVEL, LIGRA does not offer high-level graph algorithms as built-in functions and requires the user to specify even simple traversal-based algorithms using low-level programming constructs, such as the `VertexMap` and `EdgeMap` functions.

GREENMARL (Hong et al., 2012) is an imperative, domain-specific graph query language particularly designed for writing customized graph algorithms. It is part of the PGX graph analysis framework and compiles a high-level graph algorithm description into a backend-specific implementation using a source-to-source compiler. Currently available backends include graph processing units with a CUDA implementation, PREGEL with a vertex-centric implementation in JAVA, and single-node, multi-socket server machines with a parallelized C++ implementation. In contrast to TRAVEL, GREENMARL allows the user to explicitly define parallel regions in the algorithm, which are rewritten and parallelized by the compiler infrastructure. From a language perspective, TRAVEL adopts many of the language constructs available in GREENMARL and extends them with other graph-specific language constructs. Table 7.1 provides a detailed comparison of the most important language constructs of TRAVEL and GREENMARL. TRAVEL is designed to be more expressive for traversal-based algorithms and provides more fine-grained control over the traversal algorithm. For example, TRAVEL allows specifying the desired traversal depth natively in the language while in GREENMARL this can only be achieved by using additional variables to track the traversal depth manually. Additionally, GREENMARL does not allow performing custom

Table 7.1: Comparison of available language features in GreenMarl and TraveL.

	TraveL	GreenMarl
General Features		
Unbounded Traversals	✓	✓
Graph Modifications	(✓)	✗
Parallel Constructs	✓	✓
Invoking other functions	✓	✗
Type System		
Graph-Specific Types	Vertices, Edges, Paths	Vertices, Edges, Graphs
Collection Types	Multi-Sets	Sets, Ordered Sets, Sequences
Graph Traversals		
K-Hop Traversals	✓	✗
Traversal Extensions	✓	(✓)
Path Operations	✓	✗
Local Traversal Restrictions	✓	✓
Global Traversal Restrictions	✓	✗

actions, when edges are traversed. A TraveL script can be composed of multiple, isolated graph algorithms and can stored/reused as registered built-in function later on in other TraveL scripts. In a similar fashion, we allow the invocation of other built-in graph algorithms, such as graph reachability and shortest path computations.

The main differences between GreenMarl and TraveL, however, are driven by the environments in which they are used. GreenMarl and the accompanying compiler only provide a source-to-source compiler, the actual compilation into an executable unit is left to the application user. In contrast, TraveL compiles the textual algorithm representation into an LLVM module and subsequently uses the LLVM JIT compiler to generate an executable program on the fly and execute it. The main assumption of having two separate compilation units, one for the tuned, built-in graph algorithms and one for the actual GreenMarl script does not hold for the GreenMarl compiler. Instead, one compilation unit can be constructed as the built-in graph operators can be directly linked into the GreenMarl program executable. In contrast, TraveL has to deal with two different compilation phases and automatically patch the built-in graph algorithms with custom logic at runtime.

EmptyHeaded (Aberger et al., 2015) is a relational engine tailored to graph processing and provides a high-level, Datalog-like query language. The authors use the boolean algebra and recent advances from join theory, i.e., worst-case optimal joins (multiway joins) to construct a logical query plan using *generalized hypertree decompositions* (GHD). EmptyHeaded stores all relations in *tries* and formulates the main operations on a trie or set representation, respectively. The query language of EmptyHeaded supports conjunctive queries with aggregations and a simplified form of recursion. The recursion operation follows a similar semantics to Kleene-star or transitive closure. The recursion either terminates, when the relation does not change anymore or a user-defined convergence criterion is fulfilled. The code generation phase produces C++ code from the GHD representation.

7.1.3 *Code Generation*

Code generation for compiling SQL queries to efficient machine code has been an active area of research (Neumann, 2011; Nagel et al., 2014). HyPer (Neumann, 2011), a main-memory RDBMS for mixed OLTP/OLAP workloads uses the LLVM compiler framework (Lattner, 2002) to generate efficient machine code from SQL queries at runtime. Similar to TraveL, HyPer applies a mixed code generation consisting of complex logic written in C++, such as data structures, spilling to disk, and allocation of new memory, and the orchestration of the precompiled parts in LLVM-IR. HyPer exploits the possibility to easily call external C++ functions from the generated LLVM-IR code. We follow a similar model by exposing an internal, low-level graph API directly in the generated LLVM-IR code.

Cloudera Impala (Kornacker et al., 2015), a SQL engine for the Hadoop eco system, leverages the LLVM compiler framework to generate and compile code by eliminating unnecessary branching, data loads, and virtual function calls. This is similar to TraveL, where we perform vertex/edge attribute location and type resolution at compile time and inline calls to the traversal hooks. Like Impala, we use a hybrid approach that combines manual code generation emitting LLVM-IR with cross-compilation of built-in operators, such as the graph traversal, using Clang.

Tupleware (Crotty et al., 2015) is an analytics framework for compiling workflows of *user-defined functions* (UDF) into executable code for distributed execution. The code generation is based on the LLVM framework and aims at combining high-level query optimization with low-level compiler optimizations that are specific to the underlying hardware configuration. In contrast to TraveL, a workflow is a composite of an abstract workflow description consisting of high-level operations, such as *map*, *reduce*, and *loop*, and UDFs written in an LLVM-compatible programming language like Julia or Python. During compilation time, Tupleware generates LLVM-IR code from the UDF and links it to the program generated from the workflow graph. The UDF analyzer inspects the code for vectorizability and derives computation characteristics, i.e., whether the function is likely to be memory- or CPU-bound. Although the general compilation process is similar to TraveL, our approach is focused on graph analysis tasks and provides sophisticated graph-specific optimizations that are not present in Tupleware. Further, we expose a single interface to the end user by making UDFs—in TraveL called *traversal hooks*—a first class citizen of the language. We believe that our concept of traversal operators and specific extension points could be also integrated into a more general-purpose analytic systems such as Tupleware.

7.2 MODEL OF COMPUTATION

A graph traversal discovers new vertices and traverses over edges in a deterministic and well-defined manner. We use an event-oriented programming model and the notion of *traversal events* to allow end users to extend the ordinary graph traversal semantics with custom logic. By ordinary traversal semantics, we refer to the traversal order of BFT and DFT, i.e., to discover vertices level-by-level (BFT) or to discover vertices recursively (DFT).

Such traversal events include the discovery of new vertices and the traversal over edges. Although it would be possible to define other, more specialized traversal events, i.e., the repeated visit of a vertex/an edge, we argue that a restricted number of event types is sufficient to compose more complex traversal events.

Each traversal event triggers the execution of a user-defined action—called *traversal hook*— that is defined for this event type. A traversal hook can produce and access volatile and persistent state, which is shared between invocations. Additionally, a traversal hook can change the semantics of the underlying graph traversal and steer the traversal during runtime.

Figure 7.2 depicts the interaction between a traversal operator and the traversal extension with its components *traversal hook* and *traversal state*. The traversal operator calls the traversal hook for each triggered traversal event; the traversal hook can steer the traversal operation by either restricting/terminating or extending the traversal. The traversal hook

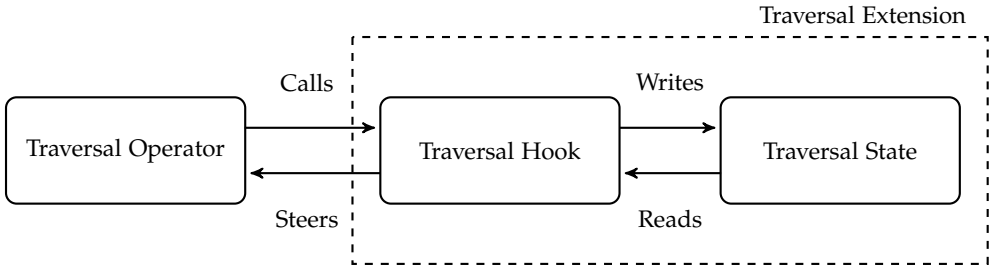


Figure 7.2: Interaction between the traversal operator and a traversal hook with state.

stores intermediate results in the traversal state, which is shared across all traversal hook invocations. Data stored in the traversal state is immediately visible in the logically subsequent traversal hook invocation. Multiple traversal hooks can be associated with a graph traversal operator, where each traversal hook is assigned to a traversal event type. For example, the user can specify two traversal hooks reacting to the discovery of new vertices, where each traversal hook by itself might perform a different action. Traversal hooks can either share the traversal state or have exclusive state that is only visible within the specific traversal hook. In each call, the traversal hook can read its traversal state and access the graph through a common graph programming interface.

7.2.1 *Traversal Events*

A traversal event $T(\tau, k)$ in a graph traversal τ describes the discovery of an item k—a vertex or an edge in the graph. The complete set of traversal events for a specific traversal operation is partially ordered with respect to the discovery order of vertices and edges. The concrete ordering of traversal events depends on the traversal strategy and the specific traversal implementation.

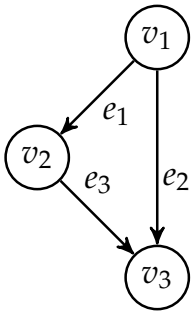
We support two traversal strategies, namely breadth-first (BFT) and depth-first (DFT). In a traversal strategy, it is implementation-specific, in which order neighbors in a BFT and paths in a DFT are evaluated. We evaluate traversal events in increasing order according to their assigned traversal number $t(v)$. For all $e \in E$ with $e = (u, v)$ $u, v \in V, u \neq v$ the following implication holds:

$$t(u) \leqslant t(v) \implies T(u) \leqslant T(e) \leqslant T(v) \tag{7.1}$$

Figure 7.3 depicts an example graph and a set of valid partial orderings for BFT and DFT traversals starting at vertex v_1, respectively. The partial ordering is a sorting of the assigned traversal number $t(v_i)$ for a vertex v_i in increasing order. For a BFT, we first traverse over edges e_1 and e_2, then discover vertex v_2 followed by vertex v_3 and so on. Whether we first traverse over edge e_1 and then e_2 is implementation-specific.

7.2.2 *Traversal Context*

A traversal hook receives a *traversal context* with information about the vertex or the edge triggering the event. Initially, the traversal hook only has access to the vertex/edge identifier—we refer to this as *minimal context*. The traversal context can be implicitly extended if the traversal hook accesses also attribute values or the neighborhood of the minimal context in the body of the traversal hook. The traversal context can be seen as the minimal unit of retrieval from the graph for each traversal hook invocation. By analyzing the traversal hook context in the compilation phase, we can determine data dependencies between traversal hook invocations and prefetch the required data for the traversal context before the actual traversal hook logic is executed.



Breadth-first:

$T(v_1) \leqslant T(e_1) \leqslant T(v_2) \leqslant T(e_2) \leqslant T(v_3) \leqslant T(e_3)$,
$T(v_1) \leqslant T(e_2) \leqslant T(v_3) \leqslant T(e_1) \leqslant T(v_2) \leqslant T(e_3)$,
[...]

Depth-first:

$T(v_1) \leqslant T(e_1) \leqslant T(v_2) \leqslant T(e_3) \leqslant T(v_3) \leqslant T(e_2)$,
$T(v_1) \leqslant T(e_2) \leqslant T(v_3) \leqslant T(e_1) \leqslant T(v_2) \leqslant T(e_3)$,
[...]

Figure 7.3: Partial ordering of traversal events from vertex v_1 for breadth-first and depth-first traversal.

7.2.3 *Traversal State*

We use three different ways to carry traversal state between traversal hook invocations: *temporary attributes*, *persistent attributes*, and *user-defined data structures*.

Temporary Attributes

A temporary attribute is a typed data container for storing either a vertex or an edge attribute and is only accessible during the query session. For example, a user might add a temporary vertex attribute `weight` to accumulate vertex-specific, intermediate results during the traversal. A temporary attribute can be referenced like a regular, persistent attribute and read/written multiple times. A graph can hold multiple temporary attributes during a single query session to capture multiple states between traversal hook invocations.

Persistent Attributes

A persistent attribute is a typed data container for storing either a vertex or an edge attribute and is made atomically visible after the query transaction commits. From a semantic point of view it is similar to a temporary attribute with the major difference being that the content is persisted at the end of the transaction in an atomic operation. When two queries run concurrently, each query instance has its own, private data container.

User-Defined Data Structures

We support different containers, which can be chosen from a set of predefined data structures, including unordered maps and lists. Both types can hold literals, vertices, edges, and simple paths.

7.2.4 *Traversal Control Flow Manipulation*

The traversal semantics describes in which order the vertices in a graph should be discovered. For example, a textbook breadth-first traversal implementation visits all neighbors of a specific vertex before exploring the adjacent vertices of the neighbors. Further, each vertex is discovered on the shortest path (with respect to the number of traversed edges) and is visited at most once.

For a realistic use case, this traversal semantics, however, can be to rigid. The user usually requires a more fine-grained control over the traversal logic to adjust it to the specific algorithm. We provide a mechanism to allow the user to steer the traversal operator and to explicitly modify the traversal semantics for a specific query. The control flow modification is specified in the traversal hook and uses signals to notify the invoking traversal operator

about the changed traversal semantics. We support two different variations of control flow modifications: *traversal restriction* and *traversal extension*.

Traversal Restriction

The traversal operator discovers vertices and expands the traversal on all edges that fulfill the conditions specified in the traversal configuration, including the traversal depth and vertex/edge filters. The evaluation of filters, however, can only be applied on a vertex/edge level. We propose a *dynamic filter*, which allows evaluating the filter condition in a data-driven manner, while being able to take a broader hook context and traversal state into account. For example, a graph algorithm that uses a budget to traverse the graph can only continue the traversal as long as the budget is not yet exhausted. We categorize traversal restrictions into *local* and *global* restrictions. A local traversal restriction operates on a hook invocation level and restricts the edge expansion for the specific hook context. In contrast, a global traversal restriction terminates *all*—potentially in parallel—running and scheduled traversal hooks.

Traversal Extension

By default, a graph traversal discovers each vertex at most once and never visits an already discovered vertex again. Further, each edge is traversed at most once to discover its target vertex. This traversal semantics allows terminating the traversal once all reachable vertices have been discovered or no undiscovered vertices are reachable anymore. Sometimes, however, a more lenient traversal semantics is desirable. Traversal hooks allow manipulating the traversal operator such that vertices can be visited multiple times and edges can be used for traversals more than once. However useful, a dedicated termination criterion has to be defined since otherwise the execution might not terminate. For example, a query might want to modify the traversal semantics to visit each vertex at most `k` times, where `k` might be some statically or dynamically computed value.

7.3 TRAVEL

TraveL is a domain-specific query language for writing complex graph algorithms. In contrast to a low-level programming interface, TraveL provides high-level, graph-specific language constructs to formulate graph algorithms in a user-friendly and intuitive way while retaining an equivalent execution performance to manually optimized implementations written in C++. TraveL exhibits a graph abstraction and natively supports fundamental graph data types, such as vertices, edges, and paths, and operations thereon. It facilitates an imperative programming model with control flow elements and data querying and manipulation operations. The core programming concept of TraveL are *traversal hooks*, which allow extending optimized, built-in graph traversal operators by user-defined program logic. In addition to imperative language constructs, such as loops and conditional statements, TraveL provides declarative language blocks for specifying filter conditions, traversal operations, and updates on vertex/edge attributes.

The TraveL execution engine relies on code generation and translates a TraveL script into a low-level intermediate representation and an accompanying program module using the LLVM compiler framework. TraveL accesses the data through a low-level programming interface, which is linked during the compilation phase to the generated TraveL program binary. Except for a low-level programming interface—which most GDBMS already offer—TraveL does not pose any prerequisites to the underlying graph processing system. In the course of this chapter, we describe the prototypical implementation of TraveL in Graphite, but argue that TraveL could be implemented on top of any other graph processing system exposing a certain set of functionality through a low-level programming interface.

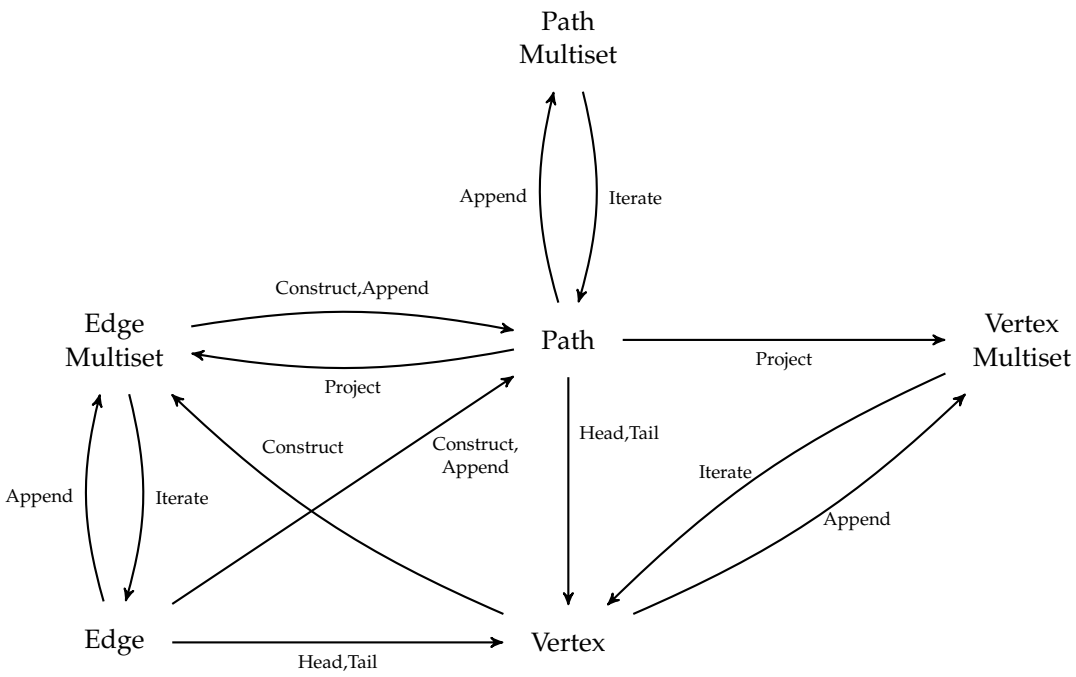


Figure 7.4: Type system and transformation operations in TRAVEL.

7.3.1 *Data Types*

TRAVEL is statically typed and supports a variety of primitive data types, including *integer*, *bigint*, *double*, and *string*. The TRAVEL type system is compatible with the accompanying backend implementation—in our case the type system of GRAPHITE. More complex attribute types, such as geometry or text types, could be integrated in TRAVEL as well.

TRAVEL supports three graph-specific types, namely *vertices*, *edges*, and *(simple) paths*. We uniquely identify each vertex and edge by a 64 bit identifier and label a vertex as `vertex:<ID>` and an edge as `edge:<ID>`, respectively. A path is an ordered sequence of interleaved vertices and edges, e.g., ⟨ `vertex:1,edge:1,vertex:2,edge:2,vertex:3` ⟩ forms a path of length two from `vertex:1` to `vertex:3`. Although the path could be reconstructed from the ordered sequence of edges only, we explicitly also store the vertices. A path is *valid*, if for any consecutive pair of edges $\langle\langle v_{k-2}, v_{k-1}\rangle, \langle v_{k-1}, v_k\rangle\rangle$ both edges share a common vertex v_{k-1}. TRAVEL supports a *multiset* type, which stores a homogeneous, unordered set with duplicates of elements. Additionally, we provide built-in support for *duplicate removal* and *sorting* on the multisets. The type of an element in the multiset can be *vertex*, *edge*, or *path*, respectively. Figure 7.4 depicts the TRAVEL type system and important transformation operations between different types.

We perform type checking of all TRAVEL expressions during the code generation phase and ensure that all subexpressions forming more complex expressions exhibit the same type. To that end, TRAVEL does not support automatic type promotion and value casting.

7.3.2 *Language Constructs*

In the following we introduce the main language features of TRAVEL. An algorithm expressed in TRAVEL is split into three parts: a preamble section, a traversal hook section, and a traversal section. Listing 7.2 depicts an exemplary TRAVEL script skeleton.

The optional preamble section consists of a set of statements to declare global variables and temporary/persistent vertex/edge attributes. The TRAVEL execution engine evaluates the preamble in a preprocessing step before the actual program runs so that declared data containers and attributes can be referenced in the TRAVEL script.

Listing 7.2: Structure of a TRAVEL script.

```
-- (optional) preamble section
CREATE ...

-- (optional) traversal hook section
HOOK EDGE "H1" (CONTEXT $e) {
  -- travel statements
}

-- (mandatory) travel section
TRAVEL "T" (VERTEX $source, INT $numHops, ...) GRAPH "G" {
  -- travel statements
}
```

The optional traversal hook section consists of a set of traversal hooks, where each hook is composed of a set of statements. We assign to each traversal hook a unique identifier, a hook type, and a hook context. The identifier can be used to reference the hook from other TRAVEL statements. We use the hook type to distinguish between vertex-centric (hook type *vertex*) and edge-centric (hook type *edge*) computations. Finally, the hook context provides a variable binding to the hook call context and can be used to reference the local vertex or edge in the body of the traversal hook.

The main clause acts as the entry point and orchestration part of a TRAVEL script. It consists of a set of statements to implement the graph algorithm and might reference declarations and hooks. A TRAVEL script can be identified using a globally unique name, which can be used to distinguish and reference different TRAVEL scripts. A TRAVEL script can receive an arbitrary number of read-only input parameters, where each parameter is a tuple consisting of a type and a variable identifier. Being able to specify input parameters allows a TRAVEL script to be invoked from other TRAVEL scripts as long as the return type and the input type match. Additionally, a TRAVEL script references the graph using a unique graph workspace identifier on which the script should run.

Declaration Statements

The preamble of a TRAVEL script consists of a set of statements to declare additional vertex/edge attributes and global variables. Each declared attribute or variable is globally visible and can be referenced using an identifier in any expression in the TRAVEL script. For attributes, additionally the identifier is unique across already existing attributes in the graph.

A vertex/edge attribute can be declared to be temporary or persistent. A temporary attribute is volatile and exists only during the execution of the TRAVEL script. Each script instance has its own private copy of the attribute and is agnostic to transient graph data modifications of other parallel running TRAVEL scripts. In contrast, a persistent attribute is non-volatile and is committed in an atomic operation at the end of the script. An attribute can be either initialized implicitly with the default value or explicitly with a value specified by the user.

```
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> A1 = 0;
CREATE PERSISTENT EDGE ATTRIBUTE<DOUBLE> WEIGHT;
```

A global variable can be of any supported primitive type in TRAVEL and either initialized with a default value or with a value specified by the user.

```
BOOL b;
INT i;
BIGINT b = 1L;
STRING s = "myString";
```

```
DOUBLE d;
```

In addition to vertex/edge attributes and variables of primitive types, TraveL supports built-in data structures to collect intermediate results. Currently, we support unordered multi-sets, lists, and unordered maps.

```
MULTISET<INT> C;
LIST<EDGE> L;
MAP<VERTEX,INT> M2;
```

Each container type can store primitive values of type *string*, *integer*, *bigint*, or *double* and complex types, such as *vertex*, *edge*, and *path*.

Variables

TraveL supports three different types of variables: *local*, *global*, and *built-in variables*. Each variable can be referenced by a variable identifier that is composed of a `$` followed by a sequence of alphanumeric characters. A local variable is used to split large, complex statements into smaller ones to improve the readability of the TraveL script. A global variable instead allows the user to maintain state during the computation. The assignment of the result of an expression determines the data type of the local variable. For a global variable, the data type has to be specified explicitly. After the initial assignment/declaration, the type of a global or local variable cannot be modified anymore.

We use two built-in variables—`$VERTICES` and `$EDGES`—to reference the set of vertices and edges in the graph, respectively. The built-in variable `$VERTICES` is of type *vertex multiset* and `$EDGES` is of type *edge multiset*.

Graph Access Expressions

We expose fundamental graph access operations directly via expressions and summarize them, their corresponding syntax in TraveL, and their return type in Table 7.2. The attribute access expression requires a variable binding or an expression, which evaluates to type *vertex* or *edge*, and an attribute name. Depending on the type of the variable binding, the attribute name will be either looked up in the vertex column group or in the edge column group. The resolution of whether the attribute is persistent or temporary and the actual data type of the specified attribute is performed in the backend. If the vertex or the edge does not exist in the graph, we raise an error and terminate the execution. TraveL currently does not support `NULL` value handling.

TraveL supports the specification of graph traversals and follows the semantics introduced in Section 5.2.1. A graph traversal starts from a single vertex or a set of vertices and performs a set of traversal iterations. A traversal expression can have an optional edge filter, which restricts the traversal to a subset of edges of the complete graph. The traversal direction can be either *forward* or *backward* and the traversal performs a limited number of traversal iterations or a recursive exploration of the graph until no more vertices can be discovered. Traversal expressions can be used in any place where a vertex multiset as return type is applicable, such as in loops, assignments, aggregations, and as core component of traversal statements.

```
$node-[@weight > 2]->
$node-[@weight > 2]->(2)
$node-[@weight > 2]->(*)
```

In addition to traversals and attribute access, TraveL supports several built-in access functions that transform between values of different types, in particular between vertices, edges, and paths. For example, for an edge or a path `$i` the expression `HEAD($i)` returns the head vertex and `TAIL($i)` returns the tail vertex, respectively. TraveL can also return the set of edges between two vertices. We use a multiset semantics here as two vertices can have

Description	Variable Type	Return Type	TraveL Expression
Multiset of vertices	-	Vertex Multiset	`$VERTICES`
Multiset of edges	-	Edge Multiset	`$EDGES`
Attribute Access	Vertex/Edge	Attribute data type	`$i@weight`
Traversal	Vertex	Vertex Multiset	`$node-[*]->`
Head vertex access	Edge/Path	Vertex	`HEAD($i)`
Tail vertex access	Edge/Path	Vertex	`TAIL($i)`
Edge constructor	Vertex	Edge multiset	`EDGES($u,$v)`
Edge constructor	Path	Edge Multiset	`EDGES($p)`
Vertex constructor	Path	Vertex Multiset	`VERTICES($p)`

Table 7.2: Graph access operations and their corresponding expressions in TraveL.

multiple edges in between. If there is no edge between the two vertices, TraveL returns an empty edge multiset. Similarly, for a path `$p`, the user can project on vertices and edges.

Relational Expressions

A relational expression is a boolean expression and evaluates to `true` or `false`. We support relational operators on vertex/edge attribute values, literals, and vertices, edges, or paths. The types of both operands in a relational expression must be equal.

```
$i <> $k
($i >= 1) AND ($i < 10)
```

TraveL supports all commonly known relational operators, including comparisons for equality (==), inequality (<>), greater/lower comparisons (<, >, <=, >=), and conjunctive (`AND`) and disjunctive (`OR`) combinations thereof.

Arithmetic Expressions

TraveL supports basic mathematical operations on numerical values, including addition (+), subtraction (-), multiplication (*), and division (/).

```
$k + 2
2 * ($k - 1)
2 * $v@weight
```

The operator precedence of arithmetic expressions can be modified by explicitly surrounding them with parentheses. The operands for an arithmetic expression can be literals, local variable bindings, and vertex/edge attribute values. TraveL currently does not support automatic type promotion and requires both operands to be of the same type.

Set Expressions

We support three types of collections in TraveL, namely multisets of vertices, edges, and paths. A multiset is an iterable, unordered bag of elements.

```
-- yields { vertex:1, vertex:2, vertex:3 }
{ vertex:1, vertex:2 } UNION { vertex:3 }

-- yields { vertex:2 }
{ vertex:1, vertex:2 } INTERSECT { vertex:2 }
```

```
-- yields { vertex:1 }
{ vertex:1, vertex:2 } MINUS { vertex:2 }

-- union of two traversal expressions
$n1-[@weight > 1]-> UNION $n2-[@weight > 1]->

-- yields {} (empty multiset)
EDGES($u,$v) INTERSECT EDGES($v,$u)
```

Two multisets of the same type can be coalesced, intersected, and subtracted from each other. A multiset expression can be only evaluated if both operands expose the same type, i.e., vertex, edge, and path multisets cannot be freely mixed. A multiset can be either created explicitly by listing the items wrapped in curly braces or implicitly as a result of another expression, e.g., a traversal expression or an edge construction expression.

Path Expressions

TraveL supports simple (acyclic) paths as a complex data type. A path can be constructed explicitly using the path constructor expression, which receives as input an ordered sequence of edges, or implicitly as a result of a graph algorithm returning a path or a set of paths. A path can be constructed if two edges in the input are "joinable", i.e., share one common vertex. A path can be also computed incrementally by appending new edges at the end of the path.

```
-- path construction
$path = PATH(edge:1,edge:2,...,edge:100);

-- extend path by another edge
$path = $path + PATH($e);
```

A path itself is not iterable, i.e., it cannot be directly used to iterate over it in a loop statement. TraveL, however, provides access methods to return the set of vertices and edges of a path, respectively. Further, TraveL can return the source and the target vertex of the path, respectively.

Assignment Statements

An assignment statement binds the result of an expression to a local variable. We provide assignment statements as a simple but effective language construct to split larger TraveL statements into smaller ones.

```
-- type is vertex
$v = vertex:23;

-- type is integer
$e = 1;

-- type is edge multiset
$edges = { edge:1, edge:2 };

-- type is vertex multiset
$neighbors = $v-[@weight > 0.5]->;

-- type is derived from the attribute type (e.g. string)
$val = $v@name;
```

All variables have to be initialized before they can be referenced and accessed in a TraveL expression. TraveL is statically typed and the data type of the object that is bound to the variable cannot change during the lifetime of a single query run.

Conditional Statements

TraveL allows the conditional evaluation of statements depending on the result of a boolean expression. As of now, TraveL only supports relational expressions as boolean expression to be evaluated in the context of conditional statements. Conditional statements can be arbitrarily nested and provide the same scoping rules for variable bindings like other high-level languages.

```
IF ($val@weight < 2) {
  -- travel statements
} ELSE {
  -- travel statements
}
```

Loop Statements

Besides set-oriented processing, TraveL also provides a built-in mechanism to iterate over a collection of elements. The object to iterate over has to be *iterable*, a requirement which is fulfilled by multisets in TraveL.

```
FOR $i : $n-[@weight > 1.0]-> {
  -- travel statements
}

FOR $i : $VERTICES {
  -- travel statements
}
```

The *iteration variable* can be referenced in the body of the loop clause and its type is derived from the type of an item in the set. In the body of the iteration statement, the set itself cannot be modified. The set to iterate on can be either a set that is bound to a local variable or the result of an expression returning a set.

Data Manipulation Statements

TraveL allows the modification of vertex and edge attribute values through update statements. Currently, the topology of the graph cannot be modified, i.e., the addition and deletion of vertices and edges is not allowed.

```
UPDATE $s {
  SET weight = 2*@weight,
  SET note = "Doubled";
}
```

An update statement uses a local variable binding to determine the vertex/edge or vertex/edge set to update. In the body of an update statement, there can be an arbitrary number of statements, each of them assigning a new value to the attribute of the context element. The right-hand side of the assignment can be composed of an expression that returns values of primitive types.

Traversal Statements

A traversal statement allows adding traversal hooks to traversal expressions and extend them with custom logic. Traversal statements play an important role in TraveL as they directly expose the hook-based programming model to the user.

```
TRAVERSE BFS $k-[@weight > 1.0]->(*) HOOK "H1";
TRAVERSE DFS $k-[@weight > 0.5]->(2) HOOK "H1";
TRAVERSE PATH $p HOOK "P";
```

We can assign a list of traversal hooks to a traversal statement, where each traversal hook is referenced by its hook identifier. A traversal statement receives either a path variable or a traversal expression describing the root vertex, vertex and edge predicates, and the traversal depth. For a path variable, the traversal operator traverses over a path and invokes the specified traversal hooks. For a traversal expression, the traversal strategy can be either breadth-first or depth-first. TRAVEL allows referencing multiple traversal hooks within the same traversal statement, even not restricting a traversal to a specific hook type.

Traversal Manipulation Statements

Since the traversal strategies *breadth-first* and *depth-first* can be too restrictive for some applications, TRAVEL provides two language constructs, namely `RESTRICT` and `EXTEND`, to modify the traversal behavior during runtime. The expression of a traversal manipulation statement is a relational expression.

```
RESTRICT $c > 2;
RESTRICT ALL $c > 2;
EXTEND $c <= 23;
```

In a default setting, a traversal is never restricted (`RESTRICT false`) and never revisits vertices/edges (`EXTEND false`) as long as new vertices can be discovered or the maximum traversal depth is not reached.

The traversal can be restricted to remove certain frontier vertices from the evaluation, which cannot be achieved by using static vertex and edge filters. For example, a global budget could be used to model travel costs in a graph; for each traversed edge, the travel budget is reduced by the cost assigned to the edge. When the budget is exhausted, the traversal should terminate. TRAVEL allows terminating either the complete traversal—using the `RESTRICT ALL` statement—or only dismisses the hook context vertex/edge using the `RESTRICT` statement. The extension of a traversal is useful to repeatedly visit vertices during the traversal. This is not possible in the default traversal strategies as they visit each vertex and edges at most once.

7.3.3 *Invoking Built-In Algorithms*

Although TRAVEL provides the expressiveness to implement arbitrary graph algorithms, it is reasonable to implement performance-critical graph algorithms directly in C++ using custom data structures and tailored parallelization strategies. We support the embedding and invocation of high-performance, built-in graph algorithms directly within a TRAVEL script. Such a hybrid execution combines the expressiveness and composability of a general-purpose language interface with the query performance of hand-crafted, specialized graph algorithms in a seamless manner. From an implementation perspective, built-in graph algorithms can be considered as larger building blocks of an extended graph API.

Listing 7.3: TRAVEL example using SSSP as built-in graph algorithm.

```
1  HOOK EDGE "S" (CONTEXT $e) {
2    RETURN $e@weight;
3  }
4
5  TRAVEL "CALL_BUILTIN" (VERTEX $u) GRAPH "G" {
6    -- all shortest paths from $u to all reachable vertices
7    $paths = SHORTEST_PATH($u) HOOK "S";
8    $sum = 0;
9    FOR $pth : $paths {
10     $sum = $sum + LENGTH($pth); -- get path length and add to $sum
11   }
12   -- compute average path length
13   $avgPathLength = $sum / COUNT($paths);
14 }
```

Listing 7.3 illustrates a TRAVEL script example that invokes a built-in single-source shortest path (SSSP) algorithm and post processes the result of the algorithm. In the example, we invoke the SSSP algorithm for a root vertex `$u` and compute afterwards the average path length to all reachable vertices. By exposing the built-in graph algorithm directly to TRAVEL we can customize graph algorithms—for SSSP we allow the user to specify a cost function to derive edge weights dynamically. To specify the edge hook for the SSSP algorithm, we use an edge hook, which dynamically computes the edge weight for each inspected edge and returns the value to the algorithm.

7.4 TRAVEL COMPILER

In this section we describe our code generation routine for compiling a TRAVEL script into executable code and the integration of TRAVEL into GRAPHITE. Figure 7.5 depicts the fundamental components of the TRAVEL engine and the interaction with the graph processing system backend.

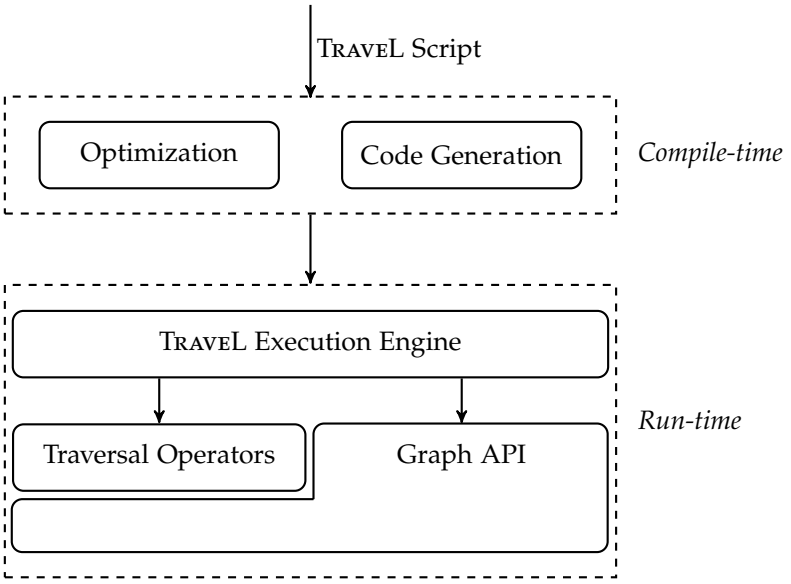


Figure 7.5: General architecture of the TRAVEL engine.

7.4.1 *General Architecture*

TRAVEL operates as a language frontend and an accompanying compiler backend and execution engine as part of a graph processing system, such as GRAPHITE, and consists of a compile-time and a run-time component. We follow the classical query optimization process as implemented in most RDBMSS, where the initial optimization pass operates on a logical representation of the query and the second pass translates the logical plan into a physical execution plan. Similarly, we translate a TRAVEL script into an abstract syntax tree and subsequently employ a two-tiered optimization process.

On the logical level, we analyze the predominant graph access patterns and perform high-level optimizations, which rewrite parts of the TRAVEL script and eliminate unreachable code fragments. In the code generation step, we use the LLVM API to generate LLVM-IR code from the optimized, high-level TRAVEL script representation. LLVM-IR is a strongly typed, portable, low-level assembler representation from the LLVM compiler framework. In the second optimization phase, we use the LLVM compiler framework to perform low-level, graph-agnostic optimizations using optimization passes available in LLVM. We describe the code generation in Section 7.4.2 and details on the applied high-level optimization techniques in Section 7.5. The TRAVEL execution engine uses the LLVM JIT compiler and translates the TRAVEL LLVM-IR representation into executable code. To bind the executable

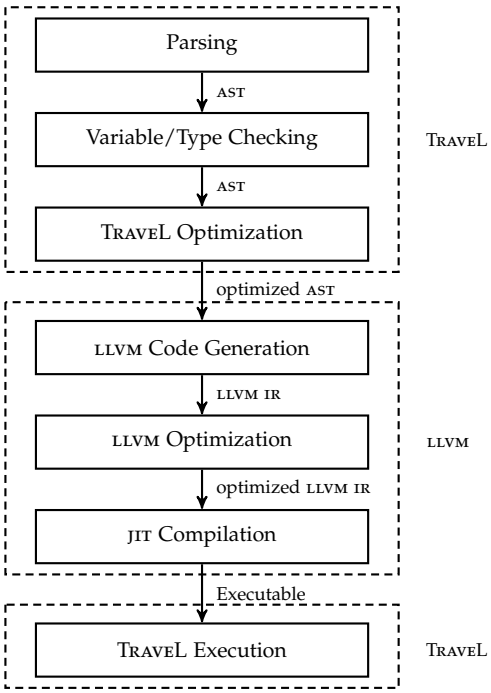


Figure 7.6: Code generation steps in TraveL.

program to the graph processing backend, we lookup and link the symbols, i.e., the function calls to the graph API and to the traversal operators.

7.4.2 *Code Generation*

We utilize the LLVM compiler framework to generate LLVM-IR code from a TraveL script and use LLVM-IR code to orchestrate the execution logic, and to delegate more complex functions, such as operators and resource management tasks, to the graph backend. Thus, we rely on a hybrid approach that seamlessly blends tuned C++ graph operator implementations with orchestration logic implemented in LLVM-IR. Generating LLVM-IR code provides several advantages over a textual C++ code generation approach: (1) LLVM provides a C++ interface to generate LLVM-IR code programmatically while retaining the ability to develop higher-level constructs, such as loops and conditional statements, in an intuitive manner. (2) LLVM can resolve code symbols, such as function declarations, from C++ code and make them available in the generated LLVM-IR code. Thus, we can expose the internal graph API written in C++ directly to the generated LLVM-IR code and thereby simplify the code generation significantly. (3) A code fragment generated in LLVM-IR can be cached for later reuse, loaded again into memory, and manipulated during runtime.

Figure 7.6 depicts the code generation steps from the input TraveL script to the compiled executable. Initially, we read and parse a TraveL script from a file or the command line and transform it into an abstract syntax tree. In a subsequent step, we perform a series of semantic verification passes, such as variable reference checking and type checking. In TraveL, all local variables have to be declared and initialized before they can be referenced. In the variable reference checking pass, we iterate over all referenced local variables and check whether they have been declared upfront. In the type checking pass, we determine the data type of each expression in the TraveL script. If the type checker identifies type-incompatible subexpressions, we throw raise an error and the abort the compilation process.

In the high-level optimization pass, we rewrite inefficient TraveL constructs using more processing-friendly TraveL language building blocks. We use static program analysis to determine data dependencies between traversal hook invocations and identify traversal

hook code fragments that could be executed independently from each other. The independence of traversal hook calls allows reordering and grouping single hook calls into bigger batches of hook invocations that can be further parallelized. Further, we analyze filter predicates that could be extracted from the traversal hook body and performed as a separate preprocessing step outside of the actual hook code. We provide a detailed description of all applied high-level optimizations in Section 7.5.

In the code generation phase, we traverse the abstract syntax tree and generate an LLVM-IR module from it. On top of this module, we run a set of LLVM optimization passes, such as *function inlining*, and finally pass the LLVM-IR module to the just-in-time execution engine of the LLVM framework for compilation and execution.

LLVM-IR Generation

We use the C++ API of the LLVM framework to programmatically generate LLVM-IR code from TraveL. In contrast to a simplistic, string-based code generation routine that appends new code fragments to a stream of characters, an API offers type safety and enforces the compiler implementer to adhere to the semantics of the language even during the code generation. In addition to the generation of LLVM-IR code, the framework also offers manipulation functions to inspect, traverse, and modify already generated LLVM modules.

Our code generation routine is based on a visitor pattern and traverses the optimized TraveL AST to emit LLVM-IR code. For each traversed object in the AST, we invoke a `CodeGen()` routine and generate the LLVM-IR code for that specific statement or expression. Listing 7.4 sketches the code routine for generating attribute access function calls to the internal graph API. We generate a function call to the internal C++ API and already calculate the corresponding physical address of the column location and pass it as a parameter to the function. In the graph API we provide specialized functions that allow retrieving single attribute values without having to cast value types and having to resolve the attribute by name in every function invocation. By that, each attribute access function is mainly a wrapper around the low-level access to the internal data structure holding the attribute. The graph API itself and the traversal operators are both written in C++ as they perform complex operations on internal data structures, which are tedious to implement natively in LLVM-IR.

Listing 7.4: Code generation routine for producing attribute access-related code.

```
1  llvm::Value* AttributeAccessExpression::codegen(CodeGenContext& context) {
2      llvm::Function* f = nullptr;
3      std::vector<llvm::Value*> args;
4      if (context.getType(m_variable) == travel::datatype::VERTEX) {
5
6          // resolve attribute and compute physical address
7          llvm::Value* pos = llvmhelper::createInt(getGraph().getAttributePos(m_attr));
8
9          // retrieve llvm variable
10         llvm::Value* vertex = nullptr;
11         auto it = context.locals.find(m_variable);
12         if (it != context.locals.end()) {
13             vertex = context.builder->CreateLoad(it->second);
14         } else {
15             throw TravelCodeGenException(...);
16         }
17
18         // add function parameters
19         args.push_back(vertex);
20         args.push_back(pos);
21
22         // determine graph api function call
23         switch (getGraph().getAttributeType(m_attr)) {
24             case INT: f = context.module->getFunction("getVertexIntAttrVal"); break;
25             ....
26             default: break;
27         }
28     } else if (context.getType(m_variable) == travel::datatype::EDGE) {
29         ...
30     } else {
31         throw TravelCodeGenException(...);
```

```
32      }
33
34      if (f) {
35          // return llvm function call
36          return context.builder->CreateCall(f, args, "");
37      } else {
38          throw TravelCodeGenException(...);
39      }
40  }
```

Listing 7.5 depicts an LLVM-IR representation of a simple vertex traversal hook accessing a vertex attribute of type double and summing up the values in a global variable `@sum`. Line 3 contains the generated code fragment that would be produced by the routine presented in Listing 7.4.

Listing 7.5: LLVM-IR representation of a simple vertex traversal hook.

```
1 ; Function Attrs: uwtable
2 define i32 @_Z13hook_vertex_Hm(i64 %v) #1 {
3   %1 = tail call double @_Z29getVertexDoubleAttributeValuemi(i64 %v, i32 2)
4   %2 = load double* @sum, align 8, !tbaa !1
5   %3 = fadd double %1, %2
6   store double %3, double* @sum, align 8, !tbaa !1
7   ret i32 0
8 }
```

Graph Data Access

TRAVEL offers built-in functionality to fetch vertices/edges and vertex/edge attribute values from the underlying graph storage (cf. Table 7.2). To access graph data, we translate each TRAVEL graph access operation into function calls that are exposed through the internal, C-style graph API. This API is part of the graph storage backend and exposes a basic set of functionality to retrieve data from the underlying graph.

We identify two potential performance issues that arise in a general-purpose graph storage with support for multiple data types and a varying set of vertex/edge attributes: (1) support for multiple data types is typically implemented either using dynamic polymorphism and virtual function calls or through function overloading, and (2) to access the actual attribute data container, the high-level attribute name has to be translated into a physical memory address for each access.

Since TRAVEL is statically typed, we can generate query-specific, type-dependent code in the code generation phase. This avoids unnecessary value conversions and attribute type resolution during runtime due to system-internal programming abstractions. In the code generation step, we determine the data type of all accessed attributes and automatically select the correct, type-dependent access function from the API. This shifts the otherwise expensive metadata lookup operation into the compilation phase of the query.

The second important consideration is to resolve, which attribute containers to access, already in the code generation phase. In particular for tight loops with loop variable dependent attribute access, having to resolve the attribute container location during each iteration again is not practical. Therefore, we determine the correct data location by resolving the attribute name and by replacing it with an internal, positional identifier. Listing 7.7 depicts a simplified code example for the generic and the code generation attribute access.

Instead of returning a generic value wrapper object as in Listing 7.7 (a), we use the code generation approach (cf. Listing 7.7 (b)) to select the correct function call and directly place them into the LLVM-IR code. Thereby, we avoid unnecessary data copying and the virtual function calls introduced by the column and attribute value abstractions.

Extending Traversal Operators

TRAVEL offers through traversal hooks a powerful concept to extend and customize built-in traversal operators. So far, we have only discussed how to generate new code and their

```
class GenericVal { ... };
class IntVal : GenericVal { ... };
class DoubleVal : GenericVal { ... };

class ColumnBase {
public:
  virtual GenericVal getAttributeVal(
          const uint64_t id) = 0;
};

GenericVal getAttributeValue(const uint64_t id,
                             const char* attr) {
  ColumnBase* c = resolveAttribute(attr);
  return c->getAttributeVal(id);
}
```

(a) Generic attribute interface.

```
class IntColumn {
public:
  int getAttributeVal(const uint64_t id);
};

IntColumn* getIntColumn(const uint32_t pos) {
    return int_cols[pos];
}

int getAttributeIntValue(const uint64_t id,
                         const uint32_t pos) {
  return getIntColumn(pos)->getAttributeVal(id);
}
```

(b) Attribute interface for code generation.

Figure 7.7: Attribute access programming interfaces.

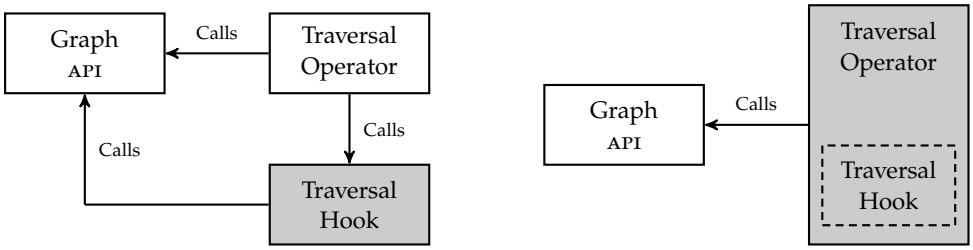
interaction with the low-level graph API. A graph traversal operator could be naturally extended by passing a function object to the traversal and calling the contained function for each event from within the operator (cf. Figure 7.8 (a)). For this approach to work, a dummy function call—potentially from an abstract base class—has to be placed in the body of the traversal. Similar to the attribute access discussion, calling the traversal hook through a virtual function call is fairly expense, in particular for large graphs and a large number of traversal hook invocations. In such a scenario, there are at least two compilation units, one for the graph storage interface and the built-in traversal operators and one for the code-generated traversal hook. The compilation unit of the graph storage is compiled when the complete system is compiled; the traversal hook can only be compiled at runtime. This fact limits the possibilities of the compiler to *inline* or otherwise optimize the interplay between the traversal operator and the traversal hook.

Instead of keeping two separate compilation units, we create for a TraveL script a customized traversal operator during runtime in a single compilation unit (cf. Figure 7.8 (b)). This has the advantage that the JIT compiler can perform a holistic code optimization by tightly integrating the traversal hook code into the traversal operator. If the traversal hook is small in size and simple in structure, the compiler can even inline the traversal hook and without having to pay the costs for an additional function call. We perform the following steps to produce such a customized traversal operator at runtime:

1. In an offline step, we statically generate an LLVM module of the traversal operator using the Clang compiler. This could be either stored in the caching infrastructure of the TraveL compiler, in a textual `*.ll`, or a binary `*.bc` file on disk. The traversal operator contains two dummy function calls, one for vertices and one for edges.
2. We generate the traversal hook as a function in a separate LLVM module as described in Section 7.4.2.
3. We merge the two LLVM modules and replace the calls to the dummy traversal hooks with the generated traversal hooks.
4. We execute several optimization passes available in LLVM, in particular the `-inline` optimization pass to embed the traversal hook code directly into the traversal operator.

7.5 TRAVEL QUERY REWRITING

In this section we exploit the benefits of the high-level abstractions offered by TraveL and propose several rewriting techniques, which leverage the program semantics introduced by the programming model (cf. Section 7.2). One of our main goals is to optimize the



(a) Traversal hook in separate compilation unit.

(b) Traversal hook compiled into traversal operator compilation unit.

Figure 7.8: Extending traversal operators in TRAVEL.

```
HOOK EDGE "H" (CONTEXT $e) {
 IF ($e@type == "knows") {
   -- perform some action
 }
}

TRAVEL "T" GRAPH "G" {
 $r = vertex:1;
 TRAVERSE BFS $r-[*]->(*) HOOK "H";
}
```

(a) Suboptimal predicate evaluation.

```
HOOK EDGE "H" (CONTEXT $e) {
 -- perform some action
}

TRAVEL "T" GRAPH "G" {
 $r = vertex:1;
 TRAVERSE BFS $r-[@type=="knows"]->(*) HOOK "H";
}
```

(b) Rewritten TRAVEL script.

Figure 7.9: TRAVEL rewriting of predicate evaluation.

execution of the traversal hooks, i.e., the code path that is executed for every discovered vertex or every traversed edge. There are two main directions to enhance the performance of traversal hook invocations: (1) reducing the number of instructions performed for each traversal hook invocation and (2) batching multiple, independent traversal hook invocations. A simple step that we perform at the beginning of the rewriting pass is to check for unreachable code fragments, such as unreferenced variables and not used traversal hooks, and eliminate them.

7.5.1 *Filter Rewriting*

TRAVEL distinguishes between *static* and *dynamic* filter conditions, i.e., a predicate that can be evaluated statically at the beginning of the query or a dynamic predicate, which depends on data gathered during the execution. An example for a static filter is to restrict the traversal to edges that fulfill a certain predicate, e.g., the edge type. In contrast, an example for a dynamic predicate could be to terminate the traversal once a certain criterion is not anymore satisfied, such as a travel budget exceeds the given limit.

The TRAVEL compiler automatically detects predicates in traversal hooks that do not depend on runtime computations and moves them out of the traversal hook function into the traversal operator. Listing 7.9 (a) depicts a TRAVEL script with a traversal hook and a static predicate in the traversal hook body. The condition is evaluated for each traversed edge and results in a branch for each traversal hook invocation. In contrast, if the predicate is not evaluated inside the traversal hook, the traversal hook itself is only invoked for matching vertices or edges. If the predicate is evaluated for every single vertex or edge, no further optimizations can be applied, i.e., by choosing between different predicate evaluation strategies.

The rewriting logic of the TRAVEL compiler moves the predicate evaluation out of the traversal hook and rewrites the traversal configuration to use an additional edge filter restricting the traversal to edges of type *knows*. Thus, we already restrict the traversal before

```
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> sum;

TRAVEL "T" GRAPH "G" {
 $r = vertex:1;
 FOR $i : $r-[*]-> {
   UPDATE $r { SET sum = sum + $i@weight; };
 }
}
```

(a) TRAVEL script without traversal hook.

```
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> sum;

HOOK VERTEX "H" (CONTEXT $v) {
 UPDATE $r { SET sum = sum + $v@weight; };
}

TRAVEL "T" GRAPH "G" {
 $r = vertex:1;
 TRAVERSE BFS $r-[*]-> HOOK "H";
}
```

(b) Rewritten TRAVEL script.

Figure 7.10: Rewriting of TraveL script logic as traversal hooks.

invoking the traversal hook and leave the decision to optimize the predicate evaluation further to the traversal operator (cf. Listing 7.9 (b)).

7.5.2 *Merging Traversal Hooks*

The user can specify multiple traversal hooks associated to a single traversal operator. By default, we consider each traversal hook independently for optimization. In some cases, i.e., when the traversal hook accesses similar attributes or exposes a similar neighborhood access pattern, we merge traversal hooks into a single traversal hook. The ordering in which traversal hooks are evaluated is specified by the user in the traversal statement. The first referenced traversal hook is invoked first, then the second and so on.

7.5.3 *Rewriting of TraveL Scripts with Traversal Hooks*

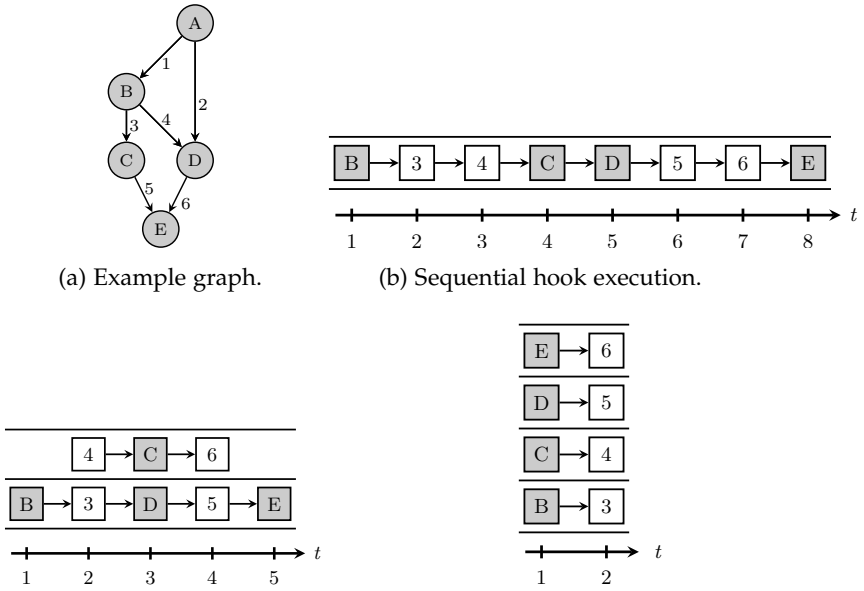
Although traversal hooks are a key component of TRAVEL and the majority of our proposed optimizations target the execution of traversal hooks, the user is not forced to write TRAVEL scripts using solely traversal hooks to express graph algorithms. Listing 7.10 (a) illustrates a TRAVEL script, which circumvents the hook-based programming model of TRAVEL.

In the program analysis phase, we search for code patterns in a TRAVEL script that mimic the intended functionality of traversal hooks. A typical pattern is the invocation of a traversal expression, which returns a set of discovered vertices, and subsequent processing on the resulting vertex multiset by iterating over it. Once we detected such a code pattern, we rewrite the TRAVEL script such we use a traversal statement and a traversal hook instead and use the code block of the for loop inside the traversal hook (cf. Listing 7.10 (b)). This, however, only works when there are now data dependencies between the original for loop body and other statements in the TRAVEL script.

The TRAVEL script shown in Listing 7.10 (a) performs a one-hop traversal, followed by a subsequent iteration over the vertices of the traversal result and computes a derived attribute value. We can rewrite the TRAVEL query by introducing a traversal hook and performing the update operation in the body of the traversal hook. By transforming an arbitrary TRAVEL script into a more canonical, hook-based version of the script, we can reuse already existing code generation techniques and avoid having to fully materialize the result of the traversal expression. Although traversal hooks are particularly well-suited for multi-hop traversals, they can also replace in their simplest form a one-hop traversal—a neighborhood query—and a subsequent postprocessing phase on the collected vertices.

7.5.4 *Traversal Hook Pipelining*

Traversal hooks are evaluated sequentially in the order of appearance of their respective traversal events, potentially causing a considerable slowdown in the total execution



(a) Example graph. (b) Sequential hook execution.

(c) Level-parallel execution. (d) Hook-parallel execution.

Figure 7.11: Variations of pipelined execution of traversal hooks from vertex B (grey boxes refer to vertex hooks, white boxes to edge hooks, respectively).

time since each traversal hook call is invoked individually. Although the strict sequential BFT/DFT ordering is a fundamental concept of the TRAVEL programming model, such a strong and rigid traversal semantics can be lessened for certain data access patterns within the traversal hooks. In the following we discuss the implications of data dependencies present within and between traversal hooks, which might force the traversal operator to retain the sequential traversal hook evaluation pattern. In the optimal case of no data dependencies between two separate traversal hook invocations, they can be executed independently from each other. Such data dependencies are analogous to dependencies found in modern CPU pipelines causing *data hazards*. We refer to the independent execution of traversal hook invocations as *traversal hook pipelining*, as it is similar to the pipelined execution of instructions in modern processors. Similarly, data dependencies can result in data hazards, i.e. blocking the pipeline caused by an unresolved data dependency. Ignoring data dependencies can result in a changed execution order and unexpected results compared to a sequential execution.

Figure 7.11 depicts an example graph and three possible traversal hook invocation orders. For a BFT starting at vertex B, Figure 7.11 (b) depicts the strictly sequential execution of the traversal hook. A sequential processing of the traversal hooks maintains the semantics of a strict traversal ordering, but does not provide any possibility to invoke traversal hooks independently from each other. Figure 7.11 (c) depicts *level-parallel* hook pipelining, where vertices and edges that are discovered on the same level are processed independently from each other. Finally, Figure 7.11 (d) depicts the optimal case, where all traversal hooks can be invoked independently from each other.

Data dependencies can occur between consecutive traversal hook invocations on shared traversal state objects, such as temporary attributes and global variables. Our aim is to schedule traversal hook invocations such that traversal hooks without data dependencies can be invoked independently from each other without modifying the sequential semantics of the TRAVEL script. We distinguish four major operations that access shared traversal state:

- **Local Reads:** A local read operation only accesses the traversal state of the object—the vertex or the edge—for which the traversal hook is invoked. This includes access to temporary attributes and map entries for the corresponding object, but not to access traversal state of any other vertex or edge.

Table 7.3: Data dependencies between invocations of traversal hooks in TRAVEL.

Traversal Hook	*Line*	*Local Read*	*Local Write*	*Non-Local Read*	*Non-Local Write*

`1 HOOK VERTEX "V" (CONTEXT $v) {`
`2  $sum = $sum + 1;`
`3  UPDATE $v { SET s = $sum; }`
`4 }`
	2	-	-	✓	✓

	3	-	✓	✓	-

`1 HOOK VERTEX "V" (CONTEXT $v) {`
`2  FOR $i : $v-[*]-> {`
`3  UPDATE $v { SET s = s + $i@s; }}`
`4 }`
	3	-	✓	✓	-

`1 HOOK VERTEX "V" (CONTEXT $v) {`
`2  UPDATE $v { SET s = LEVEL(); }`
`3 }`
	2	-	✓	✓	-

`1 HOOK VERTEX "V" (CONTEXT $v) {`
`2  UPDATE $v { SET s = 0; }`
`3 }`
	2	✓	✓	-	-

- **Local Writes:** A local write operation only allows the modification of the traversal state of the object—the vertex or the edge—for which the traversal hook is invoked.
- **Non-Local Reads:** A non-local read operation accesses the traversal state of any other vertex or edge in the graph that is not the context object of the respective traversal hook invocation.
- **Non-Local Writes:** A non-local write operation modifies the traversal state of any other vertex or edge in the graph that is not the context object of the respective traversal hook invocation.

In Table 7.3 we illustrate the different types of data dependencies that can occur between traversal hook invocations. We use a simple TRAVEL script blueprint, as depicted in Listing 7.6, with a BFT traversal starting at vertex `$root` and an accompanying dummy traversal hook `H` to describe the different types of data dependencies.

Listing 7.6: TRAVEL query pattern with dummy traversal hook.

```
INT sum = 0;
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> s = 0;

-- dummy traversal hook "H"
HOOK VERTEX "V" (CONTEXT $v) { }

TRAVEL "T" (VERTEX $root) GRAPH "G" {
 TRAVERSE BFS $root-[*]->(*) HOOK "V";
}
```

The first example reads the state of the global variable `$sum`, increments it by 1, and writes the new result back. Since `$sum` is a global variable, we identify the read/write access pattern as a *non-local read* followed by a *non-local write* operation. Similarly, in Line 3 we read the state of the global variable `$sum` and assign its value to the attribute `s` of the context vertex `$v`. In the second example, we read the value of attribute `s` for all neighbors of `$v`—we access not only the local context vertex and its state, but also the state of the adjacent vertices. The third example writes the `LEVEL()` information into the local traversal state, but

accesses global state, i.e., information about the current traversal level. The fourth example exposes no data dependencies as the read and the write operation are local to the invoked traversal hook.

7.6 APPLICATIONS

In this section we showcase the expressiveness of TRAVEL on a set of realistic graph algorithms that can be naturally implemented using the hook-based programming model.

7.6.1 *K-Hop Reachability*

A *k-hop reachability* query is a reachability query with an additional constraint and receives a numerical parameter *numHops* in addition to the two vertices u and v to test for reachability. The algorithm returns *true* if there is a simple path from vertex u to vertex v of length *numHops*, *false* otherwise.

Listing 7.7: K-hop reachability implementation in TRAVEL.

```
1  BOOL reachable = false;
2
3  HOOK VERTEX "H" (CONTEXT $v) {
4    IF (LEVEL() > $numHops) {
5      RESTRICT ALL true;
6    }
7    IF ($v == $t) {
8      IF (LEVEL() == $numHops) {
9        $reachable = true;
10     }
11     RESTRICT ALL true;
12   }
13 }
14
15 TRAVEL "IS_K_HOP_REACHABLE" (VERTEX $s, VERTEX $t, INT $numHops) GRAPH "G" {
16   TRAVERSE BFS $s-[*]->(*) HOOK "H";
17   RETURN $reachable;
18 }
```

Listing 7.15 sketches the implementation in TRAVEL, which receives three input parameters, namely the two vertices `$s` and `$t` and the maximum path length `$numHops`. Initially, we perform a BFT starting from vertex `$s`, extended by a vertex hook `"H"` (Line 16). The intuition is to traverse the graph starting from the source vertex and to check for each discovered vertex, whether it is the specified target vertex and whether the maximum path length condition is satisfied (Lines 7-8). If both conditions are fulfilled, it sets the global variable `$reachable` to *true*. We terminate the traversal (Line 11) if either both conditions are satisfied or the traversed level exceeds the maximum path length (Lines 4-6). Finally, we return the global variable `$reachable`.

7.6.2 *Collaborative Filtering*

Recommendation engines play an important role in e-commerce systems and online dating websites. Based on the preferences of each user, a recommendation engine makes suggestions to the user based on similar preferences by other users. Figure 7.12 depicts an example graph from an artificial online dating website, which represents users as vertices and relationships between users as *Likes*. We want to make recommendations for a single user based on his own likes and similar likes of other users. For each query, we return a ranked list of potential matches and exclude those that have been already liked by the user. For example, for user M, which already liked other users C_1, C_2, and C_3, the algorithm recommends users C_5, C_4, and C_6 (in decreasing order of relevance).

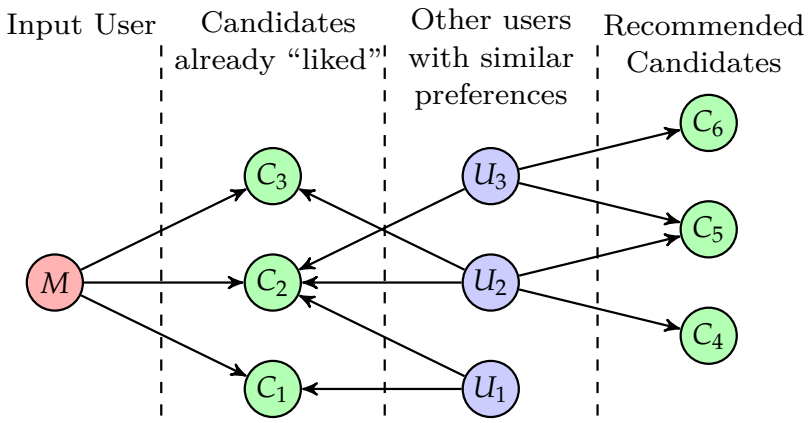


Figure 7.12: Collaborative filtering recommendation using similar user preferences.

Listing 7.8 depicts an implementation of a simple collaborative filtering algorithm in TraveL. The algorithm receives as input a user (`$user`) and returns a ranked set of user recommendations. We use a temporary vertex attribute `popularity` to compute a relevance score for each candidate match. In Line 11 we determine the set of already liked users and store the result in a local variable `$liked`. From the set of already liked users, we perform a backward traversal on incoming edges of type *likes* and perform vertex-based hook invocations for a traversal hook "REC" (Line 12). For each user with a similar preference, we invoke the traversal hook, iterate over the user's likes and update the relevance score for each candidate user (Lines 4-7). In Line 13 we compute the set of candidates by removing the set of already liked users from the set of potential candidates. In the final return statement (Line 14), we sort the candidate matches by the computed relevance score in decreasing order and return the top 10 matches.

Listing 7.8: Implementation of collaborative filtering in TraveL.

```
1  CREATE TEMPORARY VERTEX ATTRIBUTE<INT> popularity = 0;
2
3  HOOK VERTEX "REC" (CONTEXT $user) {
4    $candidates = $user-[@type="likes"]->;
5    FOR $i : $candidates {
6      UPDATE $i { SET popularity = popularity + 1; };
7    }
8  }
9
10 TRAVEL "RECOMMENDATION" (VERTEX $user) GRAPH "G" {
11   $liked = $user-[@type="likes"]->;
12   TRAVERSE BFS $liked<-[@type="likes"]- HOOK "REC";
13   $recommendations = $VERTICES MINUS $liked;
14   RETURN $recommendations ORDER DESC BY popularity TOP 10;
15 }
```

7.6.3 *Weighted Shortest Path*

The single-source shortest path (SSSP) algorithm computes for an input vertex the shortest paths to all other reachable vertices in a weighted graph. We generalize the well-known SSSP algorithm by Dijkstra (Dijkstra, 1959) to arbitrary cost functions that result in non-negative edge weights. The central data structure of Dijkstra's algorithm is a min-priority queue with efficient support for a `decrease_priority()` operation. In practice this is usually accomplished by using Fibonacci heaps (Cormen et al., 2001).

Although TraveL does not offer a built-in Fibonacci heap, we can mimic the same behavior through temporary attributes. To implement a Fibonacci heap in TraveL, we require

three operations: `add_with_priority()`, `extract_min()`, and `decrease_priority()`. In the algorithm preparation phase, we initialize the Fibonacci heap with all vertices and an infinite priority value. The same behavior can be implemented in TraveL by setting the default initialization value of the temporary attribute to `INT_MAX`. In the main loop of Dijkstra's algorithm, we iterate over all vertices in the Fibonacci heap and extract in each iteration the vertex with the lowest priority value. We achieve the same semantics by using the `TRAVERSE` statement of TraveL and treating the Fibonacci heap as a multiset of values to iterate on. For every iteration, we return and remove the vertex with the lowest priority (attribute value) and pass it as context object to the assigned traversal hook. In fact, this generalizes the traversal hook concept to iterations over arbitrary collections, where the iterator defined on the collection defines the order in which objects of the collection are discovered.

Listing 7.9: Dijkstra algorithm implementation in TraveL.

```
1  CREATE TEMPORARY VERTEX ATTRIBUTE<INT> distances = INT_MAX;
2  CREATE TEMPORARY VERTEX ATTRIBUTE<INT> queue = INT_MAX;
3
4  HOOK VERTEX "V" (CONTEXT $v) {
5    $e = EDGE($u, $v);
6    $alt = $u@distances + $e@weight;
7    IF ($alt < $v@distances) {
8      UPDATE $v { SET distances = $alt;
9                  SET queue = $alt; -- can only apply decrease_prio
10               }
11   }
12 }
13
14 -- get min
15 HOOK VERTEX "Q" (CONTEXT $u) {
16   -- extract min
17   UPDATE $u { SET queue = UNDEFINED; }
18   TRAVERSE BFS $u-[*]-> HOOK "V";
19 }
20
21 TRAVEL "DIJKSTRA" (VERTEX $source) GRAPH "G" {
22   UPDATE $source { SET distances = 0;
23                    SET queue = 0; };
24   TRAVERSE MIN_QUEUE queue HOOK "Q";
25   RETURN $distances;
26 }
```

Listing 7.9 illustrates the implementation of Dijkstra's algorithm in TraveL. The script receives a root vertex from which to compute the shortest path to all other reachable vertices and returns a distance map of all reachable vertices and their corresponding minimal distance to the root vertex. We initialize two temporary vertex attributes, one to store computed minimal distances (`distances`) and one to mimic the Fibonacci heap (`queue`). For the root vertex, we initialize both attributes to zero. In Line 24 we issue the `TRAVERSE` statement stating that the collection we are iterating over is a `MIN_QUEUE`. This gives the TraveL compiler the necessary information to use a queue data structure for the execution. When the traversal hook `Q` is invoked, its hook context vertex contains the vertex with the minimal priority from the queue. To remove the element permanently from the queue, we set the corresponding attribute value in the queue to `UNDEFINED`. In Line 18 we perform a BFT traversal starting from the extracted vertex. For each discovered neighbor, we invoke the traversal hook `V`, which computes the new distances (Line 6) and updates the distance map and decreases the priority of the context vertex in the queue (Lines 8-10). The algorithm terminates when all vertices from the queue have been processed, i.e., the attribute value for each vertex is set to `UNDEFINED`.

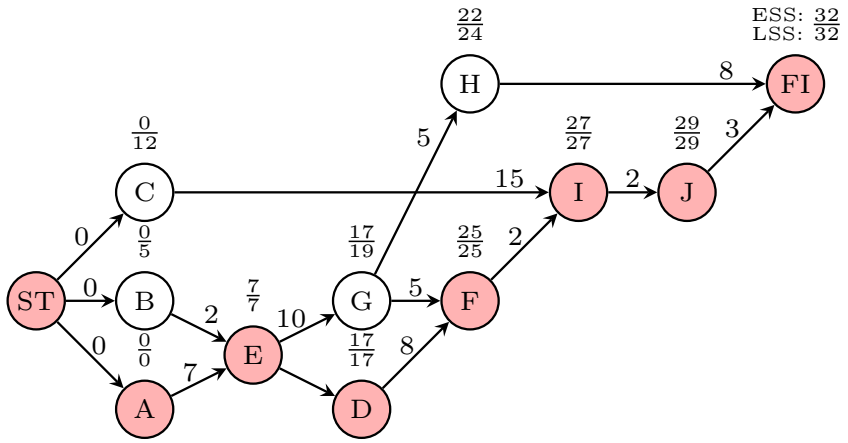


Figure 7.13: Activity graph with edge weights as activity durations, ST as initial activity, and FI as final activity. The critical path is colored in red.

7.6.4 *Critical Path Analysis*

The scheduling of activities in complex projects can be formulated as a graph problem and is usually referred to as *Critical Path Analysis*. The task is to find all critical paths in the graph, i.e., the paths that define the total project duration. A project schedule with activities and their relationships (dependencies) can be modeled as an *Activity Graph*, where each activity is modeled as a vertex and each dependency between two activities A and B is modeled as an edge. Each edge has an assigned attribute `weight` describing the duration it takes to complete activity A, before activity B can be started.

Figure 7.13 depicts an example with activities A, . . . , FI and an initial activity ST. The start activity is an artificial activity, which is used to form a single root activity (in the case of multiple initial activities). To compute the critical path, we first determine the *earliest start schedule* (ESS) for each activity using forward calculations of the *earliest finish schedule* of its predecessors. We compute the earliest finish schedule as the sum of the earliest start schedule and the duration. If there is more than one predecessor, the new earliest start schedule is the maximum of the earliest finish schedules of its predecessors. For example, the earliest start schedule for activity E is 7. The second step computes the *latest start schedule* (LSS) of each activity using backward calculations starting from the final activity FI. We compute the latest start schedule as the minimum of all its predecessor activities.

Listing 7.10: Critical path implementation in TraveL.

```
1  CREATE TEMPORARY VERTEX ATTRIBUTE<INT> earliest = 0;
2  CREATE TEMPORARY VERTEX ATTRIBUTE<INT> latest = 0;
3  LIST<VERTEX> output;
4
5  HOOK VERTEX "FORWARD" (CONTEXT $v) {
6    $predecessors = $v<-[*]-;
7    $max = 0;
8    FOR $i : $predecessors {
9      $e = EDGE($i,$v);
10     IF (($e@duration + $i@earliest) > $max) {
11         $max = $e@duration + $i@earliest;
12     }
13   }
14   UPDATE $v { SET earliest = $max; }
15 }
16
17 HOOK VERTEX "BACKWARD" (CONTEXT $v) {
18   $successors = $v-[*]->;
19   $min = INT_MAX;
20   FOR $i : $successors {
```

```
21      $e = EDGE($i,$v);
22      IF (($i@latest - $e@duration) < $min) {
23        $min = $i@latest - $e@duration;
24      }
25    }
26    UPDATE $v { SET latest = $min; }
27  }
28
29  HOOK VERTEX "CRIT_PATH" (CONTEXT $v) {
30    IF ($v@earliest == $v@latest) {
31      APPEND($output,$v);
32    }
33  }
34
35  TRAVEL "CRIT" (VERTEX $root, VERTEX $tail) GRAPH "G" {
36    TRAVERSE BFS $root-[*]->(*) HOOK "FORWARD";
37    TRAVERSE BFS $tail-[*]->(*) HOOK "BACKWARD";
38    TRAVERSE BFS $root-[*]->(*) HOOK "CRIT_PATH";
39    RETURN $output;
40  }
```

The latest start schedule of an activity is defined as the latest finish schedule decreased by the duration. For example, the latest start schedule of activity G is 19. The final step traverses the graph and collects all activities that have equal earliest/latest start schedules. Each activity that has identical earliest/latest start schedules is defined to be on a critical path. For example, the graph in Figure 7.13 has a critical path S $\rightsquigarrow$ A $\rightsquigarrow$ E $\rightsquigarrow$ D $\rightsquigarrow$ F $\rightsquigarrow$ I $\rightsquigarrow$ J $\rightsquigarrow$ FI (colored in red).

Listing 7.10 shows an implementation of the critical-path algorithm in TraveL. For the sake of simplicity, we assume that there can be only a single critical path. We collect the vertices forming the critical path in a list structure `myRes` and use two temporary vertex attributes `earliest` and `latest` to collect the ESS and LSS, respectively. We use three different traversal hooks to implement the three steps of the critical path analysis algorithm. The first step computes the earliest start schedules for each activity and stores the results in the temporary vertex attribute `earliest` (Lines 5–15). The second step computes the latest start schedules for each activity and stores the result in the temporary vertex attribute `latest` (Lines 17–27). The final step extracts all activities, where its earliest start schedule and latest start schedule are the same, into the final output structure `myRes` (Lines 29–33). The `"CRIT"` TraveL clause contains the setup code and triggers the execution of the three traversals, one for each step and with a different traversal hook registered (Lines 35–40).

7.7 EXPERIMENTAL EVALUATION

We implemented TraveL in Graphite as depicted in our architecture overview in Figure 7.5. To construct an executable from a TraveL script, we use Flex and Bison to implement the parser and the LLVM C++ API to generate code from the resulting AST. To compile and link the BFT operator at runtime, we use a pre-generated binary LLVM-IR representation of the operator that we load at system startup. We conducted all experiments on an Intel® Xeon® E5-2660 machine with 2 sockets, 10 cores per socket, 2 threads per core, each core running at 2.6 GHz. The machine runs on SLES 12 SP1 and is equipped with 128 GB of DDR4 RAM and 25 MB last level cache. For the experiments, we compiled Graphite using CLANG 3.8 with option `-O3`.

We study the effectiveness of the proposed TraveL optimizations and rewriting techniques using several micro benchmarks. These micro benchmarks are TraveL programs, but are minimal in scope such that we can evaluate the effect of each applied technique individually. Additionally, we select two realistic graph scenarios that we implemented in TraveL and compare the end-to-end performance of the generated executable using the TraveL compiler against a hand-written implementation on the generic graph API. All reported numbers are median execution times over ten runs using a single-threaded execution.

7.7.1 *Micro Benchmarks*

In this section we evaluate the effectiveness of the proposed TRAVEL optimizations and rewriting techniques on the LIVEJOURNAL data set. Specifically, we evaluate two optimizations that avoid the overhead introduced by a generic programming interface, i.e., the overhead of virtual function calls and the overhead of attribute access calls and subsequent physical storage location resolutions. Moreover, we evaluate the impact of the proposed rewriting techniques, specifically the extraction of filter-related coding from a traversal hook and the rewriting of single traversal hook invocations to batched invocations.

Overhead of Virtual Function Calls

The first experiment investigates the impact of virtual function calls on the overall execution performance of TRAVEL. We use a simple TRAVEL program as depicted in Listing 7.11 that performs a BFT starting at some vertex v and accumulates the attribute value of all discovered vertices. If we would straightforwardly generate code against the generic graph API, the resulting program would incur two virtual function calls per traversal hook invocation, one for calling the hook function and one for accessing the corresponding entry in the column group to fetch the value.

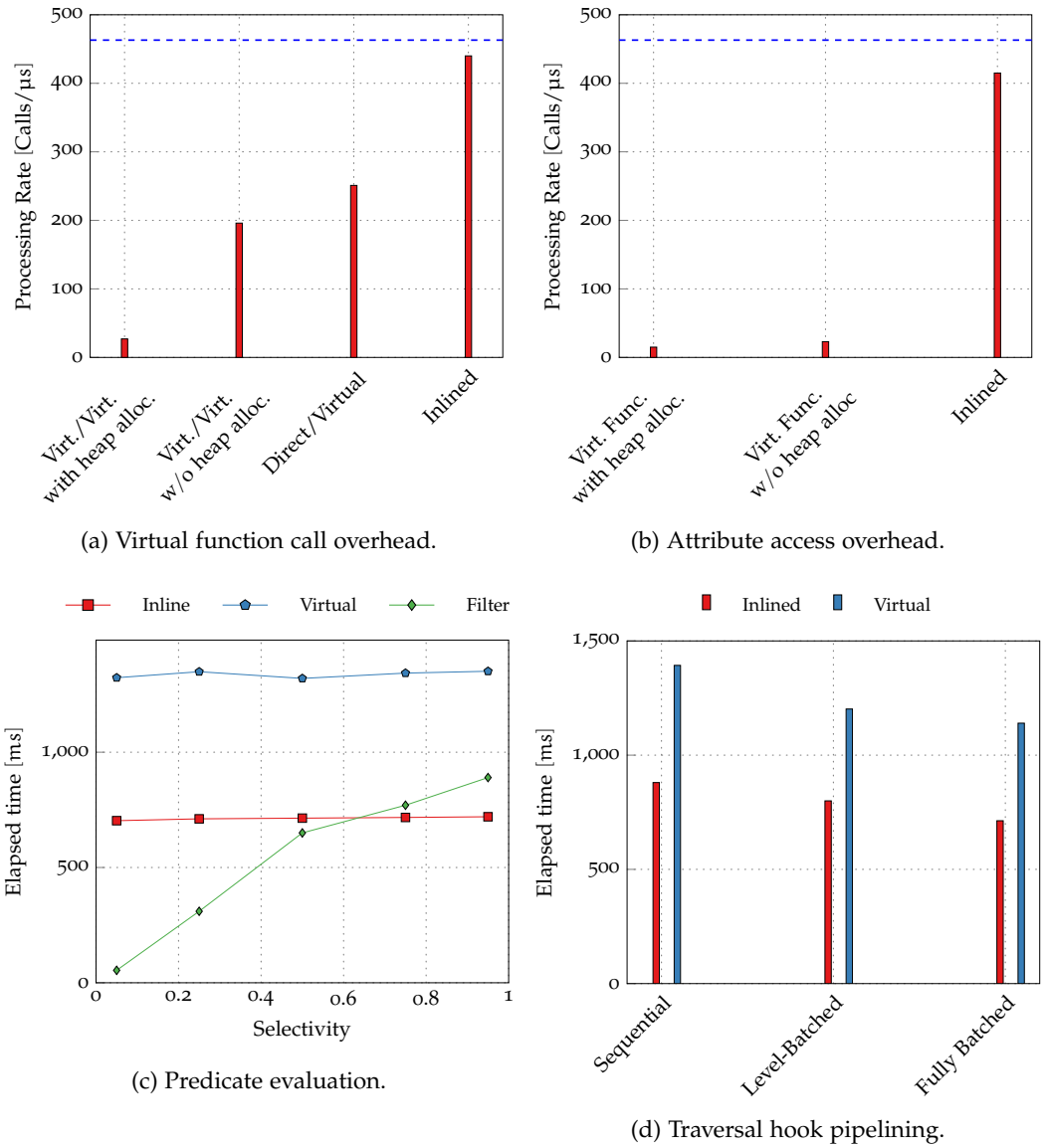
Listing 7.11: Test program to evaluate the overhead of virtual function calls.

```
1 INT sum = 0;
2
3 HOOK VERTEX "SUM" (CONTEXT $v) {
4   $sum += $v@attribute;
5 }
6
7 TRAVEL "MICRO_TEST1" (VERTEX $v) GRAPH "G" {
8   TRAVERSE BFS $v-[*]->(*) HOOK "SUM";
9 }
```

We compare the following three configurations of the TRAVEL compiler: (1) code generation using virtual function call elimination for the traversal hook call and the attribute access, (2) code generation against the generic graph API resulting in two virtual function calls per hook invocation, and (3) code generation that removes the virtual function call to the traversal hook, but retains the virtual function call to access the attribute.

We omit the physical attribute storage resolution and assume that the corresponding physical address of the attribute in memory is already known. In the generic implementation—besides the overhead of the virtual function call—, we also investigate the impact of small memory allocations that could be performed to create temporary objects during the operation. In Figure 7.14 (a) we report the number of processed vertices per µs for all evaluated implementations. Since there are no data dependencies between consecutive traversal hook invocations, we apply the traversal hook in a batch operation on the set of discovered vertices after the actual traversal. The dashed blue line depicts the highest possible processing rate, i.e., an iteration over all discovered vertices without performing any work. To avoid undesired compiler optimizations, we added the `asm("nop")` instruction to the loop body.

The generic code generation routine, which does not remove virtual function calls and performs a heap allocation for a small object during each invocation, reaches a processing rate of about 27 vertices/µs, which is about 16× slower than the fully inlined code emitted by the TRAVEL compiler. We note that although virtual function calls alone cause a performance penalty of factor 2, the majority of the overhead reported here is caused by additional memory allocations performed during the traversal hook invocations. If we remove the additional memory allocations, we reach a processing rate of 196 vertices/µs, which is already 7× faster than the generic implementation. For the fully inlined code, we remove both virtual function calls, any memory allocations, and access the attribute array directly using a positional access. We reach a processing rate of 440 vertices/µs, which is close to the highest possible processing rate. Since the number of traversal hook invoca-



(a) Virtual function call overhead.

(b) Attribute access overhead.

(c) Predicate evaluation.

(d) Traversal hook pipelining.

Figure 7.14: Evaluated micro benchmarks to quantify the effectiveness of the rewriting techniques implemented in the TRAVEL compiler.

tions is quite high for long traversals—most queries that we ran against the LIVEJOURNAL data set discovered about 95% of all vertices on average, i.e., more than 4 M vertices—the critical code path is sensitive to unnecessary overhead that is incurred by function call indirections, virtual function calls, and memory allocations.

Attribute Access

In this experiment we evaluate the overhead for accessing attribute values for a given vertex or edge introduced by a generic programming interface. Specifically, we evaluate the overhead of virtual function calls—as already discussed in the previous experiment—and the repeated resolution of attribute names to their respective physical storage location. We compare the code emitted by the TRAVEL compiler, which already resolves attribute names to their storage locations and removes unnecessary virtual function calls from the critical code path, against a naive implementation on the generic graph API. As minimal example, we use a TRAVEL program (cf. Listing 7.12), which iterates over all vertices in the graph and accumulates the attribute values of a numerical vertex attribute.

Listing 7.12: Test program to evaluate the overhead of attribute location resolutions.

```
INT sum = 0;

TRAVEL "ATTRIBUTE_ACCESS" GRAPH "G" {
  FOR $i : $VERTICES {
    $sum += $v@attribute;
  }
}
```

In the generic graph API, the function interface to retrieve an attribute for a given vertex/edge receives two input parameters, i.e., the vertex/edge and the name of the attribute. In the graph storage backend, each invocation of the function requires to resolve the attribute name to the actual physical memory address, i.e., the location of the data vector in the corresponding column group. In GRAPHITE this resolution is implemented as an unordered map, which translates attribute names into pointers to the corresponding column in the column group. Performing the attribute resolution for each invocation incurs a considerable performance overhead as we shown in Figure 7.14 (b). We measure the processing rate of attribute access operations per µs and compare two configurations, namely using virtual function calls and attribute resolution for each invocation against a fully inlined program emitted by the TRAVEL compiler. The plot trend is similar as in first experiment, but adds an additional overhead of the attribute resolution while only performing a single virtual function call per invocation. The configuration without additional memory allocation is slightly faster than the one with memory allocation, but is still about 30× slower than the inlined direct access to the attribute location.

Conditional Predicates as Traversal Edge Filters

In this experiment we evaluate the rewriting technique of transforming a conditional predicate based on some attribute attached to a vertex/edge context object into a traversal statement restricted by a filter expression. Listing 7.13 depicts an example, which performs a traversal and evaluates for each traversed vertex a condition on the respective vertex attribute and conditionally accumulates an aggregate on the attribute. Since the predicate is evaluated on each discovered vertex in the graph, we can rewrite the program such that only vertices that fulfill the predicate condition are traversed. This reduces the size of intermediate results during the traversal and subsequently reduces the number of traversal hook invocations depending on the selectivity of the predicate.

Listing 7.13: Test program to evaluate the rewriting of conditional predicate evaluation as traversal edge filter.

```
INT sum = 0;

HOOK VERTEX "H" (CONTEXT $v) {
 IF ($v@weight > 10) {
   sum = sum + $e@weight;
 }
}

TRAVEL "T" GRAPH "G" {
 $r = vertex:1;
 TRAVERSE BFS $r-[*]->(*) HOOK "H";
}
```

We compare three different code generation configurations and present the results in Figure 7.14 (c). We use two code generation versions that evaluate the predicate upon each invocation of the traversal hook. The first one performs a naive evaluation of the predicate by performing the already discussed virtual function call to the attribute access function and the attribute name resolution (—●—). The second alternative eliminates the overhead of the virtual function calls and attribute name resolution (—■—). In the third configuration, we rewrite the traversal hook and remove the predicate evaluation from the traversal hook

and instead parameterize the traversal such that the predicate is evaluated in the traversal operator (—◆—).

The execution time of the TraveL programs, which perform the predicate evaluation as part of the traversal hook, is independent of the selectivity of the predicate and takes the same time to complete for different predicate selectivities. In general, the inlined approach is about a factor 2 faster than the program using virtual function calls and attribute access resolution. If we compare the plot of the configuration using the filter-based approach, we can see that for a high selectivity, the TraveL program significantly outperforms the other two versions by up to 25×. In contrast to the hook-based predicate evaluation, which traverses the complete graph and afterwards invokes the traversal hook for each discovered vertex, the filter-based approach eliminates vertices that do not fulfill the predicate condition early during the traversal. This effectively reduces the size of intermediate results during the traversal and also limits the number of traversal hook invocations to vertices that satisfy the predicate condition.

Traversal Hook Batching

In this experiment we evaluate the effect of executing traversal hooks in batches. We use the same TraveL program as in our first micro benchmark experiment (cf. Listing 7.11) and compare three methods for executing traversal hooks. Since there are no data dependencies between consecutive traversal hook invocations, we evaluate the following three processing modes: (1) sequential execution, each traversal hook is invoked independently, (2) level-synchronous, batched execution, the traversal hooks for all vertices discovered at the same level are invoked in a batch, and (3) fully batched execution, i.e., the traversal hooks of all discovered vertices are evaluated in a single batch.

Figure 7.14 (d) depicts the results of our evaluation for the single, sequential execution, the level-synchronous, batched execution, and the fully batched execution of traversal hooks. We measure the complete execution time, i.e., the elapsed time of the traversal operator and of the traversal hook invocations. As a reference, we also include measurements against the generic graph API with additional virtual function calls and attribute name resolutions. From the results, we conclude that fully batching the traversal hook invocations is beneficial as it presumably increases locality in the instruction cache and also increases in our example spatial and temporal locality for the access to the attribute values. In contrast, the sequential execution is performed tightly interleaved with the traversal operator and invokes the traversal hook when a new vertex has been discovered. Further, for a parallelized execution of traversal hook invocations, a larger batch size is also beneficial as it minimizes the scheduling overhead to spawn new tasks in the engine. The conclusion from this experiments is that whenever possible, traversal hook batching should be applied. Which batching strategy can be applied depends on the detected data dependencies in the traversal hooks, as discussed in Section 7.5.4.

7.7.2 *System-Level Benchmarks*

In this experiment we evaluate the end-to-end performance of TraveL for two realistic graph scenarios and compare it against an equivalent hand-written implementation in C++ on the generic graph API. We use summarized BOM explosion (cf. Listing 7.14) and k-hop reachability (cf. Listing 7.15) as representative graph algorithms. We chose summarized BOM explosion because it exhibits rich data dependencies between traversal hook invocations and requires excessive access to temporary and persistent attributes. To complement this, we selected k-hop reachability as a simple but powerful routine with many applications. In contrast to the summarized BOM explosion, k-hop reachability does not exhibit data dependencies between traversal hooks but steers the traversal during runtime by terminating the execution once either the target vertex has been found or the hop condition cannot be met anymore.

Listing 7.14: Summarized BOM explosion implementation in TraveL.

```
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> sum = 0;

HOOK EDGE "H" (CONTEXT $e) {
  $h = HEAD($e);
  $t = TAIL($e);
  UPDATE $t { SET sum = $t@sum + ($e@quantity * $h@sum); }
}

TRAVEL "BOM" (VERTEX $root) GRAPH "G" {
  UPDATE $root { SET sum = 1; }
  TRAVERSE BFS $root-->(*) HOOK "H";
}
```

Listing 7.15: K-hop reachability implementation in TraveL.

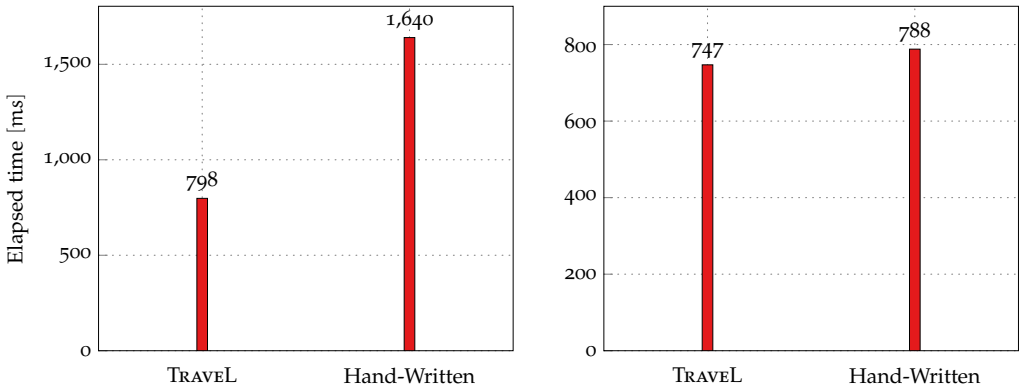
```
BOOL reachable = false;

HOOK VERTEX "H" (CONTEXT $v) {
  IF (LEVEL() > $numHops) {
    RESTRICT ALL true;
  }
  IF ($v == $t) {
    IF (LEVEL() == $numHops) {
      $reachable = true;
    }
    RESTRICT ALL true;
  }
}

TRAVEL "IS_K_HOP_REACHABLE" (VERTEX $s, VERTEX $t, INT $numHops) GRAPH "G" {
  TRAVERSE BFS $s-[*]->(*) HOOK "H";
  RETURN $reachable;
}
```

We evaluated both algorithms on the LiveJournal graph and generated for the summarized BOM explosion an additional edge attribute to reflect the quantities that are associated to edges in the part graph. We present our results for both algorithms in Figure 7.15. For the BOM explosion algorithm, TraveL outperforms the hand-written code against the generic graph API by 2×. The speedup can be explained by the large number of attribute accesses during the BOM explosion. To update the temporary quantity value of a vertex, the algorithm has to access the old value, the respective edge quantity, and the already aggregated quantity of the source vertex. This results in six virtual function calls and three attribute resolutions per traversal hook invocation. Although we already reported earlier that the overhead of the attribute access can be even higher, we note that the traversal operator itself already takes about 730 ms. The remaining time is spent for evaluating the traversal hook code. For the K-hop reachability algorithm, the TraveL compiler cannot gain much performance compared to the hand-written code. This is caused by the fact that the algorithm does not access any attributes and performs only simple comparisons in the traversal hook code. For our implementation, we only see a marginal speedup of about 40 ms, which is caused by the elimination of the virtual function calls to the traversal hook.

To summarize the experimental evaluation, we conclude that the TraveL compiler shows great potential to eliminate expensive interface-caused constructs in the algorithms by generating efficient code for the critical path of the computation, i.e., the traversal hooks, by eliminating virtual function calls and repeated physical attribute location resolutions. For simple algorithms that do not rely on constructs that can cause a severe performance



(a) Summarized BOM explosion.

(b) K-hop reachability.

Figure 7.15: Comparison of optimized code emitted by the TraveL compiler and hand-written C++ code on a generic graph API for summarized BOM explosion and k-hops reachability.

degradation, however, the TraveL compiler emits code with a comparable performance to hand-written code on a generic graph API.

7.8 SUMMARY

In this chapter, we introduced TraveL—an imperative, domain-specific query language for graph analysis—and *traversal hooks*, a simple yet powerful programming model to extend graph traversals with custom logic. TraveL aims at bridging the gap between highly tuned graph algorithms and the implementation of custom graph algorithms against a generic low-level programming interface. Besides vertices and edges, TraveL supports simple paths as built-in data type and seamlessly embeds other built-in graph algorithms, such as shortest-path, and can handle graph-shaped result types. While being a compiled, domain-specific language, TraveL provides rich opportunities for query optimization, such as avoiding expensive virtual function calls, eliminating unnecessary attribute location lookups, and identifying and batching independent traversal hook invocations. TraveL can be tightly integrated into an RDBMS as a stored procedure language while still being competitive in terms of execution performance to hand-crafted graph algorithms.

8 CONCLUSION

8.1 CONTRIBUTIONS

Increasingly, companies face the challenge of storing, manipulating, and querying terabytes of graph data for enterprise-critical applications. Existing solutions performing graph operations on business-critical data use a combination of SQL and application logic or employ a graph management system (GMS). Since the majority of these systems exclusively run on relational DBMSs, employing a specialized system for storing and processing graph data is typically not sensible. As of today, RDBMS do not provide the necessary query performance for traversal-based graph algorithms nor does a RDBMS offer an intuitive graph data model or programming abstraction.

To tackle these challenges, we developed GRAPHITE, a hybrid graph-relational data management system, and discussed the major components during the course of this thesis. GRAPHITE is a performance-oriented graph data management system at the core of an RDBMS allowing to seamlessly combine graph data with relational data in the same system. Our experimental evaluation shows that GRAPHITE can outperform native GMS by multiple orders of magnitude while providing all the features of an RDBMS, such as transactional support, backup and recovery, security and user management. For customers, this offers an interesting alternative to specialized GMS that lack many of these features and require expensive data replication and maintenance processes. In the following we summarize our main contributions and depict an overview of the most important elements in Figure 8.1.

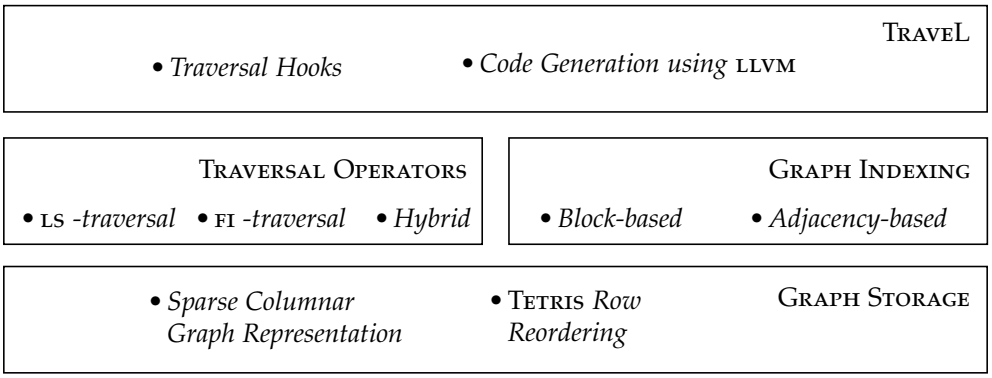


Figure 8.1: Main contributions of the thesis organized by architecture component in GRAPHITE.

COLUMNAR GRAPH STORAGE: We proposed a relational storage representation for graph data based on *column groups* to leverage the already existing and mature data management and query processing infrastructure of RDBMSs. GRAPHITE represents a graph in two wide column groups, one for vertices and one for edges. On the physical storage layer, we separate each column group into a read-optimized partition and a write-optimized partition following a similar approach as many main-memory columnar RDBMS, such as SAP HANA and VERTICA. Since the representation of vertices and edges in wide column groups might lead to sparsely populated columns, effectively resulting in a higher memory consumption, we developed a light-weight compression technique called TETRIS. TETRIS identifies entities that expose a similar set of attributes automatically, reorders them within a column group, and finally applies RLE to compress `NULL` values in each column. In contrast to other column-by-column compression techniques, TETRIS considers the complete row, including all exposed attributes, as a single entity, and achieves a better compression ratio by up

to 60%. On top of the column group representation, we implemented a light-weight, set-oriented API, which directly leverages the columnar data organization.

GRAPH TRAVERSAL OPERATORS: For the query processing layer of GRAPHITE, we focus on traversal-based graph queries. To support efficient query evaluation on large graphs, we proposed a logical graph traversal operator that can be configured to run k-hop traversals on (sub) graphs and an accompanying set of graph traversal implementations, namely the LS-traversal and the FI-traversal. The LS-traversal relies on repetitive parallelized full-column scans on the edge column group and provides good performance for graphs with a low diameter and a power-law degree distribution resulting in large intermediate results during the traversal. For extremely sparse graphs and graphs with a large diameter, such as road networks, we developed FI-traversal , an index-assisted graph traversal implementation. In the experimental evaluation, we showed that both traversal implementation exhibit strengths and weaknesses in terms of their scalability to different graph topologies and sizes, as well as to different traversal query configurations. The results lead to the conclusion that there is no universally best graph traversal implementation and the DBMS query optimizer has to select from a set of available implementations the optimal one for the given traversal query and graph. Finally, we showed that our graph traversal implementations outperform SQL-based traversal implementations as well as GDBMS-based implementations by up to two orders of magnitude.

LIGHTWEIGHT GRAPH INDEXES: To accelerate neighborhood queries—a building block for graph traversals—on the edge column group and to make their run time complexity independent from the number of edges in the graph, we developed two secondary graph index structures. The *Block-based topology index* is a light-weight, updateable index structure on top of the edge column group and effectively reduces the neighborhood lookup complexity to a constant-time lookup and a scan of a single block in the edge column group. To update the index, we use an auxiliary measure, the *index health* to quantify the usefulness of the index structure. For the optimal performance with the lowest overhead, we developed the *Adjacency-based topology index*, which is effectively a mutable adjacency list with additional mapping structures to allow combined relational/graph query processing. In the experimental evaluation, we show that an index-based traversal can outperform a scan-based counterpart by multiple orders of magnitude for small frontier set sizes, while the scan-based approach outperforms the index-based approach for large intermediate results. Therefore, we developed a hybrid graph traversal operator, which switches between scan-based execution and index-based execution depending on the frontier set size.

QUERY LANGUAGE FOR GRAPH ANALYSIS: Finally, we developed a traversal-based programming model and an accompanying domain-specific graph query language called TRAVEL. We build TRAVEL on the general programming model of *traversal hooks*, which are well-defined extension points of traversal operators allowing the user to specify custom logic that should be executed during the lifetime of a traversal query invocation. We used the LLVM compiler framework and code generation to create a specialized graph traversal operator on-the-fly during runtime as the combination of highly tuned, built-in traversal operators and user-specified code that act on a vertex- or edge level. In contrast to a message passing-based programming model, we believe that a traversal-based programming model provides a higher level of abstraction while offering a more intuitive programming interface. In addition to native support for graph traversals and traversal hooks, TRAVEL provides many more functionalities to query the graph and post-process the results of invocations of other built-in graph algorithms. By this, TRAVEL allows staying in the graph data model without having to transform intermediate results back to the relational world for further processing.

8.2 FUTURE WORK

We believe that this thesis can serve as the foundation for future research projects that aim at combining relational and graph processing in the same DBMS. In particular, areas of research that are already well-studied in the RDBMS community, but have been largely neglected by the graph community, are potential research fields to advance the state-of-the-art. Examples for such research fields include, transaction processing, query optimization, advanced graph statistics, and expressive declarative/imperative language interfaces for the formulation of graph algorithms. In the following we describe some interesting research directions that could build on the contributions made in this thesis and extend them at several layers in the system architecture of GRAPHITE.

SNAPSHOT ISOLATION FOR GRAPH INDICES: GRAPHITE does not yet provide session management, i.e., there is only a single connection to the system and the user can see and access all data and index structures. In a more realistic environment, a transactional view has to be provided not only on the base data, but also on the secondary index structures. For the adjacency-based topology index, a transactional view could be guaranteed by consulting the edge column group and their corresponding visibility data structures to get a consistent and isolated view of the data. In particular for performance-critical graph algorithms, a more sophisticated snapshotting approach on the adjacency list could be beneficial. An interesting research direction would be the evaluation of different approaches to enable lightweight snapshot isolation on the adjacency list level while not slowing down graph algorithms running on top of the secondary graph index.

COMPRESSION OF GRAPH INDICES: One important aspect we did not consider in the discussion of secondary graph index structures is data compression. Recent work on compressed graph topology index structures, e.g., k^2-ary trees by Brisaboa et al. (2009), demonstrated that the adjacency matrix representation of a graph can be highly compressed, effectively achieving a compression of the graph topology of 2 to 3 bit per edge. An interesting research question would be to identify the trade-offs for using a compressed adjacency list in terms of memory consumption and query performance in the context of a DBMS. In particular the performance trade-offs between graph-aware compression techniques, which take into account domain knowledge about the graph, and graph-agnostic compression techniques, such as delta encoding on the local neighborhood set, are an interesting starting point for further exploration.

ADVANCED GRAPH STATISTICS: We only considered basic graph statistics, such as vertex-local and graph-global degree information and the effective graph diameter, for the definition of the graph traversal operator cost models. Recent work, such as the estimation of the neighborhood function, i.e., an approximation of the number of reachable vertices within a k-hop distance from a given start vertex, can improve the estimation of the traversal costs and the number of discovered vertices significantly. Most approaches in line of research, however, compute the auxiliary data structures used for the neighborhood size estimation as part of an expensive pre-computation task, effectively making it not applicable in a DBMS context. An interesting topic for further investigation is the development of novel, lightweight graph statistics synopses that are easy to maintain, expose a low memory footprint, and provide a good estimation of the neighborhood size. Further, it would interesting to evaluate how additional predicate constraints—in our traversal queries through edge and vertex filters—can be applied to estimate the neighborhood not on the entire graph, but instead on a pre-filtered subgraph.

GRAPH OPERATORS: In the course of this thesis we focused on graph traversals as a core building block of many graph algorithms. Although graph traversals are universal graph operators with many applications, there are other graph operators, such as

path finding and value propagation algorithms, for which similar techniques as for graph traversals could be applied. Encouraged by the results for graph traversals, we envision that more graph operators will be proposed in the future, all with several physical implementations that are tailored to specific graph topology characteristics. A second important observation of this thesis is the integration of graph operators into a more complex execution plan, making most assumptions of stand-alone graph algorithm implementations invalid. Further, in a DBMS context there are typically several queries (algorithms) running in parallel and competing for scarce hardware resources. As already proposed by Then et al. (2014), work sharing between graph algorithm building blocks is an interesting research direction, which puts an emphasis not on maximum single-query performance but rather than on overall query throughput performance.

EFFICIENT DSL COMPILERS: With the introduction of TraveL, we proposed one of the first high-level DSLs for graph analysis that aim at providing an expressive algorithm interface on one side but also offer high-level constructs, such as graph traversals and traversal hooks, that a query optimizer can analyze and eventually rewrite to generate an optimal executable program out of it. We believe that the potential of combining recent advances in compiler construction research with query processing from the database community brings an unforeseen gain in terms of expressiveness and performance while bringing non-relational query processing closer to the data. One particular area of research lies in the automatic detection of available parallelism in the high-level description of a graph algorithm and the generation of highly efficient, scalable code that is close or even better than hand-written, manually tuned low-level implementations of the same graph algorithm.

A

EVALUATED DATA SETS

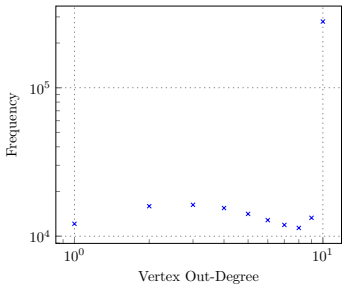
A.1 DATA SET PROPERTIES

We use representative real-world graphs from different application domains, including social networks (Orkut, LiveJournal, Twitter, Pokec), road networks (Cali), citation networks (Patents), and web networks (Wikipedia). To evaluate Graphite for larger graphs, we generated graphs using the R-MAT data generator (Chakrabarti et al., 2004) for scale factors 12 to 24. We used the matrix coefficient configuration $\langle a = 0.57, b = 0.19, c = 0.19, d = 0.05 \rangle$, which is commonly used to generate power-law degree distributions.

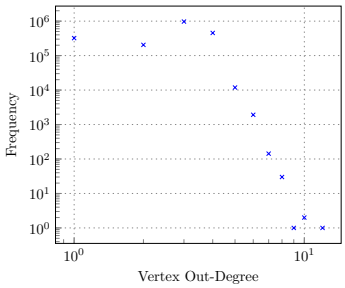
Additionally, we use data from the LDBC social network benchmark, which provides a graph data generator for producing realistic graph topololgies, a rich set of vertex/edge attributes, and realistic attribute and structural correlations. It represents a social network application with user activities during a period of time and models persons, tags, forums, messages, likes, organizations, and places as vertices and about 20 different relations between them as edges (Erling et al., 2015). Table A.1 summarizes the evaluated real-world and generated graph data sets and their basic characteristics.

Table A.1: Evaluated data sets and their topology properties.

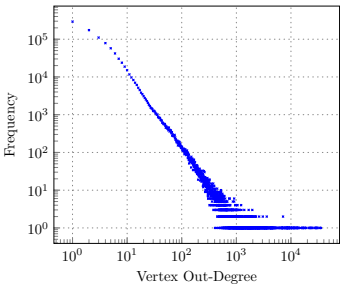
ID	$\lvert V \rvert$	$\lvert E \rvert$	$\bar{d}_{out}$	$\max(d_{out})$	$\tilde{\delta}$	*Size* (GB)
Cali	1.9 M	2.7 M	2.8	12	495.0	0.1
LiveJournal	4.8 M	68.5 M	28.3	635 K	6.5	1.6
Orkut	3.1 M	117.2 M	76.3	32 K	5.0	3.1
Patents	3.7 M	16.5 M	8.7	793	9.4	0.4
Skitter	1.7 M	11.1 M	13.1	35 K	5.9	0.3
Twitter	40.1 M	1.4 B	36.4	2.9 M	5.4	32.7
Amazon	0.4 M	3.3 M	16.8	2.7 K	7.7	0.1
Pokec	1.6 M	30.6 M	37.5	21 K	5.1	0.7
Wikipedia	25.9 M	601 M	62.1	900 K	4.7	14
R-MAT-12	3.3 K	63.2 K	24.2	1.8 K	7	0.001
R-MAT-14	13.3 K	0.3 M	22.7	4.4 K	8	0.003
R-MAT-16	48.7 K	1.1 M	24.6	9.9 K	9	0.01
R-MAT-18	0.2 M	4.2 M	26.9	22.5 K	9	0.05
R-MAT-20	0.7 M	16.8 M	29.1	51.5 K	9	0.2
R-MAT-22	2.5 M	67.1 M	31.4	114 K	10	0.9
R-MAT-24	9.3 M	268.4 M	34.6	273 K	10	3.8
LDBC-SF1	3.1 M	17.1 M	–	–	–	1.3
LDBC-SF3	8.9 M	50.7 M	–	–	–	3.8
LDBC-SF10	29.1 M	171.5 M	–	–	–	12.9
LDBC-SF30	85.7 M	520.6 M	–	–	–	39.8
LDBC-SF100	274.1 M	1.7 B	–	–	–	131.4



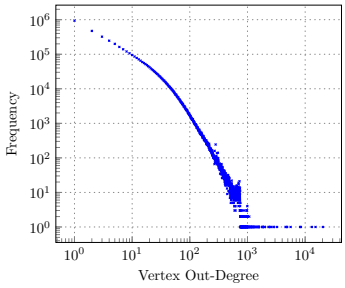
(a) AMAZON



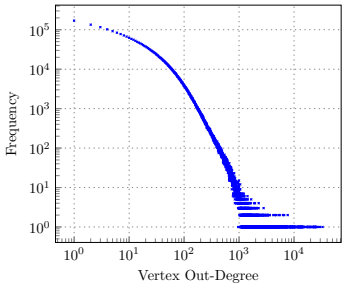
(b) CALI



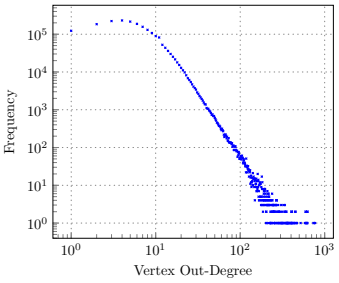
(c) SKITTER



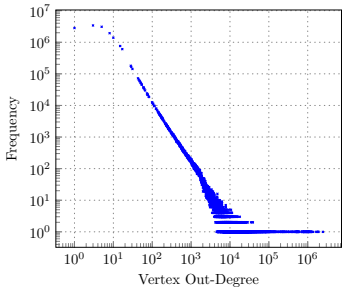
(d) LIVEJOURNAL



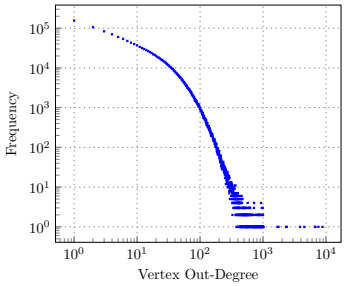
(e) ORKUT



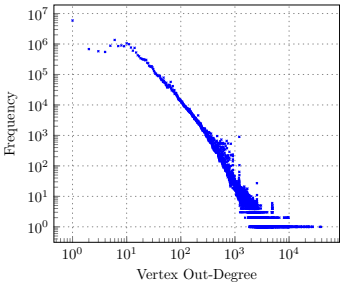
(f) PATENTS



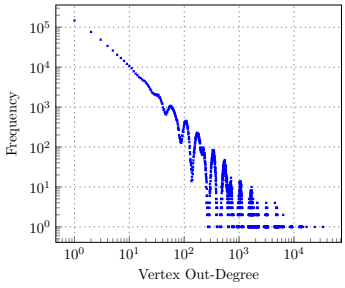
(g) TWITTER



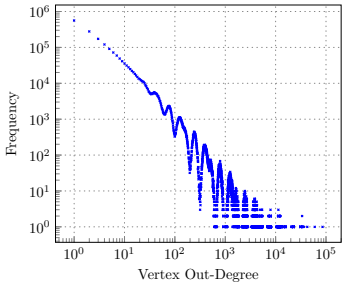
(h) POKEC



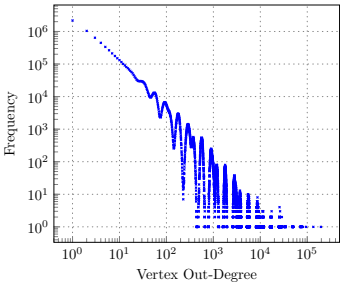
(i) WIKIPEDIA



(j) RMAT-SF20



(k) RMAT-SF22



(l) RMAT-SF24

B ADDITIONAL TRAVEL EXAMPLES

B.1 BETWEENNESS CENTRALITY

```
CREATE TEMPORARY VERTEX ATTRIBUTE<DOUBLE> sigma = 0;
CREATE TEMPORARY VERTEX ATTRIBUTE<DOUBLE> delta = 0;
CREATE TEMPORARY VERTEX ATTRIBUTE<DOUBLE> centrality = 0;

HOOK VERTEX "FORW" (CONTEXT $v) {
  $sum = 0.0;
  FOR $i : $v<-[*]- {
    $sum = $sum + $i@sigma;
  }
  UPDATE $v { SET sigma = $sum; }
}

HOOK VERTEX "BACK" (CONTEXT $v) {
  $sum = 0.0;
  FOR $i : $v-[*]-> {
    $sum = $sum + ($v@sigma / $i@sigma * (1 + $i@delta));
  }
  UPDATE $v { SET delta = $sum;
              SET centrality = centrality + $v@delta; }
}

TRAVEL "BETWEENNESS" GRAPH "G" {
  FOR $v : $VERTICES {
    UPDATE $v { SET sigma = 1.0; }
    UPDATE $VERTICES { SET sigma = 0.0; }
    UPDATE $VERTICES { SET delta = 0.0; }
    TRAVERSE BFS $v-[*]->(*) HOOK "FORW";
    TRAVERSE BFS $v<-[*]-(*) HOOK "BACK";
  }
}
```

B.2 DEGREE CENTRALITY

```
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> outdeg_centrality = 0;
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> indeg_centrality = 0;
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> combdeg_centrality = 0;

TRAVEL "DEGREE_CENTRALITY" GRAPH "G" {
  FOR $n : $VERTICES {
    UPDATE $n { SET outdeg_centrality = COUNT($n-[*]->);
                SET indeg_centrality = COUNT($n<-[*]-);
                SET combdeg_centrality = COUNT($n-[*]->) + COUNT($n<-[*]-);};
  }
}
```

B.3 RANDOM WALKS WITH RESTART

```
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> visitationCnt = 0;
INT cnt = 0;

HOOK VERTEX "H" (CONTEXT $v) {
  IF ($cnt == $maxVisits) {
    RESTRICT ALL true;
  }
  UPDATE $v { SET visitationCnt = visitationCnt + 1; }
  $cnt = $cnt + 1;
  IF ($probe < RANDOM()) {
    TRAVERSE BFS $v-[*]-> HOOK "H";
  }
}

TRAVEL "RANDOM_WALK" (VERTEX $source, INT $maxVisits, DOUBLE $probe) GRAPH "G" {
  TRAVERSE BFS $source-[*]-> HOOK "H";
}
```

BIBLIOGRAPHY

ArangoDB project website.
https://www.arangodb.com/ (Last accessed: April 2017).

Apache Flink project website.
https://flink.apache.org/ (Last accessed: April 2017).

Apache Giraph project website.
http://giraph.apache.org/ (Last accessed: April 2017).

Graph500 project website.
http://www.graph500.org/ (Last accessed: April 2017).

InfiniteGraph project website.
http://objectivity.com/INFINITEGRAPH (Last accessed: April 2017).

Neo4j project website.
http://neo4j.org/ (Last accessed: April 2017).

OrientDB project website.
http://www.orientdb.org/ (Last accessed: April 2017).

Sparksee project website.
http://www.sparsity-technologies.com/ (Last accessed: April 2017).

Apache TinkerPop project website.
http://tinkerpop.incubator.apache.org/ (Last accessed: April 2017).

TitanDB project website.
http://thinkaurelius.github.io/titan/ (Last accessed: April 2017).

W3C: SPARQL 1.1 Overview, March 2013.
http://www.w3.org/TR/sparql11-overview/ (Last accessed: April 2017).

W3C: RDF 1.1 Concepts and Abstract Syntax, February 2014.
http://www.w3.org/TR/rdf11-concepts/ (Last accessed: April 2017).

Daniel Abadi. Column Stores for Wide and Sparse Data. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, CIDR '07, pages 292–297, 2007.

Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, 2006. doi:10.1145/1142473.1142548.

Daniel Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '07, pages 411–422, 2007.

Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A Bernstein, Michael J Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J Franklin, et al. The Beckman Report on Database Research. *SIGMOD Rec.*, 43(3):61–70, 2014. doi:10.1145/2694428.2694441.

Christopher R. Aberger, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: Boolean Algebra Based Graph Processing. *CoRR*, abs/1503.02368, 2015. http://arxiv.org/abs/1503.02368 (Last accessed: April 2017).

Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010. doi:10.1109/SC.2010.46.

R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 253–262, 1989. doi:10.1145/67544.66950.

Rakesh Agrawal. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. *IEEE Trans. Softw. Eng.*, 14(7):879–885, July 1988. doi:10.1109/32.42731.

Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '15, pages 105–117, 2015. doi:10.1145/2749469.2750386.

P.A. Alsberg. Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring. *Proceedings of the IEEE*, 63(8):1114–1122, Aug 1975. doi:10.1109/PROC.1975.9903.

Sandra Álvarez, Nieves R. Brisaboa, Susana Ladra, and Óscar Pedreira. A Compact Representation of Graph Databases. In *Proceedings of the Workshop on Mining and Learning with Graphs*, MLG '10, pages 18–25, 2010. doi:10.1145/1830252.1830255.

Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008. doi:10.1145/1322432.1322433.

Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four Degrees of Separation. In *Proceedings of the ACM Web Science Conference*, WebSci '12, pages 33–42, 2012. doi:10.1145/2380718.2380723.

David A. Bader and Kamesh Madduri. Designing Multithreaded Algorithms for Breadth-First Search and St-connectivity on the Cray MTA-2. In *Proceedings of the International Conference on Parallel Processing*, ICPP '06, pages 523–530, 2006. doi:10.1109/ICPP.2006.34.

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, 1986. doi:10.1145/6012.15399.

Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, 2012.

Scott Beamer, Aydın Buluç, Krste Asanović, and David Patterson. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1618–1627, 2013. doi:10.1109/IPDPSW.2013.159.

Rudolf Berrendorf and Mathias Makulla. Level-Synchronous Parallel Breadth-First Search Algorithms For Multicore and Multiprocessor Systems. In *Proceedings of the International Conference on Future Computational Technologies and Applications*, FUTURE COMPUTING '14, pages 26–31, 2014.

Mauro Bisson, Massimo Bernaschi, and Enrico Mastrostefano. Parallel Distributed Breadth First Search on the Kepler Architecture. *CoRR*, abs/1408.1605, 2014. http://arxiv.org/abs/1408.1605 (Last accessed: April 2017).

Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An Experimental Analysis of a Compact Graph Representation. In *Proceedings of the Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pages 49–61, 2004.

Paolo Boldi and Sebastiano Vigna. The Web Graph Framework I: Compression Techniques. In *Proceedings of the International World Wide Web Conference*, WWW '04, pages 595–601, 2004.

Paolo Boldi and Sebastiano Vigna. In-Core Computation of Geometric Centralities with HyperBall: A Hundred Billion Nodes and Beyond. In *Proceedings of the IEEE International Conference on Data Mining Workshops*, ICDMW '13, pages 621–628, 2013. doi:10.1109/ICDMW.2013.10.

Paolo Boldi, Marco Rosa, and Sebastiano Vigna. HyperANF: Approximating the Neighbourhood Function of Very Large Graphs on a Budget. In *Proceedings of the International Conference on World Wide Web*, WWW '11, pages 625–634, 2011. doi:10.1145/1963405.1963493.

Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an Efficient RDF Store over a Relational Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 121–132, 2013. doi:10.1145/2463676.2463718.

Christof Bornhövd, Robert Kubis, Wolfgang Lehner, Hannes Voigt, and Horst Werner. Flexible Information Management, Exploration and Analysis in SAP HANA. In *Proceedings of the International Conference on Data Technologies and Applications*, DATA '12, pages 15–28, 2012.

NievesR. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-Trees for Compact Web Graph Representation. In *String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 18–30. 2009. doi:10.1007/978-3-642-03784-9_3.

Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths. In *Algorithm Theory - SWAT 2004*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. 2004. doi:10.1007/978-3-540-27810-8_41.

Michael L Brodie and Jason T Liu. OTM10 Keynote: The Power and Limits of Relational Technology In the Age of Information Ecosystems. In *On the Move to Meaningful Internet Systems OTM*. October 2010.

Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(Ger) Graph Analytics on a Dataflow Engine. *Proc. VLDB Endow.*, 8(2):161–172, October 2014. doi:10.14778/2735471.2735477.

Aydın Buluç and John R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011. doi:10.1177/1094342011403516.

Aydın Buluç and Kamesh Madduri. Parallel Breadth-first Search on Distributed Memory Systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, 2011. doi:10.1145/2063384.2063471.

Paul Burkhardt and Chris Waring. An NSA Big Graph experiment, May 2013.

Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*, GRADES '15, pages 7:1–7:6, 2015. doi:10.1145/2764947.2764954.

Deepayan Chakrabarti and Christos Faloutsos. Graph Mining: Laws, Generators, and Algorithms. *ACM Comput. Surv.*, 38(1), June 2006. doi:10.1145/1132952.1132954.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the SIAM International Conference on Data Mining*, pages 442–446, 2004. doi:10.1137/1.9781611972740.43.

Fabio Checconi and Fabrizio Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 425–434, 2014. doi:10.1109/IPDPS.2014.52.

Chen Chen, Souad Koliai, and Guang Gao. Exploitation of locality for energy efficiency for breadth first search in fine-grain execution models. *Tsinghua Science and Technology*, 18(6):636–646, Dec 2013. doi:10.1109/TST.2013.6678909.

Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, 2015. doi:10.1145/2741948.2741970.

Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C. S. Lui, and Cheng He. VENUS: Vertex-centric streamlined graph computation on a single PC. In *International Conference on Data Engineering*, ICDE '15, pages 1131–1142, 2015. doi:10.1109/ICDE.2015.7113362.

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, 2012. doi:10.1145/2168836.2168846.

J. Chhugani, N. Satish, Changkyu Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *IEEE International Parallel Distributed Processing Symposium*, IPDPS '12, pages 378–389, May 2012. doi:10.1109/IPDPS.2012.43.

Nicos Christofides. The optimum traversal of a graph. *Omega*, 1(6):719–732, 1973. doi:10.1016/0305-0483(73)90089-3.

Marek Ciglan, Alex Averbuch, and Ladialav Hluchy. Benchmarking Traversal Operations over Graph Databases. In *IEEE International Conference on Data Engineering Workshops*, ICDEW '12, pages 186–189, 2012. doi:10.1109/ICDEW.2012.47.

Edith Cohen. Size-estimation Framework with Applications to Transitive Closure and Reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, December 1997. doi:10.1006/jcss.1997.1534.

Edith Cohen. Scalable Neighborhood Sketching and Distance Distribution Estimation in Graph Datasets: Revisited, Unified, and Improved. *CoRR*, abs/1306.3284, 2013. `http://arxiv.org/abs/1306.3284` (Last accessed: April 2017).

Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-hop Labels. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 937–946, 2002.

Guojing Cong and Konstantin Makarychev. Optimizing Large-Scale Graph Analysis on a Multi-threaded, Multi-core Platform. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 688–697, 2011. doi:10.1109/IPDPS.2011.393.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

Derek G. Corneil and Richard Krueger. Simple vertex ordering characterizations for graph search: (expanded abstract). *Electronic Notes in Discrete Mathematics*, 22(0):445 – 449, 2005. doi:http://dx.doi.org/10.1016/j.endm.2005.06.061. International Colloquium on Graph Theory.

A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. 1998. doi:10.1007/BFb0055823.

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.*, 8(12):1466–1477, August 2015. doi:10.14778/2824032.2824045.

Zehan Cui, Licheng Chen, Mingyu Chen, Yungang Bao, Yongbing Huang, and Huiwei Lv. Evaluation and Optimization of Breadth-First Search on NUMA Cluster. In *IEEE International Conference on Cluster Computing*, CLUSTER '12, pages 438–448, Sept 2012. doi:10.1109/CLUSTER.2012.29.

Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting Relational to Graph Databases. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 1:1–1:6, 2013. doi:10.1145/2484425.2484426.

E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1(1): 269–271, December 1959. doi:10.1007/BF01386390.

David Dominguez-Sal, Peter Urbón-Bayes, Aleix Giménez-Vañó, Sergio Gómez-Villamor, Norbert Martínez-Bazan, and Josep Larriba-Pey. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. In *Web-Age Information Management*, volume 6185 of *Lecture Notes in Computer Science*, pages 37–48. 2010. doi:10.1007/978-3-642-16720-1_4.

David Dominguez-Sal, Norbert Martínez-Bazan, Victor Muntés-Mulero, Pere Baleta, and Josep Larriba-Pey. A Discussion on the Design of Graph Database Benchmarks. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, volume 6417 of *Lecture Notes in Computer Science*, pages 25–40. 2011. doi:10.1007/978-3-642-18206-8_3.

Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1): 3–8, 2012.

Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, 2015. doi:10.1145/2723372.2742786.

Christos Faloutsos. Multiattribute Hashing Using Gray Codes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 227–238, 1986. doi:10.1145/16894.16877.

Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The Case Against Specialized Graph Analytics Engines. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, 2012. doi:10.1145/2094114.2094126.

Martin Faust, David Schwalb, and Jens Krueger. Fast column scans: Paged indices for in-memory column stores. In *Proceedings of the International Workshop on In Memory Data Management and Analytics, IMDM*, pages 13–24, 2013.

Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*, GRADES '14, pages 2:1–2:6, 2014. doi:10.1145/2621934.2621936.

Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. Relational Approach for Shortest Path Discovery over Large Graphs. *Proc. VLDB Endow.*, 5 (4):358–369, December 2011. doi:10.14778/2095686.2095694.

Tao Gao, Yutong Lu, Baida Zhang, and Guang Suo. Using the Intel Many Integrated Core to Accelerate Graph Traversal. *Int. J. High Perform. Comput. Appl.*, 28(3):255–266, August 2014. doi:10.1177/1094342014524240.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Hillel Gazit and Gary L. Miller. An Improved Parallel Algorithm That Computes the BFS Numbering of a Directed Graph. *Inf. Process. Lett.*, 28(2):61–65, June 1988. doi:10.1016/0020-0190(88)90164-0.

Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems. *CoRR*, abs/1312.3018, 2013.

Rosalba Giugno and Dennis Shasha. GraphGrep: A Fast and Universal Method for Querying Graphs. In *ICPR (2)*, pages 112–115, 2002.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, 2012.

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 599–613, October 2014.

Oded Green, Marat Dukhan, and Richard W. Vuduc. Branch-Avoiding Graph Algorithms. *CoRR*, abs/1411.1460, 2014.
`http://arxiv.org/abs/1411.1460` (Last accessed: April 2017).

Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *Parallel Object-Oriented Scientific Computing*, POOSC '05, 2005.

Andrey Gubichev and Manuel Then. Graph Pattern Matching: Do We Have to Reinvent the Wheel? In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*, GRADES '14, pages 8:1–8:7, 2014. doi:10.1145/2621934.2621944.

Fred G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978. doi:10.1145/355791.355796.

Minyang Han and Khuzaima Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015. doi:10.14778/2777598.2777604.

Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 7(12):1047–1058, August 2014a. doi:10.14778/2732977.2732980.

Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, 2014b. doi:10.1145/2592798.2592799.

Wook-Shin Han, Jinsoo Lee, Minh-Duc Pham, and Jeffrey Xu Yu. iGraph: A Framework for Comparisons of Disk-based Graph Indexing Techniques. *Proc. VLDB Endow.*, 3(1-2): 449–459, September 2010. doi:10.14778/1920841.1920901.

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, 2013. doi:10.1145/2487575.2487581.

P Hanrahan. Using Caching and Breadth-first Search to Speed Up Ray-tracing. In *Proceedings on Graphics Interface/Vision Interface*, pages 56–61, 1986.

Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the International Conference on High Performance Computing*, HiPC '07, pages 197–208, 2007.

Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, pages 27–38, 2014. doi:10.1145/2628071.2628091.

Olaf Hartig. Reconciliation of RDF* and Property Graphs. *CoRR*, abs/1409.3288, 2014.

Matthias Hauck, Marcus Paradies, Holger Fröning, Wolfgang Lehner, and Hannes Rauhe. Highspeed Graph Processing Exploiting Main-Memory Column Stores. In *Proceedings of the Euro-Par 2015 International Workshops*, pages 503–514, 2015. doi:10.1007/978-3-319-27308-2_41.

Huahai He and Ambuj K. Singh. Closure-Tree: An Index Structure for Graph Queries. In *Proceedings of the International Conference on Data Engineering*, ICDE '06, pages 38–38, 2006. doi:10.1109/ICDE.2006.37.

Kai Herrmann, Hannes Voigt, and Wolfgang Lehner. Cinderella - Adaptive online partitioning of irregularly structured data. In *Proceedings of the International Conference on Data Engineering Workshops*, ICDEW '14, pages 284–291, 2014. doi:10.1109/ICDEW.2014.6818342.

Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, 2011. doi:10.1109/PACT.2011.14.

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 349–362, 2012. doi:10.1145/2150976.2151013.

Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 92:1–92:11, 2013a. doi:10.1145/2503210.2503246.

Sungpack Hong, Jan Van Der Lugt, Adam Welc, Raghavan Raman, and Hassan Chafi. Early Experiences in Using a Domain-specific Language for Large-scale Graph Analysis. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 5:1–5:6, 2013b. doi:10.1145/2484425.2484430.

Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 208:208–208:218, 2014. doi:10.1145/2544137.2544162.

Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently Answering Reachability Queries on Very Large Directed Graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 595–608, 2008. doi:10.1145/1376616.1376677.

Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-HOP: A High-compression Indexing Scheme for Reachability Query. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 813–826, 2009. doi:10.1145/1559845.1559930.

Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. Scarab: Scaling reachability computation on large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 169–180, 2012. doi:10.1145/2213836.2213856.

Alekh Jindal and Samuel Madden. GRAPHiQL: A Graph Intuitive Query Language for Relational Databases. In *Proceedings of the International Conference on Big Data*, Big Data '14, pages 441–450, 2014. doi:10.1109/BigData.2014.7004261.

Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. Graph Analytics using the Vertica Relational Database. *CoRR*, abs/1412.5263, 2014a. `http://arxiv.org/abs/1412.5263` (Last accessed: April 2017).

Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: Your Relational Friend for Graph Analytics! *Proc. VLDB Endow.*, 7(13):1669–1672, August 2014b. doi:10.14778/2733004.2733057.

U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, 2009. doi:10.1109/ICDM.2009.14.

U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: A Scalable and General Graph Management System. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1091–1099, 2011a. doi:10.1145/2020408.2020580.

U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Mining Radii of Large Graphs. *ACM Trans. Knowl. Discov. Data*, 5(2): 8:1–8:24, February 2011b. doi:10.1145/1921632.1921634.

David Kernert, Frank Köhler, and Wolfgang Lehner. SLACID - Sparse Linear Algebra in a Column-oriented In-memory Database System. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, SSDBM '14, pages 11:1–11:12, 2014. doi:10.1145/2618243.2618254.

Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, 2013. doi:10.1145/2465351.2465369.

Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proceedings of International World Wide Web Conference*, WWW '13, pages 1343–1350, 2013.

Aapo Kyrola and Carlos Guestrin. GraphChi-DB: Simple Design for a Scalable Graph Database System - on Just a PC. *CoRR*, abs/1403.0701, 2014. `http://arxiv.org/abs/1403.0701` (Last accessed: April 2017).

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 31–46, 2012.

Monica S. Lam, Stephen Guo, and Jiwon Seo. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *Proceedings of the IEEE International Conference on Data Engineering*, ICDE '13, pages 278–289, 2013. doi:10.1109/ICDE.2013.6544832.

Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL Server Column Stores. In *Proc. SIGMOD'13*, pages 1159–1168, 2013. doi:10.1145/2463676.2463708.

Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

Charles E. Leiserson and Tao B. Schardl. A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, 2010. doi:10.1145/1810479.1810534.

Daniel Lemire and Owen Kaser. Reordering Columns for Smaller Indexes. *CoRR*, abs/0909.1346, 2009.

Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting Improves Word-aligned Bitmap Indexes. *Data Knowl. Eng.*, 69(1):3–28, January 2010. doi:10.1016/j.datak.2009.08.006.

Daniel Lemire, Owen Kaser, and Eduardo Gutarra. Reordering Rows for Better Compression: Beyond the Lexicographic Order. *ACM Trans. Database Syst.*, 37(3):20:1–20:29, September 2012. doi:10.1145/2338626.2338633.

Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In *Datenbanksysteme in Business, Technologie und Web*, BTW '09, pages 486–497, 2009. (In german).

Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding Up Queries in Column Stores: A Case for Compression. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '10, pages 117–129, 2010. doi:10.1007/978-3-642-15105-7_10.

Jure Leskovec and Rok Sosic. SNAP: A General Purpose Network Analysis and Graph Mining Library. *CoRR*, abs/1606.07550, 2016.

Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++.
http://snap.stanford.edu/snap (Last accessed: April 2017).

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 177–187, 2005. doi:10.1145/1081870.1081893.

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the International Conference on World Wide Web*, WWW '08, pages 695–704, 2008. doi:10.1145/1367497.1367591.

Zhiyuan Lin, Minsuk Kahng, Kaeser Md. Sabrin, Duen Horng Chau, Ho Lee, and U. Kang. MMap: Fast Billion-Scale Graph Computation on a PC via Memory Mapping. In *Proceedings of the International Conference on Big Data*, 2014.

R. J. Lipton and J. F. Naughton. Estimating the Size of Generalized Transitive Closures. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '89, pages 165–171, 1989.

László Lovász. Random Walks on Graphs : A Survey. *Bolyai Soc. Math. Stud.*, 2:1–46, 1993.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence*, UAI '10, pages 340–349, July 2010.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. doi:10.14778/2212351.2212354.

Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.*, 8(3):281–292, November 2014. doi:10.14778/2735508.2735517.

Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun. Compression and Sieve: Reducing Communication in Parallel Breadth First Search on Distributed Memory Systems. *CoRR*, abs/1208.5542, 2012.
http://arxiv.org/abs/1208.5542 (Last accessed: April 2017).

Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *Proceedings of the International Conference on Data Engineering*, ICDE '15, pages 363–374, 2015. doi:10.1109/ICDE.2015.7113298.

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010. doi:10.1145/1807167.1807184.

Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000. doi:10.1007/s007780000031.

Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez, and Josep Larriba-Pey. DEX: High-performance Exploration on Large Graphs for Information Retrieval. In *Proceedings of the ACM Conference on Information and Knowledge Management*, CIKM '07, pages 573–582, 2007. doi:10.1145/1321440.1321521.

Norbert Martínez-Bazan, M. Ángel Águila Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep Larriba-Pey. Efficient Graph Management Based on Bitmap Indices. In *Proceedings of the International Database Engineering & Applications Symposium*, IDEAS '12, pages 110–119, 2012. doi:10.1145/2351476.2351489.

William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *J. Parallel Distrib. Comput.*, 65(8):901–910, August 2005. doi:10.1016/j.jpdc.2005.03.007.

Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In *Workshop on Hot Topics in Operating Systems*, HotOS '15, Kartause Ittingen, Switzerland, May 2015.

Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, 2012. doi:10.1145/2145816.2145832.

Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web — revisited: A trick of the heavy tail. In *Proceedings of the International Conference on World Wide Web*, WWW '14, pages 427–432, 2014. doi:10.1145/2567948.2576928.

Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *Trans. Storage*, 11(3):14:1–14:34, July 2015. doi:10.1145/2700302.

Stanley Milgram. The Small World Problem. *Psychology Today*, 61(1):60–67, 1967.

C. Mohan. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *Proceedings of the International Conference on Extending Database Technology*, EDBT '13, pages 11–16, 2013. doi:10.1145/2452376.2452378.

DoRon B. Motter and Igor L. Markov. A Compressed Breadth-First Search for Satisfiability. In *Revised Papers from the International Workshop on Algorithm Engineering and Experiments*, ALENEX '02, pages 29–42, 2002.

Inderpal Singh Mumick and Hamid Pirahesh. Implementation of Magic-sets in a Relational Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 103–114, 1994. doi:10.1145/191839.191860.

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, 2013. doi:10.1145/2517349.2522738.

Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. Code Generation for Efficient Query Processing in Managed Runtimes. *Proc. VLDB Endow.*, 7(12):1095–1106, August 2014. doi:10.14778/2732977.2732984.

Jacob Nelson, Brandon Myers, A. H. Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, Dan Grossman, Simon Kahan, and Mark Oskin. Crunching Large Graphs with Commodity Processors. In *Proceedings of the USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011.

Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011. doi:10.14778/2002938.2002940.

Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Proceedings of the IEEE International Conference on Data Engineering*, ICDE '11, pages 984–994, 2011. doi:10.1109/ICDE.2011.5767868.

Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91–113, February 2010. doi:10.1007/s00778-009-0165-y.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013. doi:10.1145/2517349.2522739.

Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join Processing for Graph Patterns : An Old Dog with New Tricks. *CoRR*, abs/1503.04169, 2015a.
http://arxiv.org/abs/1503.04169 (Last accessed: April 2017).

Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*, GRADES '15, pages 2:1–2:8, 2015b. doi:10.1145/2764947.2764948.

Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 25–28, 2014. doi:10.1109/.13.

Carlos Ordonez, Achyuth Gurram, and Nirmala Rai. Recursive query evaluation in a column dbms to analyze large graphs. In *Proceedings of the International Workshop on Data Warehousing and OLAP*, DOLAP '14, pages 71–80, 2014. doi:10.1145/2666158.2666177.

Christopher R. Palmer, Phillip B. Gibbons, and Christos Faloutsos. ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 81–90, 2002. doi:10.1145/775047.775059.

Onofrio Panzarino. *Learning Cypher*. Packt Publishing, 2014. ISBN 1783287756, 9781783287758.

Marcus Paradies, Michael Rudolf, Christof Bornhövd, and Wolfgang Lehner. GRATIN: Accelerating Graph Traversals in Main-Memory Column Stores. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*, GRADES '14, pages 9:1–9:6, 2014. doi:10.1145/2621934.2621941.

Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 29:1–29:12, 2015. doi:10.1145/2791347.2791383.

Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010. doi:10.1109/SC.2010.34.

Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive Graph Analytics on Big-Memory Machines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1105–1110, 2015. doi:10.1145/2723372.2735369.

Minh-Duc Pham, Peter Boncz, and Orri Erling. S3G2: A Scalable Structure-Correlated Social Graph Generator. In *Proceedings of TPC Technology Conference on Performance Evaluation & Benchmarking*, 2012.

Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing Large Graphs on Multi-cores with Graph Awareness. In *Proceedings of the USENIX Conference on Annual Technical Conference*, ATC'12, pages 41–52, 2012.

Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. PGX.ISO: Parallel and Efficient In-Memory Engine for Subgraph Isomorphism. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*, GRADES '14, pages 5:1–5:6, 2014. doi:10.1145/2621934.2621939.

Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.*, 6(11):1080–1091, August 2013. doi:10.14778/2536222.2536233.

Bruno F. Ribeiro and Don Towsley. On the estimation accuracy of degree distributions from graph sampling. In *Proceedings of the IEEE Conference on Decision and Control*, CDC '12, pages 5240–5247, 2012. doi:10.1109/CDC.2012.6425857.

Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2015. ISBN 978-1-4493-5626-2. 2nd edition.

Liam Roditty and Virginia Vassilevska Williams. Fast Approximation Algorithms for the Diameter and Radius of Sparse Graphs. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, STOC '13, pages 515–524, 2013. ISBN 978-1-4503-2029-0. doi:10.1145/2488608.2488673.

Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language. *CoRR*, abs/1508.03843, 2015.

Marko A. Rodriguez and Peter Neubauer. The Graph Traversal Pattern. *CoRR*, abs/1004.1001, 2010a.
http://arxiv.org/abs/1004.1001 (Last accessed: April 2017).

Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010b.
http://arxiv.org/abs/1006.2361 (Last accessed: April 2017).

Donald J. Rose, Robert Endre Tarjan, and George S Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, 2013. doi:10.1145/2517349.2522740.

Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web*, pages 403–420, 2013.

Sherif Sakr. Storing and Querying Graph Data Using Efficient Relational Processing Techniques. In *Proceedings of International United Information Systems Conference*, UNISCON '09, pages 379–392, 2009. doi:10.1007/978-3-642-01112-2_39.

Sherif Sakr and Ghazi Al-Naymat. Graph indexing and querying: a review. *International Journal of Web Information Systems*, 6(2):101–120, 2010a. doi:10.1108/17440081011053104.

Sherif Sakr and Ghazi Al-Naymat. Efficient Relational Techniques for Processing Graph Queries. *J. Comput. Sci. Technol.*, 25(6):1237–1255, 2010b. doi:10.1007/s11390-010-9402-5.

Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-sparql: A hybrid engine for querying large attributed graphs. In *Proceedings of the ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 335–344, 2012. doi:10.1145/2396761.2396806.

Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, SSDBM '13, pages 22:1–22:12, 2013. doi:10.1145/2484838.2484843.

Scott Sallinen, Abdullah Gharaibeh, and Matei Ripeanu. Accelerating Direction-Optimized Breadth First Search on Hybrid Architectures. *CoRR*, abs/1503.04359, 2015.

Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. Horton: Online Query Execution Engine for Large Distributed Graphs. In *Proceedings of the IEEE International Conference on Data Engineering*, ICDE '12, pages 1289–1292, 2012. doi:10.1109/ICDE.2012.129.

Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, 2014. doi:10.1145/2588555.2610518.

Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013. doi:10.14778/2556549.2556572.

Martin Sevenich, Sungpack Hong, Adam Welc, and Hassan Chafi. Fast In-Memory Triangle Listing for Large Real-World Graphs. In *Proceedings of the Workshop on Social Network Mining and Analysis*, SNAKDD '14, pages 2:1–2:9, 2014. doi:10.1145/2659480.2659494.

Haichuan Shang and Masaru Kitsuregawa. Efficient Breadth-first Search on Large Graphs with Skewed Degree Distributions. In *Proceedings of the International Conference on Extending Database Technology*, EDBT '13, pages 311–322, 2013. doi:10.1145/2452376.2452413.

Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, 2013. doi:10.1145/2463676.2467799.

Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013. doi:10.1145/2442516.2442530.

Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *Proceedings of Data Compression Conference*, DCC '15, pages 403–412, 2015. doi:10.1109/DCC.2015.8.

Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store Support for RDF Data Management: Not All Swans Are White. *Proc. VLDB Endow.*, 1(2):1553–1563, August 2008. doi:10.14778/1454159.1454227.

Jeremy G. Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 731–742, 2012. doi:10.1145/2213836.2213946.

J. Silvela and J. Portillo. Breadth-first search and its application to image processing problems. *Trans. Img. Proc.*, 10(8):1194–1199, August 2001. doi:10.1109/83.935035.

David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *Proc. VLDB Endow.*, 7(13):1405–1416, August 2014. doi:10.14778/2733004.2733013.

George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 550–559, 2014. doi:10.1109/IPDPS.2014.64.

George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. High-Performance Graph Analytics on Manycore Processors. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 17–27, 2015. doi:10.1109/IPDPS.2015.54.

Tom St. John, Jack B. Dennis, and Guang R. Gao. Massively Parallel Breadth First Search Using a Tree-structured Memory Model. In *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 115–123, 2012. doi:10.1145/2141702.2141715.

Michael Stonebraker and Uğur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering*, ICDE '05, pages 2–11, 2005. doi:10.1109/ICDE.2005.1.

Philip Stutz, Abraham Bernstein, and William Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *Proceedings of the International Semantic Web Conference on The Semantic Web*, ISWC '10, pages 764–780, 2010.

Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901, 2015. doi:10.1145/2723372.2723732.

Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, 2015.

Ilie Tanase, Yinglong Xia, Lifeng Nai, Yanbin Liu, Wei Tan, Jason Crawford, and Ching-Yung Lin. A Highly Efficient Runtime and Graph Library for Large Scale Graph Analytics. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*, GRADES '14, pages 10:1–10:6, 2014. doi:10.1145/2621934.2621945.

Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The More the Merrier: Efficient Multi-source Graph Traversal. *Proc. VLDB Endow.*, 8(4):449–460, December 2014. doi:10.14778/2735496.2735507.

Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.*, 7 (3):193–204, November 2013. doi:10.14778/2732232.2732238.

Silke Trissl. *Cost-based Optimization of Graph Queries in Relational Database Management Systems*. PhD thesis, 2012.

Silke Trißl and Ulf Leser. Fast and Practical Indexing and Querying of Very Large Graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 845–856, 2007. doi:10.1145/1247480.1247573.

Ray Valdes. The Competitive Dynamics of the Consumer Web: Five Graphs Deliver a Sustainable Advantage, 2012.

Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. doi:10.1145/79173.79181.

Sebastiaan J. van Schaik and Oege de Moor. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 913–924, 2011. doi:10.1145/1989323.1989419.

Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, CIDR '13, 2013.

Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph Query Processing with Abstraction Refinement - Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the USENIX Annual Technical Conference*, ATC '15, pages 387–401, 2015.

Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, Shanshan Ying, and Hai Jin. An Efficient Graph Indexing Method. In *Proceedings of the IEEE International Conference on Data Engineering*, ICDE '12, pages 210–221, 2012. doi:10.1109/ICDE.2012.28.

Wei Wei, Jordan Erenrich, and Bart Selman. Towards Efficient Sampling: Exploiting Random Walk Strategies. In *Proceedings of the National Conference on Artifical Intelligence*, AAAI '04, pages 670–676, 2004.

Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph Analysis: Do We Have to Reinvent the Wheel? In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 7:1–7:6, 2013. doi:10.1145/2484425.2484432.

Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.*, 2(1):385–394, August 2009. doi:10.14778/1687627.1687671.

Peter T. Wood. Query Languages for Graph Databases. *SIGMOD Rec.*, 41(1):50–60, April 2012. doi:10.1145/2206869.2206879.

Yinglong. Xia and Viktor Prasanna. Topologically Adaptive Parallel Breadth-First Search on Multicore Processors. In *International Conference on Parallel and Distributed Computing and Systems*, PDCS 09, 2009.

Yinglong Xia, Ilie Gabriel Tanase, Lifeng Nai, Wei Tan, Yanbin Liu, Jason Crawford, and Ching-Yung Lin. Explore Efficient Data Organization for Large Scale Graph Analytics and Storage. In *Proceedings of the International Conference on Big Data*, pages 942–951, 2014. doi:10.1109/BigData.2014.7004326.

Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 194–204, 2015. doi:10.1145/2688500.2688508.

Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast Iterative Graph Computation with Block Updates. *Proc. VLDB Endow.*, 6(14):2014–2025, September 2013. doi:10.14778/2556549.2556581.

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, 2013. doi:10.1145/2484425.2484427.

Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, 2004. doi:10.1145/1007568.1007607.

Y. Yasui, K. Fujisawa, and K. Goto. NUMA-optimized parallel breadth-first search on multicore single-node system. In *Proceedings of IEEE International Conference on Big Data*, pages 394–402, Oct 2013. doi:10.1109/BigData.2013.6691600.

Yuichiro Yasui, Katsuki Fujisawa, and Yukinori Sato. Fast and Energy-efficient Breadth-First Search on a Single NUMA System. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 365–381. 2014. doi:10.1007/978-3-319-07518-1_23.

Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: Scalable Reachability Index for Large Graphs. *Proc. VLDB Endow.*, 3(1-2):276–284, September 2010. doi:10.14778/1920841.1920879.

Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *CoRR*, abs/1301.0977, 2013.

Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '05, pages 25–, 2005. doi:10.1109/SC.2005.4.

Liang Yuan, Chen Ding, Daniel Štefankovič, and Yunquan Zhang. Modeling the Locality in Graph Traversals. In *Proceedings of the International Conference on Parallel Processing*, ICPP '12, pages 138–147, 2012. doi:10.1109/ICPP.2012.40.

Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. Fast Iterative Graph Computation: A Path Centric Approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 401–412, 2014. doi:10.1109/SC.2014.38.

Noel Yuhanna, Boris Evelson, Brian Hopkins, and Emily Jedinak. Forrester TechRadar: Enterprise DBMS, 2014. `http://www.forrester.com/TechRadar+Enterprise+DBMS+Q1+2014/fulltext/-/E-RES106801` (Last accessed: April 2017).

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, pages 10–10, 2010.

Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware Graph-structured Analytics. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 183–193, 2015. doi:10.1145/2688500.2688507.

Shijie Zhang, Meng Hu, and Jiong Yang. TreePi: A Novel Graph Indexing Method. In *Proceedings of the International Conference on Data Engineering*, ICDE '07, pages 966–975, 2007. doi:10.1109/ICDE.2007.368955.

Shijie Zhang, Shirong Li, and Jiong Yang. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of the International Conference on Extending Database Technology*, EDBT '09, pages 192–203, 2009. doi:10.1145/1516360.1516384.

Yaonan Zhang, Eric D. Kolaczyk, and Bruce D. Spencer. Estimating Network Degree Distributions Under Sampling: An Inverse Problem, with Applications to Monitoring Social Media Networks. *CoRR*, 2013.

Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph Indexing: Tree + Delta >= Graph. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '07, pages 938–949, 2007.

Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing Billion-node Graphs on an Array of Commodity SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '15, pages 45–58, 2015.

Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1323–1334, 2014. doi:10.1145/2588555.2612181.

Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference*, ATC '15, pages 375–386, 2015.