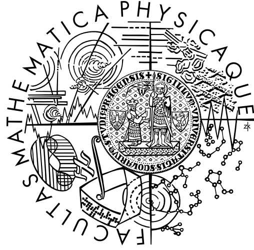


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Mgr. Tomáš Faltín

Distributed Graph Query Engine Improvements for Big Data Graphs

Department of Software Engineering

Supervisor of the doctoral thesis:	RNDr. Jakub Yaghob, Ph.D.
Advisor of the doctoral thesis:	Vasileios Trigonakis, Ph.D.
Advisor of the doctoral thesis:	Jean-Pierre Lozi, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2023

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedicated to my grandparents Stanislav, Marie, Tomáš, and Marie.

I would like to thank my supervisor Jakub Yaghob for his help, support, and reactivity during all of my PhD. I am very grateful for the friendly and sympathetic environment that he created together with other people from the department of software engineering.

A big thanks goes to all people from Oracle Labs, especially people from the PGX.D team, where I spent a big part of my Ph.D. studies as an intern. They introduced me to the exciting field of distributed graph querying, which eventually led to this thesis. A special thanks to my mentor Vasileios Trigonakis who was leading me and was a great model and leader to follow.

While I was at Oracle Labs, I met Jean-Pierre Lozi, who became another important person who helped me and advised me with research and my work on my thesis. He introduced me to the research world outside Czechia by giving me the opportunity to work with him and other great researchers at Inria Paris while still finishing and working on my thesis. He and other people in the Whisper team deserve a big thanks for giving me enough time to finish my thesis.

Last but not least, finishing the thesis would not have been possible without my family, friends, and my girlfriend Aneta, who have had patience and have given me endless support during these past years.

Title: Distributed Graph Query Engine Improvements for Big Data Graphs

Author: Mgr. Tomáš Faltín

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering

Advisors: Vasileios Trigonakis, Ph.D. and Jean-Pierre Lozi, Ph.D.

Abstract: Graph pattern matching queries enable flexible graph exploration, similar to what SQL provides for relational databases. In this thesis, we design and improve key components of a distributed in-memory graph querying engine. First, we optimize a distributed depth-first search (DFS) asynchronous pattern matching algorithm by combining it with breadth-first search (BFS), thus improving the overall engine performance by leveraging strengths of both approaches: ability to strictly bound the consumed memory of DFS and better parallelization, locality, and load balancing of BFS. Second, we further extend the distributed pattern matching with a novel solution for reachability regular path queries (RPQs) that supports variable-length patterns based on regular expressions. Our design retains the underlying runtime characteristics, allowing for efficient memory control during path exploration with great performance and scalability. Third, we improve query planning, which is one of the most crucial aspects impacting the performance of any querying system. Choosing the "best" query plan is challenging due to the many aspects influencing the performance, especially in a distributed system. We present a lightweight mechanism for gathering runtime information, which can be used to select the most effective query plan that performs optimally in the actual environment where the engine executes.

Keywords: distributed graph querying, distributed graph processing, distributed graph databases

Contents

1 Introduction **4**

2 Background **10**
2.1 Graph Data Models 11
2.1.1 Property Graph Model 11
2.1.2 Graph Related Models 13
2.2 Graph Querying 15
2.2.1 Subgraph Matching 16
2.2.2 Graph Data Structures 22
2.2.3 Graph Querying Systems 23
2.3 Other Approaches 26
2.3.1 Graph Algorithms 26
2.3.2 Graph Mining 27
2.4 Graph Query Languages 28
2.4.1 Querying and Pattern Matching 28
2.4.2 Grouping, Aggregation, and Sorting 30
2.4.3 Variable-Length Path Pattern Matching 31
2.5 Benchmarking 32
2.5.1 LDBC 32
2.5.2 TPC-H 33

3 PGX.D/Async Query Engine **34**
3.1 Architecture 34
3.1.1 Communication Management 34
3.1.2 Data Management 35
3.2 Query Planning 36
3.3 PGX.D/Async Runtime 37
3.3.1 Query Termination 40
3.3.2 Flow Control 41
3.4 Relational Post-Processing Operations 41
3.5 Evaluation 42
3.5.1 Performance Improvements with Ghost Vertices 42
3.5.2 Flow Control Performance 43

4 An Almost Depth-First Search Distributed Graph-Querying **45**
4.1 Introduction 45
4.1.1 Related Work 47
4.2 aDFS: A Pattern Matching and Querying for Distributed Graphs . . . 48
4.2.1 High-Level aDFS Architecture 49
4.2.2 aDFS Runtime 49
4.3 Evaluation 51
4.3.1 Experimental Settings 51
4.3.2 Benefits of Local BFS 53
4.3.3 Dissecting aDFS 54
4.3.4 aDFS vs. Other Engines: LDBC 55

4.3.5 aDFS vs. Other Engines: Large Schema-Less Queries 57
4.3.6 aDFS vs. Graph Mining, Dataflow Joins 58
4.3.7 aDFS Scalability 59
4.3.8 aDFS Parallel Execution 60
4.4 Concluding Remarks 60

5 Distributed Asynchronous Reachability Regular Path Queries 62
5.1 Introduction 62
5.1.1 Reachability Queries 63
5.1.2 Regular Path Queries. 64
5.2 Distributed Reachability RPQs 64
5.2.1 Query Planning 64
5.2.2 RPQd Execution Runtime 66
5.2.3 Messaging 67
5.2.4 Termination Protocol 67
5.2.5 Flow Control 68
5.2.6 Reachability Index 68
5.3 Evaluation 69
5.3.1 Experimental Settings 69
5.3.2 RPQd vs. Other Systems 70
5.3.3 RPQd Scalability 71
5.3.4 Q9 and Q10 in Detail 71
5.3.5 Effects of Reachability Index 72
5.4 Concluding Remarks 73

6 Better Distributed Graph Query Planning With Scouting Queries 74
6.1 Introduction 74
6.1.1 Query Planning 75
6.1.2 Related Work 76
6.2 Scouting Queries 77
6.2.1 Query Planning to Scouting Query Execution 77
6.2.2 Reusing Scouting Query Results 79
6.3 Evaluation 80
6.3.1 Experimental Settings 81
6.3.2 Results 82
6.4 Concluding Remarks 85

7 Conclusion 86

Bibliography 88

List of Publications and Patents 108

A LDBC Social Network Benchmark Business Intelligence Queries 110
A.1 Fixed-size Pattern Queries 110
A.1.1 Cypher 110
A.1.2 PostgreSQL 114
A.2 RPQ Queries 120
A.2.1 Cypher 120

A.2.2 PostgreSQL 124

1. Introduction

In the recent years, collections of graph-structured data are increasingly being made available for analysis [1]. Examples of such graph data are different variants of networks [2]: social, biochemical, ecological, citation, communication, mobility, and transport. Graphs provide a natural representation and visualization of data, allowing us to easily represent the interactions between data elements through relations.

A great example of a graph is a social network. Here, people can be represented as graph vertices, and the relations "a person knows another person" can be represented as graph edges. Persons can have different properties, including their name, age, address, and hobbies. Similarly, the edges can store information such as the place and date of the persons' first meeting. In addition to person-to-person connections, the graph can also include vertices representing posts and comments, with the messages stored within them as properties. These vertices can be connected to persons through the "a person is the author of a post/comment" relation, forming edges called authors. An example of a similar graph is depicted in Figure 2.1. This type of graph is referred to as a *property graph*, which we define later in Section 2.1.1.

Hand-in-hand with the graph-data availability goes the need for the graph systems allowing an analysis of the data, which tries to extract meaningful insights and information from it. To interact with graph systems for the purpose of analysis, the systems provide an interface through query languages, such as PGQL [3], Gremlin [4], GSQL [5], and Cypher [6]. The large number of query languages highlights the eminent interest in graph analysis. The novelty and relevance of this topic underlines the release of a new standard for SQL/PGQ [7] in June 2023, which is an SQL extension for graphs, and a work on GQL [8], which is the new standard query languages for graphs with potential standardization in early 2024.

For example, social network analysis can be interested in simple queries like "Who knows Jan Novak?", in PGQL:

```
SELECT friend.name
FROM MATCH (p:Person) -[:Knows]-> (friend:Person)
WHERE p.name = 'Jan Novak'
```

, or in much more complex queries, such as "Who is interacting with the posts of a person who he/she does not know, and whose age is within the range ± 5 years?", in PGQL:

```
SELECT
  person1.name AS person,
  person2.name AS potential_friend
FROM
  MATCH (person1:Person) -[:Author]-> (post:Msg),
  MATCH ANY (comment:Msg) -[:Reply]->* (post),
  MATCH (comment) <-[:Author]- (person2:Person),
WHERE
  NOT EXISTS (SELECT * FROM MATCH (person1) -[:Knows]-> (person2))
  AND person1.age - 5 <= person2.age
  AND person2.age <= person1.age + 5
```

to recommend new potential friends in the network, or "Is Jan Novak connected to everyone else by a chain of no more than 6 acquaintances?", in PGQL:

```
SELECT COUNT(*) = (SELECT COUNT(*) FROM MATCH (:Person))
FROM MATCH ANY (p:Person) -[:Knows]->{1,6} (other:Person)
WHERE p.name = 'Jan Novak'
```

to validate the famous *Six Degrees of Separation*[1] hypothesis [9, 10] for a given person.

As can be seen from the examples, graphs, and especially graph querying, give emphasis on the edges, i.e., relations between vertices. Graphs allow expressive and efficient search for long connections and patterns compared to other models, like RDF [11] or the relational model (both presented later in Section 2.1). These models allow a graph representation that is quite cumbersome, not that intuitive, and potentially not that efficient.

Graph-specific workloads have inherent characteristics in the context of data processing. According to [12, 13, 14], these are the challenges that an effective engine needs to overcome:

- *Exponential explosion.* Graph querying is hard. It is equivalent to the problem of searching for subgraph homomorphisms that is known to be a NP-complete problem (Section 2.3). Some instances of this problem that are easier to solve, e.g., isomorphic patterns with a special metric, are domain of graph mining (Section 2.3.2). Graph querying, being NP-complete, simply implies that there is an exponential number of intermediate results with the length of a query that must be processed. Therefore, the computation needs to be handled with care, otherwise the intermediate results explode and it will not fit into the main memory.

- *Poor locality.* Graphs focus on relations, which are unstructured and irregular. Edges drive the computation, and following the edge from the source to the destination typically requires a jump to a distant memory, leading to poor spatial location. With generic expression *filtering*, i.e., some edges or vertices are filtered out during the execution, the computation can become irregular, which complicates the work of hardware as it does not know where the computation will go.

- *Memory bound computation.* Graph querying is dominated by memory accesses. The querying consist of two operations: (i) graph structure exploration and (ii) graph element filtering. The graph exploration implies reading of the graph structure from memory and using the read information for further memory accesses. The graph element filtering involves evaluation of (simple) expressions whose initial data, i.e., properties, need to be loaded from the memory as well.

The aforementioned challenges show that an efficient graph data processing is complex by its nature. Other challenges arise when the engine wants to fully utilize modern CPUs that have multiple physical and virtual cores. Following graph-specific factors complicate the graph-processing parallelizations, as mentioned in [12, 13, 14]:

- *Data-driven computation.* Graph querying is data driven. The computation is dictated by the shape of the graph, i.e., its vertices and edges, but mostly by the user-submitted query, which can include a generic pattern with any dynamic filtering. As a result, parallelization based on partitioning of the computation work is hard and cannot be easily predicted because of the dynamic filtering.

- *Unstructured problem.* As already mentioned, graphs are typically unstructured and irregular. On top of that, real-world graphs, e.g., social network graphs,

[1] This idea appeared first as a game in 1929 and was popularized by the play *Six Degree of Separation* [9] in 1990. The play explored the premise that every person is connected to every other person through a chain of no more than six acquaintances.

might be skewed following a power-law distribution, making the neighborhood of some *super-vertices* extremely large, e.g., celebrities. On the opposite, some vertices, e.g., newcomers or not-so-active users, might have small number of connections.

Big data graphs emphasize all the aforementioned challenges. Big data means that the data are so big they do not fit in the main memory of a single machine [15]. One of the possibilities is to use main storage, e.g., HDD or SSD. The unused data is typically offloaded to the storage and brought to main memory when needed for the computation. Some popular graph databases, e.g., Neo4j [16] and Tigergraph [17], are using this approach. For obvious reasons, the *buffer manager*, i.e., the subsystem for handling the data offloading, is one of the major bottlenecks in these systems. This is a well-known problem in relational databases [18, 19] and graph databases are not an exception.

Another possibility, how to handle big data graphs, is to store graphs in the main memory, but scale out to multiple machines in a distributed matter. The advantage of this approach is that the engine can use multiple machines for the computation, which can result in better performance if the system scales well. This approach drew the attention of many researchers and led to development in the field of distributed graph processing [20, 21, 22, 23, 24, 25, 26, 27]. Given the popularity of cloud computing, being able to run a distributed graph engine on multiple machines and scale out when needed, these systems become interesting for industry [13, 28, 29, 17] as well.

However, the development of an efficient distributed system is difficult. In addition to all the challenges mentioned above, which need to be scaled to the entire cluster, there is a problem of *distributed memory* where the graph is partitioned across all machines. This means that each machine has access to only part of the graph. To minimize query latency, distributed systems need to efficiently balance graph partitions, which is NP-hard [30], and distribute computation equally among partitions. Because the computation is data-driven, even perfectly balanced data partitions do not guarantee well-balanced computation.

The first contribution of this thesis. (Distributed) graph querying uses two main approaches for pattern matching: breadth-first search (BFS) and depth-first search (DFS). Both approaches have their advantages and disadvantages. BFS is a typical approach used by most distributed graph querying systems [16, 31] for two main reasons. Implementing an efficient distributed version of BFS is much easier compared to DFS. Furthermore, BFS has a much better spatial locality of the IO patterns compared to DFS, which implies better performance in practice. On the other hand, BFS results in large memory consumption, which makes it suitable only for systems that do not have to limit their memory, e.g., due to data offloading to the main storage.

In contrast, limiting and controlling the memory consumption is a crucial element for systems running in the main memory only. DFS allows for limited memory consumption, as it matches one result at a time. Unfortunately, this traversal also implies a much worse spatial locality caused by eagerly following the chain of edges, basically doing pointer-chasing in the memory. On top of that, parallelization and implementation of DFS is much harder than of BFS. The first relevant implementation of a distributed DFS for querying was done by PGX.D/Async [32] that introduced an asynchronous DFS query engine that uses a local DFS with message buffering of remote edges.

In this thesis, we extend this idea further and, as our first contribution, we present ***aDFS (almost-DFS)***, a novel distributed graph querying algorithm that cleverly combines breath-first search (BFS) and depth-first search (DFS) to be able to use efficient parallel processing of graphs partitioned across multiple machines fully in-memory, while bounding the maximum amount of memory used during execution.

To the best of our knowledge, aDFS is the first graph querying system capable of strictly bounding runtime memory while operating with fully distributed in-memory computations over distributed graphs. It builds on top of PGX.D/Async [32] (Section 3) to cap the maximum amount of memory used during execution, while switching to BFS for better locality and parallelization during execution.

The second contribution of this thesis. In addition to the performance improvement, we extend the PGX.D/Async engine in another major way. As already mentioned, query languages are evolving as users find new use cases for graphs. One of the most expressive and powerful constructs in graph querying is the *regular path queries (RPQs)* [33]. RPQs allow users to specify different path patterns using regular expressions. For example, following query searches for "A number of all replies and their first and last dates starting from a given message", in PGQL:

```
SELECT
  COUNT(*) AS num_replies
  MIN(reply.date) AS first_date,
  MAX(reply.date) AS last_date
FROM MATCH ANY (:Msg) -[reply:Reply]->+ (msg:Msg)
WHERE msg.id = 104
```

PGQL combines RPQs together with *reachability queries* in the form of a search for *any variable-length paths* [34]. Reachability queries search for all the vertices reachable from a given source vertex. The semantics of PGQL allows finding any path from the source to the destination vertex, and this path is accounted only once in the final result projections. This requires eliminating path duplications and also allows the engine to avoid potential infinite cycles.

Our second contribution of this thesis is **RPQ*d***, a novel algorithm for distributed asynchronous reachability regular path queries (RPQs). RPQd builds on the work of PGX.D/Async (Section 3), extending its DFS querying engine, which allows for controllable memory consumption with a small memory footprint. However, the nature of the problem that requires support for unbounded RPQs, e.g.,

```
SELECT ARRAY_AGG(e.from)
FROM MATCH (:Person) -/e:Knows*/-> (:Person)
```

, does not allow one to fully limit the consumed memory of the matching, as aDFS does for the fixed-size matching. Therefore, our solution caps the memory consumption of the variable-size matching execution and limits the additional overflow memory to an absolute minimum in order to continue with the execution of the matching. Other algorithms, typically using some sort of shortest path algorithm, cannot guarantee this. A predictable memory footprint is required for an efficient configuration of the system and its deployment, e.g., to the cloud.

In addition to that, RPQd builds a reachability index on the fly to detect cycles in the graph and to avoid infinite loops during traversals. Moreover, our design allows the support of generic cross-filtering between variable- and fixed-size patterns. Most engines build the reachability graph/index before running the query; therefore, they cannot support generic cross-filtering. To the best of our knowledge, RPQd is the first distributed querying system that allows such feature.

The third contribution of this thesis. As was already presented through this section, an efficient distributed graph querying must overcome various challenges. A clever design of a system can help overcome some challenges, as shown in aDFS and RPQd. However, the execution depends on the complexity of the user-submitted query. The process of translating the submitted query into the execution plan is called *query planning* and dictates the order of the pattern matching operations. Optimal ordering of operations is essential for a good performance of the entire query execution.

Typically, query planning is done analytically with a potential to learn from run queries. However, distributed graph querying poses numerous challenges that are hard to cover analytically, if not outright impossible. For example, it is not easy to model the data partitioning among different machines. Also, moving data across machines requires messaging and communication that is difficult to include in the analytical model. Furthermore, all these properties can change in dynamic environments, such as in the cloud.

As our third contribution, we propose a pragmatic solution that builds on top of the standard query planner to improve the chance of finding the best execution query plan. Our solution uses ***scouting queries***, which are short exploratory executions of the actual queries used for quick benchmarking of different query plans. Thanks to that, the engine can make an informed decision on which query plan to pick.

Organization of the thesis. The doctoral thesis is organized as follows:

- **Chapter 2** discusses the background and related work regarding graph processing with a focus on graph querying. The background and related work of the topics related to our contribution is presented in respective chapters.

- **Chapter 3** introduces PGX.D/Async [32], a distributed graph querying system, which was used to evaluate all the improvements presented in this thesis.

- **Chapter 4** introduces aDFS, a novel design for distributed graph querying that allows efficient processing of practically any query fully in memory while maintaining bounded runtime memory consumption.

  This chapter is an extension of the published paper: *Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, Hassan Chafi. "aDFS: An Almost Depth-First-Search Distributed Graph-Querying System" [14] USENIX ATC '21.*

- **Chapter 5** presents a novel design for distributed reachability and regular path queries that enables memory-controlled path explorations.

  This chapter is an extension of the accepted paper: *Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Jinsoo Lee, Jakub Yaghob, Sungpack Hong, Hassan Chafi. "Distributed Asynchronous Regular Path Queries (RPQs) on Graphs" International Middleware Conference Industrial Track 2023.*

- **Chapter 6** introduces a solution to improve distributed query planning by executing short explanatory queries to benchmark the performance of different query plans.

This chapter is an extension of the published paper: *Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Sungpack Hong, Hassan Chafi. "Better Distributed Graph Query Planning With Scouting Queries" [35] GRADES&NDA ́23.*

- **Chapter 7** concludes this thesis and discusses potential future work of graph querying. The future work of our contributions are described inside respective chapters.

The parts of the text that appear in one of the above-mentioned papers are highlighted with a vertical line of a respective color on the right side, as illustrated in this paragraph.

2. Background

Graphs, mathematical structures used for modeling relations between objects, provide a natural and intuitive representation and visualization of a wide range of data. Informally, graphs consist of elements called vertices and relations between the elements called edges. As the name "graph" suggests, graphs can be visualized with dots representing elements and lines symbolizing relations between them.

There are many examples and applications of such data that played a large part in motivating the creation of special graph data models, graph engines, and graph databases. According to [36], the most important examples of such data are:

- *Social networks [37, 38, 39]*, where vertices represent people or a group of people, and the edges represent various types of relations between them, such as friendships, business relationships, collaborations, or communications.

  In addition to the relations, persons on the social network can have multiple properties, including age and name, which provide further information about them. Besides the person vertices, the network may also include other types of vertices that arise due to interactions between people, such as messages, posts, and comments.

  To differentiate between the different types of vertices, we can assign labels to each individual type. This labeling allows us to identify and categorize the vertices based on their respective roles and characteristics within the social network.

  We can see an example of a social network graph in Figure 2.1.

- *Information networks*, where the network represents information flow, for example the World Wide Web [40, 41, 42, 43], peer-to-peer networks [44, 45], preference networks [46], and knowledge-based graphs [47, 48].

- *Technological networks [49]*, where the graph corresponds to the spatial or geographical property of the data, for example, the Internet, telephone networks, or data modeled by Geographical Information Systems (GIS), which represent roads, such as car roads, airline roads, railways, rivers, and pedestrian roads.

- *Biological networks*, that represent biological data, such as BioGRID [50] or UniProtKB [51]. These networks typically cover chemical structures, neural networks, or some data from the genomic field. For example, biological networks can represent proteins as vertices and interactions between them as edges.

Each of these types of graph requires and focuses on a different type of analysis. For example, social networks might focus on distance, neighborhoods, clustering coefficient, connected components. Geographical Information Systems might focus on geometrical operations (intersection, inclusion, etc.) or topological and metrics operations (connectivity, distance, etc.). On the contrary, biological graphs focus more on pattern matching of specific patterns, querying, and similarity search.

Current interest in the field of graph analytics is evident through the large number of publications, including surveys and books, that cover various aspects of this topic. These publications span a wide range of areas, from surveys on graph data models and query languages [36, 52, 53, 54, 55], to graph mining [56, 21, 57, 58], graph

databases and graph querying [1, 59, 60, 61, 62, 63, 64, 65, 66], and distributed graph processing [67, 68, 69, 70].

Organization. In this chapter, we present all the necessary background information related to graph querying. We start by presenting the property graph model and other related graph models in Section 2.1. Section 2.2 explains the problem of graph querying in detail, and following this, Section 2.3 introduces other graph processing fields closely related to querying, such as graph graph mining and graph algorithms. Section 2.4 presents query languages and describes one of the languages, PGQL [3], which is used throughout the thesis. Finally, Section 2.5 explains the relevant graph benchmarks that we use for evaluations and benchmarking in this thesis.

2.1 Graph Data Models

A data model is *"a collection of conceptual tools used to model representations of real-world entities and the relationships among them"* [71]. In this section, we introduce a *property graph model*, which is the native graph model, along with other data models that can model graphs and are used in some relevant systems in practice, e.g., RDF [11], relational [72], and NoSQL [73] models. There exist other theoretical graph models, such as GOOD [74] or GROOVY [75], but to the best of our knowledge, they are not used by any relevant graph system at the moment. For a complete description of all graph data models, we refer to various surveys, such as [36, 53, 76, 77, 78].

2.1.1 Property Graph Model

Property graph can be informally described as follows [60]:

- It contains vertices and edges.
- Edges have names and directions with a starting and an end vertex.
- Vertices and edges can contain properties, i.e., key-value pairs.
- Vertices and edges can be labeled with one or more labels, i.e., key-only values.

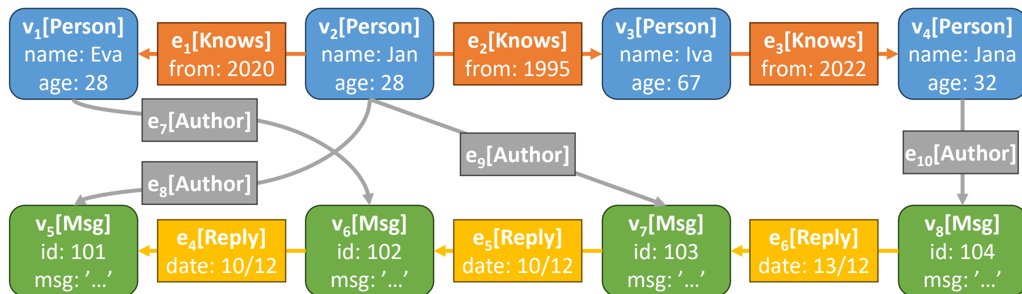


Figure 2.1: A social network example graph G_{ex}.

Figure 2.1 shows a running example of a social network graph G_{ex}. In blue, we see four vertices $\{v_1, v_2, v_3, v_4\}$ representing persons and in orange, we see edges describing a "a person knows another person" relationship. Every person has a label `Person`, a string property `name`, and an integer property `age`. Each orange edge has a label `Knows` and a property `from` storing a year when the two persons started to know each

other. The vertices in green $\{v_5, v_6, v_7, v_8\}$ represent messages that are sent in the network. Every message contains a label `Msg`, a unique identifier `id`, and a string property `msg` that contains the actual message. If "a message is a reply to another message", it is connected through a yellow edge having a label `Reply` and a property `date` storing a date of the message. The gray edges, with the `Author` label, represent relations "a person is the author of a message".

For the purpose of this work, we formally define a *property graph* [53] as follows:

Definition 1 (Property graph). *Assume that L is an infinite set of labels (for vertices and edges), P is an infinite set of property names, V is an infinite set of values, and T is a finite set of datatypes (e.g., integer). Given a set X, we assume that SET+(X) is the set of all finite subsets of X, excluding the empty set. Given a value $v \in V$, the function $type(v)$ returns the data type of v.*

A property graph is a tuple $G = (N, E, \rho, \lambda, \delta)$ where:

- *N is a finite set of vertices.*
- *E is a finite set of edges such that E does not have elements in common with N.*
- *$\rho : E \rightarrow (N \times N)$ is a total function that associates each edge in E with a pair of vertices in N, i.e., ρ is the incidence function in graph theory.*
- *$\lambda : (N \cup E) \rightarrow$ SET+(L) is a partial function that associates a vertex/edge with a set of labels from L, i.e., λ is a labeling function for vertices and edges.*
- *$\delta : (N \cup E) \times P \rightarrow$ SET+(V) is a partial function that associates vertices/edges with properties, and assigns a set of values from V to each property.*

Using this Definition 1 and the running example from Figure 2.1, we have $G_{ex} = (N, E, \rho, \lambda, \delta)$, where $N = \{v_1, v_2, ..., v_8\}$, $E = \{e_1, e_2, ..., e_{10}\}$, $L = \{$*Person*, *Msg*, *Knows*, *Author*, *Reply*$\}$, $P = \{$*name,age,id,msg,date,from*$\}$, ρ is defined as visualized as shown by the edges in the Figure (e.g., $\rho(e_1) = \{v_2, v_1\}, \rho(e_2) = \{v_2, v_3\}, \rho(e_3) = \{v_3, v_4\}$), λ is defined as visualized with colors (e.g., $\lambda(v_1) =$ *Person*, $\lambda(v_5) =$ *Msg*, $\lambda(e_1) =$ *Knows*, $\lambda(e_4) =$ *Reply*, $\lambda(e_7) =$ *Author*), and δ is defined as: $\delta(v_1, name) =$ *Eva*, $\delta(v_1, age) = 28$, $\delta(v_5, id) = 101$, and so on.

The property graph model incorporates various characteristics [1] that other property graph models can extend or restrict:

- *Direction.* All the edges are ordered pairs of vertices, i.e., edges have direction. Note that there also exist undirected graphs models, where the edges are unordered pairs of vertices. Using the presented property graph model, the undirected graphs can be emulated by including edges in both directions. The engine shown in the thesis natively supports only directed edges.
- *Multigraph.* The model allows multiple edges between the same pair of vertices. Some other graph models can prohibit multiple edges, because it simplifies the modeling. The engine in this thesis supports multigraphs.
- *Labels.* The model allows vertices and edges to have zero or more labels. There are models that allow exactly or at maximum one label per element. Other models can restrict the model by allowing labels for either vertices or edges only. The system in this thesis supports any number of labels for both edges and vertices.

- *Properties.* Each vertex or edge can contain any number of properties, i.e., key-value pairs. Data-attributed models [1] allow values to be only a chunk of binary data, i.e., $\{0,1\}^*$. Other models allow only certain types of properties, either vertex or edge properties. The engine described in the thesis supports any number of properties for vertices and edges.

Property graph extensions. The engine described in the thesis supports only the standard property graph model presented before. However, there are various extensions of the basic property graph, which we list here to provide a complete picture of the property graph model. More details and definitions can be found in books and surveys [36, 1, 79]. Some relevant property graph extensions are:

- *Objectified paths* that extend the model by allowing assigning properties and labels to *paths*, i.e., sequences of edges.
- *Objectified subgraphs* that allow assigning properties and labels to whole subgraphs.
- *Hypervertices* that allow edges between any subgraph of the graph.
- *Hyperedges* that generalize edges as sets of one or more vertices. Note that this model is useful for specialized use-cases, such as spectral clustering of relational data [79].

Temporal property graphs. A temporal property graph [80, 81, 82, 83] is a native graph model that takes into account the evaluation of the graph, i.e., changes of the graph structure and properties over time. The standard non-temporal graph model views the graph and properties as static, with a potential support for insertion, deletion, or updates. However, the temporal property graph model treats all the changes as the first class citizens. Given the complexity of the topic, there is not yet agreement on the best approach [80, 84, 85, 86], and it is one of the interesting research areas related to graphs these days. One can see that property graph models undergo a similar evolution as SQL that incorporated the temporal model in 2011 [87, 88, 89].

2.1.2 Graph Related Models

Besides the native graph data models above, there exist also other models that present graph-like features and allow modelling graphs, although they are not explicitly designed for that. We present here the most relevant models, as well as the models used by the engines that are used for comparison with our engine.

RDF Data Model

The *Resource Description Framework (RDF) [11, 90, 78]* was introduced by the W3C for the purposes of the semantic web and has been used by different domains since then, e.g., bioinformatics, social networks. It was the first model of choice for modeling graphs in the early 2000s [91, 92]. RDF defines a mechanism for annotating resources with triples *subject*, *predicate* and *object*, i.e., a resource (subject) is described by a property (predicate) and a property value (object). The subject can be an identifier called *URI* (*Uniform Resource Identifier*) or an empty node. The object is a URI, an empty node, or a literal value. The predicates can be URIs only. In this text, we refer to URI of an value as `URI(x)`.

There is also the RDF* extension [93, 94, 95] which allows more efficient encoding by allowing triples to appear recursively as subjects or objects. For simplicity, we assume the basic RDF model.

The property graph can be represented by RDF as follows [78, 95]:

- Vertices are mapped to URIs.
- An edge `e={u,v}` is connected through two RDF triples: `{URI(u),URI("FROM"),URI(e)}, {URI(e),URI("TO"),URI(v)}`.
- Every vertex property `{key,value}` is connected with its owner vertex `v` through the RDF triple `{URI(v),URI(key),URI(value)}`.
- Every edge property `{key,value}` is connected with its owner edge `e` through the RDF triple `{URI(e),URI(key),URI(value)}`.
- Every vertex label `{VL}` is connected with its owner vertex `v` through the RDF triple `{URI(v),URI("label"),URI(VL)}`.
- Every edge label `{EL}` is connected with its owner edge `e` through the RDF triple `{URI(e),URI("label"),URI(EL)}`.

Figure 2.2 shows an example of the transformation from the property graph model into the RDF model. On the left, we see a small property graph (a small part of the graph from Figure 2.1), on the right, we see the RDF graph. Subjects and objects are represented as vertices (the source vertex as the subject, the destination vertex as the object), and predicates are represented as edges with the appropriate label.

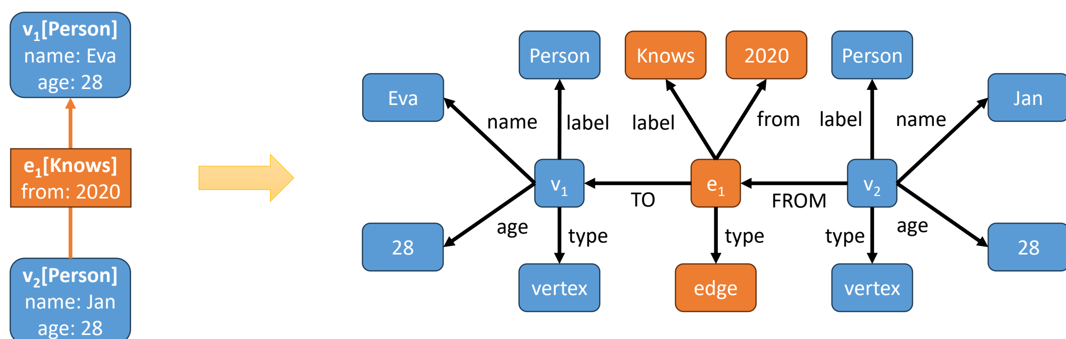


Figure 2.2: An example of a graph stored with the RDF graph model. All the presented values are URIs.

Relational Data Model

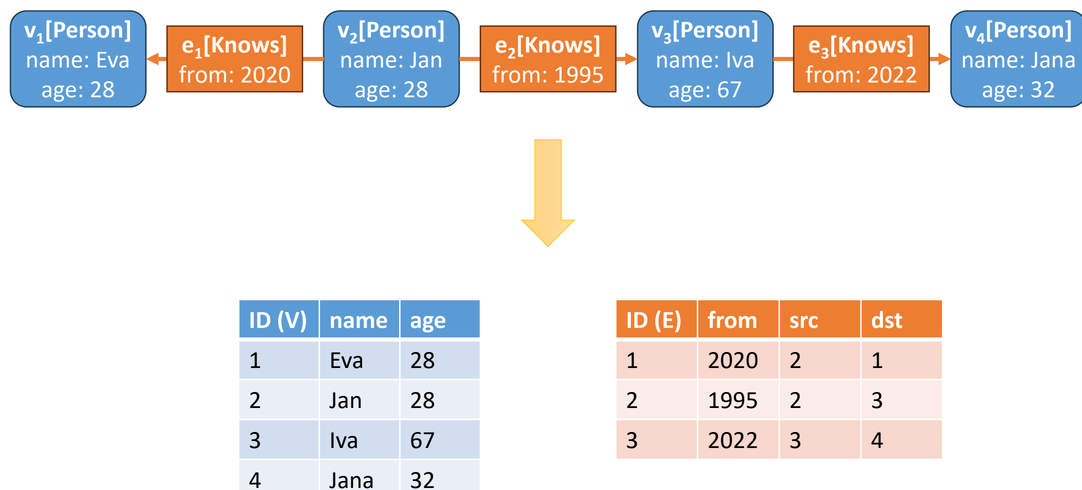
The relational data model was one of the first data models [72]. It is consistent with predicate logic, where all data are represented as tuples and grouped into different relations. Data are typically stored in tables where each row represents an element and columns typically represent attributes of that element. For defining relations between data, table can contain a special column (or more columns in case of *composite keys*) with a unique identifier per element, called a *primary key*. To implement one-to-one or one-to-many relations, another table includes a "special copy" of the primary key

called a *foreign key* which connects the relations. Many-to-many relations are implemented with a dedicated table containing all the foreign keys of the relation elements.

A property graph can be easily represented by relational data model [96] using following approach:

- Vertices and edges each are stored in individual tables, where every column represents one key of their properties.
- Vertex and edge labels can be stored as additional columns.
- The edge table has two additional columns for storing the source and destination vertices as foreign keys.

Figure 2.3 shows an example of a graph stored in the relational data model. The blue table represents the vertices, and the orange table represents the edges. The `src` column stores an index (`ID(V)`) of the source vertex, the `dst` column stores an index (`ID(V)`) of the destination vertex. The columns `name`, `age` and `from` represent the properties.



ID (V)	name	age
1	Eva	28
2	Jan	28
3	Iva	67
4	Jana	32

ID (E)	from	src	dst
1	2020	2	1
2	1995	2	3
3	2022	3	4

Figure 2.3: An example of a graph stored with the relational model.

NoSQL Data Models

There are various NoSQL databases that allow representation of graphs. Unlike the previous models, there is typically no single recommended way how to represent graphs within a specific NoSQL model. Therefore, the representation of graphs is specific to the implementation of the database. For a more thorough analysis and description of the models and databases, one can refer to various surveys, such as [59, 36, 53, 76, 77, 78].

2.2 Graph Querying

Graph queries are a key tool for graph analysis, as indicated by the large number of existing systems and graph-query languages (presented in later sections). Graph queries provide an expressive interface for interactive graph exploration with rich dynamic projection and filtering support that is analogous to SQL for relational databases. They

focus on data connections, i.e., edges, allowing users to submit queries with any pattern, filter, or projection. For instance, the following PGQL [3] query:

```
SELECT p1.name, p2.name
FROM
  MATCH (p1:Person)-[:Knows]->(p2:Person),
  MATCH (p2)-[:Knows]->(p1)
WHERE ABS(p1.age - p2.age) <= 10
ORDER BY salary_diff DESC
```

enumerates the `Persons` of similar age that `Knows` each other. Answering such a query requires finding all *homomorphic matches* (explained later in this section) of the query pattern in the target graph, while enforcing filters (e.g., `p1 IS Person`) and projecting the requested output, i.e., persons' names. The use of homomorphic subgraph matching, together with support for general projections and filtering limits the potential optimizations and requires an exhaustive search of all possible combinations. In general, the homomorphic subgraph matching problem is NP-complete [1].

A query consists of two main parts: *graph pattern matching* and *post-processing*. Graph pattern matching includes the subgraph search and projections of the returned matches. The post-processing step includes all transformations done on top of the found results:

- *Aggregations with grouping (*`GROUP BY`*)* that groups results into groups according to a certain expression and returns aggregated results for each group.
- *Group filtering (*`HAVING` that filters the aggregated groups according to a given filter.
- *Ordering (*`ORDER BY`*)* that is responsible for sorting the results according to specified criteria.

The post-processing step is presented separately, because in most systems it is separated from the process of pattern matching and is handled by two different subsystems. Performing post-processing after the matching is equivalent to the post-processing in relational databases, which is a well-researched and covered topic outside of the scope of this thesis. Despite all this, there are some cases of aggregations, e.g. `COUNT(*)` or generic (simple) accumulators in GQL [97], that the systems perform during pattern matching for performance reasons and because of its simplicity.

2.2.1 Subgraph Matching

Subgraph matching is the core functionality of graph querying. Given a graph G and a query q, the goal of a *subgraph pattern matching query* is to return all subgraphs of G that are isomorphic or homomorphic to q.

Now we formally define the problem of subgraph pattern matching queries. In theory [1], the queries are called *conjunctive graph queries*. In this thesis, we focus only on such queries. Therefore, we will refer to them simply as *queries* For a more detailed description and other types of queries, please refer to [1].

A subgraph query pattern is a set of edge predicates. Every predicate consists of an edge label and a pair of vertex variables, representing the source and the destination of the edge. Note that in practice, we often see queries without labels. For the sake of theory in this thesis, we can safely assume that all edges have a unique label `_EDGE_`.

Formally, we define a graph query as follows:

Definition 2 (Graph query). *Let $\mathcal{V}$ be a set of vertex variables, the conjunctive graph queries are all expressions of the form:* $(z_1, z_2, ..., z_m) \leftarrow a_1(x_1, y_1), ..., a_n(x_n, y_n)$, *where*

- $m \geq 0, n > 0$,
- $x_1, y_1, ...x_n, y_n \in \mathcal{V}$ *(vertices),*
- $a_1, ..., a_n \in L$ *(labels),*
- *for each* $0 < i \leq m$*, it holds that* $z_i \in \{x_1, y_1, ..., x_n, y_n\}$

m *is called the arity of the expression.*

The query semantics is given by the *variable bindings* to graph vertices. A set of variable bindings is valid iff all the predicates hold on the data graph. Formally:

Definition 3 (Query mapping). *Let* $G = (V, E, \rho, \lambda, \delta)$ *be a property graph and let* $q = (z_1, z_2, ..., z_m) \leftarrow a_1(x_1, y_1), ..., a_n(x_n, y_n)$ *be a conjunctive graph query.*

A mapping for q *on* G *is a function* μ *with domain* $\mathcal{V}$ *(vertex variables) and range* V *such that, for each* $1 \leq i \leq n$*, there exists an edge* $e_i \in E$ *where* $\rho(e_i) = (\mu(x_i), \mu(y_i))$ *and* $a_i \in \lambda(e_i)$*. The semantics of evaluating* q *over* G *is an* $m-$*ary relation* $[q]_G \subseteq V \times ... \times V$ *(*m*-times) defined as follows:*
$[q]_G = \{(\mu(z_1), ..., \mu(z_m)) | \mu$ *is a mapping for* q *on* $G\}$.

We use Figure 2.1 as an example graph G for the next queries. A query q_1: $(a, b) \leftarrow Knows(a, b)$ returns a mapping $[q_1]_G = \{(v_2, v_1), (v_2, v_3), (v_3, v_4)\}$. Another query q_2: $(p_1, p_2, m_1) \leftarrow Knows(p_1, p_2), Author(p_2, m_1)$ returns a mapping $[q_2]_G = \{(v_1, v_2, v_6), (v_3, v_4, v_8)\}$.

Query languages (presented in Section 2.4) implement conjunctive graph queries, hence both queries could be also described using PGQL [3]:

```
/* Q1 */
SELECT id(a), id(b)
FROM MATCH (a) -[:Knows]-> (b)

/* Q2 */
SELECT id(p1), id(p2), id(m1)
FROM MATCH (p1) -[:Knows]-> (p2) -[:Author]-> (m1)
```

We can see that query languages describe the query in a more natural way. For that reason, for all the examples in this thesis, we use the PGQL query language.

Homomorphic and isomorphic matching. The above presented definition of mapping μ specifies that the mapping is a function, i.e., each variable is mapped to exactly one vertex. However, multiple variables can be bound to the same vertex, which implies a *homomorphic* semantic of the subgraph search. For *isomorphic* semantics, the mapping μ is injective, i.e., each variable is mapped to a unique variable.

For example, a query that searches for "common persons that are known by two people" in the example graph in Figure 2.1. In PGQL:

```
SELECT id(a), id(b), id(c)
FROM MATCH (a:Person) -[:Knows]-> (b:Person) <-[:Knows]- (c:Person)
```

In case of the homomorphic matching, the following query returns results:
$\{(v_2, v_1, v_2), (v_2, v_3, v_2), (v_3, v_4, v_3)\}$. On the other hand, isomorphic matching returns no results because none of the mappings is mapped to unique variables.

Algorithms

Each subgraph found is encoded by a *mapping* μ of q on G. Subgraph matching algorithms are based on finding, extending, or combining a *partial mapping* of q. A partial mapping is the binding of some vertices to the query variables. Formally:

Definition 4 (Partial mapping)**.** *Given a graph pattern $q[z_1, ..., z_m]$, a graph G, and $0 \leq k \leq m$, a partial mapping μ_k of q on G is a sequence of pairs $\langle(v_1, \mu(v_1)), ..., (v_k, \mu(v_k))\rangle$ such that $\langle v_1, ..., v_k\rangle$ is a sequence of distinct vertices of q, and μ is a mapping of q on G. k is called the length of the mapping μ_k.*

There are two main algorithms when it comes to homomorphic or isomorphic subgraph matching: *depth-first search (DFS)* and *breadth-first search (BFS)*, which are sometimes called equivalently *depth-first traversal (DFT)* and *breadth-first traversal (BFT)*, respectively. We present both approaches along with algorithms, similar to those shown in [1].

Depth-First search (DFS). The DFS algorithm for subgraph homomorphism is based on *backtracking*. The algorithm builds the matching one query vertex at a time. If a partial solution cannot be extended, it is discarded, and the matching *backtracks* (returns) back to the previous partial solution and continues the matching from there.

There are two possible implementations: the first one uses a recursion, i.e., the operating system's call stack, and the other avoids the recursion and implements its own stack data structure. The Algorithm 1 presents the implementation without recursion.

As an example of the DFS algorithm, we use the graph in Figure 2.1 and the abovementioned query q_2: $(p_1, p_2, m_1) \leftarrow$ *Knows*(p_1, p_2), *Author*(p_2, m_1).

We start the algorithm with variable initialization and setting the vertex order V_q^0 to $\{p_1, p_2, m_1\}$. This step decides the vertex ordering and is typically called *query planning*. With static refinements, we simply check whether the potential vertex graph candidates have correct vertex labels and a valid number of labeled incoming and outgoing edges. The static refinement step sets: $C_{p_1}^{stat} \leftarrow \{v_2, v_3\}$ (p_1 has one outgoing `Knows` edge), $C_{p_2}^{stat} \leftarrow \{v_1, v_4\}$ (p_2 has one incoming `Knows` edge) and one outgoing `Author` edge, and $C_{m_1}^{stat} \leftarrow \{v_5, v_6, v_7, v_8\}$ (m_1 has only one incoming edge `Author`). To bootstrap the computation, we push an initial tuple into the state stack, i.e., `S.push(`$\{p_1$`,{},0})`.

The first iteration of the algorithm takes the $v_G \leftarrow v_2$, adds it into the mapping $m_c = \{p_1 \leftarrow v_2\}$, updates the state entry with $\{p_1$`,{},1}`, and adds the next query vertex into the search stack, i.e., `S.push(`$p_2, \{p_1 \leftarrow v_2\}, 0$`)`. The second iteration takes $\{p_2, \{p_1 \leftarrow v_2\}, 0\}$, creates a mapping $m_c = \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}$, updates the state entry with $\{p_2, \{p_1 \leftarrow v_2\}, 1\}$, and pushes $\{m_1, \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}, 0\}$ onto the stack. The third iteration starts with the top of the stack, i.e., $\{m_1, m = \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}, 0\}$, and refines $C_{m_1,m}^{dyn} \leftarrow \{v_6\}$, because v_6 is the only neighbor of v_2 that is also within the previous static refinement. We then update the entry to $\{m_1, m = \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}, 1\}$, and because the current candidate mapping $m_c = \{p_1 \leftarrow v_2, p_2 \leftarrow v_1, m_1 \leftarrow p_6\}$ is homomorphic, we store it into M as a result mapping.

The following iteration picks $\{m_1, m = \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}, 1\}$, i.e., does the *backtracking* step back, and requests the candidate v_G. Because there is none, it `pops` this state, and continues (backtracks) to the next iteration with $\{p_2, \{p_1 \leftarrow v_2\}, 1\}$. The algorithm continues in the same manner until it finishes and returns M with all the mappings found.

Algorithm 1: Depth-First Search Matching

Input: Graph $G = (V_G, E_G)$ and a query pattern $q = (V_q, E_q)$
Result: A set of mappings M of q into G

$M = \emptyset$;
$V_q^0 \leftarrow$ order(V_q);
foreach $v_q \in V_q^0$ **do**
  $C_{v_q}^{stat} \leftarrow$ refine(G, q, v_q); `// static refinement`
end

$S \leftarrow \emptyset$; `// search state`
m.init() ; `// candidate mapping`
S.push(V_q^0.next(), $m, 0$);
while *!S.empty()* **do**
  $\{v_q, m, n\} \leftarrow S$.peek();
  if *n = 0* **then**
    $C_{v_q,m}^{dyn} \leftarrow$ refine($G, v_q, C_{v_q}^{stat}, m$); `// dynamic refinement`
  end
  $v_G \leftarrow C_{v_q,m}^{dyn}$.get($n$); `// get` n`-th candidate graph vertex`
  if $v_G \neq \emptyset$ **then**
    $m_c \leftarrow m$.copy();
    m_c.add(v_q, v_G); `// map` $v_q \leftarrow v_G$
    if *isHomomorphic(m_c, G, q)* **then**
      $v_q^{next} \leftarrow v_q^0$.next(); `// get next query vertex`
      if $v_q^{next} \neq \emptyset$ **then**
        S.pop();
        `// update to next` v_q `for backtracking`
        S.push($v_q, m, n+1$);
        `// add next query vertex for DFS`
        S.push($v_q^{next}, m_c, 0$);
      else
        M.add(m_c);
      end
    end
  else
    S.pop(); `// done processing`
  end
end

return M;

Breadth-First search (BFS). Lots of matching algorithms use the breadth-first search algorithm as their underlying mechanism. Compared to DFS, BFS is not a recursive algorithm. It processes each query vertex match independently, and once finished with all the mappings for the given query vertex, it continues to the next one, combining it with the matches from the previous match. Another perspective to BFS is that the graph query is broken into a set of smaller subqueries, which are processed independently. In the end, the memoized solutions are combined together to produce the final mapping.

Algorithm 2: Breadth-First Search Matching

```
Input: Graph G = (V_G, E_G) and a query pattern q = (V_q, E_q)
Result: A set of mappings M of q into G

M ← ∅;
V_q^0 ← order(V_q);
foreach v_q ∈ V_q^0 do
    C_{v_q}^{stat} ← refine(G, q, v_q) ;  // static refinement
end

S ← ∅ ;  // search state
m.init() ;  // candidate mapping
S.enqueue(V_q^0.next(), m);
while !S.empty() do
    {v_q, m} ← S.dequeue();
    C_{v_q,m}^{dyn} ← refine(G, v_q, C_{v_q}^{stat}, m) ;  // dynamic refinement
    foreach v_G ∈ C_{v_q,m}^{dyn} do
        m_c ← m.copy();
        m_c.add(v_q, v_G);
        if isHomomorphic(m_c, G, q) then
            v_q^{next} ← v_q^{next}.next() ;  // get next vertex
            if v_q^{next} ≠ ∅ then
                // add next query vertex for BFS
                S.enqueue(v_q^{next}, m_c);
            else
                M.add(m_c);
            end
        end
    end
end
return M;
```

BFS algorithm is presented in Algorithm 2. The following algorithm example uses the graph from Figure 2.1 and the already mentioned query $q_2 : (p_1, p_2, m_1) \leftarrow$ *Knows*(p_1, p_2), *Author*(p_2, m_1).

The initial steps are the same as for the DFS algorithm, i.e., the static refinement step sets: $C_{p_1}^{stat} \leftarrow \{v_2, v_3\}$ (p_1 has one outgoing `Knows` edge), $C_{p_2}^{stat} \leftarrow \{v_1, v_4\}$ (p_2 has one incoming `Knows` edge and one outgoing `Author` edge), and $C_{p_3}^{stat} \leftarrow \{v_5, v_6, v_7, v_8\}$ (p_3 has only one incoming edge `Author`). To bootstrap the computation, we push an initial pair into the state queue with $S.enqueue(\{p_1, \{\}\})$. Compared to DFS, BFS

uses a queue instead of a stack for storing states. In the first iteration, BFS deques $\{p_1, \{\}\}$ and stores a dynamic refinement for the pair ($C^{dyn}_{p_1,\{\}} \leftarrow \{v_2, v_3\}$). BFS now iterates through the set of vertices from the refinement, creates copies of the initial mapping, and because both mappings are homomorphic, it enques $\{p_2, \{p_1 \leftarrow v_2\}\}$ and after that $\{p_2, \{p_1 \leftarrow v_3\}\}$. The next iteration picks $\{p_2, \{p_1 \leftarrow v_2\}\}$ and refines the next potential values for p_2 from $\{v_1, v_4\}$ to $C^{dyn}_{p_2,\{p_1 \leftarrow v_2\}} \leftarrow \{v_1\}$. The vertex v_4 is avoided, because it is not a neighbor of the matched v_2. Since the new mapping $\{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}$ is homomorphic, BFS enques $\{m_1, \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}\}$. The next iteration processes $\{p_2, \{p_1 \leftarrow v_3\}\}$: It refines $C^{dyn}_{p_2,\{p_1 \leftarrow v_3\}} \leftarrow \{v_4\}$ (v_1 is avoided because it is not a neighbor of v_3), checks the homomorphism of the mapping, and enques $\{m_1, \{p_1 \leftarrow v_3, p_2 \leftarrow v_4\}\}$. The following iteration processes $\{m_1, \{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}\}$: BFS refines the static candidates for m_1 to $C^{dyn}_{m_1,\{p_1 \leftarrow v_2, p_2 \leftarrow v_1\}} \leftarrow \{v_6\}$, since only v_6 is the neighbor of v_1 and the created mapping $\{p_1 \leftarrow v_2, p_2 \leftarrow v_1, m_1 \leftarrow v_6\}$ gets added to the final mapping set M. Similarly, the last iteration processes $\{m_1, \{p_1 \leftarrow v_3, p_2 \leftarrow v_4\}\}$ and stores the mapping $\{p_1 \leftarrow v_3, p_2 \leftarrow v_4, m_1 \leftarrow v_8\}$ to the final set M

Refinement. Both algorithms use a step called *refinement*. The purpose of it is to cut down the number of visited graph vertices by filtering out the graph vertices that will not be matched. *Static refinement* can be done based on the neighborhood structure. The engine can check that the degree of the vertex graph candidate v_G is at least the same as the degree of the pattern vertex v_q. *Dynamic refinement* employs a dynamic validity check of each incremental partial match. After mapping a graph vertex v_G to a pattern vertex v_q, all the mapped v_q's neighbors $\{v_{q_1}, ..., v_{q_k}\}$ must be also neighbors in the v_G, i.e., the newly bound vertex has the same structure. Also, in case of isomorphic matching, the engine must check that the graph vertex v_G is always mapped to only one variable.

Another important part of the refinement in the case of querying is the evaluation of filtering on the newly added graph vertex, and validation of the labels. Again, after mapping a graph vertex v_G to a pattern vertex v_q, all the pattern filters of v_q must also be valid for the bound instance v_G.

The refinement pruning can be pushed down toward the end of the execution. Some engines evaluate filters at the end of the execution, which is not great for performance given the exponential nature of the algorithm. Generally, the sooner we prune the computation, the better the performance.

DFS characteristic. Depth-first search matching is attractive because of its low memory consumption, pipelined results, and efficient refinement through advanced pruning of partial mappings [1]. Given the incremental nature of the algorithm, its memory consumption corresponds to the memory consumption of storing a single visited path. Further, the pipelined results generation makes it well-suited for the interactive analysis, e.g., queries eagerly returning top-K results.

There are two main disadvantages of DFS: the complexity of the algorithm and suboptimal IO access patterns. The complexity of DFS comes from its recursive nature. Even when replaced with an iterative algorithm, the algorithm has more complex states compared to its BFS counterparts. More importantly, the DFS access patterns require visiting a chain of neighbors, i.e., following pointers to random access memory locations, which implies a bad locality.

BFS characteristic. The main advantages of BFS compared to DFS are its simple and adaptible tuple flow, easy parallelization, and sequential IO patterns. The algorithm does not need to incorporate backtracking, which simplifies the algorithm flow.

Due to the potential splitting into smaller independent subqueries, the parallelization is straightforward, compared to DFS. Also, BFS has much friendlier access patterns. Depending on the graph representation, the accessed neighbors are in most cases stored locally next to each other.

The large disadvantage of BFS is the extensive memory consumption. All partial mappings are fully materialized before matching another query vertex specified by the query plan. The space complexity of BFS is exponential in the diameter of the matched subgraph [1].

2.2.2 Graph Data Structures

Graph querying through BFS or DFS focuses on traversing neighbors. Therefore, it requires *adjacency-centric data structure* to allow efficient neighborhood lookup for a given vertex. In property graphs with edge labels, this data structure can be partitioned by the edge label, i.e., maintaining a single index per edge label set. Here, we present two graph representations: (i) "the basic" *matrix* representation, and (ii) *CSR* representation that is used in many graph engines and is also used by PGX.D/Async. For more graph data structures, we refer to Bonifati et al. [1].

Matrix. A matrix representation is the most basic representation of graphs. A matrix m represents a graph by setting $m_{i,j} = 1$ for every graph edge (v_i, v_j), and by setting $m_{i,j} = 0$ otherwise. The matrix can be linearized into an array of bits, which can be further optimized, e.g., by compression. The space complexity is $O(|V|^2)$, which is efficient only for very dense graphs. An advantage is that many complex graph operations, e.g., the graph isomorphism test, can be implemented on top of matrix representation and leverage well-optimized libraries for linear algebra.

Compressed Sparse Row (CSR). CSR is concise and efficient lookup representation [1]. It consists of two arrays A and B. The array B stores the neighbors of v_i consecutively in such way that the first neighbor is stored at index $A[i]$ and the last neighbor (if any) is stored at index $A[i+1]-1$. Therefore, the number of neighbors of vertex v_i is $A[i+1]-A[i]$. To iterate over all neighbors of vertex v_i, the engine does a $O(1)$ lookup to find the range $[A[i], A[i+1])$ for the iteration. The array B can also be semi-sorted, i.e., every individual neighborhood is sorted, to efficiently find a neighbor for a given vertex using the binary search.

Figure 2.4 shows the representation of graph G_{ex} (Figure 2.1) in CSR. Pointers point directly to the element at the given index. Notice that the vertex v_5 without edges stores the same index as the vertex v_6. Also, the array A stores one additional element that points beyond the array to compute the upper bound of the neighborhood for the last vertex (v_8 in our example).

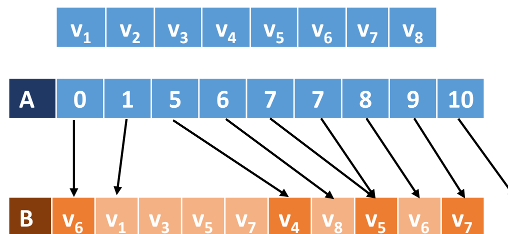


Figure 2.4: CSR representation of graph G_{ex}.

2.2.3 Graph Querying Systems

Graph querying is a domain of graph databases. Angles et al. [53] defines ***graph databases*** as *"systems specifically designed for managing graph-like data following the basic principles of database systems, i.e., persistent data storage, physical/logical data independence, data integrity, and consistency"*. In addition to graph databases, there are *graph processing systems* that allow management of graph data, but does not support all the requirements of the definition of the database systems. According this definition, PGX.D/Async, a system that we use for the evaluation of improvements presented in this thesis (introduced later in Section 3), is a graph processing system, because it does not persist data. In this section, we present graph query systems and graph databases in detail.

Native Graph Databases

Native graph databases [60] were explicitly built for managing graphs and are optimized for such workloads, e.g., by leveraging the property graph model (Section 2.1.1).

***Neo4j** [16]* is the graph database that made the property graph model popular and is the most popular graph database at the moment. It is a single-machine disk-based database supporting distributed storage for data replication. It implements the property graph model using a set of fixed-size records [60]. Thanks to that, the records are easily addressable. The *vertex record* stores a pointer to vertex labels, a pointer to the linked list of vertex properties, and a pointer to the first adjacent edge. The *edge record* stores a label, a pointer to the linked list of edge properties, two pointers to the source and destination vertex records, and a pointer to the adjacency lists of the adjacent vertices. Each property record can store up to four primitive types. For additional space, there exists a separate *dynamic store*. Thanks to this *index-free adjacency* [60] design of records, Neo4j can easily traverse relationships just by multiple pointer hops.

***Sparksee** [98]*, formerly known as DEX [99], is a graph database that implements a property graph model using B++ trees and bitmaps. Vertices and edges are identified using unique IDs. There exist multiple trees, one for labels, one per each property key, and two for vertices (one for source vertices, the second one for destination vertices). The reverse lookup, from values to keys, is done using a bitmap. Each value points to a bitmap where the position of the value 1 implies the index of key. The bitmaps for sparse graphs, i.e., bitmaps with mostly zeros, are clustered into 32 bits, and stored as `{clusterId,bitData}` pairs in the sorted tree.

***Tigergraph** [17, 97]* is a distributed database. It is a disk-based database, but it tries to store all data into main memory if it fits, and overflowed data are spilled to disk. Data are compressed and generally decompressed [100] only for displaying. The vertices and edges are referenced by hash indices. Given the fact that Tigergraph is a commercial product, there is not a lot of publicly-available information.

RDF Stores and Tuple Stores

RDF stores [101, 102, 103, 104, 105, 28], sometimes called triple stores after the RDF triples that they store, implement the RDF model (Section 2.1.2). Tuple stores [106, 107] generalize the RDF model of triples and allow storage of any size tuples.

***AllegroGraph** [108]* is a distributed multi-model database combining the document and graph model together. It supports geospatial and temporal datatypes as native data

structures. For graphs, it extends the implementation of the RDF model and stores an arbitrary number of immutable attributes along with the RDF triples.

***BlazeGraph** [109]* is a single-machine database implementing the RDF* extension [110, 93, 94, 95], which allows attaching multiple triples to one triple predicate for storing labels or properties. This allows attaching edge properties and labels more naturally than with RDF.

***Cray Graph Engine** [111]* is a scalable distributed in-memory graph query engine. It stores quads (4-tuples) instead of triples by extending the RDF triple with a graph ID. This makes it possible to easily store multiple graphs in a single storage. Quads are grouped by graph IDs and predicates into hash tables (a single hash table per a single group) storing only the subject-object pairs.

***WhiteDB** [106]* is a single-machine in-memory NoSQL tuple store. It allows storing tuples of any size. A tuple can contain any primitive datatypes or pointers to other tuples. Larger types, i.e., strings, are stored separately with a tuple storing only a pointer to it. It does not support any specific graph constructs: One needs to encode the graph structure into the tuples as shown, for example, in [107].

***GraphFrames** [31]* is a package for Apache Spark [112] which provides a graph specific functionality for Spark DataFrame [113]. It offers native support for vertices and edges, along with motif search and graph queries. GraphFrames internally uses the underlying Spark DataFrame, i.e., distributed table storage. For that reason, graph querying and motif are implemented using the BFS approach.

Other worth mentioning RDF-based engines are Amazon Neptune [28], that is built for the Amazon cloud, and Wukong [105], a distributed graph-based RDF store that leverages hardware features, such as RDMA and GPUs.

Relational Stores

Relational Database Management Systems (RDBMSes) store data in tables as was described in Section 2.1.2. *Row-oriented* tables [114, 115] store rows continuously in memory. This is an efficient representation if the system frequently accesses a few rows with most of their columns. On the other opposite, *column-oriented* tables [116, 117, 19, 118, 119] store columns continuously in memory, which is an efficient representation if the system frequently accesses only a few columns for most rows. An example of such workload is JOINing, which performs a linear scan on a column ID.

In general, relational databases are the most used, developed, and optimized databases at the moment. There is long ongoing research in that field, and covering the field in detail is out of the scope of this thesis. We mention here systems that are typically used for benchmarking [120] with other graph systems, i.e., PostgreSQL [119, 121], MonetDB [118], Umbra [19], SAP HANA [116], AgensGraph [122], and MariaDB [117].

From a graph processing point of view, it is interesting to see the recent adoption of the *worst-case optimal JOIN* [123, 124, 125], which optimizes queries with many JOINs by performing traversals similar to depth-first search in order to avoid a memory explosion. This improves the overall performance of long JOINs, i.e., consecutive JOINing of many tables. This is a typical workload for graphs during edge traversals.

Key-Value Stores

In key-value stores [13, 126, 127], the graphs are built without applying one specific graph model. Each engine uses its own approach for storing graphs.

Trinity *[13]*, newly called ***Microsoft Graph Engine***, is a distributed key-value storage with globally addressable memory RAM storage. It leverages graph access patterns to optimize memory and communication for best performance. It stores graphs as follows: The keys are vertex IDs. The values, called *cells*, store all the vertex properties as well as lists to store all the IDs of the vertex neighbors. For directed graphs, the vertex stores two neighbor lists, one for outgoing edges and the second for incoming edges; for undirected graphs, a single list is enough. For a small number of edge properties, the properties are stored along with the vertex ID in the neighbor list. In the case of many edge properties, individual edges are represented by individual cells, similar to vertices.

HyperGraphDB *[127]* is a general-purpose distributed hypergraph database built on top of key-value storage. The key-value pairs are called *atoms*. Every atom has a cryptographically strong ID used as a key. Atoms use a recursive design to store either a binary value or a list of other atom IDs. With that design, the vertices are represented as atoms with a type ID representing labels and a value ID representing properties. Edges are encoded with a type ID and a value ID representing labels and properties, and with a list of vertex IDs representing the vertices of the edge.

JanusGraph *[128]* is a distributed graph query engine. It is an example of an engine built on top of a *wide-column store*. It works as a two-dimensional key-value store. A key returns a row of sorted cells where each cell contains a key-value pair. Practically, the final cell value is identified by a row key and a cell key. JanusGraph is modular and allows picking different types of underlying data storage: Apache Cassandra [129], Apache HBase [130], or Oracle Berkeley DB [131]. The graph is modeled as follows: Each row represents a vertex. Each cell represents either properties or adjacent edges together with their properties. The cell keys are encoded in such a way that after the cell sorting, the properties go first, and the edges second.

TuGraph *[132]*, formerly known as LightGraph [133], is a large-scale distributed graph database used by Alipay [132]. Internally, it uses key-value storage implemented using multiversion B++-tree and write-ahead log. On top of that, it builds the graph index for graph processing.

Document Stores

Document stores [134, 135, 136, 137] use a document data model to organize data as a collection of documents. Documents have data in semi-structured formats, e.g., JSON [138] or XML [139]. The document data model is similar to a key-value model with values having a structure. This structure can consist of various types, collections, key-value pairs, or recursive structures, similar to what one can find in JSON or XML.

OrientDB *[135]* is a distributed document database. Documents are using a recursive design where every document has a *Record ID (RID)*, and consists of RIDs of other documents and offsets within that document. This forms *links* between documents. A vertex document stores labels, properties, a list of RIDs of incoming edges, a list of RIDs of outgoing edges, and a list of *lightweight edges*. The lightweight edges are inlined edges that have no properties or labels. The *heavyweight edges* are stored as an edge document that contains edge properties, labels, and RIDs of source and

destination vertices.

ArangoDB *[134]* is a scalable document graph database. It stores documents in a compacted binary JSON format called *VelocyPack*. Documents are stored in different collections with unique IDs for these collections. A graph is encoded as a vertex and an edge collection. Vertex documents store properties and labels. Edge documents store properties along with the source and destination vertices. All the edge documents are also connected into a linked list which allows a direct access to a neighbor vertex without revisiting the source vertex.

Object-Oriented Databases

The Object-oriented data model [140, 141, 142] stores graphs using a collection of objects linked together, similar to what one would program in object-oriented programming languages, such as Java or C#. Given this fact, graph modeling is implementation specific.

VelocityGraph *[143]* is built on top of the distributed object-oriented database VelocityDB [141]. The vertices, edges, and their properties are stored as C# objects. Each object has a unique object ID pointing to its physical storage. Every vertex and edge stores a single label and its properties in a dictionary. Edges are stored as a collection of neighbor vertices within each vertex.

InfiniteGraph *[142, 144]* builds on top of the distributed object-oriented disk-base database server Objectivity/DB [145]. It allows partitioning and distributed processing of the stored data. For querying, it uses its own Infinity DO query languages [146].

2.3 Other Approaches

In this thesis, we focus on *graph querying*. However, there are also other graph processing fields, such as *graph algorithms* and *graph mining*, which are close to graph querying. Some complex queries, such as shortest/cheapest path, might use a combination of querying and algorithms. Graph mining, on the other hand, performs a specialized pattern matching similar to querying.

2.3.1 Graph Algorithms

One of the main reasons for the appearance of the specialized graph systems, was the ability to run graph specific algorithms on the data. Nowadays, graph systems can run simple breadth-first, depth-first, shortest path searches [147], or more complex algorithms covering different categories of data analysis [148], i.e., *centrality* for determining the importance of distinct nodes in the network (e.g., Pagerank [149], Eigenvector centrality, Betweenness centrality), *community detection* for figuring how are the vertices clustered and their tendency strengthen or break apart (e.g., label propagation, weakly connected components, local clustering coefficient), *similarity* to find similar pairs of nodes based on their neighborhoods or properties (e.g., node similarity algorithms based on Jaccard or Overlap metrics), *path finding* for searching a path between two or more vertices (e.g., already mentioned breadth-first search (BFS) and depth-first search (DFS), Dijkstra and A* shortest path or random walks), *topological link prediction* to determine closeness of some vertices using the topology (e.g., common neighbors, same community, Adamic Adar), and *node embedding* to compute low-dimensional vector representations of the vertices, which is very useful in machine learning (e.g., FastRP, Node2Vec, GraphSAGE, HashGNN).

The systems typically have a set of optimized hard-coded algorithms, and allow additional flexibility by implementing new algorithms. A more high-level approach uses domain specific languages [150, 151, 152] for letting users program algorithms using predefined language constructs and transforming these constructs into system-specific operations.

The other approach employs special graph programming iterative models [153, 154, 24, 25, 29, 155, 156, 157, 158, 159, 160, 161, 162, 163] to calculate values of vertices and edges iteratively based on the values of the neighborhood. Influenced by the runtime systems, the values between iterations are propagated by using the *push model [164]*, i.e., pushing the values to neighbors, by using the *pull model [165]*, i.e., pulling the values from neighbors, or by using some kind of hybrid model [163, 160, 151]. In distributed systems, the push model means that the data are sent to the remote machine once they are available. On the contrary, the pull model implies requesting the necessary data first and processing the data once they arrive.

2.3.2 Graph Mining

Graph mining aims to extract structural properties of the graph by exploring its subgraph structures to understand the graph. More specifically, graph mining problems search for *isomorphic* projections (explained in Section 2.4.1) from a subgraph of interest to a given input graph.

Graph mining has a fixed set of problems on which it focuses. There are two versions of each problem, the first is focusing on *counting*, i.e., returning number of matched subgraphs, the second one is focusing on *enumeration*, i.e., returning all the matched subgraphs. Below are the graph mining problems. For simplicity, we mention the counting versions only:

- *Motif counting.* A motif is any connected unlabeled subgraph pattern. The problem is counting occurrences of all possible motifs up to a certain size.

- *Frequent subgraph mining.* This problem focuses on listing all labeled patterns with `k` edges that appear more than a given threshold τ. The frequency of appearance (called *support*) is measured using an *anti-monotonic* metric, i.e., given a pattern `Q` and its subgraph `P`, support$(P) \geq$ support(Q). Since simple subgraph counting (known from querying) is not anti-monotonic, graph mining uses special metrics [166, 167, 168]. The metrics allow efficient computation with optimizations. Without it, exhaustive search is unavoidable [167, 168].

- *Clique counting.* A `k`-clique is a fully-connected graph with `k` vertices. The problem counts the number of such patterns in the graph. There are different variations of this counting problem with slightly different patterns, such as counting *pseudo-cliques*, i.e., patterns where a density of the edges goes over a threshold, a counting of *maximal cliques*, i.e., cliques that are not included in any other cliques, or a search for *frequent cliques*, i.e., cliques that appear more than a given threshold.

- *Pattern matching.* This problem counts the number of subgraphs that are isomorphic to a given pattern. There is also a variation of this problem called *constrained subgraph* that constrains the searched subgraphs in a given way.

There are many general purpose graph mining engines [23, 20, 169, 170, 22, 171, 172, 173]. ***Peregrine*** *[23]* is a state-of-the-art single-machine graph mining system

that is fully pattern-aware. It also supports additional performance optimizations such as symmetry breaking or using the notions of anti-vertices and anti-edges to express absences of vertices and disconnections. ***Automine*** *[171]* is another single-machine system that is not pattern-aware, but supports memory/disk-resident graphs. It provides user algorithms with a high-level abstraction of representing subgraph patterns via vertex composition sets, enumerates their subgraph patterns, generates their schedules, for identifying the subgraph patterns, and finally compiles them in C++ by considering re-using subgraph patterns across schedules to reduce memory consumption. ***RStream*** *[170]* is a single machine graph mining system that expresses mining tasks as relational JOINs, and allows processing large graphs by offloading intermediate results to SSD.

Several distributed graph mining engines exist as well. ***Arabesque*** *[20]* is implemented using map-reduce and proposes the "think like an embedding" model, i.e., a high-level filter-process computational model that passes subgraphs to the application to decide upon their further exploration. ***Fractal*** *[169]* adapts load-balancing to the workload via a hierarchical and locality-aware work-stealing mechanism, and enumerates subgraphs with a depth-first strategy while recomputing subgraphs from scratch to avoid numerous in-memory intermediate results. ***G-Miner*** *[22]* is a task-oriented engine with load-balancing processing that uses a distributed task queue with asynchronous communication to hide communication overhead. ***BiGJoin*** *[173]* uses a data-parallel dataflow computation with a worst-case optimal approach that dynamically joins the columns with the least matches.

2.4 Graph Query Languages

In addition to graph engines and databases, many languages for graph querying exist, such as PGQL [3] (from Oracle [174]), Cypher [6] (from Neo4j [16]), GSQL [5] (from Tigergraph [17]), SPARQL [175] (for RDFs), or Gremlin [4]. Because of this fragmentation, two new graph query language standards emerged: SQL/PGQ [7] (published in June 2023) as a graph extension for SQL, and GQL [8] (planned to be standardized in April 2024) as a new graph query language, semi-compatible with SQL/PGQ. For example, the pattern matching in GQL should be identical to the pattern matching in SQL/PGQ. Additionally, GQL should also include constructs related to graph modifications, e.g., create graph, insert/update/delete.

Given the novelty of SQL/PGQ and the lack of support for some more advanced features, such as RPQs (presented in Section 5), for all query examples in this thesis, we use PGQL 2.0 [3] which is close to SQL/PGQ, as it was modeled after SQL. Nevertheless, the other languages offer similar functionality. In this section, we use PGQL to describe various features of graph query languages that we use in this thesis. More complex features not needed in our work, such as subqueries, SHORTEST and CHEAPEST paths, are omitted. For these features and more details, please refer to the PGQL documentation [3].

2.4.1 Querying and Pattern Matching

Similar to SQL queries, PGQL queries consist of multiple clauses with predefined meanings, of which some are mandatory and some are optional. We will go over all the relevant clauses in the order in which they appear in queries.

SELECT. The `SELECT` clause specifies data entities that the query returns. More specifically, it defines the columns of the result table. It can contain any graph ele-

ment, i.e., vertex, edge, its properties, or any expression consisting of those values. We can refer to the graph elements using the variable names and also create their name aliases using the keyword `AS`. To refer to all properties of a graph element, one can use `*` (star). To avoid duplicate entries within the result data rows, PGQL specifies keyword `DISTINCT` that filters out all the duplicate entries and returns only unique elements. In case of missing values, the data is reported as `NULL`. Also, one can use predefined aggregated functions, e.g., `COUNT, MIN, MAX, AVG`, to return scalar values. Note that we cannot combine the scalar values with the table results. An example of the clause is shown later.

FROM and MATCH. The `FROM` clause describes the searched pattern. The clause can consist of multiple `MATCH` clauses that specify a searched pattern or multiple `GRAPH_TABLE` clauses that define a SQL/PGQ compatible way of expressing the searched pattern.

The `MATCH` clause specifies a (partial) subgraph pattern that the engine is going to search for. It consists of one or more vertices and relations between them. A relation can be an edge or a path (Section 2.4.3). Each vertex and edge are specified using an optional variable, which is a symbolic name to reference the element in other clauses further in the query, e.g., in projections, filters, or post-processing.

A single vertex is specified using parentheses `()` with the optional name of the variable inside, i.e., `(x)`. An edge, named or anonymous, can be specified between two vertices only. To mark the edge, we use: an arrow -> for directed outgoing edges, <- for incoming edges, or a single dash - for any-directed edges (the edge pattern can be matched to the edge with any direction). The optional edge variable is specified using square brackets inside the edge pattern, i.e, `-[e]->`, `<-[f]-` and `-[g]-`, to define an outgoing edge `e`, an incoming edge `f`, and any-directed edge `g`, respectively. Multiple patterns can be chained together or specified using an individual `MATCH` clause. The semantics is the same in both cases, i.e., every result must match all the patterns together.

WHERE. The `WHERE` clause specifies filtering. Filtering is defined in the form of boolean expressions that are evaluated on every match. If the expression evaluates to `true`, the filter passes, and in case of `false`, the matched result is removed. PGQL allows usage of typical mathematical numerical and boolean expressions, e.g., `+, -, *, /, AND, OR, >, <, =(equality)`, string functions, e.g., `SUBSTR, LOWER, JAVA_REGEXP_LIKE`, functions for date-time manipulations, as well as user-defined functions, and `CASE` predicate that allows branching. In addition to manually specifying labels, there is a function `has_label()`.

An example. Here are two example queries with the graph G_{ex} (Figure 2.1):

```
/* Q1 */
SELECT p.name, p.age
FROM MATCH (p:Person)
WHERE p.age < 30

/* Q2 */
SELECT a, b, c, d
FROM
  MATCH (a:Person)-[:Knows]-(b:Person)-[:Author]-(c),
  MATCH (c)-[:Reply]-(d)-[:Author]-(a)
```

The query `Q1` matches every person that is younger than 30 years and returns its property `name` and `age`. The query `Q2` searches for any-directed 4-hop cycles in the graph

with given labels. Note that the labels are specified using `:` (colon) with filters names separated with the `|` (pipe), e.g., `(p:Person|Men|Employee)`. The results are: $\{v_1, v_2, v_7, v_6\}, \{v_1, v_2, v_5, v_6\}, \{v_2, v_1, v_6, v_5\}$, and $\{v_2, v_1, v_6, v_7\}$. If we do not include labels, other mappings, such as $\{v_1, v_2, v_1, v_2\}$ or $\{v_1, v_6, v_1, v_2\}$, would also be valid due to homomorphic matching.

2.4.2 Grouping, Aggregation, and Sorting

The aggregations are responsible for aggregating the results along all the results or a group of results. Groups are created based on a given expression. Each aggregated value is created per a group of the expression results. The aggregated value can be any valid expression, e.g., `COUNT(2 * a.prop1 + 3)`.

GROUP BY. This clause allows specifying the grouping of the results based on properties having the same value. Within those groups, one can aggregate results of result values or expressions. There are predefined aggregation operators that do what their name suggests, i.e., `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`, as well as `ARRAY_AGG` that constructs an array from all the aggregated values, and `LISTAGG` that concatenates the aggregated values with a given separator. Similarly as in projections, all expressions can be combined together with `DISTINCT` to eliminate duplicated values.

HAVING. `HAVING` specifies the aggregation filtering. It filters out particular groups in the results based on a given expression.

ORDER BY. This clause specifies the ordering of the returned results. Without it, the returned order of the results is undefined. The parameter of the `ORDER BY` can be any number of values or expressions. There are also two keywords: `ASC` and `DESC` specifing the ordering direction.

OFFSET and LIMIT. `OFFSET` specifies the start index of the first result. `LIMIT` gives a limit on the number of results returned.

An example. Here we present an example of a non-trivial query and its execution in G_{ex}. However, there is no fixed order of the operations, and every engine can optimize this process.

```
SELECT AVG(p.age), COUNT(*) AS msg_count
FROM MATCH (p:Person) -[:Author]-> (m:Msg)
GROUP BY p
HAVING msg_count < 10
ORDER BY msg_count ASC, AVG(p.age) DESC
LIMIT 2
```

The query evaluation proceed in following steps:

1. Matching the given pattern and returning mappings together with the properties that are needed for next steps: `(p,m,p.age)` $= \{(v_1, v_6, 28), (v_2, v_5, 28), (v_2, v_7, 28), (v_4, v_8, 32)\}$.

2. Creating groups with the same `p`: $g_{v_1} = \{(v_1, v_6, 28)\}$, $g_{v_2} = \{(v_2, v_5, 28), (v_2, v_7, 28)\}$, and $g_{v_4} = \{(v_4, v_8, 32)\}$.

3. Aggregating the groups. `AVG(p.age)` computes average `age` for each group, `COUNT(*)` counts the number of entries in each group, and `AS msg_count` creates an alias for the result of the counting: `(AVG(p.age),msg_count)` $= \{g_{v_1}(28, 1), g_{v_2}(28, 2), g_{v_4}(32, 1)\}$.

4. Filtering out groups with `msg_count < 10`. Because all groups meet this criteria, we continue to the next step.

5. Ordering of groups. First, we order according the `msg_count` in ascending order, and second, we order according the `AVG(p.age)` in descending order: $[(32,1),(28,1),(28,2)]$.

6. Finally, processing `LIMIT 1` by taking the first two entries from the sorted results. The query result is: `(AVG(p.age),msg_count)` = $[(32,1),(28,1)]$.

2.4.3 Variable-Length Path Pattern Matching

Section 2.4.1 introduced fixed-length path matching. However, PGQL also supports variable-length path matching that allows matching a certain part of the query (called *path*) multiple times within a given limit in such a way that every matched result can have a different number of repetitions, i.e., have different (variable) number of matched vertices and edges. The variable-length queries also allow different search strategies to find paths between matched vertices, which present two powerful, and easy to use, constructs that unlock the true potential of graph querying.

To specify the number of repetitions of the path, PGQL uses a syntax similar to regular expressions. Table 2.1 shows the quantifiers with their number of repetitions.

Quantifier	Number of repetitions
*	zero or more
+	one or more
?	zero or one
{m}	exactly `m`
{m,}	`m` or more
{m, n}	between `m` and `n` (inclusive)
{, n}	between zero and `n` (inclusive)

Table 2.1: Variable-Length Path Quantifiers.

There are different path-finding goals, i.e., strategies on how to find the given path between the source and destination vertices, and also specifying the number of returned paths. The options are: `ANY`, `ALL`, `ANY SHORTEST`, `ALL SHORTEST`, `SHORTEST K`, `ANY CHEAPEST`, `ALL CHEAPEST`, and `CHEAPEST K`. Here we focus only on `ANY` and `ALL`. The others relate to other path-finding strategies that use different search algorithms, which is out of the scope of this thesis.

ANY paths. In the previous versions of PGQL, this was called *reachability* since it uses reachability queries to find the searched path. This goal searches for any path with no restrictions on the length between the source and destination vertices. This can be used to test the existence of paths between two vertices. Reachability semantics implies that multiple found paths between the same source and destination vertex pairs are counted only once. For example, a query that "returns all pairs of persons that contributed in a discussion under a message", in PGQL:

```
SELECT p1, m1, m2, p2
FROM
  MATCH (p1:Person) -[:Author]-> (m1:Msg),
  MATCH (p2:Person) -[:Author]-> (m2:Msg),
  MATCH ANY (m1) (-[:Reply]-)* (m2)
```

returns following results/mapping: $\{p1, m1, m2, p2\} = \{$
$\{v_1, v_6, v_5, v_2\}, \{v_1, v_6, v_6, v_1\}, \{v_1, v_6, v_8, v_4\}, \{v_1, v_6, v_5, v_2\},$
$\{v_2, v_5, v_6, v_1\}, \{v_2, v_5, v_5, v_2\}, \{v_2, v_5, v_7, v_2\}, \{v_2, v_5, v_8, v_4\},$
$\{v_2, v_7, v_5, v_2\}, \{v_2, v_7, v_6, v_1\}, \{v_2, v_7, v_7, v_2\}, \{v_2, v_7, v_8, v_4\},$
$\{v_4, v_8, v_7, v_2\}, \{v_4, v_8, v_6, v_1\}, \{v_4, v_8, v_5, v_2\}, \{v_4, v_8, v_8, v_4\}\}.$
Notice that homomorphic matching together with the any-direction `Reply` edge generates many results that can include loops.

ALL paths. This goal matches all paths between a given pair of vertices. Compared to `ANY` this counts all the found paths. In order to avoid infinite cycles in the case of non-tree graphs, PGQL forbids unbounded quantifiers, i.e., `*` (star), `+` (plus), and `{m,}`.

2.5 Benchmarking

In this section, we provide a brief overview of the benchmarks typically used by graph querying systems and the benchmarks used in this thesis. More detailed information together with queries can be found in the appendix.

2.5.1 LDBC

The Linked Data Benchmark Council (LDBC) [120] is a non-profit organization aiming to standardize graph benchmarks. It consists of organizations and individuals from industry and academia. Current benchmark consists of following benchmarks:

- *Graphalytics [176]*: A benchmark for graph algorithms.
- *Financial Benchmark [177]*: A benchmark for financial workloads (work in progress in July 2023).
- *Semantic Publishing Benchmark [178]*: An industry-grade benchmark for RDF-based semantic databases.
- *Social Network Benchmark*: A benchmark for graph database management systems.

Social Network Benchmark

The Social network benchmark [179] is currently the most interesting benchmark for all graph database management systems. It uses *choke points* to design the workloads. The choke points are challenging aspects of query processing that motivate databases to employ more complex optimizations. An example of a choke point is the ability to compute the shortest paths between a vertex and a set of vertices. The benchmark consists of two workloads, *Business Intelligence (BI)* and *Interactive (I)* workloads.

Business Intelligence (BI). The Business Intelligence (BI) workload [180] is the analytical (OLAP) workload focusing on aggregation and join-heavy complex queries accessing a large portion of the graph with microbatches of insert/delete operations. It contains 20 parameterized read queries consisting of 38 choke points divided into 9 categories: aggregation, join, data access locality, expression calculation, correlated subqueries, parallelism and concurrency, graph specifics, language features, and update operations.

Interactive (I). The workload [181] captures OLTP scenarios, i.e., it focuses on complex read queries accessing the vertex neighborhood and concurrent update operations continuously inserting new data. There are four types of operations: 14 complex (CR) and 7 short (SR) read queries, 8 insert (INS) queries, and 8 delete queries. The workload is mixed using 8% CR, 72% SR, 20% INS, and 0.2% DEL operations.

Datasets. The data consist of different scales from SF1 to SF10,000. The initial dataset of SF10,000 contains more than 23 billion vertices and around 173 billion edges. The update dataset employs 26 billion insert operations and 250 million delete operations. Within this thesis, we use SF10, SF100, and SF300. The total number of individual elements is shown in Table 2.2.

2.5.2 TPC-H

TPC is a non-profit corporation developing data-centric benchmark standards that can be used easily by the industry. TPC-H [182] is a decision support benchmark of business oriented ad-hoc queries and data modifications. It simulates a decision support system that performs complex business analysis. It consists of 22 complex queries and two database refresh functions. The total number of elements is shown in Table 2.2. TPC-H is not a classic graph benchmark since it was developed for RDBMSes. Nevertheless, given its popularity for relation databases, we port the dataset as a graph and use it in some of the experiments in this thesis.

	LDBC			**TPC-H**	
	SF10	**SF100**	**SF300**	**SF100**	**SF300**
Total vertices	27M	255M	738M	786M	2.4B
Total edges	170M	1.7B	5.1B	2.0B	6.1B

Table 2.2: The number of elements in LDBC and TPC-H for different scales in millions (M) and billions (B).

3. PGX.D/Async Query Engine

PGX.D [29] is a fast distributed graph processing engine that supports the most common way to process graphs, namely *graph analytics*, i.e., graph algorithms, and *graph querying*. In this chapter, we describe parts of the engine relevant to the graph querying as well as the underlying system infrastructure. The original design of the graph querying engine, PGX.D/Async, is described in [32]. The parts related to graph analytics and infrastructure can be found in [29].

Organization. This chapter presents PGX.D/Async. Section 3.1 shows the architecture of the engine. Section 3.2 explains how query planning works. Section 3.3 describes the runtime and how the matching and other important parts work. Section 3.4 explains the post-processing part of the engine, i.e., `GROUP BY` and `ORDER BY`. Finally, Section 3.5 evaluates selected parts of the engine and explains their performance.

3.1 Architecture

Figure 3.1 shows the high-level architecture of PGX.D/Async. Different components of the systems are described in the following sections.

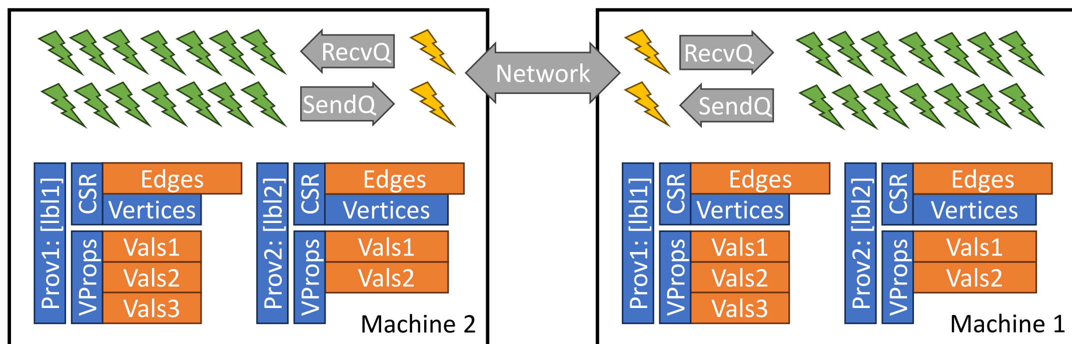


Figure 3.1: High-level architecture of PGX.D/Async.

3.1.1 Communication Management

For handling communication, PGX.D/Async uses an internally developed library that allows efficient communication over different network layers, e.g., Ethernet or InfiniBand. This library incorporates zero-copy messaging which directly registers a set of pre-allocated buffers with the network card. This library uses a standard message passing approach, similar to one in Open MPI [183].

For message communication, the system pre-allocates a fixed number of messages at the beginning of the program that are used for all the communication. There are two threads on each machine dedicated to sending and receiving. One thread manages the sender queue and the other manages the receiver queue. When sent, the message is pushed into the sender queue and eventually sent by the receiver thread. On the receiving side, the sender thread pushes the received message into the receiver queue, where it is eventually picked up by one of the other worker threads.

3.1.2 Data Management

Graphs are stored in memory using the conventional CSR (Compressed Sparse Row) format (Section 2.2.2). A CSR is created per each edge and vertex *provider*. This allows fast vertex and edge iterations based on given provider. Providers are specified in a graph schema and are typically mapped to labels. In case of no labels, edges/vertices can use a single unnamed provider or can be split into multiple ones. If there are more labels, one can pick one main label as a provider, and let the other labels be encoded as normal labels, i.e., using dictionary encoding. Another approach creates a provider per each label subset. Vertex and edge properties are stored in a columnar format and allow a direct lookup based on the vertex/edge ID. String properties incorporate dictionary encoding for performance.

All data arrays are partitioned across the cluster during the graph loading. PGX.D uses *edge partitioning* [184] as a partitioning strategy that results in a similar number of edges in each partition. The primary goal of this strategy is to balance workloads between machines [30, 185]. Once the graph is partitioned, PGX.D assigns a global 64-bit ID to every vertex. The vertex ID allows for a quick lookup of the vertex and its properties and labels.

Ghost vertices. As mentioned in Section 1, the number of neighbors in real social networks follows the exponential distribution [12]. This implies the existence of *high-degree vertices* that are connected to almost every vertex in the graph. In combination with a suboptimal partitioning, this can create an imbalance between machines even with the edge partitioning.

To mitigate this problem, the engine selects high-degree vertices and creates copies of them on every machine, called *ghost copies* [186, 29]. Each copy holds only a part of the neighborhood. As a side effect, in well-balanced edge partitioning, almost all the edges of ghost copies should be local.

Along with the *ghost copies*, there exists also a *ghost origin* that stores all the properties and is used as an edge vertex destination in CSR. Note that compared to a normal vertex, accessing all ghost-vertex neighbors requires broadcasting the request to all its copies.

Later in the evaluation Section 3.5, we show the benefits of using ghosts with a real workload.

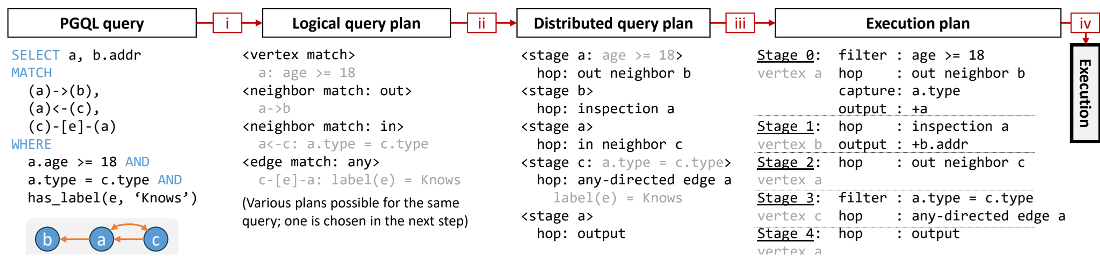


Figure 3.2: From a PGQL query to PGX.D/Async execution. Three transformation steps before execution.

3.2 Query Planning

Users submit declarative PGQL [3] queries to PGX.D/Async. As Figure 3.2 illustrates, each query goes through three transformation steps before being executed in step iv.
Step i: Logical query planner. The first step translates the PGQL query into a logical query plan, which consists of the logical operators of Table 3.1. Similar to relational query planning, a given query can be executed by multiple logical query plans. In the example of Figure 3.2, an alternative plan could rewrite the query as `(a)-[e]-(c)->(a)->(b)`. This first step directly translates the query to an admissible plan, which is then optimized in the following steps.

Step ii: Distributed query planner and optimizer. This step specializes the logical query plan by taking into account the specific characteristics of PGX.D/Async's runtime. The query planner rewrites the logical plan in terms of *stages* and transitions from one stage to another (called *hops*). A stage is responsible for matching or accessing exactly one vertex and contains all the information necessary for matching the corresponding vertex and for transitioning to the next vertex with a hop. In the example of Figure 3.2, the topmost stage "a" matches the first vertex `a` of the query, while the next one matches `b`. An out-neighbor hop takes the execution from `a` to `b`.

PGX.D/Async supports four types of hops that specialize for distributed execution: *neighbor match*, *edge match*, *output*, and *inspection*. Neighbor and edge hops have the same behavior as the corresponding logical operators in Table 3.1. An *output* hop produces a final match using the current intermediate result and is always used in the last stage of a match.

Operator	**Example**	**Description**
Vertex match	`(x)->(y)` (**x** bold)	Matches vertices without following edges
Neighbor match	`(x)->(y)` (**->(y)** bold)	Matches neighbors of the current vertex
Edge match	`(x)->...` `(y)->(x)` (**->(x)** bold)	Matches again an already matched vertex
Inspection match	`(x)->(y)->(z),` `(y)->(w)` (**(y)** bold)	Transfers the execution back to an already matched vertex
(the last hop)	—	Stores the intermediate results

Table 3.1: Graph operators used in the logical query plan.

Inspection hops are specific to distributed processing: They bring the current intermediate result back to an already matched vertex in order to continue query evaluation. In the example of Figure 3.2, after matching `a` and `b` of `(a)->(b)`, the query again needs the neighbor list of the already matched vertex `a` in order to continue with matching `(a)<-(c)`. Since the matched vertex `b` might be in a different machine than `a`, the query planner introduces an inspection step to "link" this disconnected pattern and bring back the context to the machine of `a`. If `a` resides in the current machine, an inspection hop is essentially a no-op.

In this step, PGX.D/Async rewrites the logical query plan with a cost-based op-

timizer, implemented using dynamic programming, that is based on the following heuristics:

- Heavily filtered vertices are preferred for the earlier stages of the plan.
- Inspection hops are not free and increase the plan's cost.
- The cost of an edge hop is approximately log of the cost of a neighbor hop, as it can be implemented with a binary search in the neighbor list of the source vertex.

The optimizer further detects whether a query has a single starting vertex, by extracting ID equality filters (e.g., `ID(person) = 123`). In the example of Figure 3.2, the optimizer rewrites the query as `(a)-[e]-(c)->(a)->(b)` because it avoids an inspection hop and `a` and `e` are more filtered as compared to `b`.

Steps iii–iv: Execution plan and execution. Finally, PGX.D/Async generates a concrete execution plan. Apart from stages and hops, the execution plan contains filters (on vertices and edges), as well as information on what data should be included in the intermediate results in order to execute filters of later stages and produce the final output. For example, in the query of Figure 3.2, Stage 0 must collect `a.type`, since it is required by the filter of Stage 3. Similarly, Stage 0 must put vertex `a` in the intermediate result as it is part of the projection of the query. Overall, each stage builds up the intermediate result such that another thread, local or remote, can pick it up and continue the computation. The resulting execution plan is then submitted to the PGX.D/Async runtime, on which we focus next.

3.3 PGX.D/Async Runtime

As was presented in Section 2.2, recent graph-mining and graph-querying systems [187, 23] adopt a pattern-matching approach that relies on intersecting neighborhood lists. Instead of being vertex-centric (i.e., starting from vertices and following edges), the intersection approach focuses on edges. The benefit of the intersection-based model is that it takes $O(|V|)$ steps since it allows intersecting multiple incoming edges at a time, as compared to the vertex-based approaches that are $O(|E|)$. However, intersections require complete subgraph parts to operate. This necessitates pulling/-gathering possibly large amounts of data from remote machines. To make things worse, queries enumerate all automorphisms (i.e., the exploration space could locally explode) and offer arbitrary user filters and projections, meaning that in an intersection-based model, one would need to pull not only the vertex/edge data, but also all the properties required by the query. Therefore, PGX.D/Async uses a vertex-centric approach that builds mini-frontiers based on the first query vertex and enables PGX.D/Async to operate on fully partitioned graphs with limited memory.

Runtime. The runtime of PGX.D/Async is based on the *stage* and *hop* constructs described above. PGX.D/Async initiates query execution by applying Stage 0 (matching of the first vertex variable of the execution plan) to each vertex of the graph. This bootstrapping process happens (i) across machines, i.e., each machine starts from the locally-stored vertices, and (ii) concurrently within each machine, i.e., each worker thread handles a distinct set of vertices and performs the bootstrapping process on these vertices one after the other. Hops that follow remote edges send the intermediate match (batched) to the destination remote machine where they are picked up and taken over by a local thread.

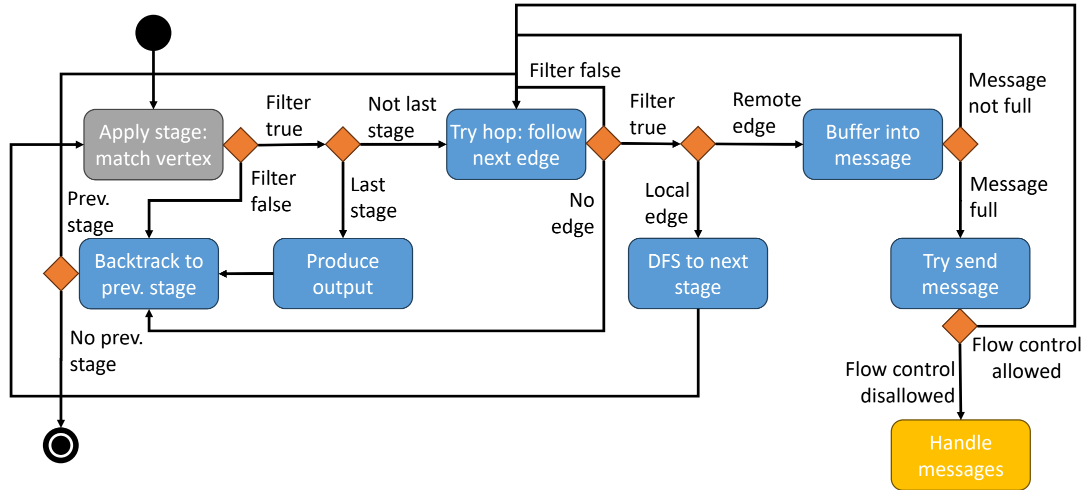


Figure 3.3: Matching operations starting from a given vertex. The yellow box "Handle messages" connects to the activity diagram in Figure 3.5.

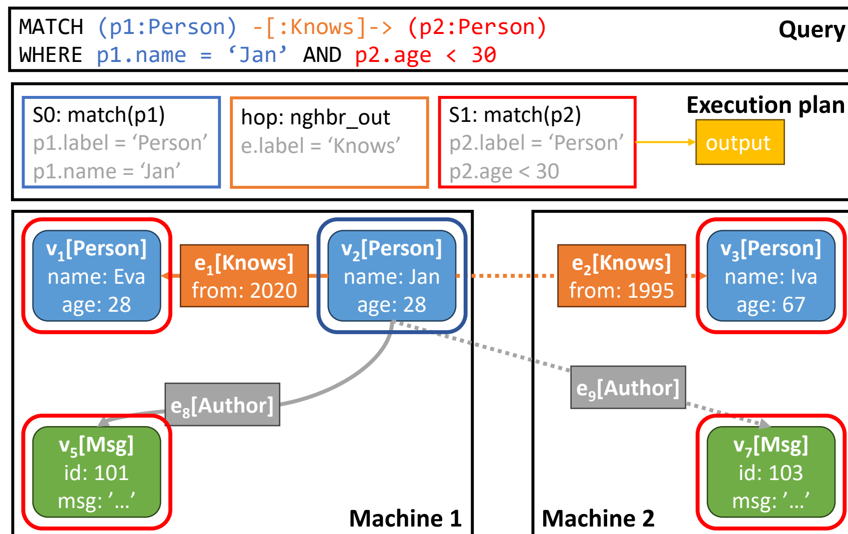


Figure 3.4: Example query on a distributed graph. Rounded rectangles represent vertices. Remote edges are dotted, local edges use full line. The execution plan depicts Stage 0 in blue, Stage 1 in red, neighbor hop in orange, and output hop in yellow. Vertices matched to Stage 0 are inside blue rectangle, vertices matched to Stage 1 are inside red rectangles.

Matching. Figure 3.3 includes a high-level activity diagram of the PGX.D/Async runtime. Completing the execution of this diagram from Stage 0 to the last query stage implements the complete matching starting from a single vertex of the graph. We explain these steps using the example of Figure 3.4. Text in the *blue italic face* represents the activities in Figure 3.3. The PGX.D/Async runtime assigns vertex v_2 in Figure 3.4 to a worker thread `t`, which tries to generate new matches. The thread first tries to match v_2 with Stage 0's `p1` using *Apply stage*. If the filter `p1.name = 'Jan'` returned `FALSE`, the thread would try to *backtrack to a previous stage* and, because there

is none, it would simply complete this invocation. If there were more top-level vertices to explore, `t` would start again with a different vertex.

In the example of Figure 3.4, we assume that the execution plan matches vertex `p1` as Stage 0. `p1` matches v_2 and `t` continues with the *Try hop: follow next edge* operation, starting from edge e_1. Since the `:Knows` label filter is satisfied and the edge is local, `t` proceeds via *DFS to next stage* to Stage 1 where `p2` is matched with the vertex v_1 that has `age = 28`. At this point, since the filter `p2.age < 30` is satisfied and there is no next stage, `t` *produces a query output* row and *backtracks* to Stage 0 to continue with the next edge. Thread `t` then *tries hop* to the edge e_8, but the label filter is not satisfied.

Thread `t` is now done with local edges and starts processing the remote ones (the system does not necessarily match all local edges first). The first one, edge e_2, has label `:Knows` (edge properties and labels are stored with the source vertex), thus `t` places the current intermediate result in a messaging buffer targeting Machine 2 (*buffer into message*). This way, the matching of this sub-tree is sent to Machine 2 with all intermediate results and thread `t` can completely "forget" about it. Once the buffer is filled up, `t` *tries to send a message* with the contents of the buffer to the destination. As Section 3.3.2 describes in more detail, flow control might temporarily block `t` from sending the message; in that case, `t` *handles messages*. Once the thread returns from performing these other tasks, it retries sending the blocked message. Finally, `t` attempts to match the last remote edge e_9, which does not match because of its label. With all the edges of vertex v_2 explored and no previous stage to backtrack to, `t` completes the invocation.

Handling incoming messages (intermediate results). Workers eagerly try to receive and process remote messages, always prioritizing the latest stage with available work. Threads try to process messages:

- Before starting new work, i.e., before *Apply stage* at Stage 0 (new top-level vertex).
- When flow control (temporarily) disallows message sending—in that case, the impacted thread picks up a new message to process while waiting for flow control to release the blocked message.
- Once the matching operations (see Figure 3.3) have completed on all local vertices. At that point, workers continuously wait for incoming messages to complete any pending work from remote machines.

The activity diagram for handling incoming messages is depicted in Figure 3.5. We continue the example of Figure 3.4 from the point of view of Machine 2 and a thread `r` that has just started handling incoming messages. Thread `r` starts from the latest stage and tries to unblock any potential computations first, i.e., it checks if the stage is blocked and *tries to send the blocked message*. If the flow control allows sending, it *continues the matching* from the vertex where it left before blocking. If the flow control still disallows sending, the thread backs off and *tries the previous stage*.

In case the stage was initially unblocked, the thread `r` *tries to get a received message*. If there is none, it proceeds to *try the previous stage*. In our case, `r` successfully acquires the message sent from Machine 1 and *continues/starts the matching* by iterating over intermediate results of the message and *Applying a stage to the vertices* (refer back to the activity diagram in Figure 3.3). Thread `r` eventually starts processing

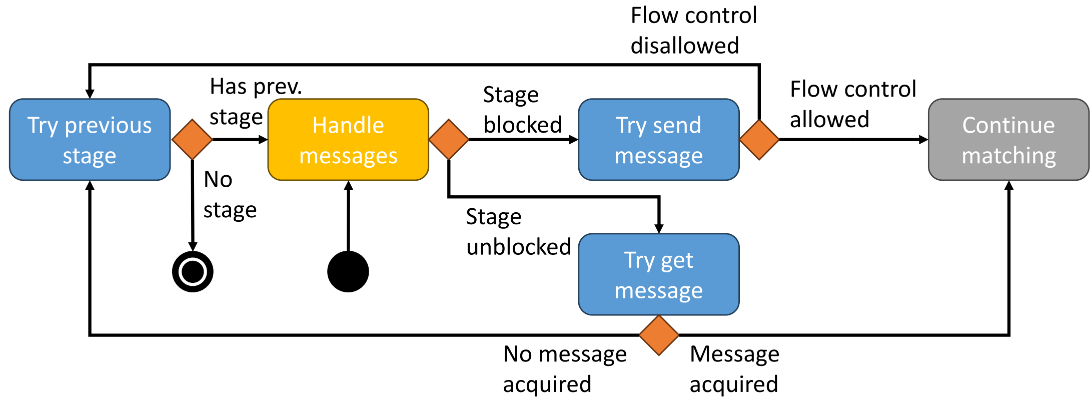


Figure 3.5: Handling incoming messages. The gray box continues to the previous activity diagram showing matching in Figure 3.3.

the intermediate result of edge e_2. Thread `r` *applies stage 2 to match vertex v_3*, evaluates the filter `p2.age < 30` to `false`, *backtracks to previous stage*, and continues with other intermediate results. The matching continues until the whole message is processed or the matching gets blocked by the flow control.

3.3.1 Query Termination

Detecting the termination of a query (or even of a single stage) is not straightforward in PGX.D/Async, because workers operate asynchronously. PGX.D/Async solves the termination-detection problem using a lightweight protocol inspired from the one by Potter et al. [188] to incrementally detect the termination of each stage.

Each machine tracks the completion of a stage locally and notifies stage completion to other machines with special `COMPLETED` messages. Machine k can finish processing Stage n if it knows that all machines have finished processing all stages up to $n-1$ and k has processed all received messages for this stage. At this point, k sends a `COMPLETED` message to all other machines, informing them that they will not receive further messages from k for Stage n. With this termination protocol, the termination of a query is performed incrementally: Stage 0 cannot receive any messages from previous stages by design, hence machines can send their first `COMPLETED` message as soon as they finish bootstrapping the query. Then, Stage 1 can complete after all Stage 0 `COMPLETED` messages are received and the machine has finished processing all messages delivered to Stage 1, and so on.

The termination protocol is deadlock-free: The last stage n can always complete locally, since it includes the always-local output hop. Accordingly, all messages from Stage $n-1$ are eventually processed by Stage n, thus allowing more and more work to be sent from Stage $n-1$ to Stage n. Recursively, as messages from Stage $n-1$ are sent, more work of Stage $n-2$ can be performed and buffered to Stage $n-1$ messages. With this same logic, Stage 0 can be reached, leading to a continuous non-blocking flow from Stage 0 to Stage n. On the receiving end, there needs to be enough receive buffers to guarantee that at least one message for Stage n can be received.

3.3.2 Flow Control

PGX.D/Async allows specifying the total memory size `M` of the messaging buffers that hold the intermediate results in any machine, making it possible to cap runtime memory utilization. Besides these buffers, PGX.D/Async only needs a small per-thread, per-stage, additional memory allocation to hold the current ongoing local match and metadata for thread blocking.

In order to enforce this memory cap, PGX.D/Async employs a simple flow control protocol. PGX.D/Async partitions the buffers that hold intermediate results across the query stages, such that no stage can consume all buffers (required to prevent starvation). When a buffer with intermediate results is full, the corresponding worker requests permission to send the contents of the buffer to the target machine. The flow control protocol keeps track of the amount of data `D` that has been sent to that machine but not yet processed. If `D` is above a threshold (computed based on the memory cap `M`; a machine does not accept more than `M / #Machines` worth of intermediate results from any other), flow control blocks the message transmission (controlled per stage, not for the whole query) and the thread continues with some other work before retrying to send the message. Once a message has been processed, the handling thread informs the source machine that its chunk of intermediate results has been completed and makes the corresponding memory available for another message. Note that this simple protocol *strictly bounds memory consumption*, i.e., no pattern can violate the memory configuration of PGX.D/Async.

3.4 Relational Post-Processing Operations

PGX.D/Async implements the `GROUP BY` and `ORDER BY` relational operators as post-processing steps after the pattern matching has produced its output. For both `GROUP BY` and `ORDER BY`, the system uses "textbook" algorithms [189]. During runtime, these relational operators use additional memory that is proportional to the size of the result set (i.e., create roughly one copy of the data).

`GROUP BY` is implemented in three steps to capture the needs of different workloads. First, the worker threads perform a parallel iteration on all result rows and do a machine-local aggregation using thread-local caches; if the cache has no space, they resort to a machine-local map. Thread-local caches (a fixed capacity maps) brings large performance benefits in case the number of distinct groups is small, e.g., in a query such `GROUP BY gender`. In the second step, the thread-local maps are merged to the machine-local map. For the implementation of maps we use an array-based hash table [190], which preallocates key/value space and is very fast given that PGX.D/Async can size the maps appropriately as it knows the maximum number of keys to be inserted. Accordingly, with a controlled number of buckets and the same hash functions for the maps, merging the thread-local cache to the machine-local one happens without any thread synchronization. Finally, In the third step, PGX.D/Async reshuffles and merges the key-aggregation pairs based on a key hash to generate the final result.

`ORDER BY` also proceeds in three steps. First, PGX.D/Async samples the result rows and generates a distribution histogram of the target values. Based on this histogram, PGX.D/Async assigns ranges of values to different machines in the system. Thus, after the second step, where the results are reshuffled based on that histogram, each machine is holding unique ranges with the invariant that max(range of machine i) $<$ min(range of machine $i+1$). Finally, PGX.D/Async performs local parallel sorting to generate the final ordered result set.

3.5 Evaluation

This evaluation section presented here explains how some parts of PGX.D/Async contribute to the overall performance. These benchmarks should also clarify some configuration settings used in later benchmarks. We present two benchmarks: the first focuses on the performance improvements with ghost vertices, and the second shows the performance of the flow control. For the details on the experimental setup, we refer to Section 4.3.

3.5.1 Performance Improvements with Ghost Vertices

We evaluate the performance of the ghosts. Figure 3.6 compares the latency of different ghost vertex configurations in LDBC with various queries with the latency of the configuration without ghost vertices. The metric is the latency speedup compared to the non-ghost configuration (0%). We use variations of the three-hop query `Q0`:

```
SELECT COUNT(*)
FROM MATCH (a:Person) -[:Knows]-> (b:Person),
     MATCH (b) -[:Knows]-> (c:Person),
     MATCH (c) -[:Knows]-> (d:Person)
```

with different filters that limit the number of traversed vertex matches.

Looking at Figure 3.6, we see that using ghost nodes is always better than not using them. Ghosts are balancing the partitions better among the machines. Without ghost vertices, the machines processing the high-degree vertices become stragglers. On the other hand, we can see that the fastest queries use $20 - 30\%$ of ghost vertices, then the performance starts to get worse again. At that point, ghost message broadcasting, which is required with every iteration over a ghost neighborhood, becomes a bottleneck. Therefore, for social networks, where the number of neighbors follows the exponential distribution, we use 20% ghost vertices.Note that this setting is valid only for neighborhoods with the exponential distribution, e.g., the `Person` vertices in LDBC. For neighborhoods with other distributions, we have not seen any benefits of using ghosts (not shown in this thesis).

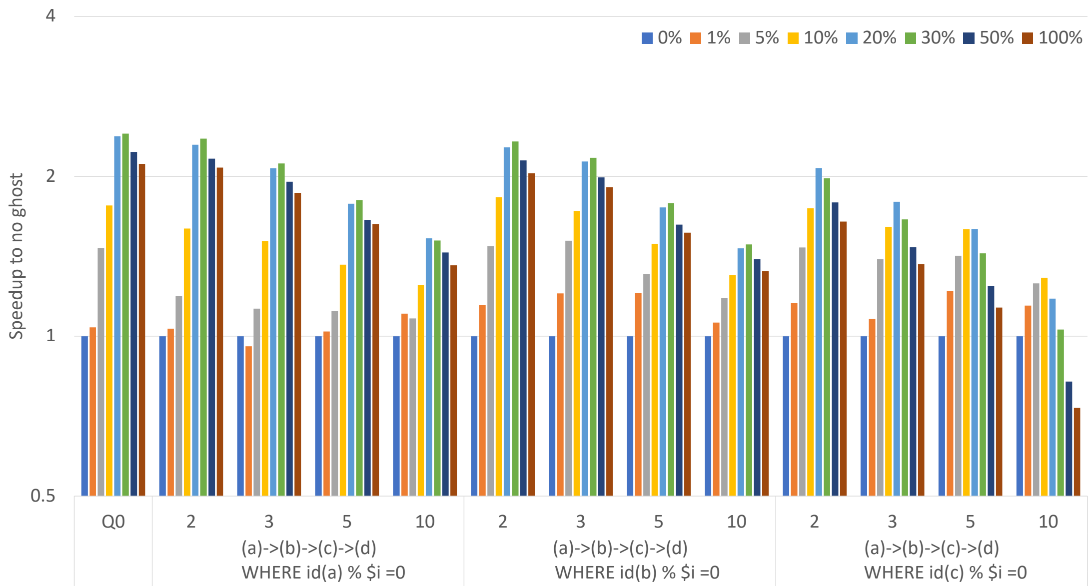


Figure 3.6: Different ghost configurations in LDBC. Each color shows a percentage of ghosts out of all graph vertices.

3.5.2 Flow Control Performance

We evaluate the performance benefits of flow control. Figure 3.7 shows the query execution latency of different flow control configurations in Twitter and Livejournal.

In this experiment, we use a buffer size of 256KB and eight machines. The per-machine limit N is the total number of outgoing buffers that the query execution is allowed to have, therefore it also dictates the maximum amount of memory M that a machine can use during the execution of the query. Since all intermediate results could be targeting a single machine at some point during execution, $M = N \times$ (size of one buffer) $\times$ (# machines).

We execute simple `SELECT COUNT(*)` queries that include basic one-hop patterns `(a)->(b)` (Q1, Q2, Q3) and two-hop patterns such as `(a)->(b)->(a)` (Q4, Q5) and `(a)->(b)->(c)` (Q6, Q7, Q8), with different filters. The figure shows that PGX.D/Async is not very sensitive to different flow control limits, unless the limit is very low, i.e., 512 messaging buffers. The runtime memory footprint of DFS is low on its own, and is not limited by flow control in most cases.

In the case of 512 messaging buffers, we see an extreme case where the flow control allows a single message per worker per stage per machine. On the other hand, the engine in such an extreme configuration, with only an additional 1GB of runtime memory per machine, the engine is able to run and finish the query.

Livejournal Q6 in detail. Figure 3.8 gives more insight into the execution of Q6 with Livejournal: `SELECT COUNT(*) MATCH (a)->(b)->(c)`. The figure shows the maximum number of incoming and outgoing messages for the busiest stage on any of the eight machines, as well as the number of cases in which the flow control limits were reached. For very low limits ($N = 512$ messages) the amount of blocking is very high, which penalizes performance (more than 3$\times$ higher latency). Still, the overhead for switching stages due to flow control is generally low: Setting N to 8,192 results in only ~10% performance loss as compared to no flow control (`OFF`), while reaching 10$\times$ fewer maximum incoming messages (2,087 vs. 21,793) and 4$\times$ fewer outgoing messages (1,636 vs. 6,430).

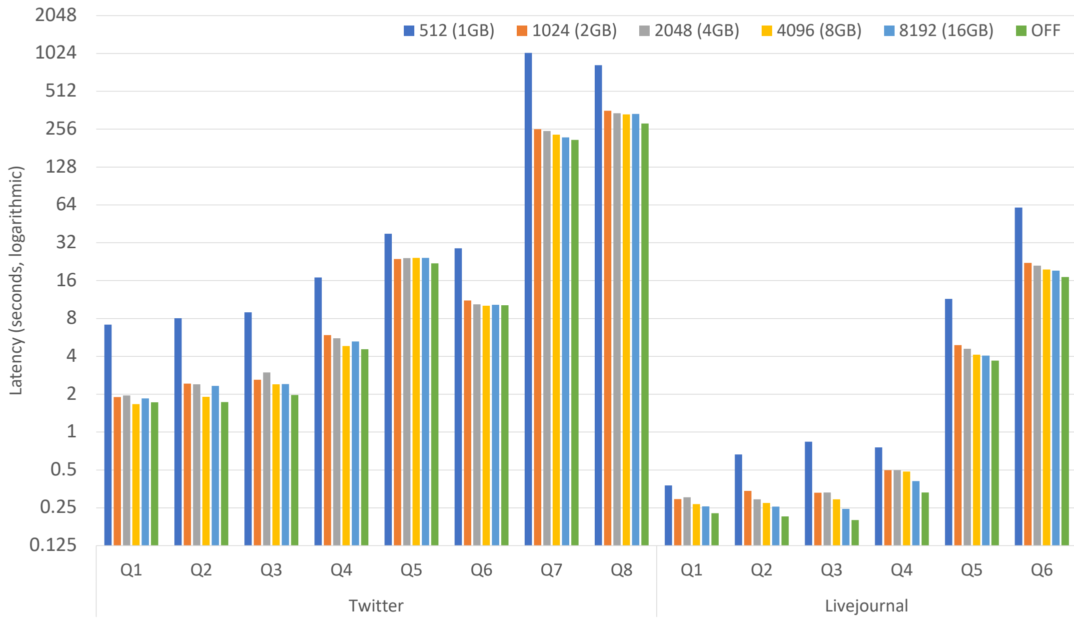


Figure 3.7: Performance of simple queries with eight machines on two graphs with different flow control limits and with flow control turned off. The numbers in parentheses indicate the total per-machine maximum memory consumption.

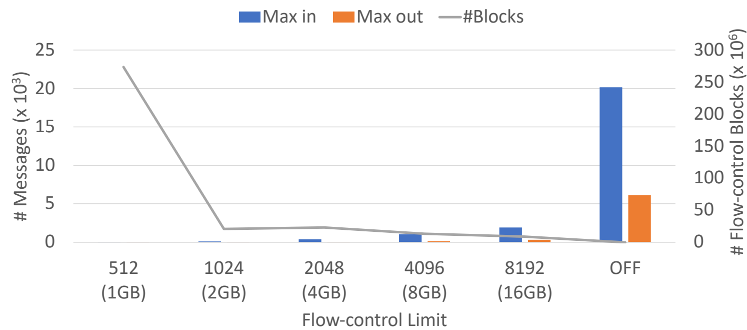


Figure 3.8: Messaging and blocking statistics on Q6 in Livejournal with different flow control limits. In parentheses: Total per-machine max. memory consumption.

4. An Almost Depth-First Search Distributed Graph-Querying

In this chapter, we introduce aDFS: A novel distributed graph-querying approach that allows to process practically any query fully in memory, while maintaining bounded runtime memory consumption and great performance. To achieve this, aDFS builds on top of PGX.D/Async (Section 3), which allows a *non-blocking dispatching of intermediate results to remote edges*, and changes the runtime into using *almost depth-first (aDFS) explorations with some breadth-first characteristics*.

We evaluate aDFS against state-of-the-art graph querying (Neo4J and GraphFrames for Apache Spark), graph mining (G-Miner, Fractal, and Peregrine), as well as dataflow joins (BiGJoin), and show that aDFS significantly outperforms prior work on a diverse selection of workloads.

Organization. The rest of this chapter is organized as follows: Section 4.1 introduces and motives the problem. It also presents other approaches and the related work. Section 4.2 presents the aDFS runtime. Section 4.3 shows the evaluation of aDFS with other state-of-the-art systems. Finally, Section 4.4 concludes this chapter.

4.1 Introduction

The dynamic user-defined patterns, filters, and projections, the focus on edges, and the homomorphic matching make graph query execution a challenging workload that needs to handle very large intermediate and final result sets, with a combinatorial explosion effect. For example, on the well-researched Twitter graph [191], the single-edge query `(a)->(b)` matches the whole graph, amounting to 1.4 billion results, and the two-edge query `(a)->(b)->(c)` amounts to 9.3 trillion matches. This means matching the `(a)->(b)->(c)->(a)` cycle needs to consider 9.3 trillion intermediate results. Compared to relational queries, graph queries can exhibit extremely irregular access patterns [13, 192] and lack of spatial locality, while calling for low-latency data access.

Query execution on graphs is typically based on one of the two classic graph-traversal strategies (presented in Section 2.2: depth-first search (DFS) or breadth-first search (BFS). Both DFS and BFS have major advantages and drawbacks for distributed graph queries:

DFS can expand one intermediate result at a time, starting from the first variable in the pattern and continuing to the next ones until the whole pattern is matched. This reduces the size of intermediate results, but it is challenging to parallelize and results in random data access patterns, which makes it impractical for distributed graph traversal: the only way to continue with strict DFS is to directly send the intermediate result to the remote machine and wait until it is picked up and completed.

Thus, graph exploration is traditionally done using BFS: For each query edge (hop), the entire result set is computed, and only then does the exploration of the next hop start. This approach has two main advantages: (i) it is easy to implement, as work is naturally divided into simple steps (hops), and (ii) it is relatively easy to parallelize, as the entire input is known before processing a hop (of course, skewed vertex degrees still pose a problem). However, BFS has one major shortcoming: Because the intermediate

result set is produced between stages, an intermediate result-set explosion can quickly occur.

Figure 4.1 illustrates this issue showing the average total per machine memory usage and execution time when matching cycles of various lengths using DFS and BFS (both implemented on top of the PGX.D/Async) on a small graph [193] (875K vertices and 5.1M edges). While both approaches are able to match cycles of length one to four with similar performance, the memory consumption of BFS explodes for five-hop cycles at approximately 60GB on each of the eight machines in the experiment, and BFS crashes with six-hop cycles after 96 minutes when one machine runs out of memory (~768GB). Meanwhile, the memory consumption of DFS is almost constant. Note that we can see the same problem with BFS engines on real queries as well (see evaluation in Section 4.3).

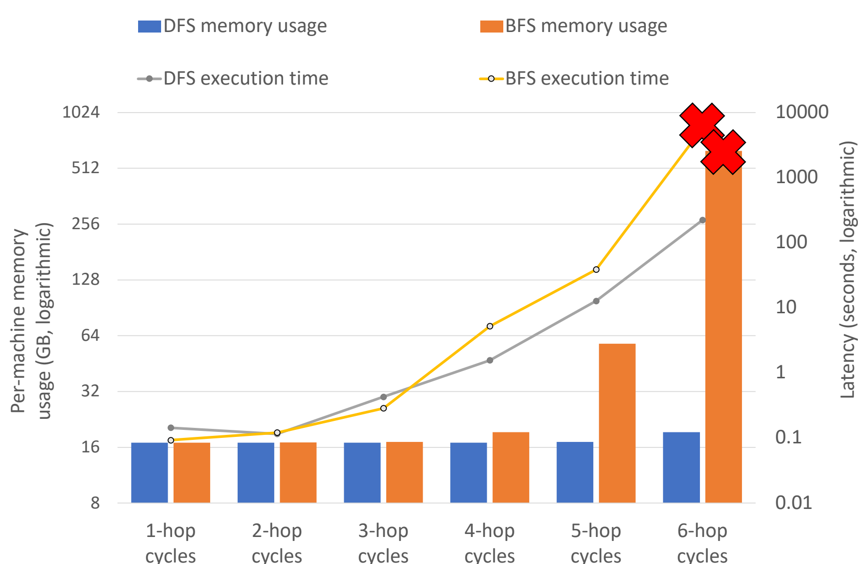


Figure 4.1: Matching cycles using DFS vs. BFS.

In this chapter, we introduce aDFS (almost DFS): A novel distributed graph querying algorithm that brings the best of both DFS/BFS worlds. aDFS improves the performance of PGX.D (Section 3) by combining BFS and DFS traversals to *bound the maximum amount of memory* required for query execution, while achieving a *high degree of parallelism*. DFS, together with a distributed flow control mechanism, guarantees that the amount of runtime memory remains within limits, while the BFS exploration allows for better locality and parallelization during execution.

Worker threads in aDFS mainly prioritize DFS execution for completing—and thus freeing—intermediate results. The execution switches to BFS when matching a remote edge (i.e., an edge pointing to a remote machine) or when the runtime detects that the query contains limited parallelism (i.e., a small set of intermediate results). To elaborate, for local edges, worker threads perform DFS, unless aDFS detects that there is a limited amount of available work on the local machine, in which case they switch to per-thread BFS exploration until there is enough parallelism. For remote edges, threads buffer the matched intermediate results and continue with matching the next edge in a BFS manner (i.e., the next edge is possibly at the same depth as the current

one). Once a buffer is full, the worker thread sends its contents to the target machine, unless it is blocked by the flow control mechanism, which enforces target memory limits. Section 4.2 expands on the design and implementation of aDFS.

Section 4.3 thoroughly evaluates aDFS and shows that it is capable of executing trillion-scale queries, with a 10GB per-machine runtime memory cap. When running our largest query, aDFS computes a 9.3 trillion count pattern on the Twitter graph with a rate of 3.5 billion matches per second. We compare aDFS to two graph systems (i.e., Apache Spark GraphFrames [31] and Neo4j [16]) and two relational databases (i.e., MonetDB [118] and PostgreSQL [119]) using adaptions of LDBC [194] benchmarks. aDFS completes the set of queries 1,315 times and 56 times faster than Neo4j[1] and PostgreSQL, respectively. The other two engines, GraphFrames and MonetDB, struggle to finish all the queries, and time out or go out of memory. On the queries that they are able to finish is aDFS 62 times and 15 times faster than GraphFrames and MonetDB, respectively.

We also compare aDFS to these four systems with schema-less graphs and show that either aDFS is 8 to 3,247 times faster than the rest, or the other systems simply fail to complete the queries. Finally, we compare aDFS with (i) three state-of-the-art graph mining systems: G-Miner [22], Fractal [169] and Peregrine [23], as well as (ii) BiGJoin [173], a dataflow join system. We show that aDFS is up to 12×, 625×, and 18× faster than G-Miner, Fractal and Peregrine, respectively, and performs comparably to BiGJoin on mining-oriented workloads.

4.1.1 Related Work

A number of single-node graph-querying systems were proposed by academia: Sun et al. [195] and Lin et al. [196] build relational and transactional systems, Graphflow [197] is an active graph database that supports evaluating one-time and continuous subgraph queries, and CECI [187] uses multiple embedding clusters and intersections of neighborhood lists to optimize subgraph matching (CECI can be distributed through graph replication, not through graph distribution as aDFS).

There are numerous industrial graph-querying solutions: Neo4j [16] is single-machine graph database, Amazon Neptune [28] is a graph engine built for the Amazon cloud, Microsoft Graph Engine is an in-memory data processing system based on Trinity [13], and TigerGraph [17] distributes GSQL [5] queries based on the source vertex data for a given query hop. Furthermore, there are also open-source distributed solutions. JanusGraph [128] uses distributed graph storage but does not distribute computation. GraphFrames [31] implements graph pattern matching with Spark using joins of dataframes. Wukong [105] is a distributed graph-based RDF store that leverages hardware features, such as RDMA and GPUs.

To the best of our knowledge, aDFS is the first truly distributed graph-querying system that works on fully-partitioned graphs and strictly bounds memory while maintaining great performance.

Graph-Mining systems. Recent single-machine systems include RStream [170], AutoMine [171], and Peregrine [23]. Distributed systems include NScale [172], Arabesque [20], G-thinker [198], BiGJoin [173], G-Miner [22], ASAP [199], and Fractal [169]. aDFS shares features with some of these systems. For example, forms of

[1]Using Neo4j Community Edition (benchmarks not audited by Neo4j).

asynchronous computations are used in G-Miner [22] (with a "task-pipeline" to hide communication overheads) and BiGJoin [173] (with data-parallel dataflow computations that pick up dynamically joined columns with the least matches). Techniques to reduce memory consumption are used by G-Thinker [198] (buffering excess subgraph-tasks in a disk-based priority queue), BiGJoin [173] (primarily using batching to limit memory consumption but not for intermediate results as with aDFS) and Fractal [169].

Fractal combines a DFS strategy with a "from-scratch processing" paradigm which leads to re-computation overheads (absent in aDFS), as well as imbalances across workers that are mitigated by work stealing: workers break the DFS strategy to steal enumerations, which can be at any level of the matched graph pattern, from other workers. aDFS uses asynchronous DFS-based graph traversals together with flow control to strictly bound memory consumption, and can switch to BFS, in the same graph pattern-matching level, to generate more local work and to buffer remote edges (see Section 4.2). Our in-depth evaluation shows that the performance of aDFS for graph pattern-matching is competitive with that of state-of-the-art graph-mining systems.

BFS/DFS. The BFS/DFS tradeoff has been explored in the context of single-machine parallel task-scheduling runtimes. Typically, DFS is used to schedule a task graph in order to curtail memory [200], and BFS is used opportunistically (often called "work stealing") to maximize parallelism [201, 202]. aDFS leverages these insights in the context of distributed graph query processing.

4.2 aDFS: A Pattern Matching and Querying for Distributed Graphs

The main design goals of aDFS are (i) enabling fast, fully in-memory distributed queries of any size, while (ii) allowing for limited, controllable memory consumption during execution. The rationale for these two goals is as follows. First, high-performance graph queries demand in-memory execution and the ever-increasing size of data calls for distribution. Second, server systems, especially in cloud deployments, are shared by multiple concurrent users, hence no single query can be permitted to saturate the system memory. aDFS achieves these two goals through the following design principles:

- **DFS-first and asynchronous communication (Section 3.3).** The eager match completion of DFS gives aDFS fine-grained control on the size of intermediate results during query execution, but strict DFS would be inefficient when matching a remote edge, i.e., an edge that leads to a remote machine. For that reason, worker threads do not block when encountering a remote edge, but place the intermediate result in a message buffer and continue with other local work instead. Buffers batch intermediate results: once full, a buffer's contents are asynchronously sent to the remote machine for further processing. Threads only need to block if flow control dictates so. This buffering results in essentially BFS exploration of the remote edges of a vertex. aDFS extends the DFS traversal and asynchronous communication from PGX.D/Async (presented in Section 3.3).

- **Flow control (Section 3.3.2).** Cross-machine communication is controlled through a flow control mechanism that caps the number of in-flight intermediate result buffers. The finite nature of these message buffers allows strictly configuring the amount of runtime memory that aDFS requires, while the flow control

mechanism guarantees query termination and deadlock freedom. For aDFS, we reuse the flow control mechanism of PGX.D/Async presented in Section 3.7.

- **Dynamic local DFS/BFS (Section 4.2.2).** Besides the buffering style BFS for remote edges, aDFS includes a dynamic approach for deciding whether to go DFS or expand with BFS for local matches in order to improve parallelism, locality, and work sharing across threads.

4.2.1 High-Level aDFS Architecture

The aDFS builds on top of the PGX.D/Async engine presented in Section 3. Therefore, aDFS can reuse the infrastructure and the architecture of PGX.D/Async. Nevertheless, aDFS requires some changes in runtime to allow BFS pattern matching together with DFS, which we present here. For a more detailed architecture description, we refer to Section 3.1.

Architecture overview. Graphs are kept in memory and partitioned across machines. For efficient traversals, graphs are stored in the classic CSR (Compressed Sparse Row) graph format. Due to graph partitioning, messaging is necessary for moving intermediate results to the machine which holds the target vertex. aDFS maintains two threads on dedicated cores on each machine for messaging; a sender and a receiver. Consequently, worker threads in aDFS place their messages in software queues, from where they are picked up by the sender.

Before running an actual query, the query is transformed into an execution plan. The transformation process, query planning, is the same as for PGX.D/Async (presented in Section 3.2). The execution plan consists of stages and hops forming connections between stages. The aDFS runtime executes this plan by matching vertices to the stages and edges to the hops. A detailed runtime execution is described below.

4.2.2 aDFS Runtime

aDFS initiates query execution by applying Stage 0 (matching of the first vertex variable of the execution plan) to each vertex of the graph. This bootstrapping process happens (i) across machines, i.e., each machine starts from the locally-stored vertices, and (ii) concurrently within each machine, i.e., each worker thread handles a distinct set of vertices and performs the bootstrapping process on these vertices one after the other.

aDFS matching. Figure 4.2 includes a high-level activity diagram of the aDFS runtime, similar to that presented for the PGX.D/Async runtime in Figure 3.3. The green boxes depicts the activities related to BFS. Completing the execution of this diagram from Stage 0 to the last query stage implements the complete matching starting from a single vertex of the graph. Text in the *blue italic face* represents the activities in that figure.

For remote edges aDFS essentially does (per-thread) BFS: A thread matching a remote edge simply buffers the intermediate result (*Buffer into a message*) and continues exploring and matching the same stage, which might produce new intermediate results.

While local processing could happen in pure DFS, doing so can result in artificially limited parallelism for queries that produce small sets of intermediate results. A characteristic example is queries with a very narrow starting Stage 0, such as `MATCH (a)->... WHERE ID(a) = X`; this narrow-start behavior appears in several real-life queries (e.g., the LDBC queries of Section 4.3). In such a query, the whole Stage 0

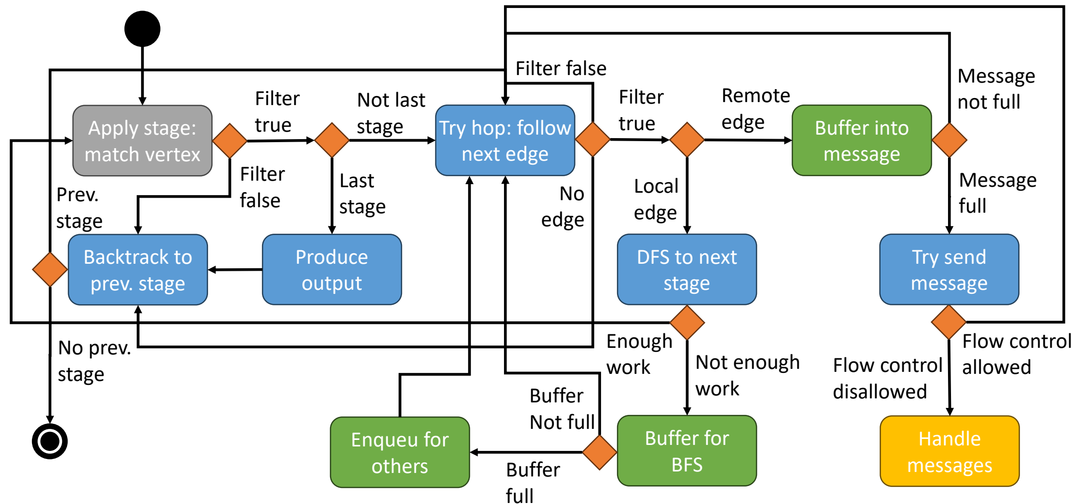


Figure 4.2: Matching operations starting from a given vertex in aDFS. The yellow box "Handle messages" connects to the activity diagram in Figure 4.3.

might produce a single intermediate result, giving limited opportunities for parallelism. For these workloads, DFS can significantly delay the expansion of intermediate results that are produced in the system (both locally and through messages).

In aDFS, we solve this DFS limitation by dynamically switching from depth-first exploration to per-thread breadth-first for local edges. aDFS maintains per-stage counts of the number of buffers with intermediate results that are ready to be taken care of by worker threads. A low number of intermediate results means that the stage has not expanded enough, hence some threads could end up not having sufficient work to perform. When threads in aDFS are processing a local edge, they use this information to decide whether to go for BFS, i.e., buffer the intermediate result in a local buffer (*Buffer for BFS*) and continue at the same stage.

Once the buffered BFS messages are full, they got enqued into a BFS software queue for other threads to pick up. Threads can acquire the message in the *Han-*

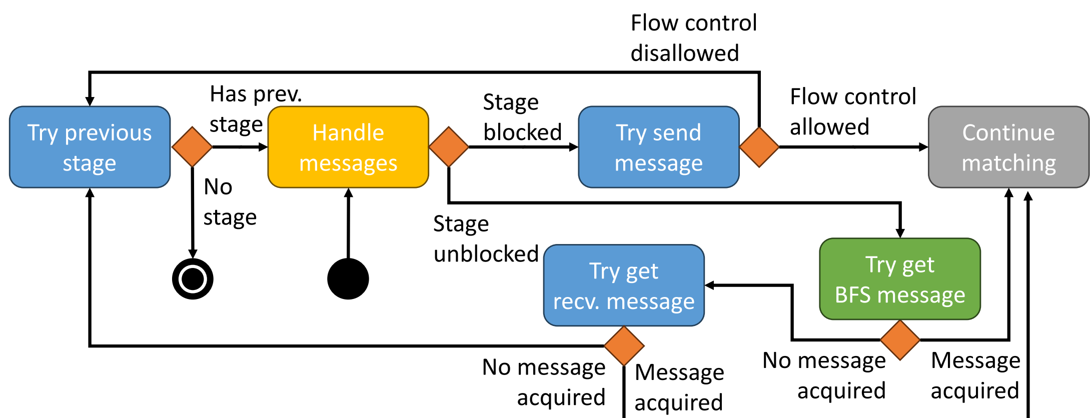


Figure 4.3: Handling incoming messages in aDFS. The gray box continues to the previous activity diagram showing matching in Figure 4.2.

dle messages before acquiring received messages from other machines. Figure 4.3 shows the action diagram of the message processing containing the BFS-related box in green. *Handle messages* starts from the last stage. The work performed when a stage is blocked is the same as in PGX.D/Async runtime presented in Figure 3.5. If the stage is unblocked, aDFS *tries to get a local BFS message* first. If it is successful, it initiates the matching from vertices of the BFS message. In the opposite case, it continues by *trying to get a received message* as the original PGX.D/Async runtime does.

In practice, we keep these local BFS buffers small, i.e., up to a few kilobytes, in order to promote quick local work creation. We further use a *DFS threshold* to decide when to work depth-first: When the sum of the number of local buffers (produced by the breadth-first expansion) plus the number of message buffers from remote machines is greater than 4$\times$ the number of threads, threads switch to DFS. Having a low threshold plus small local buffers allows aDFS to keep the maximum additional memory consumption limited: If the DFS threshold is set to n, the maximum number of threads is t, the size of local buffers is b, and the query contains s stages, the maximum additional memory in a machine is $(n + t) * (s - 1) * b$ ($n + t$ because the t threads could concurrently detect that there is not enough work and produce local buffers. $s - 1$ because the last stage does not produce intermediate results.) In the configuration used for our experiments ($t = 28$, $n = 4t = 128$, $s \leq 11$, and $b = 8{,}192$), local buffers consume less than 12MB additional memory.

4.3 Evaluation

The goals of our evaluation are (i) to understand how well aDFS performs as compared to other systems (graph, relational, mining and dataflow join systems) that could be used in similar use cases, (ii) explain how different parts of aDFS contribute to performance and memory, and (iii) show how aDFS scales as we increase the number of machines.

4.3.1 Experimental Settings

Hardware details. We use a cluster of eight nodes, each having two Intel Xeon E-2690 v4 2.60GHz CPUs with 14 cores (hyperthreads disabled/DVFS enabled), for 28 cores in total. Each machine contains 756GB of DDR4-2400 memory and LSI MegaRAID SAS-3 3108 storage. Each node includes a Mellanox Connect-X InfiniBand card, all connected to an EDR 100Gbit/s InfiniBand network.

Graphs and queries. Unless specified otherwise, our experiments use the six graphs in Table 4.1. More details about the LDBC graphs can be found in Section 2.5. For

Graph	#V	#E	Schema	Description
Livejournal [203]	484K	68.9M	No	Users and friendships
URandom	100M	1B	No	Uniform random edges
Twitter [191]	42.6M	1.47B	No	Tweets and followers
LDBC SF100 [194]	255M	1.7B	Yes	LDBC social graph
LDBC SF300 [194]	738M	5.1B	Yes	LDBC social graph
Webgraph-UK [204]	77.7M	2.97B	No	2006 .uk domains

Table 4.1: The set of graphs we use in the evaluation. We show number of vertices (#V) and number of edges (#E) in thousands (K), millions (M), and billions (B).

comparison with other engines, we use the smaller-scale SF100, since the engines (except for aDFS) struggle even with this smaller graph. For other experiments with aDFS only, we use the larger SF300 scale.

We use adaptations of 13 LDBC Business Intelligence (BI) standard queries [180] to test a graph-specific workload. We adapt the queries as the target of this work is fixed-pattern read-only queries. Through the evolution of LDBC, the queries became so complex that only one query (`Q1`) uses read-only fixed-pattern matching. The others require more complicated features, such as subqueries, variable-size patterns, or update queries. Therefore, we devised a simplified variant of these queries in order to support the benchmark specification as closely as possible. We simplified the queries by removing these extra constructs and replaced them with a fixed-pattern matching. After that, we also removed queries that were too simple, e.g., with a small number of hops, or queries that were similar to other queries. For reference, all queries can be found in Appendix A.

Here is an example of the query `Q7`. The specification [180, 205] describes the query as follows: "*Find all Messages that have a given $tag. Find the related Tags attached to (direct) reply Comments of these Messages, but only of those reply Comments that do not have the given $tag. Group the related Tags by name, and get the count of replies in each group.*" The query can be represented in PGQL as follows:

```
// original
SELECT relatedTag.name, COUNT(DISTINCT comment) AS count
FROM
  MATCH (m:Message) -[:hasTag]-> (tag:Tag),
  MATCH (m:Message) <-[:replyOf] - (comment:Comment),
  MATCH (comment) -[:hasTag]-> (relatedTag:Tag)
WHERE
  tag.name = 'Enrique_Iglesias' AND
  NOT EXISTS (SELECT * FROM MATCH (comment) -[:hasTag]-> (tag))
GROUP BY relatedTag
ORDER BY count DESC, relatedTag.name
LIMIT 100

// simplified
SELECT relatedTag.name, COUNT(DISTINCT comment) AS count
FROM
  MATCH (m:Message) -[:hasTag]-> (tag:Tag),
  MATCH (m:Message) <-[:replyOf] - (comment:Comment),
  MATCH (comment) -[:hasTag]-> (relatedTag:Tag),
  MATCH (comment) -[:hasTag]-> (tag)
WHERE tag.name = 'Enrique_Iglesias'
GROUP BY relatedTag
ORDER BY count DESC, relatedTag.name
LIMIT 100
```

The red color highlights the clause that we removed, i.e., an existential subquery, and its transformation into new query. We cannot completely remove the subgraph matching pattern because that would oversimplify the query. Therefore, we include the pattern from the subquery into the parent query to emulate the "*do not have the given $tag*" requirement from the description. Note that the new query might be even more challenging, because it must enumerate all the `(comment)->(relatedTag)` matches while with existential subqueries, a query engine can, e.g., enumerate until it finds a match and continue to another matching instance, or use *anti-edges* [23].

Methodology. We perform 10 runs of each query and report the median latency. For each experiment set, we execute the queries in a per-graph round-robin fashion in order to reduce caching effects (e.g., data in the LLC or instruction caches). There is a timeout of 4 hours for a single query. We use eight machines for aDFS, GraphFrames, G-Miner, Fractal as well as BiGJoin, and make sure all systems are configured to use InfiniBand. The four other systems are single machine.

Engines and their configurations. We configure aDFS to use up to 4,096 messaging buffers of 256KB per machine for messaging. This setting translates to approximately 1GB of intermediate results that can be produced per machine and limits the worst-case maximum memory consumption of a single machine to approximately 8GB (1GB outgoing, plus 7GB incoming). For the local-edge dynamic BFS, we use the configuration mentioned in Section 4.2.2, i.e., $t = 28$, $n = 4t = 128$, $s \leq 11$, and $b = 8{,}192$, resulting in up to a few MBs of extra memory per machine. Altogether, the aDFS runtime consumes approximately 10GB per machine. Of course, the graph (that resides in memory) and the final query results consume extra memory than these 10GB. We use such a low-memory configuration because (i) aDFS is designed for server deployments and we want to evaluate the performance at a realistic setting, where a single query cannot monopolize memory, and (ii) as we showed in Figure 3.7, this configuration is already sufficient for aDFS to perform well. We disabled ghosts vertices and used a random vertex partitioning to be able closely compare the runtimes of other engines that do not implement any of these improvements.

We first compare aDFS to two graph systems and two relational systems which we describe below. In Section 4.3.6, we further compare aDFS to three graph mining systems and a dataflow join system.

GraphFrames [31] is a distributed graph querying system built on top of Apache Spark [112]; we use version 0.8.2 on top of spark 3.2.1 with 600GB executor memory per machine. ***Neo4j*** [16] is a single-machine graph database, which stores its data on disk but uses an in-memory cache for performance (caching effects are obvious in the first run of each query). We use Neo4j Community Edition 4.0.10 and allow it to manage the full machine memory. Neo4j is configured according to the Neo4j-admin Memrec [206] utility. We use this older version, because it has the best performance with the same configuration compared to the new versions (tested up to version 5.4). ***MonetDB*** [118] is an in-memory column-store relational database. Its distributed support is rather rudimentary, resulting in worse than single-machine performance for our join-heavy workloads. Therefore, we use MonetDB v11.45.7 on a single machine, configured to use the whole 756GB of memory. ***PostgreSQL*** [119] is a relational database. We use version 15.2, tuned according the PGTune [207] for a single connection with a memory cache size of 564GB and 198GB of shared buffers. For both MonetDB and PostgreSQL, we use the optimized schema/indices designed for the original LDBC evaluation paper [194]. We choose these four systems as they cover a broad spectrum of data processing: Distributed graph dataframes, single-machine graph databases, and in-memory or traditional relational databases.

4.3.2 Benefits of Local BFS

Figure 4.4 illustrates the benefits of the local-match BFS mode on 8 machines with the following two queries:

```
// Q1
SELECT COUNT(*)
FROM MATCH (a)->()->()
WHERE ID(a) < $i

// Q2
SELECT COUNT(*)
FROM MATCH (a)->()-[e]->()->()
WHERE e.cost < 0.5 AND ID(a) < $i
```

using the Twitter graph extended with a uniform random edge property with values in $[0.0, 100.0)$. In both queries, the `ID(a) < $i` filter determines the cardinality of the first query stage and is used to narrow the starting point. In Q2, the edge filter also guarantees that the third stage includes a small number of intermediate results.

Results. The dynamicity of aDFS brings significant performance benefits, especially for queries with very narrow starting points. For example, for Q1 with `$i = 1`, Machine 0 hosts the match for Stage 0; without the breadth-first mode ("OFF"), a single thread handles all the 55K local edges which lead to Stage 1. In contrast, enabling dynamic local BFS ("ON") generates more work early on and allows splitting the work among local threads, each of which operates on approximately 2,000 vertices for Stage 1.

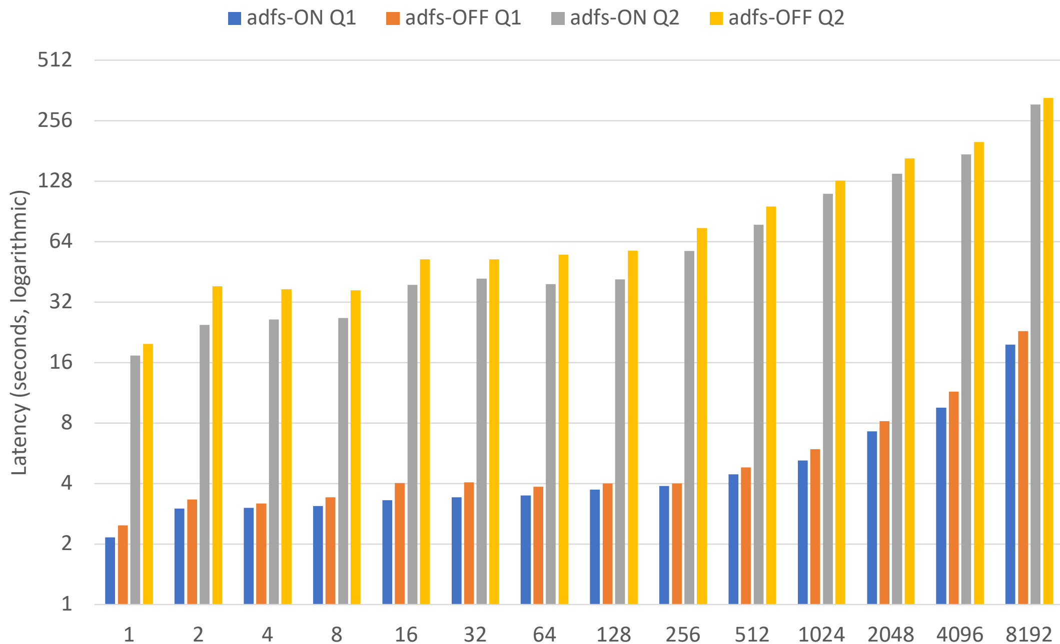


Figure 4.4: aDFS dynamicity based on the first stage cardinality

4.3.3 Dissecting aDFS

We use the LDBC benchmark to show how different design characteristics of aDFS contribute to performance and memory usage. In particular, we compare the pattern-matching latency of the default *aDFS* with flow control on (*aDFS (FC ON)*) to aDFS with disabled flow control (*aDFS (FC OFF)*) without limiting memory, *locDFS*, and *BFS*. *locDFS* here represents the PGX.D/Async engine before the aDFS performance improvement. As mentioned in Section 3, it is not purely DFS. The local traversals

are always DFS, however, the remote hops are buffered into messages and sent asynchronously later, which does a partial BFS traversal. We did not test a pure DFS traversal, since that would require a single message per hop without buffering, which inherently performs much worse than the *locDFS* solution.

Results. Figure 4.5 includes the results for these configurations. The results show that DFS is the slowest with a total execution time of 42.5s, then we have *aDFS(FC ON)* with 29.9s, *aDFS(FC OFF)* with 27.7s, and *BFS* with 27.1s. In terms of memory consumption, BFS is 5× worse than locDFS, aDFS without flow control is 4× worse, and aDFS with flow control consumes slightly more memory than *locDFS*, not only due to the local buffers, but also thanks to better parallelization, resulting in more parallel message traffic.

As we described earlier, queries often have very "narrow" execution points with a handful of intermediate results, which leads to poor parallelization with DFS. As a reference, *BFS* implements a basic BFS-only runtime. As expected, *BFS* performs the best compared to other configurations. aDFS without flow control is 2% worse in performance but consumes less memory. aDFS with flow control is 10% worse, but compensates for it with 4× less memory.

In conclusion, aDFS includes a set of design characteristics that when put together achieve great performance with low and controlled memory consumption.

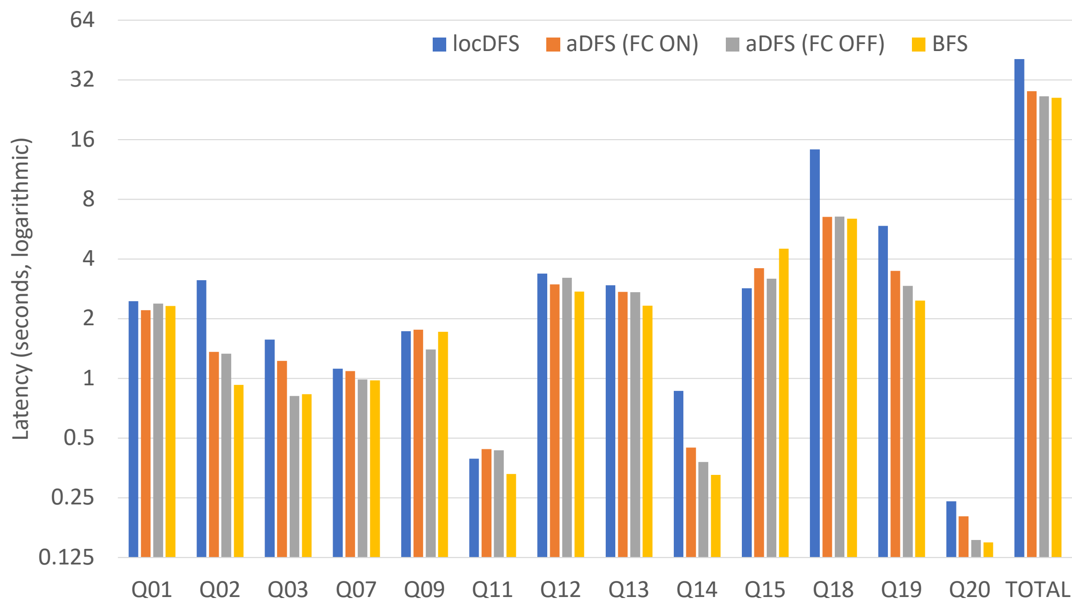


Figure 4.5: LDBC SF100: locDFS vs. aDFS vs. BFS.

4.3.4 aDFS vs. Other Engines: LDBC

We perform an end-to-end comparison of aDFS to the four aforementioned systems. We use the LDBC graph and BI queries as a graph specific benchmark.

Results. Figure 4.6 depicts the query latencies of the five systems. For most queries, Neo4j is much slower than aDFS. Even though we compare single machine engine to eight machine engine, the performance difference is huge, in total it is more than 1,315× slower, on average more than 963× slower. The minimum difference shows `Q07`, which is still 10× slower.

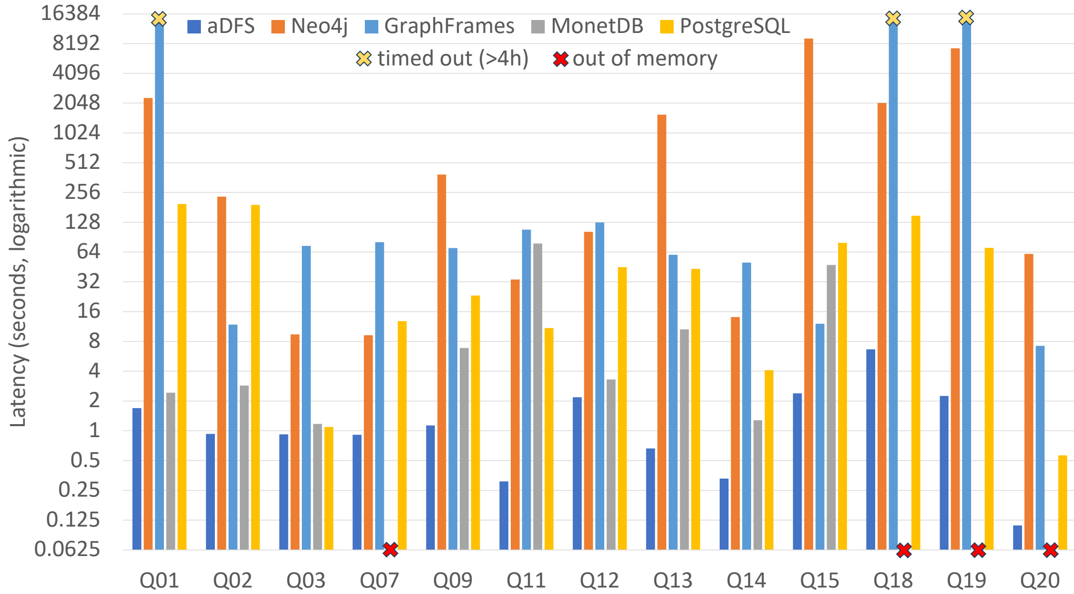


Figure 4.6: LDBC SF100: aDFS vs. other engines.

GraphFrames is a distributed engine that translates graph queries into dataframe joins, offered by Apache Spark. One disadvantage of GraphFrames is that all the vertices are stored as a single table and the edges are stores as another single table. There is no partitioning into multiple tables according to labels as in most systems. Within the four-hour time limit, GraphFrames was able to complete 10 out of 13 queries and was 62× slower in total and 97× slower on average compared to aDFS. Additionally, GraphFrames is memory hungry, consuming hundreds of gigabytes of memory compared to the small footprint of aDFS.

We compare aDFS with two relational systems, MonetDB and PostgreSQL. MonetDB was able to finish only 9 out of 13 queries due to going out of memory on a single machine during the execution. Taking into consideration only the finished queries, it was overall faster than PostgreSQL, but still 15× slower in total and 34× slower on average, compared to aDFS. On the opposite, PostgreSQL was able to finish all queries, but it was much slower, with 56× slower execution in total and 67× slower on average, compared to aDFS.

We can see that aDFS outperforms the other engines in all queries. The smallest difference is with `Q03` where the matching execution time is only 0.3 seconds (around 45% of time total time) and the remaining time is spent in post-processing. As mentioned earlier in Section 3.4, aDFS uses a basic implementation of those operations. Also, those post-processing operations are not a graph specic, because they require a relation-style of processing for which are RDBMSes (MonetDB or PostgreSQL) optimized.

Focusing on queries with pattern matching only, such as `Q11`, `Q14`, `Q15`, and `Q20`, we see that aDFS performs really well. In total, it is 2,967×, 57×, 40×, and 30× faster than Neo4j, GraphFrames, Monet, and PostgreSQL, respectively. This clearly shows the advantages of our graph system over relational system on graph-related workloads due to a focus on pattern matching. The biggest advantage of our distributed graph querying system is shown in large queries with a minimal ratio of post-processing time

to the total time, i.e., in Q18 and Q19. On those large workloads, MonetDB goes out of memory, GraphFrames runs out of time, and Neo4j with PostgreSQL are 1,056× and 3,235× slower, respectively.

In conclusion, aDFS achieves better overall performance than the four other systems while consuming lower/capped runtime memory. Both these characteristics are essential for a graph processing server which targets large workloads and possibly multiple concurrent users.

4.3.5 aDFS vs. Other Engines: Large Schema-Less Queries

The original property graph model is schema-less, which enables users to easily query the whole dataset (unlike the relational model which requires several joins and unions of results). Therefore, we now compare aDFS to the other four systems with the schema-less graphs of Table 4.1: this workload shows the full power of aDFS in handling very large queries. For the relational systems, the graphs consist of two tables: One for vertices and another one for edges. Regarding queries, we use two simple patterns, a cycle `(a)->(b)->(a)` as `Q1` and a two-hop path `(a)->(b)->(c)` as `Q2`, combined with aggregations in the `SELECT` clause (variant "a" performs a `COUNT(*)` and variant "b" `AVG` aggregations on a random vertex property). The conclusions remain the same for other patterns and projections (not shown). Note that it is impossible to evaluate more elaborate patterns, as the competing systems can barely handle the simple patterns that we use.

Results. Figure 4.7 depicts the results. In most cases, aDFS is about 2 orders of magnitude faster than the other systems. For the large queries and graphs, we also see that the other systems are either not able to complete the queries within four hours, or crash. In particular, GraphFrames crashes after having consumed its 600GB of executor memory.

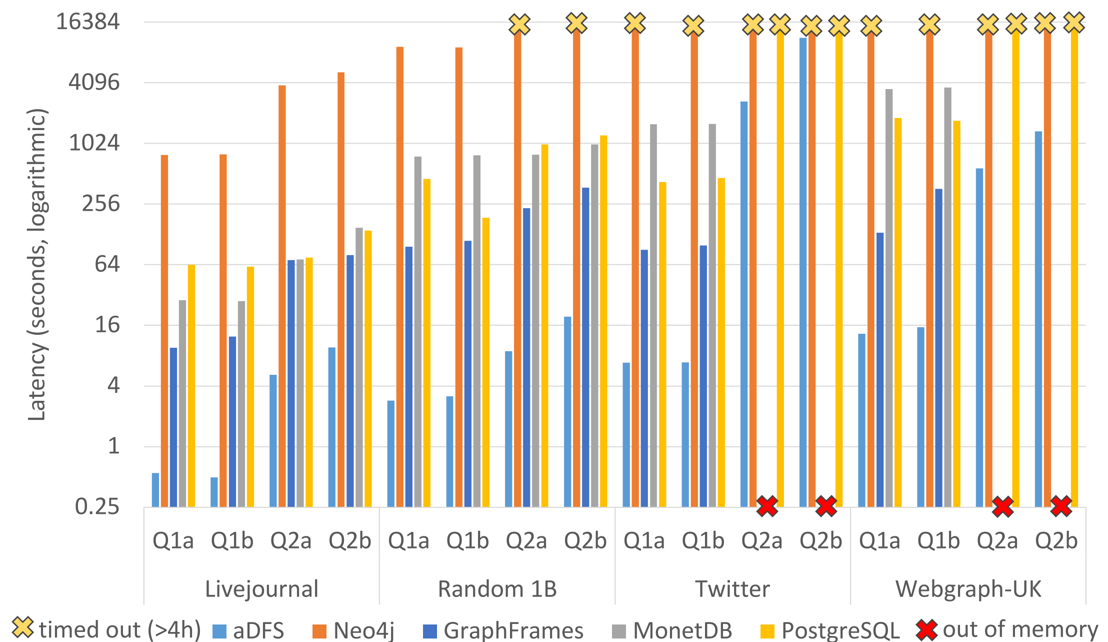


Figure 4.7: aDFS vs. other engines on simple pattern queries.

The speedups of aDFS over the other systems (for the completed queries where there is no timeout) are: 8 to 35× for GraphFrames, 541 to 3,247× for Neo4j, 14 to 269× for MonetDB, and 15 to 158× for PostgreSQL. Neither the join-based systems (GraphFrames, MonetDB, and PostgreSQL) nor Neo4j are able to handle these immense graph explorations well, although they have access to hundreds of gigabytes of memory. In particular, Neo4j spills to disk, hence the extreme performance difference compared to aDFS. Clearly, for graphs and queries at this scale, a fast graph-optimized solution such as aDFS, which easily handles these queries, is required. With the largest query (Q2a on Twitter) aDFS performs a 9.3T `COUNT` in 2,661 seconds, resulting in 3.5B matches per second, while consuming less than 10GB per-machine memory for intermediate results.

4.3.6 aDFS vs. Graph Mining, Dataflow Joins

We compare aDFS to (i) three graph-mining systems[2], namely ***G-Miner*** [22], ***Fractal*** [169], and ***Peregrine*** [23], as well as a dataflow join system, ***BiGJoin*** [173]. We use workloads from the G-Miner paper [22]: TC, i.e., Triangle Counting, and counting instances of a more complex pattern referred to as the P-pattern, with the four graphs that are used to evaluate these operations in the paper. All systems are distributed apart from Peregrine. For BiGJoin, we only perform the evaluation on TC as it does not support filters, and tune the batch size for performance (10^8). For aDFS, we express both triangles and the P-pattern as graph queries.

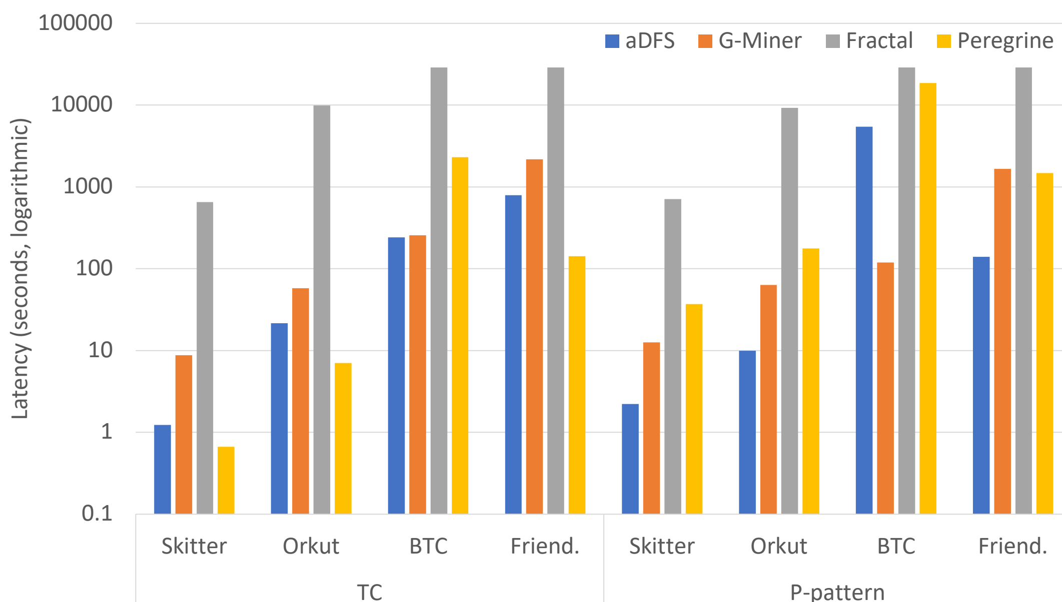


Figure 4.8: aDFS vs. graph mining engines.

Results. Figure 4.8 includes the performance of the four systems. Triangle counting (TC) highlights the difference between matching and not matching automorphisms: For the three graph mining systems, the search for "unique" triangles is baked in the pattern-matching algorithm, whereas in aDFS, we implement isomorphism with auto-

[2]We requested the artifact of Automine [171] for evaluation, but the authors were not able to provide us with it.

morphism elimination using dynamic filtering (i.e., `(a)->(b)->(c)->(a) WHERE ID(a) < ID(b) AND ID(b) < ID(c)`). This results in expensive filtering and heavier cross-machine communication than with the other systems. Still, aDFS is faster than G-Miner and Fractal for all graphs by up to 14× for G-Miner and by up to several orders of magnitude for Fractal. Peregrine outperforms all other graph mining systems including aDFS on three out of the four graphs, as it is able to intersect adjacency lists to quickly find common neighbors, an optimization that performs particularly well for triangles and which can be implemented in a straightforward manner on a single machine, where the whole graph is accessible. There is no clear winner between aDFS and BiGJoin on TC, with each system outperforming the other on two graphs. By intersecting local edges, BiGJoin's approach allows for reduced communication and better performance on the two graphs with the highest average degrees (Orkut and Friendster). The P-pattern does not require automorphism checks, as its vertices are differentiated by labels. We express it as:

```
SELECT COUNT(*)
FROM MATCH (c:c)->(b1:b)->(:a)->(c)->(b2:b)->(:d)
WHERE b1 <> b2
```

in PGQL. When matching the P-Pattern, aDFS significantly outperforms all other systems for all but one datapoint (G-Miner on BTC); it is on average 12 and 366× faster than Peregrine and Fractal, respectively, and 8× faster than G-Miner on three graphs. G-Miner achieves the best performance on BTC mainly because it replicates the target vertex label with each edge, which increases locality and reduces communication traffic. Such an optimization is not practical in a real-world system in which vertices can have many labels and properties of various types: Replicating these for each edge can have unacceptable memory overhead.

Overall, although aDFS is designed for different workloads, i.e., expressive graph queries, it is still very competitive with state-of-the-art graph mining systems and a dataflow join system on triangle counting and/or a mining-oriented workload.

4.3.7 aDFS Scalability

We use the LDBC workloads to illustrate the scalability of aDFS as we vary the number of machines. For measurements, we use SF300 with two, four, and eight machines.

Results. Figure 4.9 shows the latency of different machines with LDBC SF300. Overall, aDFS exhibits very good scalability: The average speedup is 1.9× from two to four machines, 2.0× from four to eight machines, and 4.2× from two to eight machines. The median speedup is 1.8× from two to four machines, 1.8×, from four to eight machines, and 3.5× from two to eight machines.

Overall, aDFS scales very well. In practice, systems hardly achieve perfect scalability and our engine is pretty close to that on most of the queries. The potential problems include various distributed coordination and query compilation overheads, as well as additional fixed costs. Other performance is lost because of graph sharding. When pattern matching on multiple machines, we pay the communication overhead. The LDBC queries are designed in such a way that there is no possibility of avoiding communication overhead even with ideal partitioning when we utilize all the threads and machines. aDFS is designed to scale: More machines translate to more compute resources, more buffers for intermediate results, and often more BFS exploration and higher network utilization, as the percentage of remote edges increases with the number of machines.

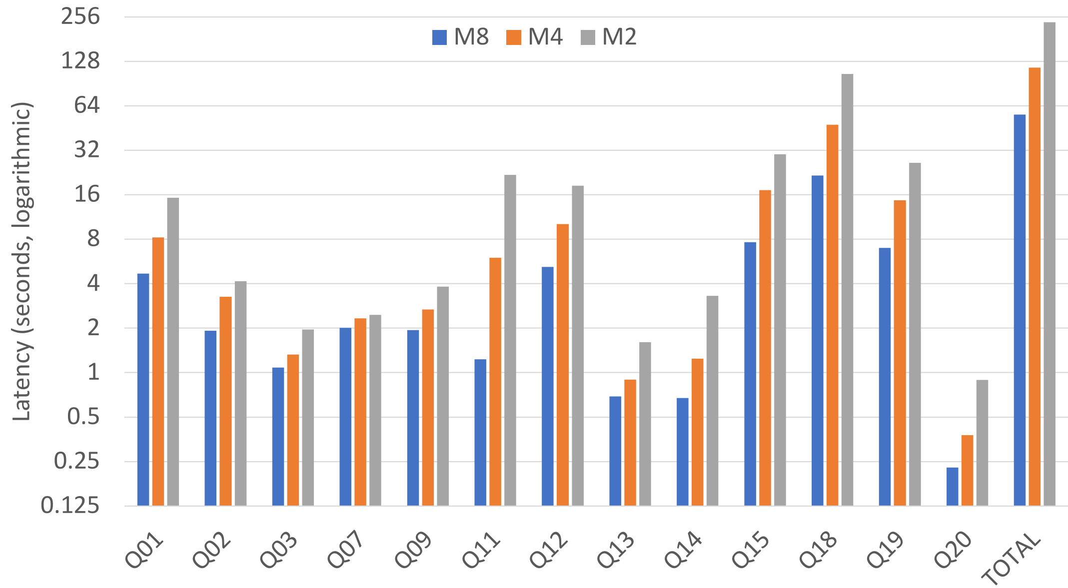


Figure 4.9: aDFS scalability on LDBC.

4.3.8 aDFS Parallel Execution

As the last experiment, we run the LDBC benchmarking suite using parallel execution. We again use SF300 to compare the sequential latency of queries when run consequently one after the other with parallel execution of the same queries with eight machines.

Results. Figure 4.10 shows the results of the sequential and the concurrent run of LDBC queries. We see that the parallel execution is faster by 2% in total.

The parallel run of the queries can be faster due to better scheduling, which hides the potential messaging latency and better balances the whole workload. As described in Section 3.3, aDFS performs either a local work or a work from a message. It can happen that a thread has no work, simply because the work is unbalanced, or because it is waiting for flow control. In these cases, the sequential execution is just waiting for a message to bring a new work for the threads. During parallel execution, the threads waiting for a message can start working on other parallel queries that have messages or local work to process.

The mentioned unbalancing can occur naturally when the data is not available on a machine. aDFS helps with the generation of thread work within a single machine if there is any available. However, in distributed graphs, the data for processing can reside on a different machine, e.g., because of a filtering.

4.4 Concluding Remarks

In this chapter, we have introduced aDFS: An almost-DFS approach to execute pattern-matching queries on distributed graphs. aDFS is able to execute virtually any query on any in-memory graph using at most a fixed and configurable amount of memory. aDFS is also very fast and scalable.

We compared aDFS to eight state-of-the-art systems with diverse characteristics — graph or relational/join-based, distributed or single machine, in-memory or disk-based — and showed that aDFS is up to orders of magnitude faster than them. We also show

that aDFS scales very well and allows an efficient parallel execution.

Limitation and future work. In the future, we plan to make the DFS/BFS switching more aggresive and dynamic. DFS has a small memory footprint, which also implies that it is slower compared to BFS in some queries, as shown in Figure 4.5. We want to allow a computation to use whole memory for BFS traversals and continue DFS only in case the memory is close to being full. This behavior can be especially useful in multi-user dynamic cloud environments.

Another direction is to use our approach on the scale of distributed graphs. aDFS helps in queries with narrow start by generating work for other threads. However, if the narrow start resides on a single machine only, other machines have no work and wait for a work from messages. Sharing the work between machines should better balance computations and lead to better performance.

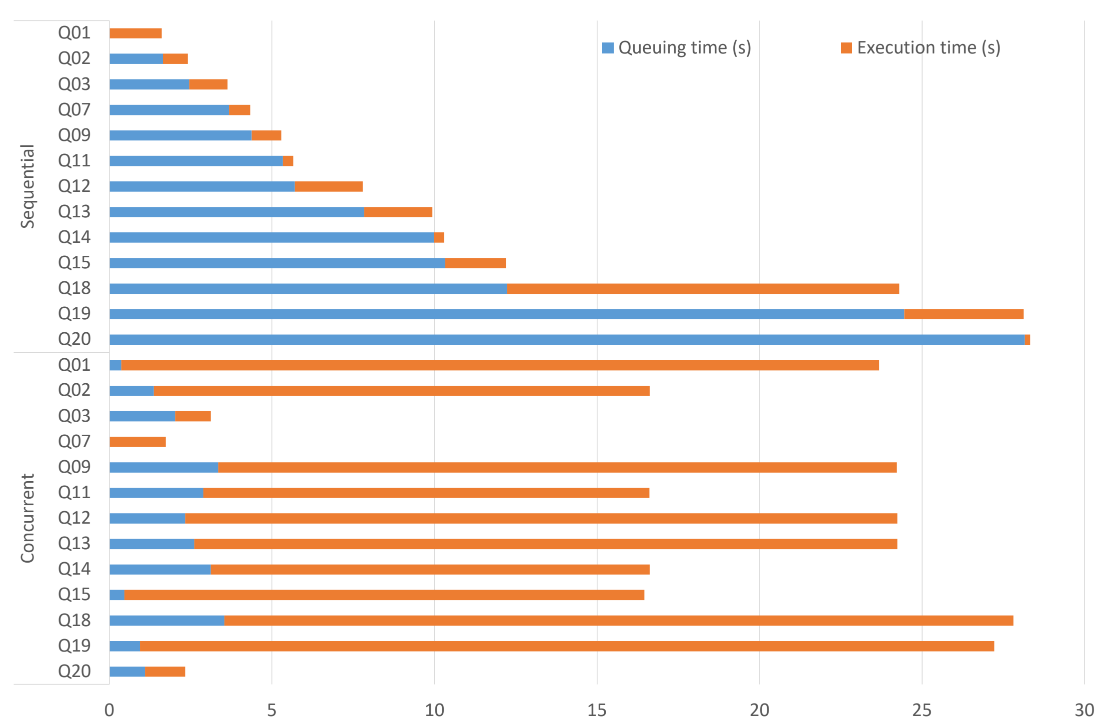


Figure 4.10: aDFS parallel execution on LDBC.

5. Distributed Asynchronous Reachability Regular Path Queries

One of the most expressive and powerful constructs in the graph querying is regular path queries, also called RPQs. RPQs enable support for variable-length path patterns based on regular expressions.

In this chapter, we introduce a novel design for distributed RPQs that is built on top of distributed asynchronous pipelined traversals to enable (i) memory control of path explorations, with (ii) great performance and scalability. Through our evaluation, we show that with sixteen machines, it outperforms Neo4j by 91× on average and a relational implementation of the same queries in PostgreSQL by 230×, while maintaining low memory consumption.

Organization. The chapter is organized as follows: Section 5.1 introduces the reachability and regular queries together with the related work. Section 5.2 presents the design and implementation of RPQs. Section 5.3 presents and discusses the RPQ evaluation. Finally, Section 5.4 concludes the chapter and highlights possible future work.

5.1 Introduction

Graph queries can express both *fixed-size patterns* and *variable-length patterns*, including regular path queries (RPQs). Variable-length patterns represent the true power of graphs, as they are far better suited in terms of expressiveness for graph query languages and in terms of performance on top of graph indices in graph engines. For instance, a user can very intuitively express the following RPQ query in PGQL:

```
PATH p AS (:Person)-[:Knows]-(:Person)
SELECT COUNT(*)
FROM MATCH (p1:Person) -/:p+/-> (p2:Person)
```

that counts the pairs of persons connected through a non-empty chain of `Knows` edges. However, expressiveness brings complexity: Variable-length patterns are a very challenging workload due to the potential explosion of intermediate explorations, especially in large graphs.

In this chapter, we present RPQd, a novel algorithm for implementing RPQs on distributed graphs. RPQd deviates from the traditional breadth-first traversal (BFT) shortest-path algorithm and instead takes advantage of asynchronous distributed traversals of PGX.D/Async (presented in Section 3). In summary, RPQd executes recursive depth-first traversal (DFT) explorations within a machine and buffers remote matches together to reduce messaging overhead. This enables RPQd to implement flow control to control the memory expansion of RPQ explorations. Unlike fixed patterns, RPQs involve explorations of variable depth, requiring RPQd to implement dynamic flow control combined with an incremental termination-detection protocol.

Additionally, RPQd implements min and max quantifiers and builds a reachability index on-the-fly to detect cycles (to avoid infinite loops) and to eliminate duplicate paths. Furthermore, the unique design enables support for generic cross-filters between RPQ and non-RPQ patterns, such as:

```
SELECT COUNT(*)
FROM MATCH ANY (p1:Person) ((pa:Person)-[:Knows]-(pb:Person))* (p2)
WHERE p1.age <= pa.age AND pb.age <= p2.age
```

that counts a number of persons who form a chain of people knowing each other and having their age in ascending order. To the best of our knowledge, RPQd is the first distributed RPQ querying system to support such a powerful feature.

We implement and test RPQd on top of PGX.D/Async. Our evaluation demonstrates that with four machines, RPQd is on average more than 29× and 96× faster than Neo4j[1] [16] and PostgreSQL [119], respectively. Focusing on three original LDBC queries, RPQd is on average 17× and 227× faster than Neo4j and PostgreSQL, respectively. Moreover, we analyze two queries in depth and show that RPQd is able to execute with low memory footprint due to its DFT matching style. Our last experiment shows how RPQd behaves on queries of varying depths and identifies the primary bottlenecks in our algorithm.

5.1.1 Reachability Queries

Reachability queries are a fundamental graph workload and are used in a large number of applications, e.g., in financial fraud detections or bioinformatics [208]. In biological networks, where a graph can be used to represent molecules, reactions, and the way they interact, reachability queries are used to find how a given molecule influences directly or indirectly the expression of genes.

Reachability tests if there exists a path between two vertices in a graph. PGQL supports variable-length paths for reachability queries in the form of *regular path queries* (RPQs). A path pattern declaration starts with the `PATH` keyword. Any pattern containing at least one vertex is a valid path pattern. Quantifiers allow for specifying an upper and/or lower bound on the number of repetitions of the path pattern that makes up the matched variable-length pattern.

Typical approaches for answering reachability queries perform the computation on a directed acyclic graph (DAG) G' built from the original graph G using its strongly-connected components [209]. Two vertices x and y are in the same strongly connected component iff x is reachable from y and vice versa. In particular, a vertex on G' corresponds to a connected component in G, while an edge between two vertices of G' means that exists an edge between two vertices belonging to the corresponding two separate strongly connected components in G. This operation can be performed in $O(n+m)$ [210].

Various approaches exist [211, 212, 213, 214, 215], finding different trade-offs between building an offline index (i.e., transitive closure, which requires $O(n^2)$ space) to answer queries faster in $O(1)$ [216, 217] runtime, and performing online search using depth-first or breadth-first search, with a runtime of $O(n+m)$ without an index. Beamer et al. [218] proposes a modern variant of BFS called DBFS, which takes $O(n+m)$ time and has lower memory requirements than the full transitive closure of the graph. Another favorite approach is to use labeling schemas [219, 220, 221, 222] running reachability queries on top of pre-labeled graph. For example, methods using two-hop indexing work by determining for each vertex a set of intermediate vertices it can reach, and a set of intermediate vertices it is reachable from, and use them to answer queries at runtime. The original method introduced in [221] labels vertices in $O(n^4)$, and executes queries in $O(m^{1/2})$. Recent work [223] gives upper bounds on the computational complexity and the number of sent messages for reachability, bounded reachability, and regular reachability queries on a distributed property graph. We also

[1] Using Neo4j Community Edition (benchmarks not audited by Neo4j).

refer to Xu and Cheng et al. [209] for more information.

5.1.2 Regular Path Queries.

Regular path queries express a path between two vertices as a regular expression on the labels of the traversed edges. They were first proposed by Cruz et al. [33] and have been extensively studied since in different variants [55, 52, 224, 225, 226], with distributed graphs being a popular use case due to possible semantic web applications. Label-constrained RPQs [227, 228, 229, 230] are limited to constraints on edge labels. General path queries extend this concept to graphs with infinitely many labels [231] which is equivalent to having arbitrary filters on properties in property graphs. Neo4j's Cypher [6] allows variable-length paths by specifying an edge with a range of repetitions. PostgreSQL [119] supports the non-standard `RECURSIVE WITH` statement for recursive queries, returning a table constructed by `UNION` of the output rows of queries running recursively on preceding query's output.

PGQL allows reachability queries with a subset of regular path semantics, in particular, it allows repetition of path patterns. When executing a query, the runtime matches all destinations reachable from a given source by following paths that respect the pattern. The pattern consists of (repeatedly) matched vertices and edges through homomorphic matching.This differs from the traditional concept of RPQs that focuses on isomorphic matching of vertices and edges to patterns, which automatically results in acyclicity and duplication elimination (homorphic and isomorphic matching is described in Section 2.2). The regular language `(a)*bb(a)+` over the label alphabet `{a,b}` can be translated into PGQL using two variable-length patterns in the same query. The only restriction of PGQL is that the "OR" operation is limited to a single vertex or label match.

5.2 Distributed Reachability RPQs

The main design goal of RPQd is to implement a fast, fully in-memory solution for distributed variable-length queries of any size on top of asynchronous DFT distributed queries, e.g., PGX.D/Async, to inherit the controllable memory consumption characteristics of asynchronous traversals.

Figure 5.1 illustrates the design of RPQd. The query in the figure matches a 2-hop pattern where the source vertex has a non-zero length path leading to the second vertex restricted with filters. Section 5.2.1 describes the translation of queries to execution plans, while Section 5.2.2 provides a description of the execution itself. We then describe important RPQd runtime aspects: (i) limiting the execution memory in Section 5.2.5, (ii) RPQ matching termination in Section 5.2.4, and (iii) duplication removal and cycle avoidance in Section 5.2.6.

5.2.1 Query Planning

Every PGQL query undergoes multiple transformations, resulting in a distributed execution plan. These transformations are depicted in Figure 5.1. RPQd extends the planning mechanism of PGX.D/Async described in detail in Section 3.2.

PGQL Query ⇒ Logical Plan. This step converts the submitted query into a logical plan using operators from Table 5.1. Operators in red highlight the operators specific to RPQs.

Multiple equivalent plans with different performance characteristics can exist, e.g., an alternative plan for our example query could swap the last two operators. The

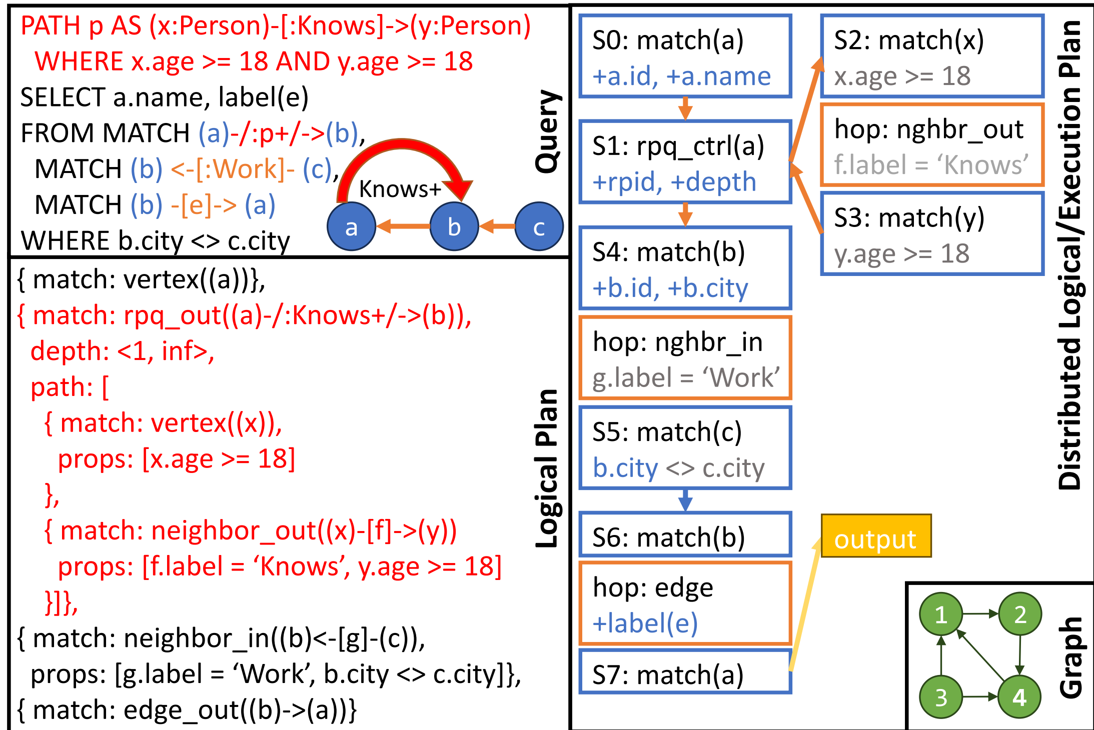


Figure 5.1: Query planning, final execution plan and example graph (in green). RPQ-related parts depicted in red. Transition hops in orange, inspection hops in blue, and output in yellow.

engine employs a standard cost-based query planner that selects the best plan using the following heuristics:

- Prefer single-match vertices, e.g., `ID(v) = 123`, as starting points.
- Prioritize vertices with heavy filtering in the early stages.
- Prefer edge matches over neighbor matches, since the cost of using the edge operator is logarithmic compared to a standard neighbor match.
- Prefer RPQ matches over neighbor matches because RPQ matching is slower and needs to run earlier due to potential match explosion.

Logical Plan ⇒ Distributed Query Plan. This step adds important distributed-specific information into the plan. All operators are transformed into a finite automaton consisting of *stages* (states) and *hops* (transitions between them). The behavior of stages and hops is described in Table 5.1.

The *inspection hop* is a special distributed operator designed to support non-linear patterns. In the Figure 5.1 example, Stage 5 uses an inspection hop to send the computation back to the machine of the matched vertex `b`. To support RPQs, a special *RPQ control stage* is added to the plan. This stage incorporates RPQ-specific logic (e.g., flow control, termination protocol) and data structures (e.g., reachability index). *Transition hops* are used to connect the RPQ path pattern with the RPQ control stage, as well as the RPQ control stage with normal stages, enabling support for 0-hop matching.

Distributed Query Plan ⇒ Execution Plan. The engine proceeds to materialize the execution plan based on the distributed query plan. This involves allocating data structures, (filter) expressions, and execution contexts. The execution context refers to preallocated intermediate result storage for each stage on a per-thread basis. It has a fixed layout and, for non-RPQs, a fixed length. RPQ context is preallocated up to a predetermined depth and dynamically allocated if further needed. In Figure 5.1, expressions are depicted in gray, while the data inserted into the context are shown in blue.

Stage/Hop	**Operator**	**Example**	**Description**
Stage	Vertex match	`(x)->(y)`	Matches vertices without following edges
RPQ control stage	RPQ match	`(x)-/:path*/->(y)`	Matches vertices and controls RPQ matching
Neighbor hop	Neighbor match	`(x)->(y)`	Matches neighbors of the current vertex
Edge hop	Edge match	`(x)->...` `(y)->(x)`	Matches again an already matched vertex
Inspection hop	—	`(x)->(y)->(z),` `(y)->(w)`	Transfers the execution back to an already matched vertex
Transition hop	—	—	Transfers the execution between stages
Output hop	(last hop)	—	Stores the intermediate results

Table 5.1: Query planning operators.

5.2.2 RPQd Execution Runtime

The engine uses an execution plan automaton to perform the graph depth-first traversals similar to PGX.D/Async (Section 3.3).Each worker thread on each machine is assigned a distinct local set of vertices and sequentially executes the initial stage of the automaton on those vertices (known as the bootstrapping process). If a vertex satisfies all its filters, the computation can transition (*hop*) to the following stage by matching a graph edge. If the edge is local, i.e., the destination vertex is stored locally, the next stage is applied recursively to the destination vertex. In the case of a remote edge, the engine serializes the working context into a message and continues the computation with another edge. If there are no more edges to process from a matched vertex, RPQd *backtracks* one stage and continues with the subsequent neighbor. Eventually, the engine returns back to the initial stage after processing the entire graph sub-tree.

From the perspective of the runtime, RPQd does DFS matching as described in Section 3.3. However, the internal logic of the RPQ-related stages and hops is extended. We will explain it using the example in Figure 5.1: For simplicity, we focus on the execution of a single worker on a single machine, assuming that all properties satisfy the query filters. We refer to `Stage i` as `Si`, and to graph `vertex i` as ⓘ. In parenthesis, we include additional comments to the given action.

The worker starts matching by applying ① on `S0`. It collects the properties `a.name` (projection) and `a.id=v1` (edge hop), *transitions* to `S1`, creates `rpid` (see

Section 5.2.6) and stores `depth=0` into the context.

The `depth` controls the number of RPQ iterations. It is incremented on each matching iteration of the control stage. If `depth<min_hop`, the RPQ path matching continues. If `min_hop<=depth AND depth<=max_hop`, (i) it atomically checks and updates the *reachability index* for duplications, then (ii) it *transitions* to non-RPQ stage moving the DFT matching closer to the output, and also (iii) *transitions* to RPQ path stages for large depths. If
`depth>max_hop`, it declines the match and backtracks.

In our example, `S1` checks the depth limit (`0<min_hop=1`) and transitions to `S2` (① match). Afterwards, the worker takes the first output neighbor edge to ②, evaluates edge filters, and hops on `S3` (② match) and transitions to `S1`. `S1` increments `depth=1`, and successfully checks the limit (`depth>=1`). It checks the reachability index, creates a new entry for path `{①,②}`, and transitions to `S4` (② match). It tries to continue using incoming neighbors (none), so it backtracks back to `S1` and transitions again to `S2` (② match), hops to `S3` (④ match), and returns back to `S1` which increments `depth=2`. After that the worker checks and creates an entry for `{①,④}`.

It transitions to `S4` (④ match) and stores `b.id=v4` (inspection hop) and `b.city` (filters), then it hops to: (i) `S5` (③ match: evaluate filters using `b.city`), (ii) `S6` (④ match: inspection hop back using `b.id=v4`), (iii) `S7` (① match: edge match using `a.id=v1`), and finally (iv) output (storing the projections from context into the output). After that, the worker backtracks and continues with further matching. Note that it cannot loop to ② because the reachability index contains the `{①,②}` entry.

5.2.3 Messaging

Messaging in RPQd follows the design of PGX.D/Async in Section 3.3. It involves batching multiple contexts into a single message, which is then sent asynchronously once it is full. When a message is received, a dedicated receiver thread places it into the message queue of the corresponding stage.

The received messages are picked up eagerly, prioritizing the latest stages and depths, and processed by the thread: (i) before bootstrapping new work from another vertex, (ii) after processing all the edges and backtracking, and (iii) when flow control prevents message sending.

The RPQ control stage handles incoming messaging for path stages using a priority queue rather than the per-stage queues of normal stages. Incoming messages are processed based on their depth (a message with a larger depth is processed first) and secondary based on their position (on the same depth, a message for later path stage is processed first). Basically, RPQd prioritizes the deepest computation following the principle of DFS traversal.

5.2.4 Termination Protocol

The engine executes the pattern matching until it visits all potential graph sub-trees. RPQd extends the lightweight distributed protocol of PGX.D/Async (presented in Section 3.3.1) to incrementally check termination conditions in stages `0` through `N-1`. The conditions for stage `i` are:

- All local work is completed.
- All remote work is received and completed.
- The previous stage `(i-1)` terminated globally.

To support RPQs, we extend the protocol with additional conditions that check all the RPQ stages incrementally in depths from `0` through `max_depth`:

- RPQ stage `i` terminates if it terminates on every depth `d` $\leq$ `max_depth`).
- RPQ stage `i` terminates on depth `d` iff:
  - All the path stages on depth $<$`d` have terminated.
  - All the path stages preceding stage `i` on the same depth `d` have terminated.

Unbounded RPQs. To address unbounded RPQs, RPQd proposes a consensus-like protocol to determine the *maximum observed depth*. Each machine independently tracks its maximum observed depth locally. When a machine M terminates stages S at depth D, it includes its maximum locally observed depth in the termination message broadcasted to other machines. By analyzing these termination messages, the engine deduces that there are no more remote messages from a specific machine, stage, and depth, indicating that the machine cannot extend the maximum observed depth further. Once all machines broadcast the same maximum observed depth `D` within the termination message for depth `D`, RPQd reaches a consensus on the maximum observed depth.

5.2.5 Flow Control

In RPQd, memory consumption for pattern matching is effectively managed during depth-first traversal. The engine uses a local DFT for fixed pattern matching, ensuring constant memory usage. Messaging between machines utilizes a fixed number of preallocated buffers, partitioned equally among stages and machines. Each machine requires at least two buffers (one for sending and the second for receiving).

Flow control is employed to regulate buffer allocation, limiting the number of buffers sent to a destination machine. Once a buffer is processed, a special `DONE` message is sent back to the source machine, indicating that it was released and made available again (see Section 3.3.2).

For RPQ stages, RPQd equally partitions buffers among stages, machines, and up to the preconfigured depth `D`. The remaining buffers are partitioned across the path stages and shared within a single path stage for all the depths $\geq D$. Additional overflow buffers are included to prevent livelocks that occur when a path stage is blocked at depth $d \geq D$, but shared buffers become available only after matching at a depth greater than `d`. The total number of consumed buffers for RPQs is $O(\#\text{machines} * (\text{P} * \text{D} + \text{P}) + \#\text{overflow_buffs})$, where $P = \#\text{path_stages}$ (a number of path stages). Unlike fixed-pattern matching, with RPQs, we cannot fully control memory consumption due to the need for overflow buffers. Still, as we detail in Section 5.3, the memory for a few per-depth overflow buffers is negligible.

5.2.6 Reachability Index

The semantic of (homomorphic) reachability queries requires that, given a source, each destination is accounted only once, e.g., following query `(a)->(b)-/:p+/->(c)` with a graph `{②->①<-③,①->④->⑨,①->⑤->⑨,①->⑥->⑨}` has only two matched results: `{{②,①,⑨},{③,①,⑨}}`.

Therefore, alternative paths must be either avoided or eliminated. Additionally, any path exploration must ensure that cycles are not infinitely followed. To solve these problems, RPQd dynamically builds a *reachability index*.

The reachability index is a distributed data structure that behaves as a map supporting atomic inserts and updates with (i) keys being reachability path IDs (`rpid`) and (ii) values being the path depths.

Path Encoding. RPQd employs a specific encoding for source path ID: `<machineId, workerId, seqId>`, where `<machineId, workerId>` (2×8-bits) is a unique worker identifier and `seqId` (48-bits) is a thread local sequence ID of the source matched path. RPQd uses the fact that every single path in DFT-based engine is processed by a single thread before entering the RPQ stage. As for the destination path ID, a simple vertex ID (64 bits) is used. `rpid` is constructed by combining source and destination path IDs (2×64-bits).

Implementation. RPQd implements the reachability index from scratch as a two-level map. The first level map is indexed by the destination vertex ID. Due to the continuous range of vertex IDs, it is implemented using an array of atomic pointers to the second-level map. The second-level map is a parallel map that stores the source path ID and the path's depth.

The reachability index is partitioned based on the vertex destination ID, i.e., the entry is stored on the path destination machine. This means that index entries cannot be used to speed up traversals by avoiding duplicated paths. This is left for a future work.

5.3 Evaluation

In this section, we evaluate the performance of RPQd and compare to state-of-the-art graph and relational databases. We also analyze the behavior of RPQd with artificial queries.

5.3.1 Experimental Settings

Methodology. Before starting the measurements, we run a single warm-up query. We run each query 10 times and report the median latency. Each experiment executes the queries in round-robin fashion in order to eliminate caching effects. We use 4 to 16 machines for running RPQd and a single machine for Neo4j and PostgreSQL.

Hardware. We use a cluster of 16 machines, each with two Intel Xeon CPU E5-2699 v3 2.30GHz CPUs with 18 cores (hyperthreads disabled/DVFS enabled), for 36 cores in total. Each machine contains 384GB of DDR4-2400 memory and LSI MegaRAID SAS-33108 storage. Each machine includes a Mellanox Connect-X InfiniBand card, all connected to an EDR 100Gbit/s InfiniBand network.

Implementation and Configuration. We implement RPQd on top of the PGX.D/Async engine (described in Section 3). We configure it to have 36 worker threads consisting of two dedicated to messaging. We use 8192 message buffers of 256KB per machine for flow control. This setting translates to approximately 2GB of intermediate results that can be produced by a single machine. This implies that the maximum worst-case memory consumption for messaging is $N * 2GB$, where N is number of machines. For RPQs, flow control works with preallocated buffers (out of the 8192 allowance) up to depth four. For larger depths, it allows five shared messages per path stage with one extra overflow message per each depth. Contexts are pre-allocated up to depth three.

Neo4j. Neo4j [16] is a single-machine disk-based graph database, but uses an in-memory cache for performance. We ignore the first slow run that brings data from disk. We run Neo4j Community Edition `4.0.10`, because it has the best performance with

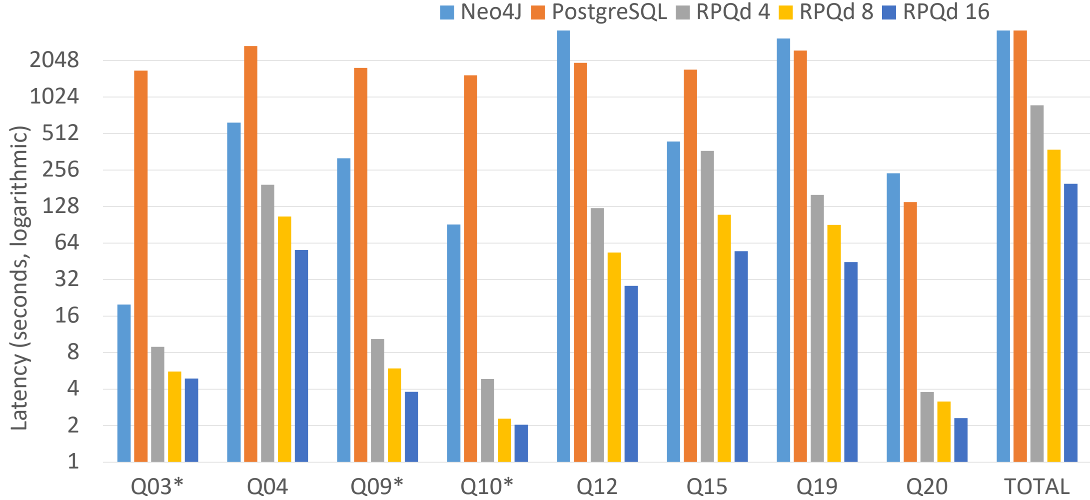


Figure 5.2: Different configurations of RPQd vs. other systems.

the same configuration compared to the newer versions (tested up to version `5.4`). To support all LDBC queries, we use the APOC [232] library, version `4.0.0.16`. Neo4j is configured according to the Neo4j-admin Memrec [206] utility: 31GB of heap initial and max size, 320GB pagecache and 36 worker threads.

PostgreSQL. PostgreSQL [119] is a single-machine open-source relational database. We run version `15.2`. The database is configured using the PGTune [207] utility: 99GB of shared buffers, 279GB effective cache size and 36 parallel workers.

Graphs and Queries. We use the latest LDBC graphs [194]: SF10 (27 million vertices, 170 million edges) and SF100 (255 million vertices, 1.7 billion edges). As a workload for comparison against the other engines, we take RPQs from the LDBC Business Intelligence (BI) queries [205] and adjust them to run on all engines – Neo4j supports reachability queries directly, PostgreSQL uses recursive queries to implement reachability. In total, we use nine queries: Three queries are the original BI (Q3, Q9, Q10) and the remaining are adaptations of the original ones. In the adapted queries, we remove expressions unsupported by RPQd, such as correlated subqueries, and use the part related to the reachability matching only. We show the RPQ queries in the appendix for reference.

5.3.2 RPQd vs. Other Systems

Figure 5.2 includes the results for RPQd, Neo4j, and PostgreSQL on LDBC SF100. Queries marked with "$\star$" are the unmodified BI queries. In terms of total time (i.e., sum of all queries), RPQd with four machines is more than $18\times$ and $16\times$ on average faster than Neo4j and PostgreSQL, respectively.

RPQd performs the best on queries that explore tree-subgraphs of LDBC (all except Q10), e.g., with `Reply` labels. In these queries, a breadth-first approach brings no advantages compared to RPQd, which is able to greatly contain memory usage.

RPQd delivers the lowest speedup and worst scalability on Q03*, due to the intermediate-result explosion on depth one, which leads to flow control blocking the execution more than 82 million times, about $5\times$ more than the number of matched vertices at that stage. Still, with the evaluated settings, RPQd delivers the purpose of consum-

ing approximately 2GB per machine for pattern matching (excluding the reachability index) and better performance than the other engines.

5.3.3 RPQd Scalability

Figure 5.2 shows that RPQd scales very well overall. Comparing to the default configuration with four machines, using eight and 16 machines is 2.3× and 4.4× faster, respectively, meaning that the speedup is almost linear. Super-linear speedups happen due to flow control: Every machine is configured with approximately 2GB of memory for pattern matching; thus, 8 or 16 machines have more memory for computations. The limited scalability to 16 machines (Q3, Q10) happens because of (i) partitioning and narrow starting queries (Q3 filters `country.name='Burma'`, thus starting from a single vertex), and (ii) less local computations that can lead to excessive flow control for some queries (to the point that it cannot be compensated by the higher flow control allowance). Note that the problem of narrow starting queries is solved by aDFS (Section 4). An integration of RPQd with aDFS is left for future work.

5.3.4 Q9 and Q10 in Detail

We analyze the detailed statistics of two queries, Q9 and Q10, exhibiting fundamentally different behaviors.

Q9 uses reachability in order to find recursively all replies to messages. The query starts from a large number of messages (see Table 5.2) and then traverses their comment trees, resulting first in an explosion of results (a message can have multiple answers), followed by an exponential decrease (few `Reply` chains are long). Due to nature of its filters and the graph, the reachability part of this query is always performed on a tree, making the reachability index superfluous. This query without reachability index on eight machines executes 3.4× faster than with the index (not shown in the thesis).

depth	0	1	2	3	4	5	6	7	8	9	10
#matches	3.1M	5.9M	4M	1.5M	375K	62K	7K	658	52	1	0

Table 5.2: Statistics of the RPQ control stage of Q9 in thousands (K) and millions (M).

Q10, starting from a predefined single person, finds all persons in a within two or three `Knows` hops. The reachability index is heavily used, allowing traversal back and forth along `Knows` edges. Table 5.3 shows the number of visited vertices with a number of eliminated (a vertex already reached at a lower or equal depth before) and duplicated (a vertex already reached at a greater depth) vertices in reachability index. The duplication can occur due to prioritizing depth-first work: It can match at higher depths while some shallower computations are still pending. This essentially represents the main limitation of a DFT-oriented engine compared to BFT. The large number of depth three eliminations is due to that most vertices visited at depth three have more than one neighbor matched already at depth two. The elimination count would be the same in a BFT engine. Since RPQd favors deeper work first, many final result materializations happen before the RPQ search at depth two is complete. This property allows the engine to have very low runtime memory usage.

None of these two queries triggers flow control: With eight machines the engine stayed below a total of 16GB memory. This is due the fact that RPQd uses DFT ap-

proach that has a small runtime memory footprint (as was shown in previous sections, e.g., Section 3.5). The number of elements saved in the reachability index is equal to the sum of the matched vertices in the RPQ control stage minus the eliminations and the duplications. Each entry in the reachability index occupies 12 bytes, resulting in a total dynamic size of 181MB for Q09 and 4.4MB for Q10 (compared to 100GB worth of data in LDBC SF100).

depth	num. matches	eliminated	duplicated
0	1	0	0
1	35	0	0
2	20K	4K	13K
3	2,700K	2,334K	0

Table 5.3: Statistics of the RPQ control stage of Q10 (K means thousands).

5.3.5 Effects of Reachability Index

Figure 5.3 shows the performance of simple artificial queries with and without the reachability index, highlighting the dynamic allocation of the index. We test a `Reply` pattern, which is the worst-case scenario for the index, while controlling the min and max depth of exploration. Hops $\{0, 0\}$ represent 0-hop where RPQd inserts entries $\{v, v\}$ for each graph vertex `v`. This shows the overhead of dynamically allocated index. All patterns with 0-min hop include this allocation overhead. Increasing inserts and updates into the index, by increasing the max-hop, has a negligible effect. Bulk (pre)allocation of the index can trade memory for performance which is left for future work.

Increasing the min-hop with reachability index leads to a counter-intuitive performance improvement. Larger min-hop values reduce the number of reachability entries created, because any traversal with depth below the min hop creates no entry. In future work, we plan to tune better the memory/performance trade-off of reachability indices.

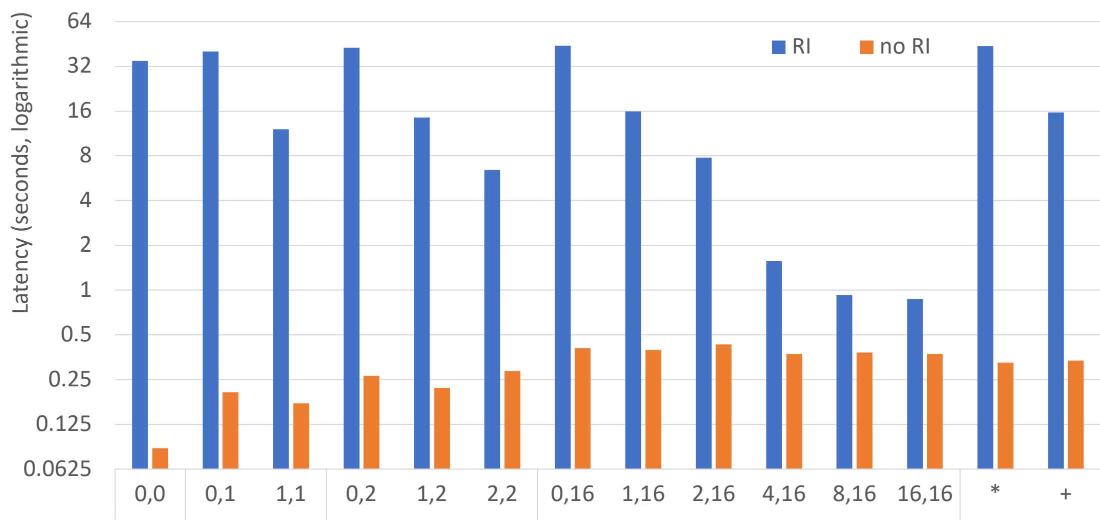


Figure 5.3: Latency of `Reply` RPQs with different depths with and without reachability index on LDBC SF10.

5.4 Concluding Remarks

In this chapter, we presented RPQd, a regular path query (RPQ) algorithm to answer reachability queries on top of distributed asynchronous depth-first graph traversals. RPQd supports both bounded and unbounded repetition of arbitrary path patterns, including regular expressions over any labels, as well as advanced features like local- and cross-filtering. It seamlessly integrates with asynchronous depth-first traversals extending its runtime with new RPQ structures. The runtime achieves strong guarantees on memory usage and outperforms state-of-the-art solutions.

Limitations and future work. RPQd builds upon the DFT engine, which implies aforementioned strengths, but also imposes limitations on the RPQ engine itself. Our approach excels in tree topology graphs and real-world workloads like social networks. However, as demonstrated in Section 5.3.5, when a graph-query combination generates numerous duplicated reachability paths, e.g., searching for long paths in complete graphs, the DFT algorithm reaches its limit. In such cases, more specialized algorithms like BFT might be a better fit if sacrificing low memory consumption for a faster evaluation is acceptable. Alternatively, when provided with a generated reachability graph, RPQd can run a fast RPQ pattern matching without compromising performance and memory consumption.

The first proposed improvement leverages the aDFS algorithm (Section 4) and its local BFS approach. This way, RPQd would leverage BFS traversals which eventually lead to fewer duplicated paths. Another possibility of improving RPQs with DFS is to use reachability index back-propagation. We could send the already-traversed paths back for reusing during traversals. This should minimize the number of duplicated paths and keep the benefits of DFS traversals, such as low memory consumption.

RPQs search for any reachable paths. However, PGQL and other query languages allow queries that search for shortest or cheapest paths [34]. One possible implementation could use depth-first RPQs together with *depth-first iterative deepening* [233]. Follow-up improvements can also support a stronger RPQ model [1]. PGQL does not support queries that allow path disjunctions, but other/future query languages might support this. It would require implementing a support for path branching, e.g., by using a state transition table.

6. Better Distributed Graph Query Planning With Scouting Queries

Query planning is essential for graph query execution performance. In distributed graph processing, data partitioning and messaging significantly influence performance. However, these aspects are difficult to model analytically, making query planning especially challenging. Therefore, this chapter introduces *scouting queries*, a lightweight mechanism to gather runtime information about different query plans, which can then be used to choose the "best" plan. In a pipelined depth-first-oriented graph processing engine, scouting queries typically execute for a brief amount of time with negligible overhead. Partial results can be reused to avoid redundant work. We evaluate scouting queries and show that they bring speedups of up to 8.7× for heavy queries, while adding low overhead for those queries that do not benefit.

Organization. The rest of this chapter is organized as follows: Section 6.1 introduces the topic and mentions related work. Section 6.2 presents the design and implementation of scouting queries. Section 6.3 evaluates scouting queries. The last Section 6.4 concludes this chapter and highlights possible future work.

6.1 Introduction

Graph queries are a highly challenging workload. The number of edges traversed by a query can easily cause a combinatorial explosion of the intermediate and final results. Therefore, efficient query planning is key to improving graph query performance. The query plan dictates the order of pattern matching operations, i.e., which vertex or edge is to be matched first, second, and so on. An initial suboptimal decision by the graph query planner can negatively impact the entire query execution. Queries over large distributed graphs, in particular, can suffer from significantly worse performance.

Below, we show a simple query example (which represents TPC-H Q16; see Section 6.3 for more details) that counts the number of suppliers per part brand, type, and size with some rather complex constraints. The query plan starting from the `PART` match results in almost 4× worse performance than starting from the `SUPPLIER` match:

```
SELECT
  p.BRAND, p.TYPE, p.SIZE,
  COUNT(DISTINCT s.SUPPKEY) AS SUPPLIER_CNT
FROM MATCH
  (p:PART)-[ps:PARTSUPP]->(s:SUPPLIER)
WHERE
  p.BRAND != 'Brand#45'
  AND NOT JAVA_REGEXP_LIKE(p.TYPE, '^MEDIUM POLISHED.*$')
  AND p.SIZE IN (49, 14, 23, 45, 19, 3, 36, 9)
  AND NOT JAVA_REGEXP_LIKE(s.CMNT, '.*Customer.*Complaints.*')
GROUP BY p.BRAND, p.TYPE, p.SIZE
ORDER BY SUPPLIER_CNT DESC, p.BRAND, p.TYPE, p.SIZE
```

However, computing a performant query plan is notoriously difficult. The problem is exacerbated in the case of distributed query engines due to the need to additionally account for data partitioning, messaging and communication costs. This is especially true with distributed graphs, where the distributed query engine must take many more

things into account compared to single-machine graph engines, such as partitioning of the data and messaging or communication costs. Properly modeling partitioning and the cost of networking for query planning is fairly complex (if not outright impossible).

The classic techniques for query planning look similar to the ones used in classic relational databases and primarily use data statistics to compute the potential cardinality of each of the matches. Computing the cardinality depends highly on the query; typically, the more complex the query (e.g., long patterns and extensive filtering), the harder it is for the query planner to produce a good estimate.

In this chapter, we introduce *scouting queries* as a pragmatic solution to improve query planning and enhance the overall performance of distributed graph pattern matching. Scouting queries are short exploratory executions of the actual query used to benchmark the performance of different query plans in order to find the best performing plan. Scouting queries follow these steps:

1. Take the top N plans with traditional query planning.
2. Execute these N plans with a short timeout (e.g., 50ms) and record statistics of their execution. If the system detects that the plan is close to completion, it simply allows this plan to run to completion.
3. Combine the per-plan cardinality metric of the traditional query planner with the scouting query statistics and choose the best candidate plan.
4. Execute the selected plan and, if there are opportunities to reuse the work of scouting queries, merge the outputs.

Scouting queries are better suited for large graphs and queries, as are typical for distributed graph engines, with any potential overheads amortized by the gains of the improved query plan. Additionally, scouting queries best fit engines with pipelined execution of pattern matching, i.e., engines that eagerly push intermediate results out as final. On top of such engines, the scouting query metrics include the actual final-result matching throughput, which gives a very good indication of the actual performance of the engine. However, as we detail in Section 6.2, scouting queries can be applied to any graph-processing engine.

We prototype scouting queries on top of the PGX.D/Async (presented in Chapter 3) which uses distributed depth-first-oriented matching, eagerly pushing intermediate matches out as final. We evaluate our prototype on several LDBC-inspired queries and 12 actual TPC-H queries expressed as graph queries. We find that the total workload execution time improves by 3.3× and 1.7× for LDBC and TPC-H, respectively, with a maximum speedup of 8.7× on a TPC-H query and only two queries where scouting selects a worse query plan than the default planner.

6.1.1 Query Planning

Query planning is a fundamental step of all declarative query languages. Since the user only describes the computation logic, it is the duty of the data management engine to come up with an execution sequence that returns the requested data. These engines have a limited set of basic operations that can be composed together and executed one after the other. The process of coming up with the right sequence of operations is called query planning, where a query plan is a specific scheduling of operations.

Table 6.1 describes the set of basic match operators that any (vertex-centric) dis-

tributed query engine needs to support for query planning in one way or another. PGX.D/Async engine uses a similar set of operators as shown in Section 3.2. The performance of any graph engine is highly influenced by the query-plan selection. Query plans dictate the pattern-matching order, i.e., which vertex or edge is matched first, second, and so on, regardless of the engine's execution model.

Operator	**Example**	**Description**
Vertex match	**`(x)`**`->(y)`	Matches vertices without following edges
Outgoing neighbor match	`(x)`**`->(y)`**	Matches outgoing neighbors of the current vertex
Incoming neighbor match	`(x)`**`<-(y)`**	Matches incoming neighbors of the current vertex
Edge match	`(x)->...` `(y)`**`->(x)`**	Matches again an already matched vertex
Inspection match	`(x)->(y)->(z),` **`(y)`**`->(w)`	Transfers the execution back to an already matched vertex

Table 6.1: Typical graph operators used in a distributed graph pattern matching engine.

Every query which involves more than a single vertex match has multiple query plans. For example, without inspection matches, a simple pattern of two directed hops – `MATCH (a)->(b)->(c)` – could be planned as `(a)→(b)→(c)` or as `(c)←(b)←(a)`. Adding inspection matches exponentially increases the number of possible plans introducing `(b)→(c)⇒(b)←(a)` and `(b)←(a)⇒(b)→(c)`.[1]

For this reason, traditional query-planner methods come up with different plans for the same query, and pick the best one according to a *cost metric*. The cost of a query plan QP ($cost(QP) = X$) represents the predicted computational "costs" of the query in arbitrary units of computations. The cost can be calculated based on a number of factors. One of the most important aspects of graph querying is the cardinality of the individual matches. The cardinality of a match is an estimation of the number of data points it needs to process.

Of course, different query planners can quantify different aspects of the query plan. Different operators can require different amount of work, and thus have different costs. Filters can be taken into consideration in order to reduce the selectivity of operations based either on fixed heuristics or static and runtime statistics. Choosing the optimal query plan based on cost-based analysis is a well-explored topic and is outside of the scope of this thesis. Our solution, scouting queries, builds directly on top of the traditional query planning approaches.

6.1.2 Related Work

With data sets and queries becoming increasingly complex, a traditional static cost-based query optimization as introduced by Selinger et al. [234] can become inefficient. The statistics and assumptions on which this dynamic programming approach relies can be inaccurate or sometimes even invalid. The limitations of the optimize-then-execute paradigm have led to a plethora of new approaches that rely on runtime feed-

[1] $\rightarrow$, $\leftarrow$ and $\Rightarrow$ are the query planner operators, i.e., outgoing and incoming edge match, and inspection match, respectively.

back to correct the query plan [235]. This class of techniques is called *adaptive query processing*.

Runtime statistics refinement [236, 237, 238, 239, 240, 241, 242, 243] is a technique where statistics collection is triggered during query execution, resulting in little to no overhead. The newly collected statistics can be used for current or future query execution. Proactive re-optimization is another technique where the query optimizer is invoked when estimation violations occur [244, 245, 246, 247, 248]. Further improvements such as leveraging intermediate results or strategically delaying re-optimization have also been proposed [245, 247, 249]. Multi-plan choices have the query optimizer concurrently run multiple query plans, sometimes on different data subsets [250, 251, 252]. This approach is the most closely related to our scouting queries, but our solution is applied on distributed graph query execution with emphasis on the engine's throughput within a specified time frame, rather than on a restricted subset(s) of the data. To the best of our knowledge, this particular emphasis has not been previously documented in the literature. Approaches such as *Smooth Scan* [253] avoid sensitivity to the quality of the statistics and estimations.

A hot topic in database research is embedding machine-learning models in the query optimizer to improve its efficiency. These vary from using supervised learning on previous execution plans to generating plans for future queries [254], to training recommender systems on textual similarities between SQL queries, with the assumption that textually similar queries should have similar query plans [255]. Using latency to reward a reinforcement learning model in the query optimizer [256] has also been proposed. Adjusting inaccurate statistics by learning from the query planner's past mistakes [243] complements those works. Somehow similar to scouting queries, Trummer et al. [257] propose learning about the best join order while training on slices of data until the best order is found. Other approaches consist of enhancing instead of substituting the query optimizer, like *BAO* [258], which learns the best execution plans for past queries and chooses from multiple query plans suggested by a traditional query optimizer. While these approaches suffer from limitations, they herald even greater improvement possibilities [254].

6.2 Scouting Queries

Scouting queries aim to augment traditional query planning, solving its shortcomings by gathering information about the actual final runtime of the pattern matching part of graph queries. They do so by executing various query plans of the same query for short duration and collecting statistics to help decide which plan will be the best once fully deployed.

6.2.1 Query Planning to Scouting Query Execution

In what follows, we detail the steps to integrate scouting queries in a (distributed) graph engine.

Preselect top N plans. We can use any traditional planner to preselect the top N query plans. Most algorithms assign an explicit confidence value indicating how likely a query plan is to be the optimal one. This confidence is internally used for ranking the N query plans at the beginning, but can also be used at a latter step to combine the power of traditional query planning with scouting queries.

Run scouting queries. We create a scouting query for each of the N query plans. Scouting queries are short executions of a query, which run for a limited time (in the

order of milliseconds) and collect runtime information about their execution. Note that graph queries typically include pattern matching followed by post processing, such as `GROUP BY` and `ORDER BY`. Scouting queries execute only the pattern matching part of the query. We use these executions to choose the best performing query plan – out of N – based on the collected data. In order to collect statistics along the whole matching pattern, it is important for scouting queries to execute on top of a depth-first-oriented engine (DFT), i.e., an engine that eagerly pipelines intermediate results out as final. Breadth-first-oriented (BFT) engines (e.g, Neo4j [16]) collect all intermediate results per each match and then proceed to subsequent matches.

On the one hand, scouting must run almost instantly compared to the actual execution of the query in order to not add high overhead. On the other hand, scouting should have enough time to explore the complete query pattern. One should ideally configure the time limit for each scouting query based on the expected performance of the graph engine. For example, if the engine is capable of delivering throughput in the million matches per second, running the scouting query for a few milliseconds is enough for almost any query. One must also account for the overhead of starting the query. Altogether, scouting is best-suited for large graphs and/or large queries, where the benefits can easily outweigh the overhead.

Scouting query statistics policies. The best scenario is for scouting to find some final (output) results of pattern matching. The output throughput can be the main indication of how good the query plan of the scouting query is. Our general assumption is that the throughput of a limited execution of a query plan is roughly the same as that of the entire execution (the experimental results in Section 6.3 validate this assumption). This means that if the query plan QP1 returns more results than query plan QP2 in the same amount of time, we expect query plan QP1 to be better than plan QP2. Our assumption could fail in theory, as the engine could be lucky while executing a worse query plan. Imagine a query `SELECT COUNT(*) FROM MATCH (a)->(b) WHERE ID(a) < 10` where the best query plan starts from matching `(a)` because of the filtering (assuming all IDs are > 0) and continues to the matching of `(b)`. In some extreme cases, the bad plan starting matching from `(b)` can have higher or equal throughput during scouting, e.g., it matches `(b)` and follows to `(a)` with $ID(a) = 1...5$.

In order to alleviate this problem, we can mix the scouting findings with the confidence values of the default query planner. The query planner returns the confidence for each of the N preselected query plans. The formula for picking the best query plan can be as follows: $argmax(confidence(QP) * (throughput(QP) + 1))$. Of course, depending on the engine and the traditional query planner, one can devise different policies for weighing the query-plan selection. (In our evaluation, Section 6.3, we use for instance only the scouting query performance as the selection metric.)

Of course, there are queries that could have a small number or no results, which means throughput 0 for all scouting queries. To solve that issue, the engine uses a secondary metric (which requires deeper integration to the execution engine). The engine records the number of visited vertices for each vertex match and combines those as a metric. Matches at latter parts of the query mean that results flow faster towards the output, i.e., the matching happens faster.

Additionally, depending on the engine and the optimization criterion, any other statistics/metrics can be deployed. The aforementioned metric optimizes for the engine performance. One can easily create a metric that aims for a different optimization criterion, e.g., when the engine is low on memory, it can prefer memory consumption

over performance. In that case, while executing a scouting query, it can monitor the memory consumption and pick the query plan with the lowest consumption. Another policy, important in cloud environments, is the one for minimizing the overall engine cost. To minimize the cost for query execution, the engine can monitor the usage of different modules (at their price) during scouting execution and pick the query plan with the potentially lowest price for the user.

Execution of scouting queries. Every scouting query executes starting from a set of vertices. If the graph vertex is fixed by the query, e.g., `SELECT COUNT(*) FROM MATCH (a)->(b)->(c) WHERE id(a) = 0`, we use the filter for bootstrapping the computation – for the plans that start with `(a)`. If there are multiple options for the starting vertex, we select the starting vertices randomly. By choosing the vertices randomly, the collected statistics are more representative compared to incremental vertex selection, e.g., in sorted order based on the vertex ID.

The actual execution of the scouting is up to the engine. As we mentioned above, in order for the queries to be short, the execution must be somehow limited. One way is to limit the execution time of the queries by the engine. To support this the engine needs an efficient support for execution cancellation. Using cancellation, the engine runs each of the scouting queries for the given amount of time and then the query is stopped.

Another approach at obtaining meaningful results in a short time is to limit the parts of the graph that are traversed. Instead of traversals navigating all edges, the engine randomly chooses edges followed for pattern matching at each step. In this case, the scouting queries are performing a random walk with the given query on the searched graph. This can be implemented by adding a random filter on every element of the scouting query that returns false with a certain probability, thus pruning further exploration of some paths. This approach is not as clean or effective as the time-capped method of the previous paragraph, but can be used to deploy scouting queries in BFT-based engines, which have no control of pushing output results out eagerly.

Furthermore, while monitoring the number of matches for each vertex match, the engine keeps track on whether the scouting query already traversed significant parts of the graph. In that case, it gives up on the execution of other scouting queries and lets the engine execute the current plan. To minimize the potential of running a worse query plan, we execute scouting queries in order of the default query-planner confidence. We predict when a query plan should continue by estimating the amount of remaining time after a scouting query. For example, if the time limit for running each scouting query is 10ms and we have 5 scouting queries, and if the first scouting query traverses 40% of the graph in that time, we then make the assumption that the query will continue with a similar pace, hence it could finish execution in another 10-20ms in comparison to running the remaining four scouting queries in 40ms.

6.2.2 Reusing Scouting Query Results

One potential overhead of scouting is when throwing away perfectly correctly-computed query matches. We can alleviate this issue by introducing scouting query result reusing. By definition, all query plans return the same output results, but they can differ in the order that intermediate results are expanded to generate final. DFT-oriented engines return output matches eagerly compared to BFT. DFT engines explore systematically all the matching subtrees with the same prefix and once they move to another prefix there are no more results with that same prefix.

We use this observation for building query-plan groups. If two query plans have the same matching prefix, they belong to the same group. For instance, if we set the prefix length to one (i.e., group query plans which start with the same vertex match) for the query example `MATCH (a)->(b)->(c)` we have following grouping of query plans, starting from different vertices:

- *start from (a)*: **(a)**→(b)→(c), **(a)**→(b)⇒(c)←(b)
- *start from (b)*: **(b)**←(a)⇒(b)→(c), **(b)**←(a)⇒(c)←(b), **(b)**→(c)⇒(b)←(a), **(b)**→(c)⇒(a)→(b)
- *start from (c)*: **(c)**←(b)←(a), **(c)**←(b)⇒(a)→(b)

The query plans within the same group can directly share scouting results and the finally selected plan will reuse those results in its execution. The results can be shared if a query traverses a whole subtree for the given matched prefix. For our example with matched prefix `(a=1)->(b=2)` it means traversing and trying to match starting from root vertex `(2)`. Thanks to DFT, we know that there are no further matches after traversing the whole subtree and we do not need to visit that part of the graph with the same prefix again.

For reusing the results, we first split the query plans into groups according to their prefix. The length of the prefix can be set statically or dynamically after analyzing the top N scouting query plans. Because the engine needs to reuse results of scouting from the same group, it keeps track of which matched prefixes were fully traversed and the result for those prefixes. After running the next scouting query from the same group, the engine should avoid those specific traversed prefixes in order not to duplicate the same work. After finishing all scouting queries from the group, we have a set of traversed prefixes and their results. After selecting the winning query plan, we can easily continue the computation from the non-visited matched prefixes and the final output is the union of results from the executed query and all the partial results collected during scouting. Notice that even when each group contains a single query plan, we can reuse the results collected during scouting execution.

The above-mentioned approach is the preferred approach for reusing the results. Another possibility for avoiding duplicated work is to mark all the matched and visited paths in the graph and store results for each of them. One can notice that this approach is memory consuming. Nevertheless, compared to the prefix approach, this can be used together with the random-walk scouting queries or potentially other approaches.

It is worth noting that result sharing with prefixes further avoids re-traversing the non-matching paths of those prefixes. In our previous example, the `(a=1)->(b=2)` prefix could lead to 10 matched `(c)` vertices and 20M non matched, e.g., because of a filter `WHERE c.value = 43`. Having completed this prefix with scouting covers both the matching and non-matching `c`s.

6.3 Evaluation

In this section, we evaluate the potential benefits, as well as the overhead of scouting queries in distributed graph queries.

6.3.1 Experimental Settings

Implementation and Configuration. We implement a prototype of scouting queries on top of the PGX.D/Async graph query and pattern matching engine (Section 3), which is the perfect target, as it is a distributed engine with eager, DFT-style completion of pattern matching. We configure the solution to scout the top $N+1$ query plans, where N is the number of neighbor matches per query. For instance, pattern `(a)->(b)` has $N = 1$, hence we run two scouting queries (for `(a)->(b)` and `(b)<-(a)`). This way, we allow more scouting queries for longer patterns, which are expected to also result in longer queries, while maintaining the overhead relatively contained. Furthermore, PGX.D/Async uses intermediate-result buffering for remote edges, which can result in intermediate results flowing slower towards output. To reduce the effect of buffering in the short scouting execution (which could bias the scouting metrics), we reduce the size of buffers by $\frac{1}{16}$ for scouting queries. Alternatively, one could incorporate the number of buffered intermediate results in the scouting performance metric, weighing intermediate results in the later parts of the query plan more than those in earlier parts.

Additionally, we configure scouting queries with 50ms timeout and use only the scouting query throughput as the query plan selection criterion, such that we evaluate strictly the efficiency of scouting queries. Finally, we do not implement the result sharing solutions described in Section 6.2.2, thus the scouting query executions are strict overhead on top of the selected query plan execution.

Hardware. We use a cluster of eight machines, each with two Intel Xeon CPU E5-2699 v3 2.30GHz CPUs with 18 cores (hyperthreads disabled/DVFS enabled), for 36 cores in total. Each processor contains 384GB of DDR4-2400 memory and LSI MegaRAID SAS-3 3108 storage. Every machine includes a Mellanox Connect-X InfiniBand card, all connected to an EDR 100Gbit/s InfiniBand network.

Methodology. Our approach mainly affects the pattern-matching part of graph queries and is oblivious to the post-processing operators, such as GROUP BY and ORDER BY. Accordingly, in our experiments, we report the pattern matching execution time for both non-scouting and scouting and additionally the scouting execution overhead from all N queries for scouting.

We perform 10 runs of each query and report the median latency. Scouting is not deterministic: With some queries, scouting could dictate different query plans in different runs. We thus report the plans that are selected, and how often they did so, under the result bars. For each experiment set, we execute the queries in a per-graph round-robin fashion in order to reduce caching effects.

Graphs and queries. Our experiments use two classic graphs and the corresponding queries. First, we use the latest LDBC graph [180] SF300 and a set of 12 queries derived from the LDBC Business Intelligence (BI) standard queries [194]. This is not the official LDBC standard. The scope of this evaluation covers user-provided fixed-pattern queries, thus the latest LDBC BI queries are outside of the scope of this work. We use the older/first version of BI: Out of the 24 original queries, four represent simple path patterns (i.e., Q2, Q4, Q12, Q23) and are directly used in our experiments. The remaining ones include expressions not supported by PGX.D/Async, such as RPQs or subqueries. We devise simplified variants of these to support the benchmark specification as close as possible. (An example of such transformation is presented in Section 4.3).

Second, we express the classic TPC-H [182] relational database workload as a graph (both the data and the queries, scale factor 300, with 2.36 billion vertices and 6.14 billion edges) and optimize with scouting queries. For instance, the workload includes region vertices, which are connected with countries, in which customer vertices reside. With PGQL, Q3 is expressed as:

```
SELECT
  o.O_ORDERKEY, o.O_ORDERDATE, o.O_SHIPPRIORITY,
  SUM(l.L_EXTENDEDPRICE *(1 - l.L_DISCOUNT)) AS REVENUE,
FROM
  MATCH (l:LINEITEM) -[:LINEITEM_ORDERS]-> (o:ORDERS)
  MATCH (o) -[:ORDER_CUSTOMER]-> (c:CUSTOMER)
WHERE
  c.C_MKTSEGMENT = 'BUILDING'
  AND o.O_ORDERDATE < DATE '1995-03-15'
  AND l.L_SHIPDATE > DATE '1995-03-15'
GROUP BY o.O_ORDERKEY, o.O_ORDERDATE, o.O_SHIPPRIORITY
ORDER BY REVENUE DESC, o.O_ORDERDATE
LIMIT 10
```

We rewrite and use the 12 TPC-H standard queries that require no subquery support.

6.3.2 Results

Figures 6.1 and 6.2 include the results of PGX.D/Async with and without scouting queries on LDBC and TPC-H workloads, respectively.

Overall. Scouting improves the total workload execution time, including scouting overhead, by 3.3× for LDBC and 1.7× for TPC-H.

For both LDBC and TPC-H, we see that scouting queries result in the same query

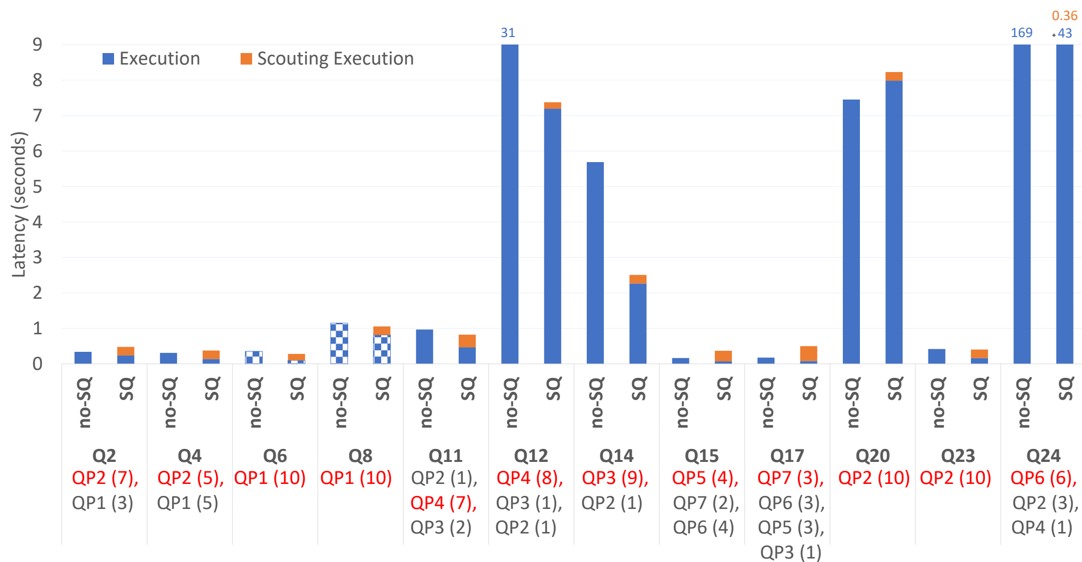


Figure 6.1: Query execution latencies for LDBC without (no-SQ) and with scouting queries (SQ). Mosaic-pattern bars represent queries where the default query planner and scouting queries give the same query plan. The query plans (QPi) and how many times each was chosen are listed beneath each query, sorted from best to worst, with the one preferred by scouting highlighted in red.

plan as the default query planner for only 4 out of 22 queries. Interestingly, even for these four queries the overall performance is slightly faster with scouting queries, even though we have not enabled result reuse. Our analysis shows that the scouting query warmup gives a good performance boost to the actual execution (especially for Q4 on LDBC that is tiny). The exact overhead from scouting queries is shown in orange and depends on (i) the query execution duration and (ii) the number of neighbor matches the query includes. As mentioned earlier, we run $N + 1$ scouting queries, where N is the number of neighbor matches in that query.

A second class of queries are the ones that marginally benefit from scouting queries (i.e., Q23 on top of LDBC and Q10 on TPC-H). For those, scouting queries cause the engine to use a different query plan, however, this plan is not so much faster than the original plan. For instance, query Q10 on TPC-H benefits by 40% (280ms faster) with the new query plan, but the scouting query overheads ($N = 4$) cancel this speedup.

A third category includes queries that have overall worse performance with scouting. This happens either because the queries are very short (Q2, Q4, Q15, Q17 on LDBC) and the scouting overheads are higher than any possible benefits, or because scouting queries choose a worse plan than the default query planer. The latter only happens to Q20 from LDBC and Q19 from TPC-H. In Q20/LDBC, scouting consistently in all 10 runs returns query plan 2, while in Q19/TPC-H, scouting takes the wrong decision 6 out of 10 runs, resulting in 1.6× slower execution time, 1.8× with the scouting overhead. In both cases, the scouting timeout is too short to select the best query plan. For Q19/TPC-H, the collected statistics are almost identical in both query plans, hence it is a matter of "luck", which plan is selected. When choosing the optimal timeout, one must strike a balance between collecting more precise statistics and the overhead of executing scouting queries.

The five LDBC queries that are altogether slower with scouting are relatively short (average execution time of 1.7s) and have an 1.8× average slowdown. With TPC-H,

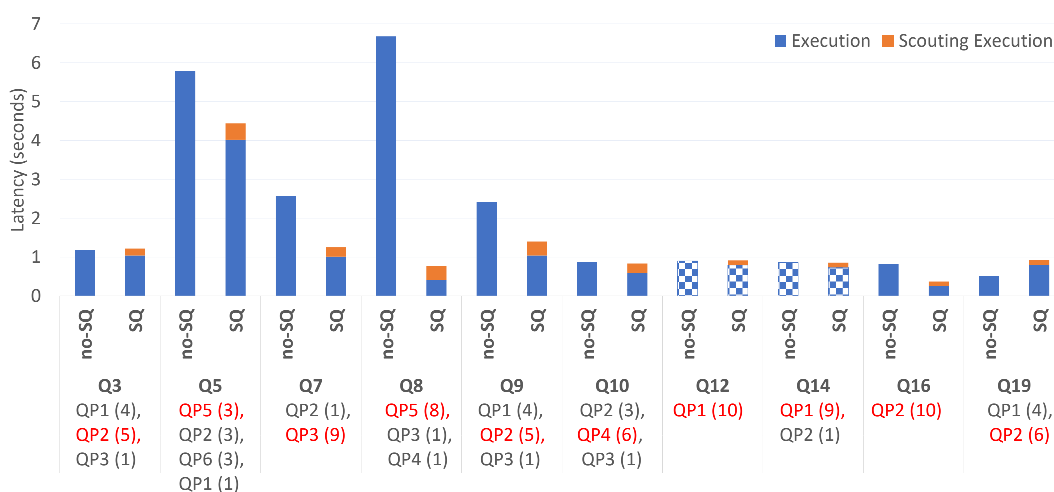


Figure 6.2: Query execution latencies for TPC-H without (no-SQ) and with scouting queries (SQ). Mosaic-pattern bars represent queries where the default query planner and scouting queries give the same query plan. The query plans (QPi) and how many times each was chosen are listed beneath each query, sorted from best to worst, with the one preferred by scouting highlighted in red.

only three queries are slower (average execution time 0.9s) with 1.3× average slowdown.

Finally, the remaining queries (Q12, Q14, and Q24 on LDBC and Q5, Q7, Q8, Q9, and Q16 on TPC-H) represent the queries where scouting helps the most. In these cases, the default query planner cannot find the best plan, while the actual scouting-query execution successfully unveils a better plan. Intuitively, these tend to be longer queries with complex connections and filters. The result is speedups from 1.3 to 8.7× maximum, with 3.4× and 3.2× average speedups for LDBC and TPC-H, respectively.

In summary, the experimental results meet our intuition: Scouting is better suited for large complex queries where (i) traditional query planning has a hard time finding the best plan and (ii) the reduction of the long execution times outweighs any scouting overheads.

Deep dive TPC-H Q7. We now analyze Q7 from TPC-H in order to explain the complexities of query planning in distributed graph query engines that bring about the need for dynamic planning approaches such as scouting queries. We choose Q7 as it combines the complexities of long patterns with heavy filtering. Q7 calculates the total money transfers between France and Germany over two years:

```
SELECT
  N1.NAME AS SUPP_NATION, N2.NAME AS CUST_NATION,
  EXTRACT(YEAR FROM li.SHIPDATE) AS L_YEAR,
  SUM(li.EXTENDEDPRICE * (1 - li.DISCOUNT)) AS REVENUE
FROM MATCH
  (li:LINEITEM)-[:LINEITEM_SUPPLIER]->(s:SUPPLIER),
  (li)-[:LINEITEM_ORDERS]->(o:ORDERS),
  (o)-[:ORDER_CUSTOMER]->(c:CUSTOMER),
  (s)-[:SUPPLIER_NATION]->(N1:NATION),
  (c)-[:CUSTOMER_NATION]->(N2:NATION)
WHERE
  N1.NAME IN ('GERMANY', 'FRANCE')
  AND N2.NAME IN ('GERMANY', 'FRANCE')
  AND N1 != N2
  AND li.SHIPDATE >= DATE '1995-01-01'
  AND li.SHIPDATE <= DATE '1996-12-31'
 GROUP BY SUPP_NATION, CUST_NATION, L_YEAR
 ORDER BY SUPP_NATION, CUST_NATION, L_YEAR
```

The query plan proposed by the default planner (QP1) starts from the supplier nation, goes to the supplier's line-items, to the orders, to the customers, and then to the second nation. Scouting queries instead choose the third query plan (QP3) that starts from the customer nation, moves to the customer, the orders, the line-items, the suppliers, and the suppliers' nation. The default query planner gives 3× lower cardinality to QP1 than QP3, thus it is certain about the choice of this particular plan.

However, the default query planner misestimates how selective the line-item filter is, bringing line-item matching as early in the plan as possible. As we see in Table 6.2, this choice leads to an early explosion of visited vertices in Stage 2 `(li)`, compared to the smaller and later explosion in Stage 3 `(li)` for QP3. Looking at the query, this explosion makes a big difference, leading to a greater number of intermediate matches that need to extract data out of the line-item vertices for projections and to support the group-by operation, i.e., `EXTRACT(YEAR FROM l_shipdate)`, and `SUM(l_extendedprice * (1 - l_discount))`.
This data needs to be carried across machines in a distributed engine, leading to over-

heads. In practice, expressing all these fine details in a default query planner is almost impossible.

Stage	QP1	Visited	Matched	QP3	Visited	Matched
0	`N1`	225	18	`N2`	225	18
1	`s`	5,488	5,488	`c`	78,683	78,683
2	`li`	3,179,315	966,944	`o`	746,157	746,157
3	`o`	663,104	663,104	`li`	2,594,222	787,010
4	`c`	370,814	370,814	`s`	533,522	533,522
5	`N2`	370,814	14,877	`N1`	533,522	21,551

Table 6.2: Scouting query execution statistics of TPC-H Q7. *Visited* and *Matched* correspond to vertices.

6.4 Concluding Remarks

This chapter introduces scouting queries, a mechanism built on top of traditional query planners to enable selecting the best query plan for pattern matching in (distributed) graph queries. An efficient query plan is a critical part of any query engine significantly affecting its performance. Our approach uses runtime statistics collected during the execution of candidate query plans and is thus able to correct any misestimations of the default query planner. Our evaluation shows that scouting queries can significantly improve performance on real graphs, while maintaining low overhead.

In future work, we intend to test BFT-style scouting matching, scouting-result reusing and different scouting metrics. Furthermore, we plan to use scouting queries in a feedback loop to improve query planners, both manually and with machine-learning techniques.

7. Conclusion

In this thesis, we presented *aDFS*: A distributed asynchronous in-memory graph querying algorithm capable of executing a fixed pattern-matching query of any size with a strictly bound configurable amount of memory. To the best of our knowledge, aDFS is the first truly distributed graph-querying system working on fully-partitioned graphs that is capable of doing this. aDFS cleverly combines the two main approaches for graph querying: breadth-first search and depth-first search, leveraging their advantages and mitigating their disadvantages. The depth-first traversal allows capping the computation memory to the bare minimum needed for the execution. The asynchronous approach adds the possibility to "hide" the performance penalty caused by sending messages across the machines. The breadth-first approach further improves the already great performance and scalability of the system by distributing the parallel work more equally among workers and using better memory access locality. We compared aDFS to eight state-of-the-art systems with diverse characteristics — graph or relational/join-based, distributed or single machine, in-memory or disk-based — and showed that aDFS is up to orders of magnitude faster than them.

The second contribution is RPQ*d*, an extension of DFS engine to support variable-length pattern matching queries in the form of regular path queries with support for reachability. RPQd supports both bounded and unbounded repetitions of arbitrary path patterns in the form of regular expressions over any edge labels, as well as advanced powerful features like local- and cross-filtering. Due to our unique design, to the best of our knowledge, this is the first distributed querying system that supports cross-filtering between RPQs and normal queries (shown in Section 5.1). The implementation ensures strong guarantees of memory usage during computation. Given the support of unbounded queries, it cannot limit the consumed memory to the size of query, but it can minimize potential memory overflows to a bare minimum to allow predictable memory consumption, which is a highly demanded feature, e.g., in distributed elastic cloud environments. We evaluated RPQd against Neo4j and PostgreSQL and showed that with sixteen machines, it outperforms Neo4j by $91\times$ on average and a relational implementation of the same queries using recursive queries in PostgreSQL by $230\times$, while maintaining low memory consumption.

The last contribution of this thesis is the improvement of distributed graph query planning, which is an essential part of any query system and has a large influence on the overall performance. Current analytical approaches are not sufficient to cover the full complexity of a distributed system (e.g., graph partitioning, messaging, elasticity, or dynamic (cloud) environments), which might lead to selecting a suboptimal query plan and ultimately result in poor performance. We propose a simple yet powerful solution that uses fast depth-first scouting queries that collect runtime statistics during the exploration. These statistics are then used to make educated decisions about the performance of potential query plans. We show that the potential overhead of scouting queries can be mitigated by reusing the found results and by "prewarming" the system, leading to faster query execution. We evaluated scouting queries and showed that they bring speedups of up to $8.7\times$ for heavy queries.

Future research directions. Each chapter of our contribution presents a section related to future work at the end. In addition to that, we present here future research directions with regard to whole field of graph querying.

Most graph queries consist of multiple parts, e.g., pattern matching and post-processing with aggregations, that are typically processed by different subsystems. We presented two pattern matching approaches, aDFS and RPQd, which are both able to limit/cap the memory consumption. However, if the other subsystems (e.g., post-processing) do not limit the memory as well, the system as a whole might go out-of-memory. Therefore, being able to cap the memory through the whole pipeline execution might be crucial for running the system with multiple users in cloud.

In this thesis, we focus on graph querying of static graphs. However, in most cases, the amount of data is constantly growing, and graph analytics should take this into account. There is a new model of temporal graphs [80, 81, 82, 83] that treats graph modifications as first-class citizens. Adding native support for graph modifications to (distributed) graphs and allowing efficient graph processing would require non-trivial changes in various parts of these systems.

Graph query languages frequently add new features to accommodate growing number of user use-cases. We see a focus on specialized features combining standard querying (well-known from relational databases) with graph-related algorithms (e.g., a shortest-path search). From our experience, an efficient implementation of such features in distributed query engines is a highly challenging research area with a practical impact on the users of big-data graph analytics. An example of a complex feature can be full support for efficient distributed subqueries. There are also two completely new query languages – SQL/PGQ and GQL – whose implementation on top of distributed graphs will undoubtedly uncover many research problems.

Perspective. Distributed graph querying has proven to be a valuable analytics approach that helps to analyze graph-specific data, e.g., social networks, but also helps to improve the performance of other related fields, e.g., worst-case optimal joins [173] for relation querying. Our clever combination of BFS and DFS, bringing a memory efficient querying with great performance, inspired other graph systems [259, 260, 261] to use a similar approach. We also described an algorithm that extends this idea for RPQs, which is a powerful constuct in graph querying. Others can build on this idea and use it to allow support for RPQs. The last contribution introduces an approach that helps with finding the most efficient query plan in practice, even in complex and dynamic environments. This approach can be leveraged by any (not only graph) querying systems.

All the improvements presented in this thesis help to develop an efficient distributed graph system and are applicable to any distributed DFS graph-querying system.

Bibliography

[1] Angela Bonifati, George Fletcher, Hannes Voigt, Nikolay Yakovets, and HV Jagadish. *Querying graphs*, volume 10. Springer, 2018.

[2] Mark Newman. *Networks*. Oxford university press, 2018.

[3] PGQL 2.0 Specification – Property Graph Query Language. `https://pgql-lang.org/spec/2.0/`.

[4] Gremlin – A graph traversal language. `https://github.com/tinkerpop/gremlin/wiki`.

[5] GSQL. `https://docs.tigergraph.com/gsql-ref/current/intro/`.

[6] Neo4j Cypher Query Language – Developer guides. `https://neo4j.com/developer/cypher/`.

[7] Property Graph Queries (SQL/PGQ). `https://www.iso.org/standard/79473.html`, June 2023.

[8] GQL Standard – Graph Query Language. `https://www.gqlstandards.org`.

[9] Guare J. Six Degrees of Separation: A Play. 1990.

[10] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the national academy of sciences*, 99(suppl_1):2566–2572, 2002.

[11] World Wide Web Consortium et al. RDF 1.1 concepts and abstract syntax. 2014.

[12] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17:5–20, 03 2007.

[13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.

[14] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost Depth-First-Search distributed Graph-Querying system. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 209–224. USENIX Association, July 2021.

[15] Jianqing Fan, Fang Han, and Han Liu. Challenges of big data analysis. *National science review*, 1(2):293–314, 2014.

[16] Neo4j – Graph database platform. `https://neo4j.com`.

[17] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. TigerGraph: A Native MPP Graph Database. *arXiv:1901.08248 [cs]*, 2019.

[18] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 981–992, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Thomas Neumann and Michael J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[20] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 425–440, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Saif Ur Rehman, Asmat Ullah Khan, and Simon Fong. Graph mining: A survey of graph mining techniques. In *Seventh International Conference on Digital Information Management (ICDIM 2012)*, pages 88–92. IEEE, 2012.

[22] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[23] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[25] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA, 2014. USENIX Association.

[26] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *Proceedings of the 21st international conference on World Wide Web*, pages 949–958, 2012.

[27] Sarra Bouhenni, Said Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys (CSUR)*, 54(2):1–35, 2021.

[28] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. Amazon Neptune: Graph Data Management in the Cloud. In *ISWC (P&D/Industry/BlueSky)*, 2018.

[29] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent advances in graph partitioning*. Springer, 2016.

[31] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *GRADES*, 2016.

[32] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017.

[33] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. A graphical query language supporting recursion. *ACM SIGMOD Record*, 16(3):323–330, 1987.

[34] Pgql variable-length paths. `https://pgql-lang.org/spec/2.0/#variable-length-paths`.

[35] Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Sungpack Hong, and Hassan Chafi. Better Distributed Graph Query Planning With Scouting Queries. In *Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–9, 2023.

[36] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1), feb 2008.

[37] Robert A Hanneman and Mark Riddle. Introduction to social network methods, 2005.

[38] Ulrik Brandes, Linton C Freeman, and Dorothea Wagner. Social networks, 2013.

[39] Barry Wellman, Janet Salaff, Dimitrina Dimitrova, Laura Garton, Milena Gulia, and Caroline Haythornthwaite. Computer networks as social networks: Collaborative work, telework, and virtual community. *Annual review of sociology*, 22(1):213–238, 1996.

[40] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the World-Wide Web: A survey. *ACM Sigmod Record*, 27(3):59–74, 1998.

[41] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Dandapani Sivakumar, Andrew Tompkins, and Eli Upfal. The Web as a graph. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–10, 2000.

[42] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. The web as a graph: Measurements, models, and methods. In *Computing and Combinatorics: 5th Annual International Conference, COCOON'99 Tokyo, Japan, July 26–28, 1999 Proceedings 5*, pages 1–17. Springer, 1999.

[43] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer networks*, 33(1-6):309–320, 2000.

[44] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design issues and challenges for RDF-and schema-based peer-to-peer systems. *ACM SIGMOD Record*, 32(3):41–46, 2003.

[45] B Pourebrahimi, K Bertels, and S Vassiliadis. A survey of peer-to-peer networks. In *Proceedings of the 16th annual workshop on Circuits, Systems and Signal Processing*, pages 570–577, 2005.

[46] Mursel Tasgin and Haluk O Bingol. Community detection using preference networks. *Physica A: Statistical Mechanics and Its Applications*, 495:126–136, 2018.

[47] Eric G Freedman and Priti Shah. Toward a model of knowledge-based graph comprehension. In *International conference on theory and application of diagrams*, pages 18–30. Springer, 2002.

[48] Michael Schuhmacher and Simone Paolo Ponzetto. Knowledge-based graph document modeling. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 543–552, 2014.

[49] Jon Kleinberg. The convergence of social and technological networks. *Communications of the ACM*, 51(11):66–72, 2008.

[50] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. BioGRID: a general repository for interaction datasets. *Nucleic acids research*, 34(suppl_1):D535–D539, 2006.

[51] UniProtKB. `http://www.uniprot.org`.

[52] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[53] Renzo Angles. The Property Graph Database Model. In *Alberto Mendelzon Workshop on Foundations of Data Management*, 2018.

[54] Peter T Wood. Query languages for graph databases. *ACM Sigmod Record*, 41(1):50–60, 2012.

[55] Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 175–188, 2013.

[56] Danai Koutra and Christos Faloutsos. *Individual and collective graph mining: principles, algorithms, and applications*. Springer Nature, 2022.

[57] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)*, 38(1):2–es, 2006.

[58] Lei Tang and Huan Liu. Graph mining applications to social network analysis. *Managing and mining graph data*, pages 487–513, 2010.

[59] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. CoRR abs/1910.09017 (2019). *arXiv preprint arXiv:1910.09017*, 1910.

[60] Jim Webber Ian Robinson and Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013.

[61] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. Management and analysis of big graph data: current systems and open challenges. *Handbook of big data technologies*, pages 457–505, 2017.

[62] Rohit Kumar Kaliyar. Graph databases: A survey. In *International Conference on Computing, Communication & Automation*, pages 785–790. IEEE, 2015.

[63] Vijay Kumar and Anjan Babu. Domain suitable graph database selection: A preliminary report. In *3rd International Conference on Advances in Engineering Sciences & Applied Mathematics, London, UK*, pages 26–29, 2015.

[64] NS Patil, P Kiran, NP Kiran, and Naresh Patel KM. A survey on graph database management techniques for huge unstructured data. *International Journal of Electrical and Computer Engineering*, 8(2):1140, 2018.

[65] Jaroslav Pokornỳ. Graph databases: their power and limitations. In *Computer Information Systems and Industrial Management: 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings 14*, pages 58–69. Springer, 2015.

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

[67] Safiollah Heidari, Yogesh Simmhan, Rodrigo N Calheiros, and Rajkumar Buyya. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Computing Surveys (CSUR)*, 51(3):1–53, 2018.

[68] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):305–324, 2017.

[69] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[70] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal*, 29:595–618, 2020.

[71] Avi Silberschatz, Henry F Korth, and S Sudarshan. Data models. *ACM Computing Surveys (CSUR)*, 28(1):105–108, 1996.

[72] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[73] Andreas Meier, Michael Kaufmann, Andreas Meier, and Michael Kaufmann. Nosql databases. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*, pages 201–218, 2019.

[74] Jan Paredaens, Jan Van den Bussche, Marc Andries, Marc Gemis, Marc Gyssens, Inge Thyssens, Dirk Van Gucht, Vijay Sarathy, and Lawrence Saxton. An overview of GOOD. *ACM SIGMOD Record*, 21(1):25–31, 1992.

[75] Mark Levene and Alexandra Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data & Knowledge Engineering*, 6(3):205–224, 1991.

[76] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on NoSQL stores. *ACM Computing Surveys (CSUR)*, 51(2):1–43, 2018.

[77] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[78] M Tamer Özsu. A survey of RDF data management systems. *Frontiers of Computer Science*, 10:418–432, 2016.

[79] Michael M. Wolf, Alicia M. Klinvex, and Daniel M. Dunlavy. Advantages to modeling relational data using hypergraphs versus graphs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2016.

[80] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[81] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[82] Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[83] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. Distributed temporal graph analytics with GRADOOP. *The VLDB journal*, 31(2):375–401, 2022.

[84] Swapnil Gandhi and Yogesh Simmhan. An interval-centric model for distributed computing over temporal graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1129–1140. IEEE, 2020.

[85] Vera Zaychik Moffitt and Julia Stoyanovich. Temporal graph algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages*, pages 1–12, 2017.

[86] Benjamin Steer, Felix Cuadrado, and Richard Clegg. Raphtory: Streaming analysis of distributed temporal graphs. *Future Generation Computer Systems*, 102:453–464, 2020.

[87] Christian S Jensen and Richard T Snodgrass. Temporal data management. *IEEE Transactions on knowledge and data engineering*, 11(1):36–44, 1999.

[88] Tom Johnston. *Bitemporal data: theory and practice*. Newnes, 2014.

[89] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.

[90] Jonathan Hayes et al. A graph model for RDF. *Darmstadt University of Technology/University of Chile*, 2004.

[91] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.

[92] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300, 2014.

[93] Olaf Hartig and Bryan Thompson. Foundations of an alternative approach to reification in RDF. *arXiv preprint arXiv:1406.3399*, 2014.

[94] Olaf Hartig. Foundations of RDF* and SPARQL*:(An alternative approach to statement-level metadata in RDF). In *AMW 2017 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017.*, volume 1912. Juan Reutter, Divesh Srivastava, 2017.

[95] Olaf Hartig. Foundations to Query Labeled Property Graphs using SPARQL. In *SEM4TRA-AMAR@ SEMANTiCS*, 2019.

[96] Kangfei Zhao and Jeffrey Xu Yu. All-in-one: graph processing in RDBMSs revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1165–1180, 2017.

[97] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, page 377–392, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Sparsity Technologies. Sparksee. https://www.sparsity-technologies.com/#sparksee.

[99] Norbert Martínez-Bazan, M Ángel Águila-Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, pages 110–119, 2012.

[100] Mark A Roth and Scott J Van Horn. Database compression. *ACM sigmod record*, 22(3):31–39, 1993.

[101] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

[102] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[103] Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[104] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422, 2007.

[105] Rong Chen and Haibo Chen. Wukong: A distributed framework for fast and concurrent graph querying. *ACM SIGOPS Operating Systems Review*, 55(1):77–83, 2021.

[106] WhiteDB Team. WhiteDB. http://whitedb.org/.

[107] Scott M Meyer, Jutta Degener, John Giannandrea, and Barak Michener. Optimizing schema-last tuple-store queries in graphd. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1047–1056, 2010.

[108] Franz Inc. AllegroGraph. https://allegrograph.com/.

[109] Blazegraph DB. Blazegraph. https://www.blazegraph.com/.

[110] Olaf Hartig. Reconciliation of RDF* and Property Graphs, 2014.

[111] Christopher D Rickett, Utz-Uwe Haus, James Maltby, and Kristyn J Maschhoff. Loading and querying a trillion rdf triples with cray graph engine on the cray xc. *Cray User Group*, 2018.

[112] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[113] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[114] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.

[115] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. Relational memory: Native in-memory accesses on rows and columns. *arXiv preprint arXiv:2109.14349*, 2021.

[116] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[117] MariaDB Foundation MariaDB plc. MariaDB. `https://mariadb.org`.

[118] MonetDB – An open-source database system. `https://www.monetdb.org`.

[119] PostgreSQL – The world's most advanced open source database. `https://www.postgresql.org`.

[120] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *SIGMOD Rec.*, 43(1):27–31, may 2014.

[121] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[122] AgensGraph. `https://bitnine.net/agensgraph/`.

[123] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[124] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *Acm Sigmod Record*, 42(4):5–16, 2014.

[125] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(12):1891–1904, 2020.

[126] Redis Labs. RedisGraph. `https://redis.io/docs/stack/graph/`.

[127] Borislav Iordanov. Hypergraphdb: a generalized graph database. In *Web-Age Information Management: WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010 Revised Selected Papers 11*, pages 25–36. Springer, 2010.

[128] The Linux Foundation. JanusGraph. http://janusgraph.org/.

[129] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[130] The Apache Software Foundation. Apache HBase. https://hbase.apache.org.

[131] Oracle. Oracle Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html.

[132] TuGraph. https://www.tugraph.org.

[133] Yue Zhao, Kenji Yoshigoe, Mengjun Xie, Suijian Zhou, Remzi Seker, and Jiang Bian. Lightgraph: Lighten communication in distributed graph-parallel processing. In *2014 IEEE International Congress on Big Data*, pages 717–724. IEEE, 2014.

[134] ArangoDB Inc. ArangoDB. https://www.arangodb.com/.

[135] Callidus Software Inc. OrientDB. https://orientdb.com/.

[136] Fauna. FaunaDB. https://fauna.com/.

[137] Microsoft. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db/.

[138] Tim Bray. The javascript object notation (json) data interchange format. Technical report, 2014.

[139] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[140] Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. In *Deductive and object-oriented databases*, pages 223–240. Elsevier, 1990.

[141] VelocityDB Inc. VelocityDB. https://velocitydb.com/.

[142] Rick F van der Lans. InfiniteGraph: Extending Business, Social and Government Intelligence with Graph Analytics. Technical report, Technical report, 20, 2010.

[143] VelocityDB Inc. VelocityDB. https://velocitydb.com/QuickStartVelocityGraph.

[144] Diogo Fernandes, Jorge Bernardino, et al. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. *Data*, 10:0006910203730380, 2018.

[145] Andrew E Wade. Full distribution in Objectivity/DB. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 69–69, 1992.

[146] InfinityGraph DO Language. https://support.objectivity.com/sites/default/files/docs/ig/latest/index.html#page/topics%2FwelcomeInfiniteGraphPlatform.html%23.

[147] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[148] Neo4j - Graph Algorithms. https://neo4j.com/docs/graph-data-science/current/algorithms/.

[149] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, 1999.

[150] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 349–362, New York, NY, USA, 2012. Association for Computing Machinery.

[151] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[152] Farzin Houshmand, Mohsen Lesani, and Keval Vora. Grafs: Declarative Graph Analytics. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[153] 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 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[154] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[155] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[156] 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 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 4, USA, 2012. USENIX Association.

[157] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[158] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 410–424, New York, NY, USA, 2015. Association for Computing Machinery.

[159] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 285–300, USA, 2016. USENIX Association.

[160] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 301–316, USA, 2016. USENIX Association.

[161] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[162] Keval Vora. {LUMOS}:{Dependency-Driven} Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, 2019.

[163] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[164] 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 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 375–386, USA, 2015. USENIX Association.

[165] Samuel Grossman and Christos Kozyrakis. A new frontier for pull-based graph processing. *arXiv preprint arXiv:1903.07754*, 2019.

[166] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In *Advances in Knowledge Discovery and Data Mining: 12th Pacific-Asia Conference, PAKDD 2008 Osaka, Japan, May 20-23, 2008 Proceedings 12*, pages 858–863. Springer, 2008.

[167] Shoji Hirano and Shusaku Tsumoto. Identifying Exacerbating Cases in Chronic Diseases Based on the Cluster Analysis of Trajectory Data on Laboratory Examinations. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, pages 151–156. IEEE, 2007.

[168] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.

[169] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, 2019.

[170] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. {RStream}: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 763–782, 2018.

[171] Daniel Mawhirter and Bo Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523, 2019.

[172] Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: Neighborhood-Centric Large-Scale Graph Analytics in the Cloud. *The VLDB Journal*, 25(2):125–150, apr 2016.

[173] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proc. VLDB Endow.*, 11(6):691–704, feb 2018.

[174] Oracle. Oracle Spatial and Graph. `https://www.oracle.com/database/technologies/spatialandgraph.html`.

[175] SPARQL Query Language for RDF – SPARQL Protocol and RDF Query Language. `https://www.w3.org/TR/rdf-sparql-query/`.

[176] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.*, 9(13):1317–1328, sep 2016.

[177] Shipeng Qi, Heng Lin, Zhihui Guo, Gábor Szárnyas, Bing Tong, Yan Zhou, Bin Yang, Jiansong Zhang, Zheng Wang, Youren Shen, et al. The ldbc financial benchmark. *arXiv preprint arXiv:2306.15975*, 2023.

[178] LDBC Semantic Publishing Benchmark (SPB). `http://ldbcouncil.org/benchmarks/spb/`.

[179] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. The LDBC Social Network Benchmark. *CoRR*, abs/2001.02299.

[180] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.*, 16(4):877–890, 2022.

[181] David Püroja, Jack Waudby, Peter Boncz, and Gábor Szárnyas. The LDBC Social Network Benchmark Interactive workload v2: A transactional graph query benchmark with deep delete operations. In *TPCTC*. Accepted.

[182] TPC-H Decision Support Benchmark. `https://www.tpc.org/tpch/`.

[183] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, pages 97–104. Springer, 2004.

[184] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078*, 2012.

[185] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124, 2004.

[186] Roger Pearce, Maya Gokhale, and Nancy M Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 549–559. IEEE, 2014.

[187] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*, 2019.

[188] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. Distributed RDF Query Answering with Dynamic Data Exchange. In *ISWC*, 2016.

[189] Ingo Müller. *Engineering Aggregation Operators for Relational In-Memory Database Systems*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2016.

[190] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News*, 43(1):631–644, 2015.

[191] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What Is Twitter, A Social Network Or A News Media? In *WWW*, 2010.

[192] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1), 2007.

[193] SNAP: Network Datasets – Google web graph. `https://snap.stanford.edu/data/web-Google.html`.

[194] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. An Early Look at The LDBC Social Network Benchmark's Business Intelligence Workload. In *GRADES Workshop*, 2018.

[195] 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 2015 ACM SIGMOD International Conference on Management of Data*, pages 1887–1901, 2015.

[196] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on typed graphs. *Proceedings of the VLDB Endowment*, 10(3):265–276, 2016.

[197] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698, 2017.

[198] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380. IEEE, 2020.

[199] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 745–761, USA, 2018. USENIX Association.

[200] Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, 2007.

[201] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[202] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM (JACM)*, 46(2):281–321, 1999.

[203] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *SIGKDD*, 2006.

[204] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Web Graph. *SIGIR Forum*, 42(2), 2008.

[205] LDBC SNB BI workload. `https://github.com/ldbc/ldbc_snb_bi`.

[206] Neo4j Admin Memrec – Neo4j memory recommendations. `https://neo4j.com/docs/operations-manual/current/tools/neo4j-admin/neo4j-admin-memrec/#_example`.

[207] PGTune – PostgreSQL configurator. `https://pgtune.leopard.in.ua`.

[208] Jacques Van Helden, Avi Naim, Renato Mancuso, Mattew Eldridge, Lorenz Wernisch, David Gilbert, and Shoshana J Wodak. Representing and analysing molecular and cellular function in the computer. *Biological chemistry*, 381(9-10):921–935, 2000.

[209] Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. *Managing and Mining Graph Data*, pages 181–215, 2010.

[210] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[211] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1009–1020. IEEE, 2013.

[212] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. Scarab: scaling reachability computation on large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 169–180, 2012.

[213] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: an independent permutation labeling approach. *The VLDB Journal*, 27:1–26, 2018.

[214] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[215] Zhiwei Zhang, Jeffrey Xu Yu, Lu Qin, Qing Zhu, and Xiaofang Zhou. I/O cost minimization: reachability queries processing over massive graphs. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 468–479, 2012.

[216] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

[217] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006.

[218] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012.

[219] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 193–204, 2013.

[220] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S Yu. Fast computation of reachability labeling for large graphs. In *Advances in Database Technology-EDBT 2006: 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006 10*, pages 961–979. Springer, 2006.

[221] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[222] Ruoming Jin and Guan Wang. Simple, fast, and scalable reachability oracle. *arXiv preprint arXiv:1305.0502*, 2013.

[223] Wenfei Fan, Xin Wang, and Yinghui Wu. Performance Guarantees for Distributed Reachability Queries. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[224] George HL Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. 2016.

[225] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 194–204, 1999.

[226] Wim Martens and Tina Trautner. Evaluation and enumeration problems for regular path queries. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[227] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems*, 40:47–66, 2014.

[228] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 345–358, 2017.

[229] Xin Chen, You Peng, Sibo Wang, and Jeffrey Xu Yu. DLCR: Efficient Indexing for Label-Constrained Reachability Queries on Large Dynamic Graphs. *Proc. VLDB Endow.*, 15(8):1645–1657, 2022.

[230] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond. *Proc. VLDB Endow.*, 13(6):812–825, 2020.

[231] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 122–133, 1997.

[232] APOC – Awesome Procedures On Cypher. `https://neo4j.com/labs/apoc/`.

[233] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[234] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[235] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[236] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. *ACM SIGMOD Record*, 28(2):181–192, 1999.

[237] Nicolas Bruno and Surajit Chaudhuri. Conditional selectivity for statistics on query expressions. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 311–322, 2004.

[238] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 211–222, 2001.

[239] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. A Pay-as-You-Go Framework for Query Execution Feedback. *Proc. VLDB Endow.*, 1(1), 2008.

[240] Surajit Chaudhuri and Vivek Narasayya. Automating statistics management for query optimizers. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):7–20, 2001.

[241] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 161–172, 1994.

[242] Utkarsh Srivastava, Peter J Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. Isomer: Consistent histogram construction using query feedback. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 39–39. IEEE, 2006.

[243] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. LEO-DB2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001.

[244] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 107–118, 2005.

[245] Kwanchai Eurviriyanukul, Norman W Paton, Alvaro AA Fernandes, and Steven J Lynden. Adaptive join processing in pipelined plans. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 183–194, 2010.

[246] Navin Kabra and David J DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 106–117, 1998.

[247] Quanzhong Li, Minglong Shao, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. Adaptively reordering joins during query execution. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 26–35. IEEE, 2006.

[248] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 659–670, 2004.

[249] Yali Zhu, Elke A Rundensteiner, and George T Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 431–442, 2004.

[250] Gennady Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 538–547. IEEE, 1993.

[251] Gennady Antoshenkov and Mohamed Ziauddin. Query Processing and Optimization in Oracle Rdb. *The VLDB Journal*, 5(4), 1996.

[252] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *Proceedings of the 31st international conference on Very large data bases*, pages 757–768, 2005.

[253] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth Scan: Robust Access Path Selection without Cardinality Estimation. *The VLDB Journal*, 27(4), 2018.

[254] Mohamed Ramadan, Ayman El-Kilany, Hoda M. O. Mokhtar, and Ibrahim Sobh. RL_QOptimizer: A Reinforcement Learning Based Query Optimizer. *IEEE Access*, 10, 2022.

[255] Jihad Zahir and Abderrahim El Qadi. A Recommendation System for Execution Plans Using Machine Learning. *Mathematical and Computational Applications*, 21(2), 2016.

[256] Ryan Marcus and Olga Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. *CoRR*, abs/1809.10212, 2018.

[257] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. SkinnerDB:

Regret-Bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.*, 46(3), 2021.

[258] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1275–1288, 2021.

[259] Qihang Chen, Boyu Tian, and Mingyu Gao. FINGERS: exploiting fine-grained parallelism in graph mining accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 43–55, 2022.

[260] Jingji Chen and Xuehai Qian. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 413–426, 2023.

[261] Yibo Wu, Jianfeng Zhu, Wenrui Wei, Longlong Chen, Liang Wang, Shaojun Wei, and Leibo Liu. Shogun: A Task Scheduling Framework for Graph Mining Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[262] Ldbc snb bi cypher queries. `https://github.com/ldbc/ldbc_snb_bi/tree/main/cypher/queries`.

[263] Ldbc snb bi postgresql queries. `https://github.com/ldbc/ldbc_snb_bi/tree/main/umbra/queries`.

List of Publications and Patents

Publications

Distributed Asynchronous Regular Path Queries (RPQs) on Graphs.
Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Jinsoo Lee, Jakub Yaghob, Sungpack Hong, and Hassan Chafi. 2023.
International Middleware Conference Industrial Track 2023 (accepted)

Better Distributed Graph Query Planning With Scouting Queries.
Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Sungpack Hong, and Hassan Chafi. 2023.
In Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES & NDA '23). Association for Computing Machinery, New York, NY, USA, Article 3, 1–9. https://doi.org/10.1145/3594778.3594884

aDFS: An Almost Depth-First-Search Distributed Graph-Querying System.
Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, Hassan Chafi. 2021.
USENIX Annual Technical Conference 2021: 209-224, Online, United States. hal-03249229

BDgen: A Universal Big Data Generator.
Tomáš Faltín, Michal Hanzeli, Vojtěch Šípek, Jan Škvařil, Dušan Variš, and Irena Holubová Mlýnková. 2017. In Proceedings of the 21st International Database Engineering & Applications Symposium (IDEAS '17). Association for Computing Machinery, New York, NY, USA, 200–208.
https://doi.org/10.1145/3105831.3105847

Patents

Dynamic asynchronous traversals for distributed graph queries.
Patent number: 11675785. Date of Patent: June 13, 2023. Assignee: Oracle International Corporation. Inventors: Vasileios Trigonakis, Tomas Faltin, Jean-Pierre Lozi, Vlad Ioan Haprian, Sungpack Hong, Hassan Chafi.

Regular path queries (RPQs) for distributed graphs.
Patent number: 11456946. Date of Patent: September 27, 2022. Assignee: Oracle International Corporation. Inventors: Petar Tonkovic, Vasileios Trigonakis, Tomas Faltin, Sungpack Hong, Hassan Chafi.

Space-efficient methodology for representing label information in large graph data for fast distributed graph query.
Patent number: 11074260. Date of Patent: July 27, 2021 Assignee: Oracle International Corporation. Inventors: Arnaud Delamare, Vasileios Trigonakis, Vlad Ioan Haprian, Oskar Van Rest, Sungpack Hong, Hassan Chafi, Tomas Faltin, Jean-Pierre Lozi.

Duplication elimination in depth based searches for distributed systems.
Publication number: 20220179859. Type: Application. Filed: December 9, 2020. Publication date: June 9, 2022. Inventors: Tomas Faltin, Vasileios Trigonakis, Jean-Pierre Lozi, Sungpack Hong, Hassan Chafi.

A. LDBC Social Network Benchmark Business Intelligence Queries

All queries presented in the appendix are not the official LDBC [194] standard queries and have not been audited by LDBC. We devised simplified variants of LDBC Business Intelligence (BI) standard queries in order to support the benchmark specification as closely as possible. The original queries are available in the official LDBC repository [205]: Cypher queries [262] and PostgreSQL queries [263]. We present here the queries in Cypher and SQL for comparison.

A.1 Fixed-size Pattern Queries

Fixed-size pattern queries were used for evaluation purpose of aDFS approach in Chapter 4.

A.1.1 Cypher

Q1

```
MATCH (message:Message)
WHERE message.creationDate < datetime('2011-12-01')
WITH count(message) AS totalMessageCountInt
WITH toFloat(totalMessageCountInt) AS totalMessageCount
MATCH (message:Message)
WHERE message.creationDate < datetime('2011-12-01')
  AND message.content IS NOT NULL
WITH
  totalMessageCount,
  message,
  message.creationDate.year AS year
WITH
  totalMessageCount,
  year,
  message:Comment AS isComment,
  CASE
    WHEN message.length <  40 THEN 0
    WHEN message.length <  80 THEN 1
    WHEN message.length < 160 THEN 2
    ELSE                           3
  END AS lengthCategory,
  count(message) AS messageCount,
  floor(avg(message.length)) AS averageMessageLength,
  sum(message.length) AS sumMessageLength
RETURN
  year,
  isComment,
  lengthCategory,
  messageCount,
  averageMessageLength,
  sumMessageLength,
  messageCount / totalMessageCount AS percentageOfMessages
ORDER BY year DESC,
         isComment ASC,
         lengthCategory ASC
LIMIT 10;
```

Q2

```
MATCH
  (message:Message) -[:HAS_TAG]-> (tag:Tag)
  -[:HAS_TYPE]-> (tagClass:TagClass {name: 'MusicalArtist'})
WHERE message.creationDate >= datetime('2012-09-09')
  AND message.creationDate < datetime('2012-12-18')
RETURN
  tag.id,
  tagClass.id,
  COUNT(*) AS cnt
ORDER BY cnt DESC
LIMIT 10;
```

Q3

```
MATCH
  (:Country {name: 'Burma'}) <-[:IS_PART_OF]- (:City)
  <-[:IS_LOCATED_IN]- (person:Person)
  <-[:HAS_MODERATOR]- (forum:Forum)
  -[:CONTAINER_OF]-> (post:Post)
  <-[:REPLY_OF]- (message:Message)
  -[:HAS_TAG]-> (:Tag)
  -[:HAS_TYPE]-> (:TagClass {name: 'MusicalArtist'})
RETURN
  forum.id,
  forum.title,
  forum.creationDate,
  person.id,
  count(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  forum.id ASC
LIMIT 10;
```

Q7

```
MATCH
  (tag:Tag {name: 'Enrique_Iglesias'})
  <-[:HAS_TAG]- (message:Message)
  <-[:REPLY_OF]- (comment:Comment)
  -[:HAS_TAG]-> (relatedTag:Tag),
  (comment) -[:HAS_TAG]-> (tag)
RETURN
  relatedTag.name,
  count(DISTINCT comment) AS count
ORDER BY
  count DESC,
  relatedTag.name ASC
LIMIT 10;
```

Q9

```
MATCH
  (person:Person) <-[:HAS_CREATOR]- (post:Post)
  <-[:REPLY_OF*3]- (reply:Message)
WHERE  post.creationDate >= datetime('2011-10-01T00:00:00')
  AND  post.creationDate <= datetime('2011-10-15T00:00:00')
  AND reply.creationDate >= datetime('2011-10-01T00:00:00')
  AND reply.creationDate <= datetime('2011-10-15T00:00:00')
RETURN
  person.id,
  person.firstName,
  person.lastName,
  count(DISTINCT post) AS threadCount,
  count(DISTINCT reply) AS messageCount
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 10;
```

Q11

```
MATCH
  (a) -[k1:KNOWS]-> (b) -[k2:KNOWS]-> (c) <-[k3:KNOWS]- (a),
  (a:Person) -[:IS_LOCATED_IN]-> (:City)
  -[:IS_PART_OF]-> (:Country {name: 'Belarus'}),
  (b:Person) -[:IS_LOCATED_IN]-> (:City)
  -[:IS_PART_OF]-> (:Country {name: 'Belarus'}),
  (c:Person) -[:IS_LOCATED_IN]-> (:City)
  -[:IS_PART_OF]-> (:Country {name: 'Belarus'})
WHERE a.id < b.id AND b.id < c.id
  AND datetime('2010-06-01T00:00:00') <= k1.creationDate
  AND datetime('2010-06-01T00:00:00') <= k2.creationDate
  AND datetime('2010-06-01T00:00:00') <= k3.creationDate
RETURN count(*) AS count;
```

Q12

```
MATCH
  (person) <-[:HAS_CREATOR]- (message:Message)
  -[:REPLY_OF]-> (post:Post)
WHERE message.content IS NOT NULL
  AND message.length < 20
  AND message.creationDate > datetime('2010-07-22T00:00:00')
  AND post.language IN ['ar', 'hu']
WITH
  person,
  count(message) AS messageCount
RETURN messageCount
ORDER BY messageCount DESC
LIMIT 10;
```

Q13

```
MATCH
   (message:Message) <-[:LIKES]- (likerZombie:Person),
   (zombie) <-[:HAS_CREATOR]- (message)
RETURN
  zombie.id,
  COUNT(*) AS zombieScore
ORDER BY zombieScore DESC
LIMIT 10;
```

Q14

```
MATCH
   (country1:Country {name: 'Chile'}) <-[:IS_PART_OF]- (city1:City)
  <-[:IS_LOCATED_IN]- (person1:Person),
   (country2:Country {name: 'Argentina'})
  <-[:IS_PART_OF]- (city2:City)
  <-[:IS_LOCATED_IN]- (person2:Person),
   (person1) <-[:HAS_CREATOR]- (c:Comment) -[:REPLY_OF]-> (:Message)
  -[:HAS_CREATOR]-> (person2)
RETURN COUNT(*);
```

Q15

```
MATCH
   (p1:Person) -[:KNOWS]-> (:Person) -[:KNOWS]-> (p2:Person)
WHERE p1.id <> p2.id
RETURN COUNT(*);
```

Q18

```
MATCH
   (person1:Person) -[:KNOWS]- (mutualFriend:Person)
  <-[:KNOWS]- (person2:Person)
  -[:HAS_INTEREST]- (q:Tag {name: 'Frank_Sinatra'}),
   (person1) -[:KNOWS]-> (person2)
WHERE person1 <> person2
RETURN
  person1.id AS person1Id,
  person2.id AS person2Id,
  count(DISTINCT mutualFriend) AS mutualFriendCount
ORDER BY
  mutualFriendCount DESC,
  person2Id ASC
LIMIT 10;
```

Q19

```
MATCH
  (personA:Person) -[e:KNOWS]-> (personB:Person),
  (personA) <-[:HAS_CREATOR]- (:Message)
  -[replyOf:REPLY_OF]- (:Message)
  -[:HAS_CREATOR]-> (personB)
RETURN
  e,
  COUNT(*) AS cnt
ORDER BY cnt DESC
LIMIT 10;
```

Q20

```
MATCH
  (company:Company {name: 'Falcon_Air'})
  <-[:WORK_AT]- (person1:Person),
  (person1:Person) -[:KNOWS]-> (:Person)
  <-[:KNOWS]- (person2:Person)
WHERE person1.id <> person2.id
RETURN COUNT(*);
```

A.1.2 PostgreSQL

Q1

```
WITH message_count AS (
    SELECT 0.0 + count(*) AS cnt
      FROM message
     WHERE 1=1
       AND m_creationdate < '2011-12-01T00:00:00.000+00:00'
)
, message_prep AS (
    SELECT extract(year from m_creationdate) AS messageYear
         , m_c_replyof IS NOT NULL AS isComment
         , CASE
             WHEN m_length <  40 THEN 0 -- short
             WHEN m_length <  80 THEN 1 -- one liner
             WHEN m_length < 160 THEN 2 -- tweet
             ELSE                     3 -- long
           END AS lengthCategory
         , m_length
      FROM message
     WHERE 1=1
       AND m_creationdate < '2011-12-01T00:00:00.000+00:00'
       AND m_ps_imagefile IS NULL
)
SELECT messageYear, isComment, lengthCategory
     , count(*) AS messageCount
     , cast(avg(m_length) AS INT) AS averageMessageLength
     , sum(m_length) AS sumMessageLength
     , count(*) / mc.cnt AS percentageOfMessages
  FROM message_prep
     , message_count mc
 GROUP BY messageYear, isComment, lengthCategory, mc.cnt
 ORDER BY messageYear DESC, isComment ASC, lengthCategory ASC;
```

Q2

```
SELECT
  tag.t_tagid as TagId,
  tagclass.tc_tagclassid AS tc,
  count(*) AS cnt
FROM message
  JOIN message_tag
    ON message.m_messageid = message_tag.mt_messageid
  JOIN tag
    ON message_tag.mt_tagid = tag.t_tagid
  JOIN tagclass
    ON tag.t_tagclassid = tagclass.tc_tagclassid
  WHERE
    tagclass.tc_name = 'MusicalArtist'
    AND message.m_creationdate >= '2012-09-09T00:00:00.000+00:00'
    AND message.m_creationdate <  '2012-12-18T00:00:00.000+00:00'
 GROUP BY tag.t_tagid, tagclass.tc_tagclassid
 ORDER BY cnt DESC
 LIMIT 10;
```

Q3

```
SELECT
  forum.f_forumid,
  forum.f_title,
  forum.f_creationdate,
  forum.f_moderatorid AS "personId",
  count(DISTINCT message.m_messageid) AS messageCount
FROM country
JOIN place AS city
  ON country.ctry_city = city.pl_placeid
JOIN person
  ON city.pl_placeid = person.p_placeid
JOIN forum
  ON person.p_personid = forum.f_moderatorid
JOIN message AS post
  ON forum.f_forumid = post.m_ps_forumid
JOIN message
  ON post.m_messageid = message.m_c_replyof
JOIN message_tag
  ON message.m_messageid = message_tag.mt_messageid
JOIN tag
  ON message_tag.mt_tagid = tag.t_tagid
JOIN tagclass
  ON tag.t_tagclassid = tagclass.tc_tagclassid
WHERE
  country.ctry_name = 'Burma'
  AND tagclass.tc_name = 'MusicalArtist'
GROUP BY forum.f_forumid, person.p_personid
ORDER BY messageCount DESC, forum.f_forumid
LIMIT 10;
```

Q7

```sql
SELECT
  relatedTag.t_name,
  COUNT(DISTINCT comment) AS count
FROM message AS m
JOIN message_tag AS mt1
  ON m.m_messageid = mt1.mt_messageid
JOIN tag
  ON mt1.mt_tagid = tag.t_tagid
JOIN message AS comment
  ON m.m_messageid = comment.m_c_replyof
JOIN message_tag AS mt2
  ON comment.m_messageid = mt2.mt_messageid
JOIN tag AS relatedTag
  ON mt2.mt_tagid = relatedTag.t_tagid
JOIN message_tag AS mt3
  ON
    (comment.m_messageid = mt3.mt_messageid
    AND mt3.mt_tagid = tag.t_tagid)
WHERE tag.t_name = 'Enrique_Iglesias'
GROUP BY relatedTag.t_name
ORDER BY count DESC, relatedTag.t_name
LIMIT 10;
```

Q9

```sql
SELECT
  person.p_personid,
  person.p_firstname,
  person.p_lastname,
  COUNT(DISTINCT message) AS threadCount,
  COUNT(DISTINCT reply) AS messageCount
FROM person
JOIN message
  ON person.p_personid = message.m_creatorid
JOIN message AS c1
  ON message.m_messageid = c1.m_c_replyof
JOIN message AS c2
  ON c1.m_messageid = c2.m_c_replyof
JOIN message AS reply
  ON c2.m_messageid = reply.m_c_replyof
WHERE
  message.m_c_replyof IS NULL
  AND message.m_creationdate >= '2011-10-01 00:00:00'
  AND message.m_creationdate <= '2011-10-15 00:00:00'
  AND reply.m_creationdate >= '2011-10-01 00:00:00'
  AND reply.m_creationdate <= '2011-10-15 00:00:00'
  AND c1.m_creationdate >= '2011-10-01 00:00:00'
  AND c1.m_creationdate <= '2011-10-15 00:00:00'
  AND c2.m_creationdate >= '2011-10-01 00:00:00'
  AND c2.m_creationdate <= '2011-10-15 00:00:00'
GROUP BY person.p_personid, person.p_firstname, person.p_lastname
ORDER BY messageCount DESC, person.p_personid ASC
LIMIT 10;
```

Q11

```
SELECT COUNT(*)
FROM person AS a
JOIN knows AS k1
  ON a.p_personid = k1.k_person1id
JOIN person AS b
  ON k1.k_person2id = b.p_personid
JOIN knows AS k2
  ON b.p_personid = k2.k_person1id
JOIN person AS c
  ON k2.k_person2id = c.p_personid
JOIN knows AS k3
  ON (c.p_personid = k3.k_person2id
     AND k3.k_person1id = a.p_personid)
JOIN place AS place1
  ON a.p_placeid = place1.pl_placeid
JOIN place AS city1
  ON place1.pl_containerplaceid = city1.pl_placeid
JOIN place AS place2
  ON b.p_placeid = place2.pl_placeid
JOIN place AS city2
  ON place2.pl_containerplaceid = city2.pl_placeid
JOIN place AS place3
  ON c.p_placeid = place3.pl_placeid
JOIN place AS city3
  ON place3.pl_containerplaceid = city3.pl_placeid

WHERE
  a.p_personid < b.p_personid AND b.p_personid < c.p_personid
  AND '2010-06-01 00:00:00' <= k1.k_creationdate
  AND '2010-06-01 00:00:00' <= k2.k_creationdate
  AND '2010-06-01 00:00:00' <= k3.k_creationdate
  AND city1.pl_name = 'Belarus' AND city1.pl_type = 'country'
  AND place1.pl_type = 'city'
  AND city2.pl_name = 'Belarus' AND city2.pl_type = 'country'
  AND place2.pl_type = 'city'
  AND city3.pl_name = 'Belarus' AND city3.pl_type = 'country'
  AND place3.pl_type = 'city';
```

Q12

```
SELECT COUNT(message) AS messageCount
FROM person AS p
JOIN message
  ON p.p_personid = message.m_creatorid
JOIN message AS post
  ON message.m_c_replyof = post.m_messageid
WHERE
  post.m_c_replyof IS NULL
  AND message.m_content <> ''
  AND message.m_length < 20
  AND message.m_creationDate > '2010-07-22 00:00:00'
  AND post.m_ps_language IN ( 'ar', 'hu' )
GROUP BY p
ORDER BY messageCount DESC
LIMIT 10;
```

Q13

```sql
SELECT
  zombie.p_personid,
  COUNT(likerPerson) AS zombieScore
FROM person AS likerPerson
JOIN likes
  ON likerPerson.p_personid = likes.l_personid
JOIN message
  ON likes.l_messageid = message.m_messageid
JOIN person AS zombie
  ON message.m_creatorid = zombie.p_personid
GROUP BY zombie.p_personid
ORDER BY zombieScore DESC, zombie.p_personid
LIMIT 10;
```

Q14

```sql
SELECT COUNT(*)
FROM country AS country1
JOIN place AS city1
  ON country1.ctry_city = city1.pl_placeid
JOIN person AS person1
  ON city1.pl_placeid = person1.p_placeid
JOIN message AS comment1
  ON person1.p_personid = comment1.m_creatorid
JOIN message AS comment2
  ON comment1.m_c_replyof = comment2.m_messageid
JOIN person AS person2
  ON comment2.m_creatorid = person2.p_personid
JOIN place AS city2
  ON person2.p_placeid = city2.pl_placeid
JOIN country AS country2
  ON city2.pl_placeid = country2.ctry_city
WHERE
  country1.ctry_name = 'Chile'
  AND city1.pl_type = 'city'
  AND country2.ctry_name = 'Argentina'
  AND city2.pl_type = 'city';
```

Q15

```sql
SELECT COUNT(*)
FROM person AS p1
JOIN knows AS k1
  ON p1.p_personid = k1.k_person1id
JOIN person AS p2
  ON k1.k_person2id = p2.p_personid
JOIN knows AS k2
  ON p2.p_personid = k2.k_person1id
JOIN person AS p3
  ON k2.k_person2id = p3.p_personid
WHERE
  p1.p_personid < p2.p_personid
  AND p2.p_personid < p3.p_personid;
```

Q18

```sql
SELECT
  person1.p_personid AS person1Id,
  person2.p_personid AS person2Id,
  COUNT(DISTINCT mutualFriend) AS mutualFriendCount
FROM person AS person1
JOIN knows AS k1
  ON person1.p_personid = k1.k_person1id
JOIN person AS mutualFriend
  ON k1.k_person2id = mutualFriend.p_personid
JOIN knows AS k2
  ON mutualFriend.p_personid = k2.k_person2id
JOIN person AS person2
  ON k2.k_person1id = person2.p_personid
JOIN person_tag
  ON person2.p_personid = person_tag.pt_personid
JOIN tag AS q
  ON person_tag.pt_tagid = q.t_tagid
JOIN knows AS k3
  ON (
    person1.p_personid = k3.k_person1id
    AND k3.k_person2id = person2.p_personid
  )
WHERE
  q.t_name = 'Frank_Sinatra'
  AND k1.k_person1id < k1.k_person2id
  AND k2.k_person1id < k2.k_person2id
  AND k3.k_person1id < k3.k_person2id
  AND person1.p_personid <> person2.p_personid
GROUP BY person1.p_personid, person2.p_personid
ORDER BY mutualFriendCount DESC, person2Id ASC
LIMIT 10;
```

Q19

```sql
SELECT
  k1.k_creationdate,
  COUNT(*) AS cnt
FROM person AS personA
JOIN knows AS k1
  ON personA.p_personid = k1.k_person1id
JOIN person AS personB
  ON k1.k_person2id = personB.p_personid
JOIN message AS pC1
  ON personA.p_personid = pC1.m_creatorid
JOIN message AS pC2
  ON (
    pC1.m_messageid = pC2.m_c_replyof
    AND personB.p_personid = pC2.m_creatorid
  )
WHERE k1.k_person1id < k1.k_person2id
GROUP BY k1.k_creationdate
ORDER BY cnt DESC
LIMIT 10;
```

Q20

```
SELECT COUNT(*)
FROM organisation AS company
JOIN person_company
  ON company.o_organisationid = person_company.pc_organisationid
JOIN person AS person1
  ON person_company.pc_personid = person1.p_personid
JOIN knows AS k1
  ON person1.p_personid = k1.k_person1id
JOIN person AS mutualFriend
  ON k1.k_person2id = mutualFriend.p_personid
JOIN knows AS k2
  ON mutualFriend.p_personid = k2.k_person2id
WHERE
  k1.k_person1id < k1.k_person2id
  AND k2.k_person1id < k2.k_person2id
  AND company.o_type = 'company'
  AND company.o_name = 'Falcon_Air'
  AND k2.k_person1id != person1.p_personid;
```

A.2 RPQ Queries

RPQ queries were used for evaluation purposes of RPQd engine in Chapter 5.

A.2.1 Cypher

Q3*

```
MATCH
  (:Country {name: 'Burma'}) <-[:IS_PART_OF]- (:City)
  <-[:IS_LOCATED_IN]- (person:Person)
  <-[:HAS_MODERATOR]- (forum:Forum)
  -[:CONTAINER_OF]-> (post:Post)
  <-[:REPLY_OF*0..]- (message:Message)
  -[:HAS_TAG]-> (:Tag)
  -[:HAS_TYPE]-> (:TagClass {name: 'MusicalArtist'})
RETURN
  forum.id,
  forum.title,
  forum.creationDate,
  person.id,
  count(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  forum.id ASC
LIMIT 10;
```

Q4

```
MATCH
   (person:Person) <-[:HAS_CREATOR]- (message:Message)
  -[:REPLY_OF*0..]-> (post:Post)
WHERE person.id < 1000000
RETURN
  person.id,
  person.firstName,
  person.lastName,
  person.creationDate,
  COUNT(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 10;
```

Q9*

```
MATCH
   (person:Person) <-[:HAS_CREATOR]- (post:Post)
  <-[:REPLY_OF*0..]- (reply:Message)
WHERE  post.creationDate >= datetime('2012-05-31T22:00:00')
  AND  post.creationDate <= datetime('2012-06-30T22:00:00')
  AND reply.creationDate >= datetime('2012-05-31T22:00:00')
  AND reply.creationDate <= datetime('2012-06-30T22:00:00')
RETURN
  person.id,
  person.firstName,
  person.lastName,
  count(DISTINCT post) AS threadCount,
  count(DISTINCT reply) AS messageCount
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 10;
```

Q10*

```
:param personId => 19791209310731
:param country => 'Pakistan'
:param tagClass => 'MusicalArtist'
:param minPathDistance => 2
:param maxPathDistance => 3
MATCH (startPerson:Person {id: $personId})
CALL apoc.path.subgraphNodes(startPerson, {
        relationshipFilter: "KNOWS",
    minLevel: 1, maxLevel: $minPathDistance-1})
YIELD node
WITH startPerson,
  collect(DISTINCT node) AS nodesCloserThanMinPathDistance
CALL apoc.path.subgraphNodes(startPerson, {
        relationshipFilter: "KNOWS",
    minLevel: 1, maxLevel: $maxPathDistance})
YIELD node
WITH nodesCloserThanMinPathDistance,
  collect(DISTINCT node) AS nodesCloserThanMaxPathDistance
WITH [n IN nodesCloserThanMaxPathDistance
  WHERE NOT n IN nodesCloserThanMinPathDistance]
    AS expertCandidatePersons
UNWIND expertCandidatePersons AS expertCandidatePerson
MATCH
  (expertCandidatePerson) -[:IS_LOCATED_IN]-> (:City)
  -[:IS_PART_OF]-> (:Country {name: $country}),
  (expertCandidatePerson) <-[:HAS_CREATOR]- (message:Message)
  -[:HAS_TAG]-> (:Tag) -[:HAS_TYPE]-> (:TagClass {name: $tagClass})
MATCH (message) -[:HAS_TAG]-> (tag:Tag)
RETURN
  expertCandidatePerson.id,
  tag.name,
  count(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  tag.name ASC,
  expertCandidatePerson.id ASC
LIMIT 10;
```

Q12

```
MATCH
  (person:Person) <-[:HAS_CREATOR]- (message:Message)
  -[:REPLY_OF*0..]-> (post:Post)
WHERE
  person.id < 1000000
  AND message.content IS NOT NULL
  AND message.length < 50
  AND message.creationDate > datetime('2010-07-22')
  AND post.language IN ['ar', 'uz']
RETURN
  person.id AS _,
  COUNT(message) AS cnt
ORDER BY cnt DESC
LIMIT 10;
```

Q15

```
MATCH
   (forum:Forum) -[:CONTAINER_OF]-> (post:Post),
   (post) <-[:REPLY_OF*]- (c2:Comment),
   (c1:Comment) -[:REPLY_OF]-> (c2),
   (p1:Person) <-[:HAS_CREATOR]- (c1),
   (c2) -[:HAS_CREATOR]-> (p2:Person)
WHERE forum.creationDate >= datetime('2011-06-01')
  AND forum.creationDate <= datetime('2012-05-31')
  AND p1.id = 19791209303405 AND p2.id = 19791209308983
RETURN COUNT(*)
LIMIT 10;
```

Q19

```
MATCH
   (person:Person) <-[:HAS_CREATOR]- (comment:Comment)
  -[:REPLY_OF*0..]-> (message:Message),
   (message) -[:HAS_CREATOR]-> (stranger:Person)
WHERE person <> stranger AND person.birthday > date('1989-01-01')
RETURN COUNT(DISTINCT comment.id + stranger.id);
```

Q20

```
MATCH
   (tagClass:TagClass) <-[:IS_SUBCLASS_OF*0..]- (:TagClass)
  <-[:HAS_TYPE]- (tag:Tag) <-[:HAS_TAG]- (message:Message)
WHERE
  tagClass.name = 'Writer'
  OR tagClass.name = 'Single'
  OR tagClass.name = 'Country'
RETURN
  COUNT(message) AS messageCount;
```

A.2.2 PostgreSQL

Q3*

```
WITH RECURSIVE replyof AS (
    -- initial
    SELECT
        message.m_messageid AS src,
        message.m_messageid AS dst,
        0 AS depth
    FROM message
UNION ALL
    -- recursion
    SELECT
        replyof.src AS src,
        message.m_c_replyof AS dst,
        replyof.depth + 1 AS depth
    FROM replyof
    JOIN message ON replyof.dst = message.m_messageid
    WHERE message.m_c_replyof IS NOT NULL
)
SELECT
  forum.f_forumid,
  forum.f_title,
  forum.f_creationdate,
  person.p_personid,
  count(DISTINCT message_tag.mt_messageid) AS messageCount
FROM place AS country
JOIN place AS city
  ON country.pl_placeid = city.pl_containerplaceid
JOIN person
  ON city.pl_placeid = person.p_placeid
JOIN forum
  ON person.p_personid = forum.f_moderatorid
JOIN message AS post
  ON forum.f_forumid = post.m_ps_forumid
JOIN replyof
  ON post.m_messageid = replyof.dst
JOIN message_tag
  ON replyof.src = message_tag.mt_messageid
JOIN tag
  ON message_tag.mt_tagid = tag.t_tagid
JOIN tagclass
  ON tag.t_tagclassid = tagclass.tc_tagclassid
WHERE
  country.pl_name = 'Burma'
  AND post.m_c_replyof IS NULL -- is post
  AND tagclass.tc_name = 'MusicalArtist'
GROUP BY
  forum.f_forumid,
  forum.f_title,
  forum.f_creationdate,
  person.p_personid
ORDER BY
  messageCount DESC,
  forum.f_forumid ASC
LIMIT 10;
```

Q4

```
WITH RECURSIVE replyof AS (
    -- initial
    SELECT
        message.m_messageid AS src,
        message.m_messageid AS dst,
        0 AS depth
    FROM message
UNION ALL
    -- recursion
    SELECT
        replyof.src AS src,
        message.m_c_replyof AS dst,
        replyof.depth + 1 AS depth
    FROM replyof
    JOIN message ON replyof.dst = message.m_messageid
    WHERE message.m_c_replyof IS NOT NULL
)

SELECT
  person.p_personid,
  person.p_firstname,
  person.p_lastname,
  person.p_creationdate,
  COUNT(DISTINCT m1) AS messageCount
FROM person
JOIN message AS m1
    ON person.p_personid = m1.m_creatorid
JOIN replyof AS p1
    ON m1.m_messageid = p1.src
WHERE
    person.p_personid < 1000000
GROUP BY
  person.p_personid,
  person.p_firstname,
  person.p_lastname,
  person.p_creationdate
ORDER BY messageCount DESC, person.p_personid ASC
LIMIT 10;
```

Q9*

```
WITH RECURSIVE replyof AS (
    -- initial
    SELECT
        message.m_messageid AS src,
        message.m_messageid AS dst,
        0 AS depth
    FROM message
UNION ALL
    -- recursion
    SELECT
        replyof.src AS src,
        message.m_c_replyof AS dst,
        replyof.depth + 1 AS depth
    FROM replyof
    JOIN message ON replyof.dst = message.m_messageid
    WHERE message.m_c_replyof IS NOT NULL
)

SELECT
  person.p_personid,
  person.p_firstname,
  person.p_lastname,
  count(DISTINCT post1) AS threadCount,
  count(DISTINCT reply) AS messageCount
FROM person
JOIN message AS post1
  ON person.p_personid = post1.m_creatorid
JOIN replyof
  ON post1.m_messageid = replyof.dst
JOIN message AS reply
  ON replyof.src = reply.m_messageid
WHERE
  post1.m_c_replyof IS NULL -- is post
  AND post1.m_creationdate BETWEEN
    '2012-05-31T22:00:00' AND '2012-06-30T22:00:00'
  AND reply.m_creationdate BETWEEN
    '2012-05-31T22:00:00' AND '2012-06-30T22:00:00'
GROUP BY
  person.p_personid,
  person.p_firstname,
  person.p_lastname
ORDER BY
  messageCount DESC,
  person.p_personid ASC
LIMIT 10;
```

Q10*

```sql
WITH RECURSIVE fixed_knows_3 AS (
    -- initial
    SELECT
        p_personid AS dst,
        0 AS depth
    FROM person
    WHERE person.p_personid = 19791209310731
UNION
    -- recursion
    SELECT
        knows.k_person2id AS dst,
        fixed_knows_3.depth + 1 AS depth
    FROM fixed_knows_3
    JOIN knows
        ON fixed_knows_3.dst = knows.k_person1id
    WHERE depth <= 3
),
fixed_knows_2_3 AS (
    SELECT DISTINCT dst
    FROM fixed_knows_3
    WHERE
        fixed_knows_3.depth = 2
        OR fixed_knows_3.depth = 3
)
SELECT
  person2.p_personid,
  tag.t_name,
  COUNT(DISTINCT message) AS messageCount
FROM fixed_knows_2_3
JOIN person AS person2
    ON fixed_knows_2_3.dst = person2.p_personid
JOIN place AS city
    ON person2.p_placeid = city.pl_placeid
JOIN place AS country
    ON city.pl_containerplaceid = country.pl_placeid
JOIN message
    ON person2.p_personid = message.m_creatorid
JOIN message_tag
    ON message.m_messageid = message_tag.mt_messageid
JOIN tag
    ON message_tag.mt_tagid = tag.t_tagid
JOIN message_tag AS message_tag2
    ON message.m_messageid = message_tag2.mt_messageid
JOIN tag AS tag2
    ON message_tag2.mt_tagid = tag2.t_tagid
JOIN tagclass
    ON tag2.t_tagclassid = tagclass.tc_tagclassid
WHERE
    person2.p_personid <> 19791209310731
    AND tagclass.tc_name = 'MusicalArtist'
    AND country.pl_name = 'Pakistan'
GROUP BY person2.p_personid, tag.t_name
ORDER BY messageCount DESC, tag.t_name, person2.p_personid
LIMIT 10;
```

Q12

```sql
WITH RECURSIVE replyof AS (
    -- initial
    SELECT
        message.m_messageid AS src,
        message.m_messageid AS dst,
        0 AS depth
    FROM message
UNION ALL
    -- recursion
    SELECT
        replyof.src AS src,
        message.m_c_replyof AS dst,
        replyof.depth + 1 AS depth
    FROM replyof
    JOIN message ON replyof.dst = message.m_messageid
    WHERE message.m_c_replyof IS NOT NULL
)

SELECT
  COUNT(message) AS cnt
FROM person
JOIN message
  ON person.p_personid = message.m_creatorid
JOIN replyof
  ON message.m_messageid = replyof.src
JOIN message AS post1
  ON replyof.dst = post1.m_messageid
WHERE
  post1.m_c_replyof IS NULL -- is post
  AND person.p_personid < 1000000
  AND message.m_content IS NOT NULL
  AND message.m_length < 50
  AND message.m_creationdate > '2010-07-22'
  AND post1.m_ps_language IN ('ar', 'uz')
GROUP BY
  person.p_personid
ORDER BY
  cnt DESC
LIMIT 10;
```

Q15

```sql
WITH RECURSIVE replyof AS (
    -- initial
    SELECT
        message.m_messageid AS src,
        message.m_messageid AS dst,
        0 AS depth
    FROM message
UNION ALL
    -- recursion
    SELECT
        replyof.src AS src,
        message.m_c_replyof AS dst,
        replyof.depth + 1 AS depth
    FROM replyof
    JOIN message ON replyof.dst = message.m_messageid
    WHERE message.m_c_replyof IS NOT NULL
)

SELECT COUNT(*)
FROM forum
JOIN message AS post
  ON forum.f_forumid = post.m_ps_forumid

JOIN replyof
  ON post.m_messageid = replyof.dst
JOIN message AS c2
  ON replyof.src = c2.m_messageid

JOIN message AS c1
  ON c1.m_c_replyof = c2.m_messageid

JOIN person AS p1
  ON c1.m_creatorid = p1.p_personid

JOIN person AS p2
  ON c2.m_creatorid = p2.p_personid
WHERE
  post.m_c_replyof IS NULL -- is post
  AND c1.m_c_replyof IS NOT NULL -- is comment
  AND c2.m_c_replyof IS NOT NULL -- is comment
  AND forum.f_creationdate BETWEEN '2011-06-01' AND '2012-05-31'
  AND p1.p_personid = 19791209303405
  AND p2.p_personid = 19791209308983
LIMIT 10;
```

Q19

```sql
WITH RECURSIVE replyof AS (
    -- initial
    SELECT
        message.m_messageid AS src,
        message.m_messageid AS dst,
        0 AS depth
    FROM message
UNION ALL
    -- recursion
    SELECT
        replyof.src AS src,
        message.m_c_replyof AS dst,
        replyof.depth + 1 AS depth
    FROM replyof
    JOIN message ON replyof.dst = message.m_messageid
    WHERE message.m_c_replyof IS NOT NULL
)

SELECT
  COUNT(DISTINCT comment.m_messageid + stranger.p_personid)
FROM person
JOIN message AS comment
  ON person.p_personid = comment.m_creatorid
JOIN replyof
  ON comment.m_messageid = replyof.src
JOIN message
  ON replyof.dst = message.m_messageid

JOIN person AS stranger
  ON message.m_creatorid = stranger.p_personid
WHERE
  comment.m_c_replyof IS NOT NULL -- is comment
  AND person <> stranger
  AND person.p_birthday > '1989-01-01';
```

Q20

```
WITH RECURSIVE issubclassof AS (
    -- initial
    SELECT
        tagclass.tc_tagclassid AS src,
        tagclass.tc_tagclassid AS dst,
        0 AS depth
    FROM tagclass
UNION ALL
    -- recursion
    SELECT
        tagclass.tc_tagclassid AS src,
        issubclassof.dst AS dst,
        issubclassof.depth + 1 AS depth
    FROM issubclassof
    JOIN tagclass
      ON issubclassof.src = tagclass.tc_subclassoftagclassid
)

SELECT
  COUNT(message_tag) AS messageCount
FROM tagclass AS tc1
JOIN issubclassof
  ON tc1.tc_tagclassid = issubclassof.dst
JOIN tagclass AS tc2
  ON issubclassof.src = tc2.tc_tagclassid
JOIN tag
  ON tc2.tc_tagclassid = tag.t_tagclassid
JOIN message_tag
  ON tag.t_tagid = message_tag.mt_tagid
WHERE
  tc1.tc_name = 'Writer'
  OR tc1.tc_name = 'Single'
  OR tc1.tc_name = 'Country';
```