Efficient Single Step Traversals in Main-Memory Graph-Shaped Data

Author: Marcus Pinnecke

June 21, 2016

Advisors:
Prof. Dr. Gunter Saake
Sebastian Dorok, M.Sc.
University of Magdeburg

Dr. Sebastian Breß
German Research Center for Artificial Intelligence
Pinnecke, Marcus:

Efficient Single Step Traversals in Main-Memory Graph-Shaped Data

Abstract

Management of graph-shaped data gained a momentum to both industry and research. Traversal queries through a graph-shaped dataset are easy to express, and can be efficiently executed using graph databases. High-performance traversals through graph-shaped data is claimed to be enabled by native graph storage (i.e., encoding data using graph data structures), and native graph processing (i.e., operating on data with graph-domain specific operations). A common belief is that native graph storage databases are inherently superior to non-native graph storage databases (e.g., relational databases) in terms of traversal efficiency. This claim is especially supported by graph database vendors, but not yet proven or disproven objectively.

In this work, we study in context of main-memory systems how the primitives of arbitrary traversal algorithms (i.e., single step traversal queries) are affected by native graph storage, and non-native graph storage in terms of execution performance. We focus on single step traversal queries that address navigation in graph-shaped data. We compare classic graph encoding and a state-of-the-art graph database micro-index as representatives of native graph storage, and table scanning and indexing by several binary search trees as representatives of non-native graph storage.

We evaluate the representatives for native and non-native graph storage on both artificial datasets, and real world graph datasets. To be aware of confounding variables, we implement a unified main-memory-only experimental query engine to avoid bias from internal behavior of some blackbox systems (e.g., main-memory systems vs. disk-based systems).

Our experimental results show that high efficient traversal algorithm in main-memory systems require indexing adjacent records and incident relationships rather than the property of being a native graph storage or a non-native graph storage.
Acknowledgements

I would like to thank my advisors Prof. Gunter Saake, Sebastian Dorok, and Dr. Sebastian Brefz for their encouraging supervision, the fruitful discussions, and their valuable comments on this thesis.

Finally, I thank my family, and especially my wife Christin for their patience and support during the process of writing this thesis.
Contents

List of Figures ix
List of Tables xi
List of Acronyms xiii

1 Introduction 1

2 Background 7
  2.1 Graph Database Models .......................... 7
  2.1.1 RDF Model .................................. 7
  2.1.2 Property Graph Model .......................... 10
  2.2 Graph Theory .................................. 14
     2.2.1 Fundamentals ................................ 14
     2.2.2 Pattern Matching ............................ 15
     2.2.3 Path Finding ................................ 16
  2.3 Native Graph Databases .......................... 17
  2.4 Summary ...................................... 19

3 Basic Pattern Matching Queries 21
  3.1 Single Step Traversals ........................... 21
  3.2 Physical RDF Graph Model ......................... 24
  3.3 Basic Pattern Matching ........................... 25
  3.4 Summary ...................................... 28

4 Executing Basic Pattern Matches 29
  4.1 Non-Native Graph Storage ......................... 30
     4.1.1 Scan-Matcher ................................ 30
     4.1.2 Red-Black-Matcher ........................... 36
  4.2 Native Graph Storage ............................ 41
     4.2.1 Native-Matcher ............................... 41
     4.2.2 Adjacency-Matcher ........................... 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Summary</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Implementation and Evaluation</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Implementation</td>
<td>49</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Griffin</td>
<td>50</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Internals</td>
<td>50</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Summary</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Micro-Benchmarking</td>
<td>52</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Environment</td>
<td>53</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Dataset Generation</td>
<td>53</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Scan Configuration Space</td>
<td>55</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Influence of Selectivity</td>
<td>56</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Scalability</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Case Study: Reachability Queries</td>
<td>60</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Measurement</td>
<td>61</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Datasets</td>
<td>62</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Dataset Impact</td>
<td>65</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Query Performance</td>
<td>69</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Storage Consumption</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>74</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
<td>77</td>
</tr>
<tr>
<td>6.1</td>
<td>Graph Traversals in Relational Engines</td>
<td>77</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison of Graph Databases, and - Models</td>
<td>78</td>
</tr>
<tr>
<td>6.3</td>
<td>Database Comparisons</td>
<td>79</td>
</tr>
<tr>
<td>6.4</td>
<td>Architecture, and Platform Comparison</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison on Different Datasets</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td>Wrap-Up</td>
<td>81</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary</td>
<td>81</td>
</tr>
<tr>
<td>7.2</td>
<td>Conclusion</td>
<td>82</td>
</tr>
<tr>
<td>7.3</td>
<td>Future Work</td>
<td>83</td>
</tr>
</tbody>
</table>

Bibliography | 85 |
List of Figures

1.1 Reachability performance: graph database vs. hand-written approach . 2
2.1 RDF graph containing information about Google . . . . . . . . . . . . 8
2.2 Property graph containing information about Google . . . . . . . . . . 11
3.1 Example citation graph data model . . . . . . . . . . . . . . . . . . . . . 23
3.2 Visualization of query plan for $q(u)$ . . . . . . . . . . . . . . . . . . . . 23
3.3 Different views on RDF-graphs . . . . . . . . . . . . . . . . . . . . . . . . 24
4.1 RED-BLACK-MATCHER indexing strategy overview . . . . . . . . . . . 38
4.2 NATIVE-MATCHER schema . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.1 Affect of selectivity value to execution time on a large-scale graph . . 57
5.2 Affect of increasing number of edges to execution time . . . . . . . . . . 59
5.3 Reachability query properties on real-world datasets . . . . . . . . . . 66
5.4 Reachability query execution times depending on hop count . . . . . . 70
5.5 Storage consumption on real-world datasets depending on hop count . 71
5.6 Logical query plan of relationship-centric queries . . . . . . . . . . . . 76
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Basic pattern matching: pattern vectors</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Support of basic patterns per matcher.</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Reachability Dataset Network Statistics</td>
<td>63</td>
</tr>
<tr>
<td>5.2</td>
<td>Converted Dataset Sizes</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Complexities, storage consumption, and index build times.</td>
<td>74</td>
</tr>
</tbody>
</table>
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td>EEN</td>
<td>Enron E-Mail Network</td>
</tr>
<tr>
<td>FB</td>
<td>Facebook</td>
</tr>
<tr>
<td>GOG</td>
<td>Google Web Graph</td>
</tr>
<tr>
<td>IRI</td>
<td>Internationalized Resource Identifier</td>
</tr>
<tr>
<td>Neo4j</td>
<td>Neo4j’s property graph database</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol And RDF Query Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
</tbody>
</table>
1. Introduction

Graph databases, as a class of non-relational databases, gained interest to both computer science, and industry. In contrast to the relational model based on Codd, graph databases expose a data model in terms of nodes and edges in schema-free multi-graphs [Cod70, RWE13]. Edges express explicit relationships between records at instance level rather than between groups of records as in the relational model. The graph data model has its advantages in applications where the data schema is not known in advance, or where the connection between single records are more important than the single records themselves, e.g., in applications regarding the web of data (Semantic Web and Linked Data) [AVH04, BHIBL08].

From a certain perspective, every database that manages graph-shaped data is a kind of a graph database [RWE13]: On the one hand, data can be stored using graph specific data structures (native graph storage), or it can be stored using tables as in relational databases (non-native graph storage). On the other hand, the data itself can be queried using graph specific operations, such as reachability between two records over \( k \) hops (native graph processing), or it can be queried using graph-domain independent operations such as joining tables using SQL (non-native graph processing). Consequentially, the most native graph database is a native graph storage-, and native graph processing graph database (or native graph database for short). The most non-native graph database is non-native for storage, and non-native for processing. Thus, a Relational Database Management System (RDBMS) that is lifted somehow to manage graph-shaped data, is a non-native graph database. The common belief is that native graph databases are better suited for graph data in general for both usability, and performance.
1. Introduction

Motivation

The database community already identified modern hardware as promising to greatly improve relational database system performances [ABH+13]. Further research considers co-processors to accelerate general-purpose query processing [BBR+13], or domain-specific data processing such as variant calling [DBS14]. Therefore it is logical to examine how modern hardware can improve graph database performances. It is not clear, whether native storage for efficient navigation in graph-shaped data is inherently superior to non-native storage in context of modern hardware.

The goal of this thesis is to determine whether native graph storage is better suited for efficient graph navigation in context of main-memory databases - independent of the concrete databases in use.

Navigation in graph data is the backbone of graph traversal operations. A graph traversal is moving through a graph-shaped dataset according a certain strategy. Traversals are a fundamental graph operation for an important class of queries such as reachability-, shortest path-, or pattern matching queries. Therefore, study graph database traversals is the key to understand whether native graph storage is necessary.

To study graph traversals, we have to study their primitives: navigation operations that are used to build single step traversals. A single step traversal is the ability to exploit adjacent (node neighborhood) and incident (node-/edge relationship) information of the graph data for exactly one step.
Problem Statement

There is no unified conclusion to the issue whether efficient navigation in graph-shaped data require native graph databases, but only hints depending on the workload, or the database systems under study. For instance, Paradise et al. showed RDBMS are well suited for reachability queries up to a certain hop count [PLB15], while Angles et al. reported unreasonable query times for a similar setting for the half hop count [APPDSL13]. In addition, Fan et al. showed relational databases are competitive to graph databases [FRP15], while Neumann et al. , and Vicknair et al. reported graph databases outperform RDBMS significantly [NW08, VMZ10].

During the determination, whether native storage or non-native storage in context of modern hardware should be favored, confounding variables must be taken into account carefully. For a better understanding, consider the following example. In Figure 1.1, we compare the execution times for reachability queries between random nodes for \( k = 1, 2, \ldots, 8 \) hops of Neo4j’s graph database, and a modern-hardware optimized scan-based strategy. The scan-based strategy outperforms the graph database counterpart in several orders of magnitude. Unfortunately, we compare apples with oranges, since we compare a general-purpose fully-fledged disk-based database (including effects of certain buffer managers) with a plain specialized main-memory hand-written operation. Is scanning competitive if the graph database would be in-memory, too?

With the effect of confounding variables in mind, the question *does efficient navigation in graph-shaped data require a native graph storage?* is hard to answer, but necessary.

We summarize reasons why current research only hints to a somehow ambiguous answer regarding performance comparison of graph databases and relational databases:

1. Different graph models, such as simple graphs or multi-graphs
2. Different graph encoding, such as triple tables or property tables
3. Different databases under study, such as Neo4j, RDF-3x or DEX
4. Different architectures and platforms, such as managed code and native code
5. Different evaluations, such as custom-build benchmarks, social network benchmarks, or RDF benchmarks

Goals and Contribution

To achieve our goal, we examine the performance of navigation-relevant single step traversals operations depending on native and non-native graph storage of the graph dataset. One key aspect is to avoid bias due to some hidden internal behavior of existing databases. Therefore, we execute our evaluation in a *whitebox* manner: we implement for
each category two representative strategies, and evaluate them in the same environment under the same conditions. Namely, we examine

- high efficient parallized scan operations (see Paradise et al. [PLB15]), and a tree-based index (see Neumann et al. [NW08]) as non-native storage approaches, and
- an object cache (see Robinson et al. [RWE13]), as well as an adjacency graph representation as native storage approaches.

We evaluate each of these strategies under equal conditions in a unified, native compiled, main-memory evaluation system, our custom-build graph database engine core called Griffin

Our solution is built upon the following:

1. **Common interface.** We identify how to manage and query graph-shaped data with focus on navigation, independent of the concrete type of database involved. For this, we identify a common interface for all approaches mentioned in order to run them in a unified environment.

2. **Main-Memory optimization.** We adapt and optimize the mentioned approaches into the same (main-memory) environment. This is challenging since it involves a high variability in terms of storage layout, and each of these approaches comes from different architectures.

3. **In-Depth Analysis.** We identify how each approach is stressed by the involved dataset, where each approach can, and where each approach cannot be applied for certain single step traversals. Further, we consider how each approach influences not only the performance of graph navigation but also affects the storage size and loading time.

4. **Direct comparison.** We examine whether efficient graph navigation is a consequence of native graph storage, or whether similar performance compared to native graph storage can be achieved using non-native graph storage in context of main-memory systems.

We achieve this by a formal perspective for the first goal, a qualified discussion about the concepts for the second goal, and an experimental study for the third and fourth goal.

---

1Source repository: https://github.com/graphworx/griffin-core
Our scientific contributions are:

- **Basic pattern match operations.** We examine single step traversals as proposed by Rodriguez et al. [RN11], and break them into more primitive operations we call *basic pattern matches* (**Chapter 3**).

- **Comparable native and non-native graph storage approaches.** We adapt and optimize existing native and non-native graph storage approaches into a main-memory context (**Chapter 4**).

- **Comprehensive comparison.** We evaluate each of these approaches in depth, and finally run a more complex query on real-world data sets. Namely, we execute reachability queries with *Breadth-First Search (BFS)*, and show how the search depth per approach affect the query performances (**Chapter 5**).

**Outline**

The structure of the remaining work is as follows:

- **Chapter 2.** We introduce background information for readers not familiar with graph databases.

- **Chapter 3.** We introduce our physical graph data model, on which we establish the concept of basic pattern matching. We distill a set of possible basic pattern matching queries as core primitives for navigation in graph-shaped data.

- **Chapter 4.** The forth chapter contains the concepts of native and non-native graph storage approaches. For each, we show the original concept, and distill relevant properties which characterize these approaches.

- **Chapter 5.** In chapter five, we evaluate the concepts of this thesis in terms of query performance on artificial and real-world datasets. This chapter ends with our interpretation of the results.

- **Chapter 6.** In the sixth chapter, we set our work into relation of exiting work.

- **Chapter 7.** The last chapter contains a summary of this thesis, and the final conclusion. It ends with our suggestion for further research.
2. Background

This chapter contains background information related to graph databases and graph-querying. We briefly introduce two important classes of graph data models, the Resource Description Framework (RDF) graph model (Section 2.1.1 on page 7), and the property graph model (Section 2.1.2 on page 10). Afterwards, we provide theoretical background regarding terminology, and important concepts such as path finding and graph traversals (Section 2.2 on page 14). At the end of this chapter we show how traversal queries are executed in current graph databases by example of Neo4j’s property graph database (Neo4j).

2.1 Graph Database Models

As the relational model based on the work of Edgar F. Codd in 1970 [Cod70] forms the fundamentals for RDBMS, graph databases rely on the mathematical theory around graphs, although several graph databases threat their data modeling and query slight different. More specific, two popular models based on graph theory are the RDF model, and the property graph model. The first can be viewed as some kind of subset to the latter. The mayor difference between both models is: in RDF everything of the data-set either a node or an edge. In contrast, the property graph model allow to semantically bundle certain meta data to both nodes, and edges in form of properties and labels.

2.1.1 RDF Model

The RDF models information in the web. RDF is tightly coupled with the initiative to to built the semantic web by semantic ontologies. It was standardized by the World Wide Web Consortium (W3C).
We introduce the terminology and model of RDF based on the specification by W3C\(^1\).

### 2.1.1.1 Model

The core idea behind W3C is the description of information by providing statements about resources in a graph-based manner without the need for a certain schema for entities. These statements are triples consisting of the components: the subject, the predicate and the object. Each triple \((\text{subject}, \text{predicate}, \text{object})\) is an RDF statement and defines a relationship provided by the predicate between two resources, the subject and the object. Hence, a W3C triple is equivalent to a graph containing two nodes and a directed (labeled) edge between them. A node is either a resource, or an internationalized form of the Uniform Resource Identifier (URI) concept, the Internationalized Resource Identifier (IRI). An IRI is used to identify something in the web uniquely. However, in terms of RDF, a resource can be an arbitrary thing, such as a person or a document. Typically, a resource has a certain data type such as a string or a number. Additionally, a predicate is a binary relation which exposes an information about the subject and the object. Therefore, a triple describes something (the subject) by a certain property (the predicate) and a certain value (the object). The set of RDF statements forms a directed labeled multi-graph, the RDF graph.

**Figure 2.1:** RDF graph containing information about Google

**Example.** Assume we want to express information about Google using RDF. We know that Larry Page and Sergey Brin developed the Google search engine. Google can be accessed by the web address `http://www.google.com` which is alias for the IP-address `216.58.213.36`. Google crawls websites in order to enable their web search. It’s likely Google crawls itself and therefore can finds itself when typing ”Google” in the search field. In RDF, we express these information using the following RDF-statements:

- \(("Larry Page", "developed", "Google Search")\),
- \(("Sergey Brin", "developed", "Google Search")\),
- \(("Google Search", "has address", "http://www.google.com")\) and
- \(("http://www.google.com", "is alias for", "216.58.213.36")\)

---

2.1. Graph Database Models

- ("Google Search", "crawls", "http://www.google.com")
- ("Google Search", "finds", "Google Search")

Ideally, we make use of IRIs in RDF statements which are drawn from a public available domain such that the RDF graph shares information with other RDF graphs. In this case, we might use "http://www.example.org/terms/developed" rather than "developed". However, we show in Figure 2.1 on page 8 a visualization of the corresponding RDF graph, in which we use rounded rectangles for nodes of resources and non-rounded rectangles for IRI nodes. One important aspect in the RDF philosophy besides sharing information with other RDF graphs, is the possibility of reasoning. Reasoning in RDF is a method to infer information based on the propositional calculus built in RDF. For instance, we can use the standardized RDF vocabulary and express that Larry Page is of type Person, and Google Search is of part of a web service. By interconnection to other RDF graphs we might infer that Larry Page is a human being while Google Search is a technology - although these facts are not contained in our knowledge base.

The RDF model strongly focus on a computational perspective, such that reasoning or other techniques are uniformly possible. However, the RDF model is intentionally designed very strictly. In contrast to other models, such as the property graph model, edges in a RDF graph are not allowed to contain any value. For instance, if one wants to express the costs cost(a,b) for traveling between two cities a and b, the cost value cannot be added to the predicate indicating a route between a and b directly. For this purpose, further techniques such as RDF reification or blank nodes are necessary. Both techniques are used to express statements about statements but typically bloat the RDF graph or complicate the model. Solutions to this issues are proposed by the research community recently, such as RDF+ which allows further meta information on statements [NBS14].

2.1.1.2 Query Language

Querying data in a RDF graph is typically performed with the SPARQL query language. SPARQL Protocol And RDF Query Language (SPARQL) is a declarative Structured Query Language (SQL)-inspired query language which was standardized by the W3C². Among other functionality such as data manipulation, SPARQL allows querying RDF data in terms of Pattern which can be connected for instance using a conjunction of several pattern.

For the purpose of this section, we introduce a fundamental form of a simple query formulated in SPARQL:

Similar to SQL, a returned SPARQL resultset can be projected to a set of variables. Variables are declared using a dedicated character, such as ? or $, and are bound in the pattern. The where clause contains a pattern matching definition for instance over basic graph pattern. Optionally, a pattern matching definition can contain a list of patterns delimited by the "." character. A list of patterns is a conjunction of patterns, i.e., the binding of a variables must satisfy the constraints given in all pattern. A possible evaluation of these conjunctions might initially evaluate each single pattern and then join the results using an appropriated algorithm. However, for a more in-depth introduction to the concepts and grammar of SPARQL, we refer the book of Curé and Blin [CB14].

Typical graph traversal functionality is not the focus of the RDF design. Therefore, traversal algorithm including path finding are not part in the official SPARQL definition standard at the moment of writing. But first progress is made here, such as the database community suggested an extension of SPARQL along with a built-in support in the underlying engine [GBS13].

**Example.** Assume we want to query the RDF graph mentioned in the previous section by the following: "What is developed by Larry Page which can be found using Google Search?". We can represent this using the following SPARQL query:

```
select $x where
{
  "Larry Page" "developed" $x .
  "Google Search" "finds" $x
}
```

As we expect, the resultset contain Google Search itself. However, if our RDF graph would contain more information it will be likely to get interesting results. One possibility is to take the advantages of connection to other (public) RDF graphs which is, as already mentioned, one of the key features behind RDF.

### 2.1.2 Property Graph Model

Property graphs allow data modeling with the power of directed, labeled, attributed multi-graph which allows flexible modeling of a wide range of applications [RN10]. One of the most known systems implementing the property graphs model is Neo4j which is open-source, and free for non-commercial use.

We introduce the terminology and model of property graphs based on the book of Robinson et al., and the paper of Rodriguez et al. [RWE13, RN11].
2.1. Graph Database Models

2.1.2.1 Model

The basic idea behind the property graph is modeling a domain as (labeled) nodes and relationships between nodes with least restrictions as possible. Both nodes and relationships can contain user-defined properties (i.e., often in terms of key-value pairs). However, in contrast to RDF in which statements about resources are given, the property graph is a more generalized description. Roughly speaking, both nodes and relationships instances are unique objects inside the graph and become meaning by their properties and labels. Properties are arbitrary mappings between a user defined key (i.e., a string literal) and a certain value. The type of values vary, but most implementations provide primitive values such as integers or strings, and more complex types such as certain collections. Modeling with property graphs often target practitioners rather than the computational aspect of the graph as in the RDF counterpart.

![Property graph containing information about Google](image)

**Figure 2.2:** Property graph containing information about Google

**Example.** Assume we want to express the same information about Google as we did in Section 2.1.1.1 on page 8. For the purpose of this example, we use the language Cypher which allows data definition and querying. In Cypher adding data to a graph is indicated by the keyword CREATE followed either by a node definition, or a relationship definition. Multiple data definitions in one statement do not require multiple CREATE keywords.

The following simplified form can be used to create a node:

```
create (Identifier { Property (, Property)* }) , ...
```

This statement creates a new node in the graph which has certain properties attached. Each property is expressed by a term Property which contains a colon delimited key-value pair of the form key: value. A nodes (optional) Identifier is an intermediate name to link to this node in the context of further statements. The identifier is not imported into the graph and is syntactic sugar for ease of use.

The following simplified form can be used to create a relationship between two nodes:

```
create ( Left ) Direction [:Name { Property (, Property)* }] Direction ( Right ), ...
```

This statement creates an relationship between the node on the Left side and on the Right side of this statement. Both nodes may refer to an Identifier of a node created
earlier. The direction of the new relationship is indicated by the left and right Direction such that the relationship hold from left to right (i.e., ()−[]−()->()), from right to left (i.e., ()<-[[]]−()->()), or in both directions (i.e., ()−[[]]−()->()). The Name determines the actual relationship name which will be stored along with its properties in the database.

Using the Cypher language, we can now create the example graph:

```cypher
create
(larry { firstname: 'Larry', lastname: 'Page'}),
(sergey { firstname: 'Sergey', lastname: 'Brin'}),
(google { name: 'Google Search'}),
(site { url: 'http://www.google.com', alias_for: '216.58.213.36'}),
(larry) -[:developed]->(google),
(sergey)-[:developed]->(google),
(google)-[:has_address]->(site),
(google)-[:crawls { timestamp: 1458139590}]->(site),
(google)-[:finds]->(google)
```

We show the resulting graph in Figure 2.2 on page 11. Since the property graph model is less restricted compared to the RDF model, we can easily define semantic-coupled entities, e.g., we can attach a first name and a last name attribute to the nodes larry and sergey. While this is a logical unit in the property graph model, in RDF this would require more (independent) statements. Furthermore, while the entire graph in RDF is expressed in form of RDF statements, nodes in property graphs might be isolated which is not possible in RDF since a statement about a resource connects always two nodes. However, we slightly changed our example by adding an additional information, a timestamp, to the crawls relationship. We did this to demonstrate the beneficial use of properties for edges, which might be hard to reason about but elegant for data modeling purposes.

### 2.1.2.2 Query Language

Querying data in a property graph depends on the vendor-specific implementation since there is no standardization at this point in time. Typically, query formulation can be done using a binding to a specific programming language, a specialized scripting language, a web service oriented Representational State Transfer (REST)-interface, an imperative or declarative Application Programming Interface (API) or a query language. For instance, the SAP HANA database supports a declarative graph traversal and manipulation language called Wipe [FCP+12, RPBL13], or Paradise et al. who suggest an extensible graph traversal framework for RDBMSs [PLB15]. However, for the purpose of this section, we will focus on Cypher, a popular declarative query language for Neo4j [RWE13].

Query in Cypher can be formulated using the following simplified form:
A Pattern can contain a single node, two nodes with a directed (or undirected) relationship between them, labels and constraints on certain properties for both nodes and relationships. A basic pattern is (a)-[b]->(c) where a and c are (constant or variable) node conditions, and b is a (constant or variable) relationship condition. Moreover, the relationship can be right directed, or left directed or both. The latter one explicitly ignores the direction in the sub-graph to be matched, which is useful if the direction does not mind. Optional, Cypher supports defining a start for the traversal manually: a Cypher query can begin with the keyword start followed by a definition on a node at which the traversal has to begin.

In contrast to SPARQL queries, graph traversal is explicitly a part of the query statement in Cypher. The relationship b can be disjunction of a certain set of possible edged to be followed over a user-defined range in terms of the path length. For instance the b might be of the form :r*2..7 | :q* which means the condition holds if c is reachable starting from a over the range of two to seven hops using the relationship r, or over an arbitrary hop count using the relationship q. The WhereCondition in the where clause provides further filtering on the potential sub-graphs found for the given pattern over additional variable binding. The ReturnStatement statement states what should be added to the resultset, for instance nodes, relationships or certain properties. For a more detailed overview on Cypher, we refer to the official manual which is available online.

Example. Assume we want to query the property graph mentioned in the previous section by the same query as in Section 2.1.1.2 on page 9 - "What is developed by Larry Page which can be found using Google Search?". We can represent this using the following Cypher query:

```cypher
match (person)-[developed]->(product)<-[finds]-({name = 'Google Search'})
where person.firstname = 'Larry' and person.lastname = 'Page'
return product
```

As for the RDF counterpart, the result is again the node associated to Google Search. However, by defining an infinite depth of relationships over arbitrary path ignoring the relationship direction, we could also express the query How are Larry Page and Sergey Brin related?:

```cypher
match (a)-[relationship*]-(b)
where a.firstname = 'Larry' and a.lastname = 'Page'
```

---

b.firstname = 'Sergey' and b.lastname = 'Brin'

return relationship

The decision variant of the last query will be also expressable using our prototype system Griffin.

2. Background

2.2 Graph Theory

The graph theory is a mathematical discipline focusing on the study of graphs. A graph is the most generic data structure capable to describe arbitrary models in terms of nodes (or vertices) and edges between nodes to express structures, or relationships. Modeling real world problems using graph theory is applied in a wide range of research fields today. In geography graphs can be used to aggregate cities based on their flows of goods, or communicating people [ND61], clinical neurophysiology uses graph theory to analyze complex structures of neural networks [RPBS07], or in sociology graphs are used to express actors and interactions between actors for social network analysis [WF94] to name a few.

The simplest way to define a graph is by the means of an undirected one. We use definitions and terminology similar to them provided by Aigner [Aig84] for this purpose in the following.

2.2.1 Fundamentals

**Definition (Graph).** An graph $G$ is a pair $G = (V, E)$ containing a finite set $V$ of nodes, and a finite set $E$ of edges between some nodes in $V$. A graph is undirected, if $E$ contains sets of nodes in $V$, and is directed if $E$ contains ordered pairs of nodes in $V$.

Each edge $\{u, v\} \in E$ has two ending nodes $u, v \in V$ which are connected by the edge. If a node $v$ belongs to an edge $e$ as one of it’s ending nodes, in symbols $v \in e$. If $u = v$ holds, we call the edge $\{u, v\}$ a circle. Are two nodes $u$ and $v$ connected by more than one edge, the graph is called a multi-graph containing multiple edges. A simple graph does not contain multiple edges or circles. Two nodes in $V$ are neighbors (adjacent to each other) if they are connected by an edge. We use $\mathcal{N}(v)$ as the set of all neighbors for a given node $v \in V$, and define the degree $\deg(v) = |\mathcal{N}(v)|$. A node $v \in V$ is incident to an edge $e \in E$ if $v$ is an ending node of $e$. A node which is not incident to any edge, is called isolated. Starting with a not isolated node $u$, a node $v$ is given by traversing over an edge $\{u, v\}$ if $v \in \mathcal{N}(u)$. A path of length $\ell$ is a $\ell$-tuple $p = (v_1, v_2, ..., v_\ell)$ of nodes $v_1, v_2, ..., v_\ell$ with edges $\{v_{i-1}, v_i\} \in E$ where all nodes in $p$ are pairwise different (for $i \in \{1, 2, ..., \ell\}$). A path $p$ from $v_1$ to $v_\ell$ can be traversed, yielding $v_i$ at the $i$-th step of traversing.
Many interesting problems can be solved using (simple) graphs, such as the graph coloring problem. Although some literature in the field of graph databases rely on simple graphs or graphs without multiple edges between nodes, we have to mentioned that this assumption does not hold in the context of real-world graph databases, since both RDF and property graph databases explicitly address multi graphs (see [RN11] for property graphs). However, as it turns out, the notation of a directed multi-graph must be enriched to labeled graphs to satisfy the requirements to describe the model of RDF triple stores. Additionally, we have to introduce the concept properties (attributes) on top of the definition of RDF w.r.t. the property graph model as introduced in Section 2.1.2 on page 10.

Modeling a problem domain using the notation of graphs allows the application of a wide range well-studied graph applications, such as the determination of the minimum spanning tree, or the transitive closure, centrality measurements, or algorithm to detected connected and isolated sub graphs. In this thesis, we address two types of graph database queries which we will introduce in a formal way in the following, namely a core operation in pattern matching queries, and a reachability query as decision variant of the path finding problem.

### 2.2.2 Pattern Matching

To introduce the pattern matching problem, we use the definition similar to these provided by Gubichev et al. [GT14]: Given a graph and and another (query) graph, a pattern matching problem is to determine sub-graphs in the first graph which matches the second graph. More formally, the inputs of a pattern matching algorithm are a directed (labeled, property) multi-graph $G$, and a pattern which is described by another graph $H$. The graph $H$ contains constraints to edges and nodes in several terms. For instance, $H$ can define a certain structure over nodes and edges, or it can allowed or disallowed certain values on labels and properties for both edges and nodes. The pattern matching problem is to determine all sub-graphs $G_1, G_2, \ldots \subset G$ which are isomorphic in terms of the structure provided by $H$, and which nodes and edges satisfy the conditions given by $H$.

Evaluating a pattern matching query is computational expensive in general, since subgraph isomorphism is already known to be $\mathcal{NP}$-complete. For certain cases and assumptions however, algorithm to determine sub-graphs might achieve acceptable, or even good execution times [KST93, Ull76]. In context of graph databases, algorithms using back-tracking strategies (e.g., in Neo4j) and algorithms using multiple selections and joins of sub-patterns can be found (e.g., in RDF-3x).

Graph traversing can be used as a starting point for a pattern matching query evaluation [RWE13]. We can define simple patterns (or basic graph pattern over triples in terms
of RDF\textsuperscript{4}) which can be efficiently evaluated using a single operation. These simple patterns are defined by a graph $H$ which contains two nodes $u$ and $v$ and an edge $e$ between both. Using certain constraints on $u$ and $e$, queries such as ”What are the incident nodes $v_1, v_2, ...$ to $e$ starting by $u$?” can be formulated. More precisely, these simple patterns allow to query

- the neighborhood $\mathcal{N}(x)$ for $x \in \{u, v\}$ including or excluding certain edges,
- the set of edges between $u$ and $v$,
- nodes incident to a certain edge, and
- exact matches on $u, v$ and $e$.

Evaluating simple patterns using single operations explicitly take advantages of the graph-shaped data and can be computed in a efficient manner, which takes more effort when using the relational model for the same task [RN11].

To avoid confusions in the terminology: We introduce the concept of basic pattern matching in Chapter 3 which can be a basis for RDF’s pattern matching of simple graph patterns. Basic pattern matching addresses adjacency and incidency information, while pattern matching can involve selection operation based on node and edge values, and more.

2.2.3 Path Finding

Given a graph and two nodes, the path finding problem is to determine one path between both nodes if any. More formally, the input of a path finding algorithm are a graph $G = (V, E)$, a start node $u_{\text{start}}$ and an end node $u_{\text{end}}$ in $V$. The path finding problem is to determine a path $p = (u_{\text{start}}, v_1, v_2, ..., u_{\text{end}})$ using an algorithm $A$. Depending on the requirements, $A$ might return the set $P(u_{\text{start}}, u_{\text{end}})$ of all paths between $u_{\text{start}}$ and $u_{\text{end}}$, or a certain subset of $P(u_{\text{start}}, u_{\text{end}})$ satisfying some requirements. These requirements might be the shortest path rather than an arbitrary one which can be solved using Dijkstra’s algorithm, or some kind of back-tracking strategy for instance.

The general idea to solve a path finding problem is to traverse the graph starting from $u_{\text{start}}$ until a result is generated. A straightforward algorithm is BFS with a typical complexity of $\mathcal{O}(|V| + |E|)$ with $|E| \leq |V|^2$, if not tuned further. The BFS algorithm explores the neighborhood $\mathcal{N}(u^*)$ first looking for $u_{\text{end}}$, where $u^*$ is initialized with $u_{\text{start}}$. If $u_{\text{end}} \in \mathcal{N}(u^*)$ BFS stops. Otherwise the algorithm will continue this process with each not explored node in $\mathcal{N}(u^*)$ recursively until $u_{\text{end}}$ is found, or the entire graph was traversed.

\textsuperscript{4}SPARQL Query Language for RDF, W3C Recommendation 15 January 2008, https://www.w3.org/TR/rdf-sparql-query/
The decision variant of the path finding problem is called the *reachability* problem and ask, if \( P(u_{\text{start}}, u_{\text{end}}) \neq \emptyset \).

In context of graph databases, BFS is applied for instance in the *Traversal API* of the *Neo4j* to express traversal queries in a declarative manner [RWE13]. However, our prototype *Griffin* supports reachability queries using the BFS algorithm in order to answer the question if two given nodes are connected over some path.

### 2.3 Native Graph Databases

Some graph database vendors strictly differ between *native* and *non-native* graph databases in both, storage and processing. We summarize the requirements provided in [RN11, RWE13, RN11]: a graph database is a native graph database when the following requirements are satisfied:

1. **R\(_1\)**. The database utilize the inherently connections between nodes, and relationships to boost-up query performance, and
2. **R\(_2\)**. The database has the capability to express relationships between records of arbitrary types "on the fly", and
3. **R\(_3\)**. The database is able to query related records given a certain record with a complexity which is not affected by the number of relationships (or records) in total.

One can argue that RDBMS do not satisfy these requirements: scanning a collection of edges in order to compute the neighborhood \( \mathcal{N}(u) \) of a given node \( u \), is *not* a native graph processing approach since scanning does not consider neighborhood information which are present in the data (R\(_1\) is violated). Moreover, scanning an edge table given a fixed node \( u \) depends on the total number of edges stored rather than the number of edges \( u \) is related to (R\(_3\) is violated). The same argumentation holds for organizing the relationship management with an index structure similar to a \( B^+ \)-tree. As it turns out, the object cache is similar to a hash-index for an m:n relationship tables in the RDBMS counterpart. However, by definition, the relational model violate R\(_2\).
Disk-based native graph storage (by example of Neo4j). Neo4j is a disk-oriented database. The graph is decomposed into several optimized binary files in the file system, which are frequently partially loaded into main memory. Each of these files has a certain responsibility, such as the containment of nodes, properties and the relationships. Neo4j organizes a graph edge by a record containing the following information:

- **In-use flag**: A flag which indicates whenever a record of the file is used, or can be overwritten.

- **In-coming and out-coming node**: The node identifiers defines the starting and ending node of the edge. Node-specific properties are not stored in the record but in a dedicated file.

- **Relationship type**: An identifier of a relationship type which provides the type of relationship assigned to this edge.

- **Relation chain pointers**: A set of four pointers which enable the native graph storage per record, i.e., each record contains four link fields that refer to related records. These fields link to record indices relative to the current record in two directions: the *previous* record index with the same starting (ending) node, and the *next* record index with the same starting (ending) node.

- **Edge property pointer**: Any edge in a property graph database can be enriched by custom properties. Each edge-record contains a pointer to its first attached property. The objects representing properties are organized in a dedicated file.

To minimize the costs of disk read operation, the database caches records loaded from these files into the "file system cache", and the "object cache" (see Section 4.2.1). Neo4j queries its data internally in an iterator-like fashion fetching data from the object cache. In case of a cache miss, the object cache has to read the requested data from the edge-file. For this purpose, the first corresponding record in the edge-file is determined. Once the file pointer is seeked to its position, the relation chain enables fast seeking to related records depending on the iterator orientation (start-, or end node) and direction (next, or previous edge), since the offset can be calculated efficiently due to the fixed-size structure of the records.
2.4 Summary

In this chapter we provide background information related to the processing of graph-shaped data in graph databases.

We showed two prominent classes of graph databases, the RDF model and the property graph model. The RDF model is associated with the semantic web initiative, and treats the entire dataset as graph of nodes, and edges. In contrast, the property graph model enriches both nodes and edges with user-defined attributes (properties). These attributes are key-value pairs, and allow further semantic per record. For both database we provided a brief overview how data can be queried using the SPARQL resp. the Cypher query language.

We provided an overview on the theoretical background of graph databases in terms of the general graph theory. We introduced the concepts of several graph types, most important labeled directed multi-graphs. In contrast to simple graphs, a multi-graph allows multiple edges between nodes. We further introduced the concepts of adjacent nodes, and nodes incident to edges. In addition, we outlined important problems classes, namely pattern matching and path finding. We concluded, that these (and other problems) based on graph traversals from a computational perspective. We introduced the concept of native graph databases by the example of Neo4j. A native graph database is characterized by its ability to exploit the graph structure of the data, i.e., by accessing related (linked) data directly and independent of the graph size in general.
3. Basic Pattern Matching Queries

This chapter deals with the concept of graph traversals in terms of single step traversals, and how these single step traversals can be reduced further to a common interface, that we call basic pattern matches. As a consequence, we can focus on efficient evaluations of the basic pattern matching queries mentioned to achieve efficient graph traversals in general - independent of a concrete database type.

We start with a summary of the work by Rodriguez et al. who provide a set of single step traversal functions in the context of property graph databases (Section 3.1). Afterwards, we provide a formal but yet physical graph model that is actually RDF-like (Section 3.2). We use this physical model to introduce our concept of basic pattern matching (Section 3.3). Finally, we express navigation-related single step traversals by basic pattern matching queries through the usage of some helper functions, namely projection and union operation (Section 3.3).

3.1 Single Step Traversals

In the relational model, entity classes of a domain $D$ are separated during domain modeling at a first step. Hence, the domain $D$ is decomposed into a set of (entity) tables $R_1, R_2, ..., R_n$, and (relationship) tables $S_1, S_2, ..., S_m$. The latter are required to express many-to-many (m:n) relationships in the domain $D$: each $S_k$ contains foreign-key references to the key of related records in the involved tables $R_{i_1}, R_{i_2}, ...$ (for $k \in \{1, 2, ..., m\}$, $i_1, i_2, ... \in \{1, 2, ...n\}$). The combination of related data is typically done by joining, i.e., to receive data of $S_k$-related records $r_1 \in R_i$ and $r_2 \in R_j$, both tables $R_i$ and $R_j$ are joined with each other over $S_k$ (for $i \neq j$, $k \in \{1, 2, ..., m\}$, $i, j \in \{1, 2, ..., n\}$).

In contrast, modeling the domain $D$ with a graph model typically does not require the act of domain decomposition. Typically, the relationships between objects in $D$
are explicitly expressed rather than described over "artificial" m:n tables. The graph model makes it easier to enable graph traversals due to their flexible data model and their focus on relationships between objects. Traversal is commonly known as the act of navigation through the dataset (the graph) according to some strategy.

Traversal operations in the relational model typically require a sequence of selection- and join-operations to navigate from one record to another. In contrast, the objects $o_1$ and $o_2$ of $D$ are modeled as nodes $u_{o_1}$ and $u_{o_2}$ without the need of a further assignment to some broader class in a graph model (i.e., none of these nodes belong to some kind of "table"). The relationship $e$ between $o_1$ and $o_2$ forms a directed edge between the node representing $o_1$, and the node representing $o_2$. Performing a traversal is more natural then, since jumping from $o_1$ to $o_2$ only requires to determine the neighborhood $\mathcal{N}(u_{o_1})$ in order to fetch $u_{o_2}$. In addition, traversing over an edge $\{u_{o_1}, u_{o_2}\}$ is a projection to either $u_{o_1}$ or $u_{o_2}$ (from a logical perspective).

### 3.1.0.1 Functions

In *The Graph Traversal Pattern*, Rodriguez et al. introduced a set of function signatures\(^1\) along with an description of their functionality in context of property graph databases[RN11]:

1. $e_{\text{out}}$: takes a bag of nodes and returns a bag of edges, such that the out-going edges of these nodes are returned.
2. $e_{\text{in}}$: takes a bag of nodes and returns a bag of edges, such that the in-coming edges of these nodes are returned.
3. $v_{\text{out}}$: takes a bag of edges and returns a bag of nodes, such that the out-going (ending) nodes of these edges are returned.
4. $v_{\text{in}}$: take a bag of edges and returns a bag of nodes, such that the in-coming (starting) nodes of these edges are returned.
5. $\varepsilon$: takes a bag of nodes and edges along with a property key, and returns a set of values, such that the values associated to the key for these nodes and edges are returned.
6. $e_{\text{lab} \pm}$: takes a bag of edges and a label, and include (+) or exclude (−) edges with the given label for subsequent functions.
7. $\varepsilon_{\text{p} \pm}$: takes a bag of edges and nodes, a property key, and a value and include (+) or exclude (−) edges and nodes with the given key-value pair for subsequent functions.

---

\(^1\)Please note, that we use the term *bag* for a multi-set, i.e., the elements in this structure are allowed to occur multiple times rather than only once as in regular sets. Moreover we restrict our description here to a informal one for simplicity reasons.
8. $\varepsilon_{\pm}$: takes two bags of node and edges, and return a bag of nodes and edges, such that the provided element is included (+) or excluded (−).

By the use of these functions, graph traversals can be easily expressed by function compositions, and argument rewriting.

For a better understanding of the application of these functions, consider the following example in which we provide a typical traversal query.

### 3.1.0.2 Example

Assume a scientific citation graph, in which authors cite papers of other authors in their paper\(^2\). We show the corresponding data model in Figure 3.1. Consider the following query

$q(u): Which authors cited the author u?$

Using the single step traversal functions and function composition, the query $q(u)$ can be formulated as follows:

$$q(u) = (v_{in} \circ e_{isAuthorOf} \circ e_{in} \circ e_{in} \circ e_{cited} \circ e_{in} \circ v_{out} \circ e_{isAuthorOf} \circ e_{out})(u)$$

We show in Figure 3.2 a visualization of the function $q(u)$\(^3\). The first traversal step is to get the bag of all out-going edges of $u$ (i.e., $e_{out}$ is applied). If the data schema contains more than the one type "isAuthorOf" as out-going relationships for authors, we have to exclude all others, i.e., we include only "isAuthorOf" (i.e., $e_{isAuthorOf}$ is applied). This results in a bag containing all "isAuthorOf" out-going edges of $u$. From this, the ending

---

\(^2\)A real-world data set can be found under [http://blog.related-work.net/data/](http://blog.related-work.net/data/)

\(^3\)We consider the evaluation order for function composition, i.e., $(f \circ g)(x) = f(g(x))$. Thus, we show in the figure an application of $q$ on $u$ from right to left in terms of $q$’s inner functions.
nodes are selected (i.e., $v_{out}$ is applied). This might result in a set of papers written by $u$. However, we fix $p_u$ in the example as one paper of $u$ for didactic reasons. The next step is to determine all in-coming edges of the paper $p_u$ (i.e., $e_{in}$ is applied) which in turn are restricted to the type “cited” (i.e., $e_{cited}$ is applied). The result is a bag of papers that cite $u$’s paper $p_u$ (i.e., $v_{in}$ is applied). For didactic reasons, we fix $p_v$ as one of these papers citing the work of $u$. From $p_v$ all in-coming edges are retrieved (i.e., $e_{in}$ is applied) which are restricted to the type “isAuthorOf” (i.e., $e_{isAuthorOf}$ is applied). From this bag of edges, the in-coming nodes are retrieved (i.e., $v_{in}$ is applied) which forms the final result, i.e., all nodes representing authors that are author of $p_v$. Please note here, that there is no (explicit) join involved in this query.

In the following section we are interested in how the set of single step traversal function can actually be evaluated. Therefore we introduce a graph model on which we introduce our basic pattern matching operation.

### 3.2 Physical RDF Graph Model

In the following section we introduce a formal graph model that focus on the relationship management of graph-shaped data. Our model contains certain physical aspects which will allow us to formulate algorithm which rely on e.g., the underlying storage layout.

As normally, graphs are composed of edges and nodes, i.e., a graph consists of three parts: a set of nodes (node-store), a set of relationships (relationship-store), and an indexed set of edges (edge-store). In Figure 3.3 we show how these concepts belong together in the following. Nodes and relationships are primarily numeric values referencing to concrete values, e.g., by a dictionary encoding or as reference to more complex object. Thus, the entire graph is expressed in terms of identifiers (see Figure 3.3 part (a)). This indirection allows us to formulate algorithm on edges, independent of the concrete context of nodes and relationships, i.e., nodes and relationships might be single

**Figure 3.3:** Different views on RDF-graphs containing an edge-store, and dictionaries for nodes-, and relationship values.
values (e.g., in a RDF model), contain properties (e.g., in the property graph model), or refer to tuples in certain tables (i.e., in a relational engine). In context of our thesis, we implement a RDF graph using binary search trees for its values (see Figure 3.3 part (c)). However, each edge is built up of three components, the identifier of a node, the identifier of a relationship, and the identifier of another node. Hence, we interpret a single edge as an right-directed edge from the first to the second node with the given relationship. The entire graph is encoded in a logical table containing triples of a start node, a relationship, and an end node (see Figure 3.3 part (b)).

Notation. An edge connects two nodes via a relationship. In context of the property graph model, both nodes and relationships might be enriched with further attributes in addition (or as replacement of) the corresponding value. However, an edge is a triple $e = (e_1, e_2, e_3)$ where $e_1$ and $e_3$ are node identifiers, and $e_2$ is a relationship identifier. We interpret the triple $(e_1, e_2, e_3)$ as a right-directed edge starting from $e_1$ (the start node), ending by $e_3$ (the end node) using the relationship $e_2$.

3.3 Basic Pattern Matching

The following section provides a definition for querying data in our graph model. Therefore, we introduced the concept of basic pattern matching. Basic pattern matching is a trinary-equi selection $(\sigma_{\text{start}}, \sigma_{\text{rel}}, \sigma_{\text{end}})$ operation (i.e., equality conditions per-node component) to restrict the edges in the edge-store.

Pattern vector. We begin with the definition of our concept of a pattern vector. A pattern vector $p = (p_1, p_2, p_3)$ is a triple defining the shape of edges to be matched for basic pattern matching queries. More precisely, $p$ has the same semantics as an edge: it defines a right-directed edge starting from the node $p_1$, and ending at the node $p_3$ via exactly one relationship $p_2$. In contrast to the definition of edges, a pattern vector is not required to be drawn from the domains of node identifier and relationship identifier, i.e., the components in a pattern vector can refer to node- and relationship identifier which actually do not exist in the graph. In addition, a component $p_i$ in $p$ can be set to the special symbol * which indicates that there are no constraints on the value $e_i \in e$ for an edge $e$.

Basic pattern match. For a graph $G$, and a pattern vector $p$, a basic pattern match is a function which takes the edge-store of $G$, and a pattern vector $p$ as input. The function returns a subset of edge indices in the edge-store such that the corresponding edges satisfy the conditions given by the pattern vector, i.e., the edges match the pattern. More specific, the function checks for each edge if the $j$-th component in the edge is equal to the $j$-th component in the pattern vector $p$ (for $j \in \{1, 2, 3\}$). Thus, a basic pattern match is a trinary equi-selection

$$(\sigma_{\text{start}}=\alpha, \sigma_{\text{rel}}=\beta, \sigma_{\text{end}}=\gamma) = (\alpha, \beta, \gamma)$$
Table 3.1: Basic pattern matching: Interpretation and names of pattern vectors. The symbol "*" indicates a wildcard, i.e., no condition to the column.

<table>
<thead>
<tr>
<th>Name</th>
<th>Pattern</th>
<th>Interpretation (filter condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1: Exact match</td>
<td>(α, β, γ)</td>
<td>exact match of edge (α, β, γ)</td>
</tr>
<tr>
<td>q2: Nodes out</td>
<td>(α, β, *)</td>
<td>α is start node with out-going relationship β</td>
</tr>
<tr>
<td>q3: Connected by</td>
<td>(α, *, γ)</td>
<td>α is start node, γ is end node</td>
</tr>
<tr>
<td>q4: Nodes in</td>
<td>(*, β, γ)</td>
<td>the node γ has an in-coming relationship β</td>
</tr>
<tr>
<td>q5: Edges in</td>
<td>(*, *, γ)</td>
<td>the node γ has in-coming relationships</td>
</tr>
<tr>
<td>q6: Edges out</td>
<td>(α, *, *)</td>
<td>the node α has out-going relationships</td>
</tr>
<tr>
<td>q7: Connects</td>
<td>(*, β, *)</td>
<td>right-directed relationship β connect nodes</td>
</tr>
<tr>
<td>q8: Full scan</td>
<td>(*, *, *)</td>
<td>all edges</td>
</tr>
</tbody>
</table>

where some of the selection σ are allowed to be always true, and which returns positions of satisfying records rather than the records itself. Thus, a basic pattern match is equivalent to a filter operation

$$σ_{start=α∧rel=β∧start=γ}(R)$$

on a logical table R with three columns where some expression can be left out (e.g., $σ_{start=α}(R)$ only).

We can express several restriction to the graph data depending on the pattern vector definition. Each pattern vector returns edge indices matching the conditions on their values. In Table 3.1 we provide an overview of possible pattern vector definitions including a verbal interpretation, and a naming.

To bridge basic pattern matches into single step traversal functions, we further provide a projection, and composition function.

**Projection.** Given a graph G, and a subset I of the set of edge indices in the edge-store. A projection π is a function which takes an edge-store, a set of edge indices I, a number $j \in \{1, 2, 3\}$, and returns a set of node identifiers or relationship identifiers by projecting the edge-store to the $j$-th column considering only elements as given by I. For short, we say G is $j$-projected w.r.t. I. If $j = 1$ the edges are projected to their starting node identifier, if $j = 2$ the edges are projected to their relationship identifier, and they are projected to their ending node identifier otherwise.

**Composition.** A composition τ is a function that takes an edge-store, a pattern vector p, and a number $j \in \{1, 2, 3\}$, and returns an $j$-projected evaluation of p. It first evaluates p on the edge-store, and returns a set I of edge indices satisfying p. Then, it invokes a projection π to the column $j$. For short we write $τ^j(p)$ when the edge-store is clear in the context. To increase the understandability of $j$, we write start for $j = 1$, rel for $j = 2$, and end for $j = 3$. 
3.3. Basic Pattern Matching

Single Step Traversals through Basic Patterns

In the following, we provide an alternative definition to navigation-related single step traversal function as mentioned in Section 3.1 by the use of $\tau$, and hence in terms of relational operations:\(^4\)

1. $e_{\text{out}}: 2^{D_n} \rightarrow 2^{D_e}$ with $e_{\text{out}}(X) = \bigcup_{u_i \in X} (\tau_{\text{rel}}(u_i, *, *))$
2. $e_{\text{in}}: 2^{D_n} \rightarrow 2^{D_e}$ with $e_{\text{in}}(X) = \bigcup_{u_i \in X} (\tau_{\text{rel}}(*, *, u_i))$
3. $v_{\text{out}}: 2^{D_e} \rightarrow 2^{D_n}$ with $v_{\text{out}}(X) = \bigcup_{u_i \in X} (\tau_{\text{end}}(*, u_i, *))$
4. $v_{\text{in}}: 2^{D_e} \rightarrow 2^{D_n}$ with $v_{\text{in}}(X) = \bigcup_{u_i \in X} (\tau_{\text{start}}(*, u_i, *))$

Please note that we cannot map the other single step traversal function directly to $\tau$ since our model does not support properties. In addition, the model proposed by Rodriguez supports multiple values for edges and nodes ("labels") which we have to simulate with artificial edges to values representing these labels. However, despite these facts, the navigation-related functionality of single step traversals are mapped to $\tau$, while the other functions do more relate to the meta data of the data inside the graph.

As a consequence it is sufficient to provide efficient strategies for $\rho$ (selections) to express efficient navigation single step traversal function, and thus for traversals of arbitrary length in general.

---

\(^4\)Please note that $\tau$ also allows to express queries such as \textit{how two nodes are related to each other} by executing a \textit{connected by} query. This is not necessary in context of graph traversals, but can be further considered for a more advanced query algebra.
3.4 Summary

In this chapter we introduced the concept of single step traversals based on the work of Rodriguez et al. [RN11]. We used this to extract important single step traversal function related to graph navigation in Section 3.1.0.1.

Based on a RDF formal model in Section 3.2, we introduced our concept of basic pattern matching queries in Section 3.3. A basic pattern matching query is a trinary-equi selection \((\alpha, \beta, \gamma)\) to restrict edges in a graph.

Important basic pattern matching queries are:

- the *nodes-out* query \((\alpha, \beta, \ast)\) which returns nodes pointed by a node \(\alpha\) over a relationship \(\beta\), and
- the *edges-out* query \((\alpha, \ast, \ast)\) which returns all incident edges, and all adjacent nodes starting by \(\alpha\).

We showed that the set of basic pattern matching queries can be used as primitives for a query algebra which includes the navigation-related single step traversal functions in Section 3.3. Finally we argued that an efficient navigation in graph-shaped data based on efficient basic pattern matching evaluations.
4. Executing Basic Pattern Matches

In this chapter, we introduce native and non-native graph storage approaches to evaluate basic pattern matches.

Namely, we consider the following non-native operators:

- **Scanning.** All records are sequentially accessed to evaluate the conditions for a single step traversal. Therefore, the edges in a graph are stored in a tabular-fashion. This approach directly benefits from a broad knowledge corpus of the database research community, enabling several parallelism strategies, access pattern and evaluation strategies (used similar in GRAPHITE [PLB15] for instance).

- **Tree-based indexes.** Records are organized in six clustered $B^+$-trees, one per triples-, pairs-, and single values. The ordering on keys in this tree is the lexicographical ordering of the triples-, pairs-, and single values. Answering a query is achieved by navigating through the tree from the root to a leaf. The leafs contains the requested information (used in RDF-3x [NW08] for instance).

Further, we consider the following native operators:

- **Native graph processing.** Records are enriched with an ”micro-index” such that related records point to each other. Moreover, the index ”object cache” maps nodes and relationships to related edges (used in Neo4j [RWE13] for instance).

- **Adjacency lists.** The neighborhood of nodes is directly mapped to the corresponding nodes. Connected nodes are assigned as a collection to each related node, i.e., when two nodes are connected each of these nodes has a reference to the other. This data structure is a common data structure to store adjacency information in a graph, and is typically implemented as an array of arrays.
For each approach, we show the original concept we are building on, and distill relevant properties which characterize these approaches. If we cannot directly apply the original approaches (e.g., they rely heavily on block oriented storage as by $B^+$-trees), we describe alternatives with the same characteristics for our main-memory environment. We show further how the queries are executed, and how the index can be built. This chapter ends with an overview of the applicability of each approach for each basic pattern matching query.

4.1 Non-Native Graph Storage

This section contains graph storage and access approaches that do not link related recording. Thus, these approaches are non-native graph storage approaches.

4.1.1 Scan-Matcher

Table scans are a core physical primitive operation in relational engines, and can be used when indexes are not available or not reasonable. Table scans are regular used to implement selection operators in a physical query plan. Basically, a scan takes a table and a predicate (i.e., a set on conditions defined on the table’s schema) as input. The output is all records that satisfy the given condition. In traditional architectures of databases (i.e., disk-based system), full table scan are used to be considered as a fallback operation since for many use-cases a full table scan is the most expensive operation.

Since a full table scan is a sequential operation it is natural to utilize the highly parallelism capacities of modern many-core CPU architectures. In context of in-memory database architectures, table scan implementation further can take advantages of this parallelism such that the selection is limited by the memory bandwidth to the main memory rather than the costs for computation itself (i.e., table scans are memory-bound rather than compute-bound). In context of columnar system the amount of data to be read can be further limited to the required columns rather than to the entire rows as in row-oriented systems. However, in context of graph databases scanning occurs at least when linear search is required. For instance, fetching nodes satisfying a condition on their property keys typically requires a linear search when the graph database does not speedup this operation by the use of an index, e.g., a binary tree.

Recent research in graph databases showed certain possible algorithmic strategies to evaluate traversal queries utilizing parallelized scanning. We highlight here the work of Paradies et al. who used scanning as core primitive for their proposed traversal strategies [PLB15].

In this thesis we address table scanning as one possible implementation of the basic pattern matching operator $\rho$. We call this physical operator SCAN-MATCHER and use the symbol $\rho_\sigma$ for short.
4.1. Non-Native Graph Storage

**Design space.** We outline the design-space of the physical operator $\rho_\sigma$:

1. **Storage layout.** How the edge-store is organized in memory, i.e., are the elements stored column-wise or row-wise physically.

2. **Access pattern.** How the evaluation of predicates on multiple columns is executed, i.e., is it a branching strategy, or is it a bulk-processing strategy that is used.

3. **Parallelism strategy.** How the parallel processing on the edge-store is organized, i.e., is the edge-store split into parts (chunks) for one CPU core, or are the columns of the edge-store computed at once in parallel by assigning one column to one CPU core.

Further we could consider to provide both a parallel and a sequential variant of $\rho_\sigma$. We do not consider the sequential variant since one can not expect benefits of a sequential scan compared to a parallelized scan (at least in terms of execution performance).

4.1.1.1 Storage Layout

In contrast to other approaches presented later in this chapter, the scan operator $\rho_\sigma$ is highly influenced by the underlying storage layout of the edge-store from a physical perspective. Not only the access pattern changes between columnar and row-oriented storage but also the CPU data cache utilization, i.e., a columnar design typically is more cache-friendly leading to better execution performances.

In the context of this thesis, we consider both approaches for our evaluation. Namely, the edge-store can be organized either column-wise ($C^{\text{col}}$), or row-wise ($C^{\text{row}}$).

4.1.1.2 Access Pattern

Additional to storage layout of the edge-store on which $\rho_\sigma$ operates, there is a design freedom on how the predicate on multiple columns is evaluated. Assume the predicate includes two attributes of the edge-store’s schema $\mathcal{C}$ (e.g., ”start = $\alpha$ ∧ rel = $\beta$”).

The evaluation can be performed on two ways: either we use (1) a branching strategy, or (2) a bulk-processing strategy:

**Branching Strategy.** The branching strategy is the typical use of && for boolean expressions: this strategy loops over the first column evaluating the boolean expression on the first column (i.e., $e_i[\text{start}] = \alpha$ for each $e_i \in C$) until a true occurs. If the evaluation of the first column returns true, the second column is evaluated (i.e., $e_i[\text{rel}] = \alpha$ for each $e_i \in C$). The result of this evaluation is combined using a logical and, and the outer loop on the first column is continued until all records are processed. The advantage of this strategy is, that successfully evaluated elements can be directly added...
to the final resultset. The drawback is, that there are likely more cache misses since switching in another branch typically load further data (of the second column) into the CPU data cache which might be evicted in the next step immediately.

**Bulk-processing strategy.** The bulk-processing strategy performs the evaluation in multiple steps, one for each column in the predicate definition. Each step \( i \) requires a list of positions in the table which defines the set of elements in the edge-store to be evaluated in this step. The \( i \)-th step returns a list of positions in the edge-store such that the corresponding elements in the edge-store satisfy the predicate on the \( i \)-th column\(^1\). The first step receives the position list passed to \( \rho \sigma \). The last step yields the final result. Each step between the first and the last produce intermediate lists which are refined in the next step. This behavior is this strategies major drawback since it requires multiple reads and writes. The advantage rely on the efficient CPU data cache utilization, i.e., the entire column (or the corresponding chunk) is read sequentially which likely leads to a higher cache hit rate.

In context of this work, we consider both approaches. We use \( \rho^{\text{branch}} \sigma \) for the branching strategy, and \( \rho^{\text{bulk}} \sigma \) for the bulk-processing strategy.

### 4.1.1.3 Parallelism strategy

The parallelism strategy defines how the evaluation of \( \rho \sigma \) is split across multiple CPU cores. In the following we outline two common strategies for this purpose: horizontal, and vertical partitioning.

From a physical database design perspective, the both partition types typically deals with the physical management of a table, i.e., whenever the table is partitioned by rows (horizontal partitioning), or by columns (vertical partitioning). Since this decision affects the performance significantly (e.g., consider indexing and view materialization), partitioning is an active area of physical database research [ANY04].

\[^1\]An alternative approach to the process of building a list of satisfying elements is to build a list of statements on the satisfaction of each element. The latter is typically implemented with the use of a bitmask \( \ell_{\text{mask}} \) which contains a number of \( |R| \) boolean elements (for the table \( R \)). The \( i \)-th element in \( \ell_{\text{mask}} \) correspond to the \( i \)-th element in the table. The boolean value of the \( i \)-th element in mask is the satisfaction value. The benefits of bitmasks rely on their efficient combination by logical operations (e.g., xor). Their drawback rely on the higher costs for value changes for the \( i \)-th element. Since this operation is bit-wise but the smallest data addressable is a byte, 7 bit are touched to change the value of one entry. At least in context of our pre-evaluation, these costs were not amortized such that we choose the first approach.
In context of parallel execution of selection operators, we refer to horizontal partitioning, and vertical partitioning in terms of parallel processing as follows: A parallel processing can be done either by

- **Vertical partitioning**: process columns of the table in parallel, or
- **Horizontal partitioning**: process rows in the table in parallel.

**Vertical partitioning.** The general approach for vertical partitioning is to assign the processing of $i$-th column of $A$ to the $i$-processor. When $i$ is larger than the number of processors, a scheduling strategy must be applied to avoid over-utilization of a single processor (e.g., by a round-robin strategy). When $i$ is smaller than the number of processors, the vertical approach waste processing power since a under-utilization occur. In this case some processors are not assigned to a tasks such that they do not participate on the evaluation.

**Horizontal partitioning.** In contrast to vertical partitioning, horizontal partitioning splits a table $A$ row-wise into disjunct sets of rows $A_1, A_2, ..., A_n$ (chunks) equal to the number $n$ of targeted processors ($n$ is usually the total number of processors). As a consequence, $A$ is scanned in parallel by $n$ processors where each processor evaluates the predicate on its assigned chunk $A_i$, and returns the elements\(^2\) in $A_i$ satisfying the selection condition. After a synchronization step which guarantees the end of each parallel evaluation, the partially results generated per processor in parallel, are merged into the final result.

The tables processed in context of this thesis contain a fixed number of three columns. As a consequence we consider horizontal partitioning only, since commodity hardware supports more than three threads to be executed truly in parallel, and so do more professional hardware.

4.1.1.4 Query Execution

A pattern vector $p$ using S-Matcher is evaluated by invoking asynchronous kernels. In this context, a kernel is the logic to partially evaluate the scan predicate for $p$ on an edge-store. A partially evaluation is a complete evaluation of $p$ on elements but only in the bounds of a given range in the edge-store $C$. An evaluation is complete if and only if the evaluation considers all elements in a given range of elements in $C$. All kernels invoked in order to evaluate $p$ on $C$ completely, operate concurrently and in parallel on distinct ranges of edges. These distinct ranges partition $C$. The complete evaluation of $p$ on elements in $C$ is the union of all partial evaluations of the involved

\(^2\)In the context of main-memory database architectures, this process typically operates on positions lists (indices) in a table rather than the actual values for efficiency reasons.
kernels. More in detail, a kernel is a (higher-order) function \( f \) that takes a pattern vector \( p \), a begin index \( i_{\text{begin}} \) and end-index \( i_{\text{end}} \), and an edge-store \( C \), and computes a subset of edge indices \( I_f \subset [i_{\text{begin}}, i_{\text{end}}] \) such that edges with identifiers in \( I_f \) matches the condition of \( p \). We provide for each pattern vector (i.e., for each query \( q_1, q_2, \ldots, q_7 \)) a single kernel which is aware whenever column-wise (\( C_{\text{col}} \)), or row-wise (\( C_{\text{row}} \)) storage layout is used, and whenever branching strategy (\( \rho_{\text{branch}} \)), or bulk strategy (\( \rho_{\text{bulk}} \)) is intended. However, \( f \) is invoked on pair-wise different right-open intervals of elements in \( C \), such that the union of all these intervals form the set of edge identifiers in \( C \). The width of a single interval is determined by the number of records, and the number of threads that can be executed in parallel.

In algorithm 1 we show pseudo code to a procedure \textsc{Parallel} which takes a pattern vector \( p \) and a (compatible) kernel \( f \). This procedure partitions the set of edge identifiers in a given edge-store \( C \) such that \( f \) is invoked multiple times on different threads on different ranges. Each \( f \) returns a partial evaluation of \( p \) on \( C \) such that the union of these partial results form the final result. More in detail, we initialize as many partial result index sets as parallel threads are supported by the CPU architecture (line 2). We start these threads with the following logic: in each thread we compute the thread’s chunk size as equal share in respect to the total number of threads (line 4). In addition, we compute each kernels begin- and end index on elements in \( C \) considering the last element index in \( C \) as the maximal feasible end index (line 5-9). Finally, we invoke \( f \) on the range in \( C \) given the pattern vector \( p \). This leads to a partial evaluation by \( f \) in the given range (line 10). After a synchronization phase, we compute the union of the partial results\(^3\) an return.

\(^3\)We copy the partial results sequentially into the final result at the main thread.
Algorithm 2 Querying \textit{nodes out} pattern using the index structure S-Matcher-Kernel ($\rho^\text{branch}_\sigma$)

1: \textbf{procedure} Query S-Matcher-Kernel $\rho^\text{branch}_\sigma(\alpha, \beta, i_{\text{begin}}, i_{\text{end}}, C)$
2: \hspace{1em} let $I_{\text{out}} = \emptyset$
3: \hspace{1em} for $i = i_{\text{begin}}$ to $i_{\text{end}} - 1$ do
4: \hspace{2em} let $e_i = (e_1, e_2, e_3)_i = C[i]$
5: \hspace{2em} if $e_1$ is $\alpha$ then
6: \hspace{3em} if $e_2$ is $\beta$ then
7: \hspace{4em} $I_{\text{out}} \leftarrow I_{\text{out}} \cup i$
8: \hspace{2em} return $I_{\text{out}}$

In algorithm 2, and in algorithm 3 we show pseudo code of the $\rho^\text{branch}_\sigma$ kernel resp. the $\rho^\text{bulk}_\sigma$ kernel. Both kernel evaluate a \textit{nodes out} pattern on a given node identifier $\alpha$ and a relationship identifier $\beta$. The kernel to evaluate $\rho^\text{branch}_\sigma$ (Algorithm 2) first initializes the resultset $I_{\text{out}}$ as an empty set (line 2). For each edge in the given range $[i_{\text{begin}}, i_{\text{end}})$ of indices in a given $C$, the predicate is evaluated following the branching strategy. Hence, if the starting node identifier for the $i$-th edge is equal $\alpha$, and the relationship identifier on the same edge is equal to $\beta$, $i$ is qualified for the resultset. Hence, $i$ is added to $I_{\text{out}}$ for $i_{\text{begin}} \leq i < i_{\text{end}}$ (line 4-7). The set $I_{\text{out}}$ is the final result, and therefore returned.

Algorithm 3 Querying \textit{nodes out} pattern using the index structure S-Matcher-Kernel ($\rho^\text{bulk}_\sigma$)

1: \textbf{procedure} Query S-Matcher-Kernel $\rho^\text{bulk}_\sigma(\alpha, \beta, i_{\text{begin}}, i_{\text{end}}, C)$
2: \hspace{1em} let $I_{\text{out}} = \emptyset$
3: \hspace{1em} let $I'_{\text{out}} = \emptyset$
4: \hspace{1em} for $i = i_{\text{begin}}$ to $i_{\text{end}} - 1$ do
5: \hspace{2em} let $e_i = (e_1, e_2, e_3)_i = C[i]$
6: \hspace{2em} if $e_1$ is $\alpha$ then
7: \hspace{3em} $I_{\text{out}} \leftarrow I_{\text{out}} \cup i$
8: \hspace{2em} for each $i \in I_{\text{out}}$ do
9: \hspace{3em} let $e_i = (e_1, e_2, e_3)_i = C[i]$
10: \hspace{4em} if $e_2$ is $\beta$ then
11: \hspace{5em} $I'_{\text{out}} \leftarrow I'_{\text{out}} \cup i$
12: \hspace{1em} return $I'_{\text{out}}$

In contrast, the final result is built in two steps following the bulk strategy (algorithm 3). Here, an intermediate resultset $I_{\text{out}}$, and the final resultset $I'_{\text{out}}$ are initialized (line 2-3). The first step is to compute an intermediate result $I_{\text{out}}$ that contains edge identifiers which satisfy a equality to $\alpha$ for their starting node identifiers (line 4-7). Since this
does not consider the second condition $\beta$, the intermediate result $I_{\text{out}}$ has to be refined. Consequently, the second step is to evaluate edges with edge identifier in $I_{\text{out}}$ in respect to the equality for their relationship identifiers and $\beta$ (line 8-10). Does an edge with identifier $i \in I_{\text{out}}$ satisfy this condition, $i$ is added to the final resultset $I'_{\text{out}}$ which is returned finally (line 11-12).

We note here, that the S-MATCHER index structure is capable for all possible simple pattern queries.

4.1.2 Red-Black-Matcher

In this section we describe an index structure Red-Black-Matcher which is inspired by the indexing strategy of the state-of-the-art RDF engine RDF-3x of Neumann et al. [NW08]. The core idea of Red-Black-Matcher is to organize the edge-store by seven red-black trees, one per query pattern. More specific, for each subset of the edge-store columns (i.e., "(start), (relationship), (end), (start, relationship), (start, end),....,(start, relationship, end)"), one such tree is used. Each such subset defines how to project edges in the edge-store. Each projected edge value is a node inside the tree. For each node, the edge identifier on which the projected edge value was taken from is attached in a list. When two edges share the same projection, their edge identifier lists are combined. Consequentially, the request defined by a certain basic pattern is delegated to the corresponding tree where the matching node contains the result. For instance, if a nodes out query is invoked on a given node identifier $\alpha$, and a given relationship identifier $\beta$, the (start, relationship)-tree is considered. The requested edge identifiers are stored in the trees node matching $(\alpha, \beta)$. The node with the key $(\alpha, \beta)$ can be found in logarithmic time by navigating through the tree.

Following, we outline the original indexing strategy used in RDF-3x, six separate $B^+$ trees. We summarize and conclude properties of the usage of these trees, and argue why it is reasonable to use red-black trees in our prototype.

4.1.2.1 Triple Storage and Indexing in RDF-3x

RDF-3x is an RDF engine that treat the entire set of triples as ”giant triples table” which is internally organized in several seperate clustered $B^+$ trees (using compression) [NW08]. For each permutation of the triples components (i.e., the triple itself, pair- and single-values) one such $B^+$ tree is used. In RDF-3x triple (pair-, and single-) values are used as the key inside the $B^+$ tree where the leafs contain the triple values. More specific, Neumann et al. suggests to use the dictionary compressed value for these triple (pair-, single-) values instead of the concrete values on the entire process. A lexicographical ordering is used on the triples (pairs, single) value in the dictionary inside the $B^+$ trees. Since keys close to another are likely very similar, this ordering
is then exploited for compression using a byte-level delta-encoding approach. As a consequence of this indexing strategy, a (SPARQL pattern) query can be answered using a range scan on the tree’s leafs efficiently while the storage consumption is acceptable.

**Outline of properties for indexes in RDF-3x.** RDF-3x is optimized as a disk-based database, i.e., it heavily rely on $B^+$ trees which are strongly optimized for block-oriented storage.

Since this thesis focus on main-memory data structures, we are interested in taking advantages from key-properties of the index structures in RDF-3x without the penalty of disk IO. To achieve this goal, we infer the following properties of the indexes used in RDF-3x:

1. **One index per subset.** RDF-3x uses one index structure per subset, i.e., each triple, each pair, and each value is indexed separately. Each pattern query correspond to one of this index structures.

2. **Logarithmic time complexity for record lookup.** Identifying a record in one index structure has a logarithmic time complexity which depends on the branching number, the number of children per node, and the number of elements in the three.

3. **Linear effort for result collecting.** Once a record is found, a leaf $\ell$ is yielded. A range query is performed by scanning from $\ell$ until the range scan condition is no longer satisfied. The scanning operation typically runs with a linear complexity between these leafs.

To avoid unnecessary waste of memory and storage, node and leaf values in each $B^+$ tree are compressed in RDF-3x. Hence, there is a dictionary component which provides a mapping between unique identifiers and values.

As mentioned earlier, we already assume edges containing only dictionary compressed values rather than concrete values. We do not address the fact of further compression techniques (e.g., delta-compression) since this is off-topic in the context of this thesis. However, besides this limitation, with Red-Black-Matcher we suggest an in-memory indexing strategy which leverages multiple internal index structures utilizing dictionary encoding, which have a logarithmic querying complexity and a constant time complexity for the ”leaf scans” part. In fact, we organize the pre-computed results per key as a ready-to-return data structure rather than by collecting these information via scanning.

In Figure 4.1 we depict an overview on the indexing strategy in Red-Black-Matcher.
4. Executing Basic Pattern Matches

4.1.2.2 Building Blocks

**Red-Black-Matcher** is optionally attached as an index to a collection \( C \) of edges. It internally holds seven distinct red-black trees \((A_{q_1}, \leq), (A_{q_2}, \leq), ..., (A_{q_7}, \leq)\) where \( \leq \) is the lexicographic order of elements in \( A_q \) (for \( q \in \{q_1, q_2, ..., q_7\} \)). A red-black tree is a self-balancing binary search tree with a logarithmic search worst case running time complexity, which uses node-coloring (i.e., red and black nodes) to achieve its balanced state [Bra08]. However, each red-black tree \( A_q \) is associated to the corresponding basic pattern match query \( q \) for \( q \in \{q_1, q_2, ..., q_7\} \) (see Table 3.1).

Each node \( \text{node} = (\text{key}, \text{result}) \) in each tree \( A_q \) (the **index**) contains two components: the search key \( \text{key} \), and a collection \( \text{result} = I_{A_q}(k) \) which contains the edge indices in \( C \) for which the value of \( \text{key} \) participate in the edges with indices in \( I_{A_q}(k) \) (for \( q \in \{q_1, q_2, ..., q_7\} \)). The required ordering to organize the tree \( A_q \) is \( \leq \) which is a lexicographic order on the key values.

In the case that some components of some edges are equal, the index \((A_q, \leq)\) contains only **distinct** keys where the results of nodes with colliding keys are merged together. More specific, in case the projected pair (or single value) of two, or more edges collide (e.g., the keys \( key_1 = key_2 \)), the results are merged (i.e., \( I_{A_q}(key_1) \) and \( I_{A_q}(key_2) \)).

For a better understanding, consider the following example. Assume two edges \( e_1 = (5, 2, 3) \), and \( e_2 = (5, 2, 4) \): Both do not collide when the second and third component forms the key, i.e., \((2, 3) \neq (2, 4)\) in \((A_{q_4}, \leq) = \{(2, 3), (2, 4)\}\). In this case \( I_{A_{q_4}}((2, 3)) = \{1\} \) and \( I_{A_{q_4}}((2, 4)) = \{2\} \). They do collide when the first, and second component form the key, i.e., the key \((5, 2)\) refer to both edges \( e_1 \) and \( e_2 \) in \((A_{q_2}, \leq)\). In this case the result is \( I_{A_{q_2}}((5, 2)) = \{1, 2\} \). Obviously, the lesser components form a key, the more collisions happen and the larger the results \( I_{A_q} \) will be per node (for \( q \in \{q_1, q_2, ..., q_7\} \)).

4.1.2.3 Querying Concept

From a physical perspective, once the indexes \( A_q \) are constructed, querying a certain result (i.e., a list of edge identifiers) with the key \( \text{key} \) is achieved by navigating through
Algorithm 4 Building the index structure Red-Black-Matcher

1. procedure Build Red-Black-Matcher(graph $G = (N, R, C)$)
2. let $(A_q, \leq)$ empty red-black trees for $q \in \{q_1, q_2, ..., q_7\}$
3. for $i = 1$ to 7 in parallel do
4.   for each $e_j = (e_1, e_2, e_3) \in C$ do
5.     let $key = \text{Project}(e_j, i)$
6.     if $(A_q, \leq)$ not contains node $(key, \cdot)$ then
7.       let $result = I_{A_q}(key) = \{j\}$
8.       let $node = (key, result)$
9.       $(A_q, \leq) \leftarrow ((A_q, \leq) \cup node)$
10.   else
11.     let $node = (key, result)$ find $(key, \cdot)$ in $(A_q, \leq)$
12.     $node[1] = result \leftarrow result \cup \{j\}$
13.     refresh $node$ in $(A_q, \leq)$
14. sync
15. return $(A_q, \leq)$ for $q \in \{q_1, q_2, ..., q_7\}$

16. procedure Project(edge $(e_1, e_2, e_3)$, query type index $j$)
17.   if $j$ is 1 then return $(e_1, e_2, e_3)$
18.   else if $j$ is 2 then return $(e_1, e_2)$
19.   else if $j$ is 3 then return $(e_1, e_3)$
20.   else if $j$ is 4 then return $(e_2, e_3)$
21.   else if $j$ is 5 then return $(e_3)$
22.   else if $j$ is 6 then return $(e_1)$
23.   else return $(e_2)$

the search tree $(A_q, \leq)$ from node to node until the node with the key $key$ is found (for one fixed $q$ in $\{q_1, q_2, ..., q_7\}$). Since the search tree $(A_q, \leq)$ is balanced, determining the node with the key $key$ is achieved by a logarithmic number of "less equal" or "greater than" comparisons on nodes keys from the root node to the node with the key $key$. Since we attach the list of edge identifiers on which $key$ participate, the list associated to $key$ in the found node just has to be returned in order to answer the query. Copying the pre-computed resultset is in contrast to a linear-scan-based approach as in $B^+$ trees. While a disk-based $B^+$-tree might read the resultset items one-by-one from disk, the cost of our approach is the cost of copying a single memory address once the binary search is done. Hence, once the node associated to a certain key is found, copying the "ready-to-return" result can be achieved using constant time.
Outline of properties for indexes in Red-Black-Matcher. We outline the properties of Red-Black-Matcher in comparison to the indexing approach of RDF-3x:

1. **One index per permutation.** Both, Red-Black-Matcher, and RDF-3x uses one index structure for a certain query pattern.

2. **Logarithmic time complexity for record lookup.** Depending on the configuration in respect to the branching, and number of elements stored, the $B^+$ tree of RDF-3x has a bit better complexity compared to our approach. However, the querying times in Red-Black-Matcher are lower in practice since we are not affected by disk IO costs, and do not require a buffer manager.

3. **Constant effort for result determination.** Once a record is found in Red-Black-Matcher, it can be returned immediately by following the copy-by-reference paradigm. In contrast, RDF-3x requires a scan of relevant leafs in its trees in order to collect the result which is far more costly.

4.1.2.4 Index Building Process

In algorithm 4 we show pseudo code which builds the seven indexes in Red-Black-Matcher in parallel

Building the Red-Black-Matcher index structures is straightforward: The procedure takes a graph $G$, and returns seven fully initialized red-black trees $(A_q, \leq)$ for $i \in \{1, 2, ..., 7\}$. Thus, the procedure first initialize seven new red-black trees, one per query pattern $q_i$ (line 2). The nodes in each red-black tree $(A_q, \leq)$ are pairs $(key, result)$ where key matches the signature of the query pattern $q_i$, i.e., for $i = 2$ the red-black tree $A_{q_2}$ expects nodes with keys built from starting node identifiers, and relationship identifiers. However, since Red-Black-Matcher holds seven red-black trees, we fill these trees in parallel, one tree per thread (line 3-13). In each thread $i \in \{1, 2, ..., 7\}$ the edge-store is iterated. Per iteration step $j$ (i.e., the $j$-th edge in the graph), a nodes key is constructed by a projection of the $j$-th edge according to the query pattern $q_i$. For instance, if the third edge is $e_3 = (1, 4, 6)$ and the second thread (i.e., the thread responsible for $q_2$ resp. $(A_{q_2}, \leq)$) is considered, the key for $e_3$ is $(1, 4)$. This projection is done by a projection function (line 5, and line 16-23). Once a key for the $j$-th edge is built for a certain $q_i$, we have to check whenever there is a node with the given key. Therefore, we query the corresponding red-black-tree to find a node with the given key.

We do not go into detail, how a red-black tree is internally organized and queried. Instead we point into the literature [Bra08]. For a understanding it is sufficient, that a red-black tree satisfies the properties of an sorted associative container that contains $(key, value)$ pairs, such that finding a certain pair with a given key component can be achieved in logarithmic time.
(line 6). There are two cases: either we do not find such a node, then we have to add a new node with the given key and \( j \) as one edge identifier (line 7-9). Or, we do find a node. Then we have to add the edge identifier \( j \) to the existing result of the node with the given key (line 11-13). After a synchronization step which guarantees each thread has finished its work, we can return the procedures result.

### 4.1.2.5 Querying Process

We show in algorithm 5 a pseudo code which provides the logic how a *nodes out* query is answered using Red-Black-Matcher.

**Algorithm 5** Querying *nodes out* pattern using the index structure Red-Black-Matcher

1: procedure Query Red-Black-Matcher(\( \alpha \), \( \beta \), red-black tree \( (A_{q_2}, \leq) \))
2:   let key = \( (\alpha, \beta) \)
3:   if \( (A_{q_2}, \leq) \) not contains node then
4:     return \( \emptyset \)
5:   else
6:     let node = \( (key, result) \) find \( (key, \cdot) \) in \( (A_{q_2}, \leq) \)
7:     return result

The evaluation of the given pattern is straightforward and quite efficient: We query the corresponding red-black tree for a node with the given key. This querying internally performs a navigation in the search tree. In logarithmic time, either a node is found or not. If no node is found, we return an empty list of corresponding edge identifiers. Otherwise we return the list of edge identifiers of edges which were taken into account to build the node resp. the key during the build-phase.

We note here, that the Red-Black-Matcher index structure is capable for all possible query patterns.

### 4.2 Native Graph Storage

This section contains graph storage and access approaches that do link related recording. Thus, these approaches are native graph storage approaches.

#### 4.2.1 Native-Matcher

In this section we describe our index structure Native-Matcher which is similar to the *object cache* of state-of-the-art *native graph* databases, i.e., the object cache approach can be found in Neo4j for instance. The core idea of this index structure is
to organize record (-references) such that related records "point" to each other. Hence, the \textsc{Native-Matcher} enables \textit{native graph} capabilities.

In the following we outline properties whenever a database is native graph database, or not. Afterwards, we show how native graph database organize their data storage and indexing. This leads to the \textit{object cache} index, which we will explain. On this concept, we provide our index structure \textsc{Native-Matcher} that we will use for our evaluation in Chapter 5.

4.2.1.1 Object Cache

The object cache is a main-memory caching/indexing data structure in Neo4j which is intended to enable graph traversals with less penalty of disk access compared to construct the required information by reading directly from the related files. The object cache provides two properties: (1) it caches records from a set of physical files in order to avoid costly disk read operations, and (2) it combines information of multiple physical files such that an entry in the object cache contains not only edge information but complex node and relationship objects. To enable fast lookups, the entries in the object cache are grouped according some strategies.

In Section 3.2 we provide a graph model which allows us to break traversals down to a unified processing concept, the basic pattern matching. We outline that physical operators to evaluate basic pattern matching share the same function signature. However, in the following, we describe \textsc{Native-Matcher} as one possible physical operator which is strongly based by object cache. Where ever reasonable, we outline the differences to the object cache as provided by Neo4j.

4.2.1.2 Native-Matcher Characteristics

The \textsc{Native-Matcher} is built upon the management of edge indices per node identifier grouped by in-coming and out-going edges, and relationship types.

\textbf{Building Blocks.} In Figure 4.2 we provide a schematic view on of the \textsc{Native-Matcher}. This index structure contains per each node in the graph a set of cache
pairs. Each cache pair per node has a collection of cache entries for both, in-coming and out-going edges, for which the node is either the starting node (out-going edges), or the node is the ending node (in-coming edges). Each of these collection of edges is grouped by the relationship contained in the edge. More formally, each node $u_i$ is associated with a pair $(\text{cache}_{\text{in}}(u_i), \text{cache}_{\text{out}}(u_i))$ for $i$ as a valid node identifier from the domain of nodes $D_n$. These pairs combine the cache entries for $u_i$ for out-going and in-coming edges grouped by a relationship type. The idea is to assign all edge identifier $j$ for each edge $e_j \in C$ to $u_i$ for which $u_i$ is the starting node (cache$_{\text{out}}(u_i))$, and for which $u_i$ is the ending node (cache$_{\text{in}}(u_i))$ where it is distinguished between the kind of relationship attached to the edges. As a consequence of this strategy, querying out-going edges for a node $u_i$ (see Table 3.1) is processed by reading cache$_{\text{out}}(u_i)$. In addition, querying out-going edges with a certain relationship $r$ for $u_i$, is processed by reading cache$_{\text{out}}(u_i)$ restricted to the relationship $r$. A single cache entry in cache$_x(u_i)$ for $x \in \{\text{in}, \text{out}\}$ is a pair $(r, A)$ where $r$ is a relationship identifier, and $A$ is a set of edges indices in the edge-store. The semantic is as follows:

- **Relationship identifier.** Each cache entry contains a value $r$ from the domain of valid relationship identifiers $D_r$. Each edge for which its index occurs in $A$ has this value $r$ as the relationship, i.e., $e_2 = r$ for $(e_1, e_2, e_3) = e_k \in C$ all $k \in A$. We call $r$ the relationship grouping key, since the cache entry with the value $r$ is basically a grouping of edge indices which has $r$ in common, and for which $u_i$ participates (see next bullet point).

- **Set of related edge indexes.** Each cache entry contains a set $A$ with elements from the domain of valid edge indices $I$ in the edge-store. If the cache entry is part of the out-going cache (cache$_{\text{out}}(u_i)$), each edge $(e_1, e_2, e_3) = e_k$ with $k \in A$ has $u_i$ as its starting node, i.e., $u_i = e_1$. If, on the other hand, the cache entry is part of the in-going cache (cache$_{\text{in}}(u_i)$), each edge $(e_1, e_2, e_3) = e_k$ with $k \in A$ has $u_i$ as its ending node, i.e., $u_i = e_3$.

Native-Matcher differs from Neo4j’s object cache in two properties: (1) the object cache deals with complex (Java) objects that are build, stored, and refreshed in the object cache. In contrast Native-Matcher treat this issue in terms of identifiers and collections only, and (2) in Neo4j an edge’s relationship can have a certain type which is the grouping key rather than the a relationship identifier in Native-Matcher. The object cache groups the entry related to a certain relationship type where relationship objects has a certain type. In contrast, we threat relationships at instance level, i.e., the type might be attached to a relationship object behind the relationship identifier.
4.2.1.3 Index Building Process

In algorithm 6 we provide a pseudo-code which shows how the Native-Matcher index is build from ground up. To outline the core idea, we restrict this logic to the creation of cache\textsubscript{out} since the other way around is similar. We suggest to run this process in parallel to fully utilize the available CPU cores, and to shorten the time needed to build the index.

Algorithm 6 Building the index structure Native-Matcher (restricted cache\textsubscript{out})

1: procedure Build Native-Matcher\hfill (graph $G = (N, R, C)$)
2: \hspace{1em} let Pairs\textsubscript{per thread} $\leftarrow$ empty collection of (empty collections of cache pairs)
3: \hspace{1em} for \hspace{1em} thread\textsubscript{id} \hspace{1em} in \hspace{1em} 0 \hspace{1em} to \hspace{1em} number of threads - 1 \hspace{1em} do \hspace{1em} in \hspace{1em} parallel
4: \hspace{2em} let chunk\textsubscript{size} $=$ $|C|$/(number of threads)
5: \hspace{2em} let begin $=$ chunk\textsubscript{size} $\times$ thread\textsubscript{id}
6: \hspace{2em} if thread\textsubscript{id} + 1 \hspace{1em} is \hspace{1em} equal\hspace{1em} to \hspace{1em} (number of threads) \hspace{1em} and \hspace{1em} begin $<$ $|C|$ then
7: \hspace{3em} let end $=$ $|C|$
8: \hspace{2em} else
9: \hspace{3em} let end $=$ min(begin + chunk\textsubscript{size}, $|C|$)
10: \hspace{2em} for \hspace{1em} $k = $ begin \hspace{1em} to \hspace{1em} end - 1 \hspace{1em} do
11: \hspace{3em} let $u_i = e_k[1]$
12: \hspace{3em} let $r = e_k[2]$
13: \hspace{3em} let $p = (\text{cache}_\text{in}(u_i), \text{cache}_\text{out}(u_i)) \leftarrow$ Pairs\textsubscript{per thread}[thread\textsubscript{id}][u\textsubscript{i}]
14: \hspace{3em} let Cache Entry $= (r', A)$ from cache\textsubscript{out}(u\textsubscript{i}) in $p$ where $r'$ \hspace{1em} is \hspace{1em} $r$
15: \hspace{3em} $A \leftarrow A \cup k$
16: \hspace{3em} refresh $P$
17: \hspace{2em} sync
18: \hspace{2em} return (merge Pairs\textsubscript{per thread})

The input of the Build Native-Matcher algorithm is a graph $G = (N, R, C)$, and the output is a collection of cache pairs as described earlier. First, we define Pairs\textsubscript{per thread} which is an indexed collection of collection of cache pairs (line 2). We assume here for simplicity reasons, that the $i$-th element in Pairs\textsubscript{per thread} can be accessed using a projection operator $[i]$ which automatically deals with creation of the $i$-th element when first called. However, we run the main logic in parallel (line 3-16): For each thread the interval [begin, end) of indices in the collection of $C$ is calculated by a common pattern in parallel computing, i.e., we compute a chunk size (line 4) which determines along with the current thread the start position (line 5), and the end position (line 7 resp. line 9). The end position is either the start position plus the chunk size, or the end of the collection $C$. However, for each index $k$ between the start and the end position, we take the starting node $u_i$, and the relationship identifier $r$ of the edge $e_k$ in $C$. We then
receive the per-thread collection of cache pairs for the \( u_i \)-th index (line 13). From this, we query the cache entry \((r', A)\) which has \( r \) as the related relationship identifier (line 14) in order to add the current edge identifier \( k \) to the set of relevant edge identifiers \( A \). This change in the collection is afterwards propagated resp. stored (line 16). After a synchronization phase which guarantees all threads to be joined (line 17), we merge the partial generated results and return this as the final result.

### 4.2.1.4 Query Process

We show in algorithm 7 a pseudo code which provides the logic how a nodes out query is answered using Native-Matcher.

**Algorithm 7** Querying nodes out pattern using the index structure Native-Matcher

```plaintext
1: procedure QUERY Native-Matcher(\( \alpha, \beta, \text{Native-Matcher Pairs collections} \))
2: let \((\text{cache}_\text{in}(\alpha), \text{cache}_\text{out}(\alpha)) \leftarrow \text{Pairs}[\alpha] \)
3: let Cache Entry \((r, A)\) from \(\text{cache}_\text{out}(\alpha)\) where \(r\) is \(\beta\)
4: return \(A\)
```

The evaluation of the given pattern is straightforward and quite efficient: We set \( \alpha \) (the starting node identifier), \( \beta \) (the relationship identifier), and a completely constructed index as the parameters of the query in algorithm 6. From Pairs we select the \( \alpha \)-th cache pair (line 2). This pair contains in-coming, and out-going edges in respect to \( \alpha \). From this we choose the out-going edges and return the set of edge identifiers from the cache entry which relationship identifier is equal to \( \beta \).

We note here, that the Native-Matcher index structure neither improved each possible pattern vector for a basic pattern match query, nor that the result of Native-Matcher can taken as-is for certain pattern vector definitions (e.g., the connected by pattern requires an index scan).

### 4.2.2 Adjacency-Matcher

In this section we describe an index structure Adjacency-Matcher which is an implementation of the adjacency list approach. An adjacency list is a common data structure to represent the neighborhood of nodes in a graph, such that the neighborhood \( \mathcal{N}(u) \) is organized as a collection which is directly attached to a graphs node \( u \).

#### 4.2.2.1 Properties

The core idea of Adjacency-Matcher is to utilize efficient random-access in a sequential-aligned block of fixed-sized elements to map node identifiers to related edges identifiers (i.e., it uses arrays of arrays). This idea is indeed not novel, since this idea is one possibility to implement an adjacency list \( A \). But it is necessary to mention it here.
for completeness reasons in our evaluation. One should keep in mind that adjacency lists promise the best query performance for neighborhood queries on arbitrary nodes in the graph.

The index Adjacency-Matcher provides two adjacency lists: the first maps a node to edges in which this node is the starting node of each of these edge ($A_{\text{start}}$), and the second maps a node to edges in which this node is the ending node of each of these edges ($A_{\text{end}}$). More specific, let $d \in \{\text{start, end}\}$, the index $k$ for an element $a_k$ in $A_d$ corresponds to the node with the node identifier $u^{\text{id}} = k$. Further, the element $a_k$ itself is a set $a_k = \{i_1, i_2, ..., i_\ell\}$ of edge identifiers drawn from the graphs edge identifier domain $\mathcal{I}$ such that for all node identifiers $u^{\text{id}}$ in the graph the following holds:

1. $A_{\text{start}}[u^{\text{id}}] = A_{\text{start}}[k] = \{i \in \mathcal{I} \mid e_i \in C \land e_i[1] = u^{\text{id}}\}$
2. $A_{\text{end}}[u^{\text{id}}] = A_{\text{end}}[k] = \{i \in \mathcal{I} \mid e_i \in C \land e_i[3] = u^{\text{id}}\}$

where $C$ is the edge-store in the graph, and $1 \leq (k = u^{\text{id}}) \leq |A_d|$.

### 4.2.2.2 Querying Process

The determining of the neighborhood of a node $u^{\text{id}} = k$ is constant effort: seek to the $k$-th index in $A_d$, and return the address of the collection which is referenced from the $k$-th element in $A_d$.

The query performance comes to the cost of storage consumption: Adjacency-Matcher as an uncompressed adjacency list is highly affected by the number of nodes in the graph $G$ since the number of elements in $A$ is the number of nodes in $G$. This becomes dramatically for isolated nodes or if the graph is very sparse since more memory is waste the sparser a graph is.

We note here, that the basic patterns edges in, and edges out directly benefit from the Adjacency-Matcher index structure due to Adjacency-Matcher answers these queries directly. The other basic patterns require a post-processing in order to match the additional condition, e.g., the nodes out pattern on a node $\alpha$, and a relationship $\beta$ requires to refine the results $\rho(p) = \{i_1, i_2, ..., i_n\}$ of an edges out pattern $p$ on $\alpha$. This refining must consider that each edge in $\rho(p)$ must satisfy the condition on the relationship $\beta$ in addition. This additional filtering introduces further overhead since these patterns are not directly supported by Adjacency-Matcher.
4.3. Summary

Table 4.1: Support of basic patterns per matcher. The symbol ✓ marks the direct support of a basic pattern, blank fields mark required post-processing (e.g., further selection), or fallback (i.e., execute scanning instead).

<table>
<thead>
<tr>
<th>Basic Pattern</th>
<th>Scan</th>
<th>Native</th>
<th>Red-Black</th>
<th>Adjacency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$: Exact match</td>
<td>✓</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_2$: Nodes out</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_3$: Connected by</td>
<td>✓</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_4$: Nodes in</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_5$: Edges in</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_6$: Edges out</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_7$: Connects</td>
<td>✓</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$q_8$: Full scan</td>
<td>✓</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4.3 Summary

In the previous chapter, we showed that the concept of basic patterns is a core functionality to evaluate single step traversals. In this section we provide an overview of four state-of-the-art approaches to evaluate single step traversals.

For each, we contribute an extended or modified version to evaluate basic patterns. We adapt existing approaches for a main-memory setting:

- **Scan-Matcher**: A high-efficient parallelized and memory-bound scan approach. It is aware of the tables storage layout (i.e., the records are aligned column- or row-wise in memory), the access pattern (i.e., the evaluation can be either in a bulk-fashion or in a branching-fashion). For querying we chose a vertical parallelism strategy. Hence, we evaluate a query by delegating the evaluation of table partitions to threads, one per CPU core. The result is then merged after a synchronization step, and returned.

- **Native-Matcher**: An storage-layout-independent index-structure that based on native-graph processing (especially on the object cache) concepts. Each node is mapped to a cache pair, each containing a collection of cache entries. These cache pairs contain out-going and in-going edges, i.e., if the out-going edges of a node is required, the out-going edge cache collection is considered. Each cache collection is relationship-centric organized, such that related edges are grouped by the relationship. Given a certain node and a certain relationship, querying the edges is done by seeking to the nodes out-going edges cache entry. The cache collection with the given relationship as grouping key contains the desired result.

- **Red-Black-Matcher**: An storage-layout-independent index-structure that based on the tree-based indexing strategy of RDF-3x. In contrast to RDF-3x ap-
proach, this approach uses seven (uncompressed) red-black trees to organize the index for each possible basic pattern query rather than block-storage oriented $B^+$-trees. Moreover, the resultset per query is organized as a "ready-to-return" collection which is directly attached to the trees nodes rather than in the leaf level. Consequentially, once a record is found in the tree, the desired result can be immediately returned rather than firstly collect the result item-by-item as in $B^+$ trees.

- **Adjacency-Matcher**: An storage-layout-independent index-structure that holds two adjacency lists, one for edges-in, and one for edges-out queries. An array with elements pointing to a collection containing the desired result per query type is built. Both arrays has as many elements as nodes in the graph exist such that the $i$-th element index in each array correspond to the node with the node identifier $i$. Querying this index structure for one of the two possible queries for a node with the identifier $i$ is done by seeking the $i$-th element and returning the collection pointed from here.

Each of our approaches fit into the main-memory setting of our in-memory graph database prototype Griffin. We showed and explained the design space for each approach, argued what we include or exclude in our approaches. We give a detailed overview on internals per-approach including storage- and record organization, preparation steps (e.g., index-building), and querying at a conceptional level. We also note, that not all of this approaches speedup all query types: some approaches do not accelerate certain basic pattern queries. In this case either a fallback to scanning, or a post-processing of the result from a more general query is required. We summarize the support for basic pattern by approach in Table 4.1. The support of a certain basic pattern per strategy is driven by the strategies design: while scanning and the red-black trees approach are universal strategies, an adjacency list can only accelerate edges-in, and edges-out queries for instance.
5. Implementation and Evaluation

In this chapter, we evaluate the basic pattern matching as introduced in Chapter 4.

The outline of this chapter is as follows:

1. **Implementation details.** We start with a brief description of implementation details regarding our evaluation prototype, **GRIFFIN** (Section 5.1). This section covers important implementations aspects, we would like to report.

2. **Micro-Benchmarks.** To in-depth analyze, and optimize each operation we run micro-benchmarks on artificial data (Section 5.2). We report the most important findings. The intention of this section is to share key-properties and insights per approach. The scanning approach is more weighted in this section, since scanning is a core operation of any database system and likely the intuitive choice. Thus, the scanning must be highly optimized for a fair comparison.

3. **Case-Study.** The case-study is our main evaluation for more advanced traversal algorithm that use single step traversals. We evaluate the operators in context of reachability queries on three real-world datasets: a small-size sparse multi-graph, a medium-sized dense simple graph, and a larger-sized dense simple graph (Section 5.3). We show our results regarding both, storage consumption, and query performance for reachability queries on these datasets.

### 5.1 Implementation

This section contains a brief description of our evaluation system **GRIFFIN**\(^1\) including important aspects related to its implementation.

\(^1\)Source repository: https://github.com/graphworx/griffin-core
5.1.1 Griffin

By satisfying the following design principles, GRIFFIN is a reliable and easy-to-use prototype to test and evaluate concepts and strategies for navigation in graph-shaped data:

- **Storage layout-, and processing awareness**
- **Compatibility to relational engines**
- **Effective hardware utilization**

GRIFFIN is a main-memory based prototype system that stores and queries graph-shaped relationship data in a dictionary encoded tabular representation. It utilizes modern hardware, such as high concurrent operations by many core CPU architectures. It is awareness of the tables storage layout, such that the user can switch between row-wise and column-wise representations. As a (micro) relational/graph engine core, GRIFFIN offers querying graph relational data by high efficient, parallelized and memory-bound scanning operations while at the same time supports certain indexes from both categories native and non-native graph storage. Moreover, GRIFFIN can be used to express reachability queries between two nodes over an arbitrary number of hops. The backbone for these reachability queries is a BFS algorithm that utilizes edges-out basic patterns on the graph. GRIFFIN loads graph-data from persistent storage in reasonable time using an optimized binary file format rather than a textural representation.

5.1.2 Internals

The following section provides an overview regarding general internals, used libraries, and information to operations of interest.

5.1.2.1 General

GRIFFIN is implemented in C++ using the C++11 standard. We use template programming to enable aware of several record types, and class inheritances to implement high efficient scan operations on raw data per storage-layout. Data structures in GRIFFIN tightly integrate into the Standard Template Library (STL), e.g., by the use of forward-iterator concepts for external data access. To achieve efficient iterators, we make use of the BOOST framework, namely the `boost::iterator_facade`. The external data access (i.e., the iterator concept) is used for any non-scan operation, e.g., the tree based indexing. By design, the scan-operation is the default implementation for a basic pattern matching query evaluator that can be overwritten when an index structure is used. The BOOST framework is also used when statistics are needed due to `boost::accumulators`. 
5.1.2.2 Memory Management, and Object Creation

Dynamic memory management relies primarily on the unique smart pointer concept, and memory allocation via \texttt{malloc} for mission-critical parts. For non-critical access, we rely on standard containers and standard concepts, such as \texttt{std::vector}, or \texttt{std::array}. When needed, we favor \textit{move} constructors over \textit{copy} constructors.

5.1.2.3 Graph Database Components

A graph database in \textsc{Griffin} is built-up of tree parts: (1) the edge-store, (2) the node value dictionary, and (3) the edge value dictionary. We implement the latter two dictionaries using STL containers (\texttt{std::map}), and use them for graph-structure preserving import operations from textual files. These dictionaries are not considered in our evaluation, since we operate directly on the edge-store data structures. Adding data to a graph database in \textsc{Griffin} is equivalent to add edges to the edge-store. All operations on edges either operate on positions in the edge-store, or on identifiers for nodes and relationships. Types of identifiers are free to be defined at compile time but set to \texttt{unsigned} (32bit on our evaluation machines) for the purpose of this work. Position types are general from type \texttt{size_t} (64bit on our evaluation machines).

5.1.2.4 Import Textural Graph-Data

\textsc{Griffin} operates on a main-memory edge-store only. The content of this container can be manipulated by corresponding functions during runtime. However, the content of an edge-store can be stored and loaded from a (binary) file. This capability is beneficial for operations on the same dataset over several sessions, or for efficient loading of third-party datasets, e.g., the \textit{Enron E-Mail Network} dataset.

Loading a third-party dataset is done by the following process:

1. \textbf{Graph instantiation}. An empty graph is instantiated.

2. \textbf{Dataset parsing}. A dataset-specific parser reads edges data according the dataset-related encoding.

3. \textbf{Enrichment}. Edges are added based on their string values. Internally, these string values are mapped to unique identifiers.

4. \textbf{Persistent storing}. The database contents are stored in a binary file format, one file per dictionary, and one file for the edge-store.

5. \textbf{Loading}. For operations the converted dataset files are loaded into memory. In context of this thesis, we load the edge-store file per dataset.
To provide a convenient way to operator on string values, we add `std::string`-defined edges to the graph database during step (3) rather than only identifier based once. This convenience function utilizes the value dictionaries mentioned before, to identify related unique identifier. Afterwards we operate on the plain edge-store. Both the convenience function, and the value dictionaries are used only to import external datasets in a unified, and robust way.

5.1.2.5 Random Number Generation

Whenever random number generation is needed as for the creation of artificial datasets, we use functionality provided by the STL, e.g., `std::uniform_int_distribution<T>`.

5.1.2.6 Parallel Operations

Parallel operations are implemented by using a static array of `std::thread` objects. The size of this array is set during the compile time by preprocessor symbols.

5.1.3 Summary

We provided details to our evaluation prototype Griffin. We showed our design principles: storage layout-, and processing awareness, compatibility to relational engines, and efficient hardware utilization. Afterwards we give some information regarding the implementation. Griffin is a native compiled main-memory graph database core that mainly operates on an edge-store container which models the graph by its structure. Our evaluation prototype is implemented using libraries, such as STL or Boost.

5.2 Micro-Benchmarking

In this section we share our findings regarding our micro-benchmarks, and pre-evaluation. The advantage of micro-benchmarks is a more in-depth insight in the affect of dataset parameters on the approaches.

This section is structured as follows:

- **Environment.** The experiment environment (such as the hardware) affects the results. We report our environment properties (Section 5.2.1).
- **Dataset generation.** We run our micro-benchmarks on artificial data. Generating graph data is a non-trivial task, and affects the results. We report the requirements, and describe the generation process (Section 5.2.2).
- **Scan Configuration Space.** Scanning is a backbone-operation in any databases, and has a configuration space that can be tweaked (cf. Section 4.1.1). We determine the most suitable configuration for our case-study (Section 5.2.3).
• **Influence of selectivity.** The amount of data that is returned by a query affects the query execution times. In this section we in-depth analyze how the selectivity affects the approaches under study (Section 5.2.4).

• **Scalability.** The size of the graph is determined by the number of edges. This micro-benchmark addresses how the graph size affect the execution (Section 5.2.5).

We outline here, that the micro-benchmarks focus on one single step traversal operation. Since it is likely that more complex traversal algorithms (such as breadth-first search) run thousand of single step traversal operations, each single step traversal operation must be as efficient as possible to provide a good base for complex traversal algorithm (cf. our case study in Section 5.3).

### 5.2.1 Environment

We run our experiment setting on two hardware settings: our development machine, and a dedicated server for the final results. The first one can be seen as commodity hardware while the last one serves as more professional hardware.

Our **commodity hardware** configuration is: MacBook Pro, Mac OS X 10.11.4 64bit (Build 15E65), Darwin Kernel Version 15.4.0, 2.2 GHz Intel Core i7, 4 cores, 8 threads (4 threads with hyperthreading), RAM 16 GiB 1600 MHz DDR3, L1 cache 32KiB, L2 cache 245 KiB, L3 cache 6MiB, SSD hard drive.

Our **server machine** configuration is: Ubuntu 14.04.3 LTS, 64bit, 2 x Intel Xeon E5-2609 v2 @ 2.5 GHz, 4 cores per CPU = 8 cores, 8 threads (without hyperthreading), RAM 256 GiB, L1 cache 256 KiB, L2 cache 1MiB, L3 cache 10 MiB.

For our reported results, we set the compiler optimizer level to the highest (-O3). If we report mission-critical tasks (e.g., the query times), we use the server machine, since this reflects a real-world setting. For non-mission-critical tasks (e.g., the initial conversion from textural dataset representations to binary files), we run this on commodity hardware for sake of simplicity. We always mark which hardware setting was used.

### 5.2.2 Dataset Generation

In the following, we highlight the requirements our dataset generator must satisfy, and provide a brief overview over the dataset generation process.

#### 5.2.2.1 Requirements

To determining what the each approach stresses, we have to be able to control the following:
1. **Scaling the number of edges.** This affects approaches that depend on the total number of edges in the graph (e.g., the scanning).

2. **Node- and relationship diversity.** This affects the selectivity when querying certain nodes, or relationships (e.g., the number of distinct nodes).

3. **Out- and in-degrees.** This affects the size of the neighborhood per nodes, and the density of the graph (e.g., a larger out-degree implies a larger edges-out resultset).

### 5.2.2.2 Data Generation

Our data generator $G$ gets two user-defined numbers: $n$ (the number of nodes), and $r$ (the number of relationships). Moreover, $G$ expects the targeted number of records $m$ to be created, as well as the out-degree $d$ for nodes in the graph. After invoking $G$, the result is a list of $m$ triples ready-to-import into our evaluation prototype GRIFFIN. After importing the data into GRIFFIN, the index might be created immediately. However, invoking $G$ with the given set of parameters will generate the result list in the following way: (1) we set the expected out-degree $d_n = m/d$ per node, (2) for each $i = 1, 2, ..., d_n$, we choose a source node identifier $s$ randomly drawn following a uniform distribution $U(1,n)$, followed by the generation of $d$ edges $(s,t,r)$ where $t$ and $r$ are also drawn following a uniform distribution over $n$ or $r$ respectively, (3) we ensure that there are no duplicates in the result list, and replace them if necessary. The last step (4) is to shuffle the list, such that the records positions are randomized.

The scale requirement is directly controlled by the parameter $m$. The node, and relationship diversity is a consequence of the uniform distribution when choosing the components for each record. The lower $n$ and $r$ are, the more likely is it to occur more often as part of a record. The out-degree is directly controlled over the parameter $d$, whereas the in-degree is randomized but can be changed afterwards if necessary by replacing the related components. Depending on the query and the required selectivity we replace records in the graph to match the number of results to achieve the targeted number of records to be returned. Obviously, this also affects $d$, but we do not experienced this effect as critical.

### 5.2.2.3 Measurement

The core of our micro-benchmark measurement function is based on three functionalities: (i) loop over the domain of independent variables given some step-size (e.g., increase selectivity by 5%), (ii) construct graph data using the data generator given a set of properties and the actual value of the independent variable, (iii) execute the given basic pattern using a given approach. Each experiment was rerun multiple times to receive stabilized result.
5.2. Micro-Benchmarking

We measured for each such run a certain variable, e.g., the query execution time. Based on several observations defined by the number of reruns, we constructed statistics on these values (e.g., minimum, maximum, median, and mean). We always consider the standard error in our samples to ensure confidence in our measurements. However, the last step per experiment run is to perform a clean up before the next experiment was started.

5.2.3 Scan Configuration Space

The scan operation is a universal restriction operation and a fundamental physical operator for databases. In contrast to traditional databases, we can operate on one pre-compiled table consisting of three columns known in advance. In this micro-benchmark, we consider the scan configuration space: the storage layout, and the access pattern.

Setup
We construct an artificial graph dataset with varying edge-count, selectivity values, and graph dataset parameters (e.g., the nodes out-degree).

Observation
We observed a column-wise layout as beneficial in terms of performance for large datasets and important basic patterns, e.g., the edges-out pattern as used in reachability queries.

In terms of access pattern, we observed that the branching strategy slightly outperforms the bulk-processing strategy for small to medium sized graphs. For bulk-processing, we further observed how the per-column selectivity affected this access pattern, i.e., on our generated data the bulk-processing was some times dominated by the branching strategy even on large datasets.

Explanation
The effect regarding a more performant branching strategy for some cases can be explained due to the CPU data cache, and the branch-prediction unit of the CPU. Small dataset sizes enable the database to load the majority of the graph data into the CPU data cache if the data is small to medium. If there is a predictable pattern when to choose which path for conditional code blocks, the branch-prediction unit can forecast this decision which leads to better execution times.

Regarding per-column selectivity affects on the bulk-processing, the reason is that our bulk-processing evaluation does not consider a selectivity aware column order during the predicate evaluation. This means, we strictly evaluate the first column first, followed by the second, and followed by the third. Let $n$ be the number of tuples in a table. If the first column pass $n_1 \approx n$ tuples, and the second column pass $n_2 \approx n_1$ of tuples during evaluation, while only a small number $n_3 \ll n$ of tuples satisfy all three conditions, we
inherently read and write to many positions, namely \( \approx 2n \) instead of \( n^3 \). This behavior is an implementation-artifact, and we are optimistic to improve the bulk-process even further by the usage of selectivity estimators that can be utilized to determine the column evaluation order.

**Interpretation**

We observed that for reachability queries using the edges-out basic pattern, neither this artifact, nor the per-column selectivity nor the access pattern must be considered since for edges-out queries only one column is considered. Hence, the evaluation for both branching- and bulk-processing strategy does not differ, and the column evaluation order does not matter.

For universal basic patterns, however, these options must be considered since there are typically more than one column involved.

**Wrap-Up**

We wrap-up our findings regarding the scan configuration space:

- A column-oriented storage layout suites for large datasets.
- Bulk-processing evaluation suites for large datasets.
- Branching evaluation suites for small and medium datasets.
- The column order must be considered if more than one column is involved.
- A horizontal partitioning for parallelization should be favored in general.

**5.2.4 Influence of Selectivity**

Since we are able to control the dataset generation, we can create graph data such that a query returns a certain subset of the graph. Thus, we can measure the affect of selectivity to the scan, and the indexes.

We expect a tiny selectivity value for most navigation-related use-cases, e.g., path finding in sparse graphs for social network analysis. However, we are interested in how the selectivity affects the query execution performances, in general. In Figure 5.1 on page 57 we show the execution times on an edges-out query depending on the selectivity value. We note here, that we run this experiment on a 1 GiB dataset (\( \approx 90mio \) egdes), and that a single basic pattern call (e.g., edges-out) will by typically one of many steps in a traversal query. Thus, a minor increase in the query execution time would be a major penalty for a series of these calls.

**Setup**

We constructed an artificial graph dataset of 1 GiB data per selectivity value.
5.2. Micro-Benchmarking

![Figure 5.1](image)

**Figure 5.1**: Affect of selectivity value to execution time on an ≈ 90mio. egdes graph (executed on server). 100ms is marked as threshold for interactive queries. The selectivity for which the scan exceeds this threshold is also marked.

**Observation**

With increasing selectivity all approaches require more time to answer the query as expected. This decrease in execution performance is linear for all approaches but the slopes are different. The strongest performance decrease is for scanning. The other approaches lose slightly performance even when nearly all edges must be returned. In fact, we observed the indexes were able to answer the query in less than 100ms independent of the selectivity value. Thus, the indexes are independent of whenever they are native or non-native are promising options for large-scale graph datasets with respect to performance.

**Explanation**

We explain the difference between scanning and the indexes as follows: On the one hand, scanning requires to touch each tuple independent of whenever it will satisfy the predicate or not. This inherently requires a transfer of 1 GiB from memory to the CPU. All positions of satisfying tuples must be written to the resultset, before the memory address of this resultset can be returned. After returning the resultset address, its dereferenced content is copied to the querys final resultset. Since the amount of data to scan remain the same, the performance decreases with increasing selectivity. This is caused by the effort it takes for write-operations of relevant positions in the resultset resp. the final resultset. For selectivity values near zero, the differences between scanning and indexing can be explained by the effort of reading and testing 1 GiB of data when using the scan. On the other hand, each index pre-computes the query answer
in advance. Returning the final result is returning the memory address of this pre-
computed collection. Since the effort of finding the requested pre-computed collection
should be constant due the fixed number of edges, we expected the indexes not to
decrease their performances first. This increase can be explained due to the effort it takes
to write the dereferenced pre-computed set to the final resultset. Thus, the decrease of
the indexes performance is due to the higher number of elements to write to the final
resultset. We are optimistic one could even remove this last selectivity-depending effort
for indexes such that the performance is constant. This optimization requires that the
engine ensures non-freed pointers during query processing such that the last copy-step
is not necessary. However, for tiny selectivity values the effect of selectivity-dependent
copying effort is not significant.

Interpretation
With the exception of scanning, all approaches can be used for interactive queries
requiring execution times less than 100ms when less than 90% of the data must be
returned. However, scanning can be taken into account for very-tiny selectivity values,
or on very tiny graph datasets, but is not competitive compared to indexes, especially
on large datasets where a high number of basic pattern calls are required for a given
traversal operation.

Wrap-Up
We wrap-up our findings regarding the influence of selectivity:

- All indexes achieve similar execution times.
- Copying resultsets from indexes dominates their execution time.
- Optimized index access rely on copying addresses to pre-computed results.
- Scanning inherently requires to write the resultset on the fly.
- Indexing outperforms scanning due to resultset writing.
- Real-word datasets likely deal with very tiny selectivity values (per node/edge).

5.2.5 Scalability
In this micro-benchmark we are interested in how an increasing number of edges (with
a fixed selectivity) affects the performance for indexes, and scanning. We depict in
Figure 5.2 on page 59 the results regarding the edges-out query. We show the results
on a tiny to small scaling graph (Figure 5.2 left side), on a small to medium graph
(Figure 5.2 middle), and on a medium to large graph (Figure 5.2 right side).

We used this experiment to further optimize the indexes regarding result returning.
Hence, we show the results when using a scan, an index with result-passing by value,
and an index with result-passing by reference. More specific, returning the result by
value is returning a constant reference to a resultset that was written based on the pre-computed result in the index.

**Observation**

As expected, all approaches are affected by an increasing number of edges in the graph such that it takes longer to answer the query. The strongest decrease in performance can be observed for the scan approach. For an interactive query, which requires less than 100ms for an answer, the scan is no option if there are more than 14 mio edges. In contrast, using an index (independent of whether it is a native, or non-native one), interactive queries gain 5 times more edges compared to a scan. When the result is passed by reference, and no further writes to the resultset are required, the indexes will be interactive until approximately 200 mio edges (14 times more edges compared to scanning).

**Explanation**

As for the scalability micro-benchmark, all approaches are affected by the effort of writing the final result in general. Hence, there is an inherently performance decrease when the number of edges grow. In addition, there is an additional effort to write the resultset when using the index result passing by value. Consequently, we can observe a nearly constant difference between the indexes using values, and indexes using references. This constant difference is the effort it takes to write the resultset. Writing the resultset before the final resultset is created is not necessary for indexes which use result passing by reference. However, on tiny to small graphs (left figure) scanning outperforms the index passing by value, and is competitive to any index for less than 0.1 mio edges. There is a threshold at approximately 6.3 mio edges where scanning is no reasonable option anymore (middle figure). For large-scale graphs, the only reasonable option is an index, passing the result by reference. We explain the (nearly constant) performance difference of 50ms between index result passing by reference, and index result passing by value with the following two aspects: First, the index with result
passing by value requires to copy the pre-computed set from the index to the query's resultset. Copying a certain amount of data is inherently slower than copying just an memory address. Second, in our implementation we use `std::back_inserter` from the STL to append values to the resultset for the index passing by value alternative. Hence, a certain amount of the effort responsible for the constant difference based on the sub-optimal performance of `std::back_inserter`.

**Interpretation**

The effect of `std::back_inserter` is hidden for the index passing results by reference since this effort occurs during the building time of the index rather than during querying. For scanning, however, we do not use the STL, i.e., we use `malloc` to dynamically allocate memory, and plain array write-operations which is significantly faster than the alternatives provided by the STL. However, indexes have to support the result by reference passing in order to achieve high performances. Unfortunately, using reference passing for the scan operation is not achievable due to the resultset must be written during evaluation of the predicate. Therefore, scanning is not a valid option for a high scalability value.

**Wrap-Up**

We wrap-up our findings regarding scalability:

- Index structure engineering must consider copying of and referencing to results.
- One must consider performance impacts due to indirections in the STL.
- A high-efficient index structure uses reference-passing only.
- For small graph datasets, scanning can be considered as an option.
- The larger the graph, the lesser it is reasonable to consider scanning as an option.

### 5.3 Case Study: Reachbility Queries

Navigation in graph-shaped data is one of the key-properties of graph databases. As one prominent example for traversal queries, path-finding rely on single step traversals. Since a path-finding between certain nodes typically requires the exploration of a huge amount of node neighborhoods during the evaluation, each traversal step must be as efficient as possible to enable high performance.

Typically, a generic path finding algorithm on graph-shaped data involve conditions to edges (e.g., a certain type of relationship), to nodes (e.g., a certain node type), and to directions (e.g., ignore direction of edges). Path-finding queries return an ordered set \( P(u_{\text{start}}, u_{\text{end}}) = \{v_1, v_2, \ldots, v_n\} \) in respect to a path \((u_{\text{start}}, v_1, v_2, \ldots, v_n, u_{\text{end}})\) between two given nodes \(u_{\text{start}}\), and \(u_{\text{end}}\) (if any).
One specialization of path-finding is reachability as the decision variant of path-finding. In contrast to path-finding that returns a path, reachability queries ask if $P(u_{\text{start}}, u_{\text{end}}) \neq \emptyset$, and thus return a boolean value.

Reachability queries on directed, simple graphs. A more navigation-focused version of reachability queries, are reachability queries on simple, directed graphs. This type of reachability queries are characterized by

- the starting node $u_{\text{start}}$, and the ending node $u_{\text{end}}$
- the direction of edges between nodes, i.e., a directed path between $u_{\text{start}}$, and $u_{\text{end}}$.

In contrast to the more general version, this type of reachability query does neither constraint edges to have a certain relationship type, nor nodes or edges to have a certain property. Therefore, reachability queries are an excellent choice to evaluate the concepts of this thesis since they stress the navigation capabilities of any graph database but are not limited to a certain (more advanced) graph model, such as RDF, or property graphs.

5.3.1 Measurement

In Listing 5.1 we depict a source code listing that contains our measurement process for reachability queries.

**Measurement process.** We first load a binary-converted, optimized real-world dataset containing the datasets edges $E$ into our evaluation prototype utilizing a certain strategy $S$. The strategy $S$ is either a native, or non-native approach as mentioned in Chapter 4.
Afterwards, we measured query executions for our experiments as follows: After initialization we execute a reachability-query \( P(u, v) \) between two randomly chosen nodes \( u \) and \( v \) in \( E \).

We consider the query execution time as the time duration between entering \( P(u, v) \), and returning from \( P(u, v) \).

The time duration is measured using `std::chrono::high_resolution_clock` that fits best for benchmarking, since this type of clock is intended to be realized by using a high-resolution platform-dependent clock.

**Measurement values.** We limit the maximum search depth for the reachability query to a value \( \text{depth} \) between 1, and an upper bound (e.g., 8). Hence, the reachability query starts by \( u \), and explores the neighborhood recursively to a maximum hop count of \( \text{depth} \). The query stops, if (i) \( u = v \), or (ii) \( v \) is found over hops less or equal to \( \text{depth} \), or (iii) no node until a hop count of \( \text{depth} \) starting by \( u \) contains \( v \) in its neighborhood. We rerun this process as determined by a number \( \text{rerun} \) (e.g., 200). For each reachability query, we report (a) the actual run number, (b) the actual depth, (c) the dataset \( E \), (d) the duration, and (e) results containing information about (1) the reachability of \( v \), (2) the number of basic pattern invoking in \( P(u, v) \), and (3) the maximum number of nodes considered to be explored next. Based on these reportings, we computed statistics that we used for our observations (e.g., median execution time, or standard error on execution time).

**Reachability algorithm.** A reachability query \( P(u, v) \) between nodes \( u \), and \( v \) is implemented using an iterative BFS algorithm \( A \). The algorithm \( A \) explores node neighborhoods \( \mathcal{N} \) by the use of the strategy-dependent implementation of the edges-out query, and a working queue containing pairs \((n, d)\). Each pair \((n, d)\) in the working queue holds a node identifier \( n \), and a current depth \( d \) relative to \( u \). \( A \) starts by enqueueing the pair \((u, 1)\) into the working queue. For each step until the working queue is empty, the front pair \((n, d)\) is dequeued. An edges-out basic pattern \((n, *, *)\) is invoked that returns a position list of edges with adjacent nodes relative to \( n \). If \( v \) is the end-node for one of these edges, \( A \) returns yes. Otherwise, each node \( m \in \mathcal{N}(n) \) is added to the working queue as \((m, d+1)\) if \((d+1) < \text{depth} \). If the queue is empty which happens if \( \mathcal{N} \) becomes empty for considered nodes, or the depth limit is reached, \( A \) returns no.

### 5.3.2 Datasets

So far, we introduced basic pattern queries, and evaluated these queries on artificial generated graph data. Therefore, we observed how certain parameters, such as the selectivity, affects a basic pattern query, and hence a single step in a navigation query. For an evaluation beyond an evaluation of a single step, more advanced dataset characteristics matter when considering queries involving multiple single steps, e.g., reachability queries.
Table 5.1: Reachability Dataset Network Statistics

<table>
<thead>
<tr>
<th>Property</th>
<th>FB</th>
<th>EEN</th>
<th>GOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Type</td>
<td>Social</td>
<td>Communication</td>
<td>Web</td>
</tr>
<tr>
<td>Description</td>
<td>Social circles</td>
<td>E-Mail Network</td>
<td>From Google</td>
</tr>
<tr>
<td>#Nodes</td>
<td>2,876</td>
<td>36,692</td>
<td>875,713</td>
</tr>
<tr>
<td>#Edges</td>
<td>4,199</td>
<td>367,662</td>
<td>5,105,039</td>
</tr>
<tr>
<td>#Relationship Types</td>
<td>45</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>data set size</td>
<td>4.9 MiB</td>
<td>4.1 MiB</td>
<td>75.4 MiB</td>
</tr>
<tr>
<td>diameter</td>
<td>8</td>
<td>11</td>
<td>21</td>
</tr>
<tr>
<td>effective diameter</td>
<td>4.7</td>
<td>4.8</td>
<td>8.1</td>
</tr>
<tr>
<td>Origin</td>
<td>Mcauley [ML14]</td>
<td>Leskovec [LLDM09]</td>
<td>Leskovec [LLDM09]</td>
</tr>
</tbody>
</table>

Real world datasets. One of a more advanced characteristic is the longest shortest path in the graph, i.e., the maximal number of hops between two given nodes if they are connected over some transitive path. If this number is unconstrained, a reachability query must explore the entire dataset in worst case before answering no. If this number is constrained on the other hand, the search can be stopped at a certain depth. In addition, each nodes degree influences the number of neighbors that might be considered during evaluation. Generating artifical data that consider such characteristics is out of the scope of this work. Hence, we focus on real-world datasets. In Table 5.1 we show datasets which are used for this case study:

1. The Facebook (FB) dataset is a small graph, that covers (anonymous) social network users, and their friendship. Friendship is determined by per-user friend list indicating one user as a certain friendship to other users. In addition to friendship information, certain properties are attached to users, e.g., the interest in a specific topic or the belonging to a certain group.

2. The Enron E-Mail Network (EEN) dataset is a medium graph, and covers e-mail communication. E-mail accounts are nodes. If there was at least one e-mail sent between two accounts $u$, and $v$, there is an edge between $u$, and $v$.

3. The Google Web Graph (GOG) dataset is a larger graph covering hyperlinks between web pages. Nodes are web pages, and edges are links between web pages. The dataset was released as part of the Google Programming Contest in 2002.

---

2https://snap.stanford.edu/data/egonets-Facebook.html
3https://snap.stanford.edu/data/email-Enron.html
4https://snap.stanford.edu/data/web-Google.html
Both, the EEN dataset, and the GOG dataset are directed, but simple graphs involving only one relationship type while FB based on a multi-graph involving several types of relationships.

**Importing datasets.** We imported each of the datasets using a custom parser, that import textural graph-data to Griffin. As mentioned in Section 5.1.2.4, we load afterwards a binary stored, and read-optimized version of these datasets, and operate on the edge-store ignoring eventual concrete node- or relationship (string-) values. Each dataset was interpreted, and imported as follows:

- Both EEN, and GOG are described by a single American Standard Code for Information Interchange (ASCII)-encoded file each. This file contains a pair \((u, v)\) per line. Each such pair models a directed edge from a node \(u\) to a node \(v\). Hence, we imported \((u, r, v)\) as an edge to a corresponding graph in Griffin. We select \(r\) as one single, unique, and arbitrary relationship modeling the only edge type in these datasets since both datasets rely on simple graphs.

- The FB dataset is described by several ASCII files covering friend-circles, edges, ego features, features, and feature names. A friend-circle for a user \(u\) is described by a list tuples \(c_1, c_2, ..., c_m\). Each \(c_j\) is a tuple \((v_1, v_2, ..., v_n)\) with a unique circle-id \(id(c_j)\), and \(v_i\) is a user node in the circle (for \(i = 1, ..., n, j = 1, ..., m\)). We add edges \((u, id(c_j), v_i)\) to the corresponding graph for each \(c_j\), and each \(v_i\) in the friend circle \(c_j\). An edge in in FB is a list of tuples \((r, v)\) per user \(u\). We add edges \((u, r, v)\) accordingly for each contained edge in FB. Features in FB describe properties, such as the users attraction to a certain movie, or political interest. In the FB dataset features are encoded as a dictionary that maps feature names to identifiers. These identifiers are used afterwards to decode features as feature vectors to objects in the FB dataset. Therefore, we add edges \((u, f_t, f_v)\) for users \(u\) containing a feature type \(f_t\), and a feature value \(f_v\) if the corresponding feature vector describes such a property. We explain the observed data compression from the ASCII-encoded graph to the binary encoded graph by the inefficient encoding of feature vectors in the original dataset.

Since our measurements based on the converted datasets rather than the original ASCII-encoded once, we summarize the outcome of this importing process in Table 5.2. Besides the number of edges, and the internal graph-structure, the dataset size of the graph is important. We report the compression ratio, which is greater one if the compression technique achieved space savings. The compression ratio is defined as follows:

\[
\text{compression ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}
\]
We either minimized the dataset size for edge-data due to the dictionary compression to a compression ratio up to 98.0, or we do not significantly inflate the size. The compression ratio is affected on how the edges where encoded in the original data. The more repeated strings are used in the ASCII encoded datasets, and the higher the storage difference between ASCII-encoded values, and Griffin internal identifiers are, the better we can compress. We provide the according node-, and relationship dictionary sizes which reflect this circumstance. We explicitly mention here that our netto graph dataset size is the size of the edge-store.

### 5.3.3 Dataset Impact

In this section we report our findings on how the real-world dataset affect the reachability query in terms of (a) the set considered nodes, (b) the number of edges-out queries invoked, and (c) the relative number of successfully reached nodes during a reachability call. We report this observations first since this outcome is necessary to understand the affects of datasets to the reachability queries.

#### 5.3.3.1 Maximum Queue Size

Invoking a reachability query $P(u, v)$ between nodes $u$, and $v$ using a BFS algorithm requires to determine the neighborhood of the head node in the working queue each step. During the query execution, the size of this working queue grows and shrinks depending on the number of neighborhood nodes in respect to the current node. The
maximum queue size determines the maximum number of intermediate results during the evaluation, and thus the maximum storage required to run the algorithm.

In Figure 5.3 (a) we show how how the maximum queue size is affected by the number of maximal hops per dataset.

**Expectation**
We expect the maximum queue size to have an upper bound when there is no node with an extraordinary large neighborhood, and if the graphs effective diameter (the longest shortest path for 90% of the data) is less or equal to the maximum hop count. The maximum queue size is also bound, if there are many „dead ends“ per node, i.e., ending nodes are only connected to starting nodes.

**Observation**
We observed that the maximum queue size is bound to a low value for the FB dataset, and to a more larger value for the EEN dataset. In addition, we observed that the maximum queue size grow exponentially for the FB dataset with not upper bound observable up to a hop count of eight.

**Explanation**
This can be explained as follows: Both the FB, and the EEN dataset have an effective diameter of 4.75 ± 0.5. Thus, for both the maximum queue size does not grow unbound at a hop count greater or equal five. FB has many „dead end“ nodes, and therefore the maximum queue size is strictly bound since there are no more adjacent nodes to explore once a „dead end“ is reached. Consequentially, the queue size shrinks, and the maximum queue size is around the effective diameter. In contrast, the maximum queue size grows beyond the effective diameter for the EEN graph since, the longest shortest path is
5.3. Case Study: Reachbility Queries

actually eleven. Therefore, it is likely to reach most nodes with five hops if they are reachable. Likewise, there are some nodes which can only be reached with more than five hops. Therefore, the maximum queue size for EEN grows slightly for a hop count more than five. In contrast to EEN, and FB, the GOG dataset has a longest shortest path of 21, and an effective diameter larger than eight. Consequently, the maximum queue size is unbound to this hops count. It exponentially grows further since for each found node, the number of nodes to be explored next is high compared to EEN and FB. The reason is that the GOG dataset is a web graph which pages (nodes) have a high number of links (edges) to other pages.

**Interpretation**

The maximum queue size per hop count indicates the maximum number of intermediate results to this hop count. To avoid costs for virtual memory swapping when free virtual memory is low, the maximum queue size should be bound to a maximum limit. This, however, depends on the dataset, and not the algorithm. Therefore, the google dataset, as one representative for web graphs, is a candidate dataset that produces a high number of intermediate results.

5.3.3.2 Basic pattern invoked

Each step during a reachability query requires to determine the neighborhood $\mathcal{N}(u)$ for the current current $u$. The determination of $\mathcal{N}(u)$ is edges-out basic pattern $(u, *, *)$. The more edges-out queries are required for one reachability query, the higher is the penalty of poor performances for a single edges-out query. The number of edges-out queries to be invoked depends on the cardinality $|\mathcal{N}(u)|$ for each considered node $u$. Ideally, a graph database system is performant enough to answer reachability queries even if the graph is dense, i.e., the average $|\mathcal{N}(u)|$ is high.

In Figure 5.3 (b) we show how the number of edges-out queries calls depends on the number of hops for each dataset.

**Expectation**

We expect a higher number of calls for denser graphs, and a lower number of calls for sparse graphs.

**Observation**

We observed that with increasing number of hops, more edges-out queries are in invoked as expected. For both the FB dataset, and the EEN dataset, this number has an upper bound as for the maximum queue size. We further observed the EEN datasets significantly has a far smaller maximum in comparison, while both FB and GOG have no such significant difference.

**Explanation**

As for the maximum queue size, the diameter is one factor which influences the number
of edges-out queries to be invoked. Consequentially, the plot for the edges-out query call number is similar to the plot for the maximum queue size in respect to its shape. Regarding the observed maximum, we explain this as follows: The number of edges-out invoked depends on the number of neighbors per node. At an arbitrary step during the BFS there is a node $u$ which is the head of the working queue. For this $u$ its neighborhood $\mathcal{N}(u)$ is queried using one edges-out query. For each node $v \in \mathcal{N}(u)$, $\mathcal{N}(v)$ is determined at some point. If the algorithm does not stop by finding the requested end node, the number of potential nodes to explore increases exponentially in worst case until the depth upper bound is reached. This effect can be seen on the GOG dataset. However, on both EEN, and FB, either the search stops for a certain path due to a „dead end“, or it is more likely to find the requested node the higher the hop count is.

Considering the high maximum queue size but the relatively small number of edges-out calls for EEN, we can explain the differences in the maximum of the EEN lines for both plots as follows: nodes in EEN are far more connected than on the other two datasets. Consequentially, invoking a single edges-out query results in a larger number of nodes for the next step. However, in contrast to GOG the number of edges-out query calls do not grow exponentially since the probability to find the requested node grows per each hop.

**Interpretation**

A higher number of edges-out calls make bad-performing implementations for edges-out queries even worse. The number of edges-out calls depends on the density of the graph, as well as the diameter. The more dense a graph is, the more queries must be invoked. As higher the diameter is, as less likely is it to find the requested node. Again, the GOG web graph mostly stresses reachability queries although it is a relatively sparse graph. This is due to a low probability to find the requested node in the given hop count limit of 8 hops. An even more harder graph in this terms would be in addition more dense than a web graph is.

### 5.3.3.3 Reachability Success Rate

A reachability query $P(u, v)$ returns *yes* if there is a path from $u$ to $v$, and otherwise *no*. For a bounded reachability query, the path length must be less than a given upper bound *depth*. Reaching $v$ terminates the query. If $v$ can be reached at a hop count significantly less than *depth*, one can expect stable query times even for unlimited reachability queries. Consequentially, a higher reachability success rate maximizes the probability of earlier *yes* answers to $P(u, v)$, and thus minimizes the query execution times.

In Figure 5.3 (c) we depict the relative number of successfully reached nodes for random reachability queries.
5.3. Case Study: Reachbility Queries

Explanation
We expect a small number of successes for web graphs, and a higher number for the other graph datasets. Moreover, we expect a stable success rate around the effective diameter per dataset. In addition, we do not expect to reach a 100% success rate since some nodes are not feasible from arbitrary start nodes.

Observation
We observed a low success rate for GOG, as expected. This number increased slightly as higher the number of hops are. For the EEN dataset, we can observe that about 80% of reachability queries successfully lead to a yes response for a random end node. As expected, we see a stable rate about the effective diameter of 5 hops. As for EEN, the FB dataset stabilizes the success rate about its effective diameter of 5 hops, but with a success rate about 30% rather than 80% as for EEN.

Explanation
The relative success rate of readability queries determines how likely it is to terminate the query more earlier with a yes response before the upper bound depth is reached. This probability relies in the dataset. If the requested node is in the transitive closure of the starting node up to a given upper bound, the answer will be yes, otherwise no.

Interpretation
On the one hand, a low success rate does not necessarily lead to low query performances since the success rate is affected by the number of unfeasible nodes, and „dead ends“. On the other hand, a high success rate leads to better query performances in average, since it is more likely to earlier receive a yes response, and thus stop the search. As a consequence, web graph data stresses reachability query most. In web graph datasets, there is likely a small amount of „dead ends“while at the same time, the diameter is high such that receiving an early yes cannot be expected.

5.3.4 Query Performance
The key to efficient graph navigation is high-performance query execution. While we consider how the datasets characteristics, and the number of maximal hops influence the BFS algorithm in terms of intermediate results, number of basic patterns required, and probability to early terminate the reachability query with a yes response, this section covers how the certain native and non-native evaluation strategies that can be used to realize the BFS affect the query execution time.

In Figure 5.4 we show the query execution times depending on the number of hops (the maximal path length) for reachability queries. In the left plot of the figure, we show the results for the EEN dataset, in the middle plot we show the results for FB dataset, while the right plot covers the GOG dataset. For each plot, we show the query execution times for the native approaches (the adjacency list, and the object cache),
and the non-native approaches (the scan operation, and the tree-based index). The lower the execution time, the better the result is. We show the mean execution times for reachability queries looking for the requested node in most #hops. Hence, if the requested node is not found, a path of #hops length is fully explored. If, on the other hand, the requested node is found, the path has at most a length of #hops. We chose this, since this reflects a typical case for reachability queries. However, we consider an approach as interactive, if it achieves to answer a reachability query in less than 100ms.

**Expectation**

Based on our observations on the singe-step traversal performance for the individual approaches, we expect a poor performance for the scan-based operation, and superior performances for the indexes. Likewise we expect similar results for the indexing approaches, independent of whenever it is a native, or a non-native approach. Since our in-depth analysis of the previous section (Section 5.3.3) revealed significant effort to evaluate reachability queries on the GOG dataset, we expect the worst performance for the GOG dataset.

**Observation**

Independent of the dataset, the scan operation performs worst. With the exception of the FB dataset, the scan operation is not even acceptable for an interactive query. Besides the scan operation, all indexes perform similar. In fact, all indexes are acceptable for interactive queries, even for the GOG dataset. More in detail, we observe query times below 1ms for EEN and FB, as well as below 25ms up to 4 hops, and below 50ms up to 8 hops for the GOG dataset. Moreover, we observe a similar grow for all approaches on the GOG datasets but with different factors: all approaches grow exponentially. The scan grows at most such that it exceeds a reasonable query time very quickly. The indexes, however, are likely to exceed the interactive limit for hops greater than 10. Finally, we observe on each dataset all approaches start with a similar query time for 1-
5.3. Case Study: Reachability Queries

hop queries which we do not expected. In fact, we expected a higher execution time for the scan operation already for the 1-hop query based on our outcome in Section 5.2.4.

Explanation and Interpretation

Our observations confirm our expectations. The scan operation is not competitive to its alternatives. As all approaches, the scan is affected by the number of edges in the graph, the selectivity, as well as the impacts of the datasets in respect to reachability query execution. Thus, datasets requiring a higher number of basic pattern calls make the worse performance of the scan even worst. Especially in the GOG dataset, scanning is no option. For the FB dataset, however, the scan is still interactive but this is due to FB requires a small number of edges-out, a small number of intermediate results, has a short diameter, and even a shorter effective diameter. For a single operation (e.g., a hop limit of 1), the scan is competitive for the presented datasets. This is caused by the relative small graph size which nearly fits into the CPU data cache, and the equal number of data to be processed for each approach. A general purpose engine, however, should not rely on scanning from a performance perspective when real navigation capabilities are required. Thus, indexes should be favored. We cannot observed a significant performance difference between native and non-native approaches in a main-memory context on our test datasets. This is due to finding a requested node in the edge-store (e.g., for the edges-out query) by the use of indexes is either done via direct-access (i.e., accessing the \(i\)-th field in the object-cache, or in the adjacency list), or via binary search. Even on \(10^9\) edges stored, binary search requires around 30 steps to determine the corresponding record in the edge-store. We could, however, not find a significant aspect not to rely on binary search for reachability queries from a performance perspective on our artificial- and real-world dataset.

5.3.5 Storage Consumption

![Storage Consumption Diagram](image)

**Figure 5.5:** Storage consumption on real-world datasets depending on hop count.
High-performance query execution can be always achieved to the cost of memory, e.g., by pre-computing results. There is a trade-off between the need of high-performance query execution on the one hand, and limited memory on the other hand. Therefore, we examine the query execution strategies introduced in this thesis in respect to storage consumption. We do not focus optimizing storage consumption. Consequentially, we show how costly each approach is with the option of further improvement. In Table 5.2 we report the index creation times. The indexes were built after the binary-optimized edge-store file was load into memory. The time it takes to build an index was always less than the time it takes to load the binary-optimized file from the disk. Since we focus on read-only workloads, we do not consider the update costs. However, along with the results of Section 5.3.4, we can better judge which approach is promising, and which are not in context of main-memory databases.

In Figure 5.5 we show how the storage consumption for the EEN dataset (left), the FB dataset (middle), and the GOG dataset (right) is influenced per basic pattern match query by the indexing approaches introduced in this thesis. Our baseline in the plain database (i.e., edge-store) size on which a scan operation can be executed without further space requirements. Since the indexes refer to positions in the edge-store, and the edge-store is used to materialize triples at a certain positions, the edge-store must always be presented. Thus, in our evaluation system, the edge-store size is always a lower bound for the storage consumption, independent whether an index is used or not.

**Expectation**

We expect that native approaches require more memory compared to the non-native tree-based index per basic pattern query in general. This is due to the native approaches rely on direct access to an array of node identifiers. Hence, their storage requirement increases linear with linear increasing number of nodes. In contrast, the tree-based indexing approach depends on the number of edges logarithmically but not directly on the number of nodes. Some indexes do not accelerate any basic pattern query, and thus, are not available (N/A) for not supported query types (compare Table 4.1).

**Observation**

On the EEN dataset (left), we observe the plain storage consumption of the original ASCII-encoded dataset, and the main-memory binary optimized counterpart is nearly identical (compare Table 5.2). We observed the tree-based approach requires about 60% additional storage for all basic pattern matching queries with the exception of the exact-match-, and the connected-by query. The exact-match query index consumes about 200% additional storage, while the connected-by query consumes approximately 150% additional storage. The object cache index requires additional 25% more storage than the tree-based counterpart for the same basic match queries. The adjacency list only supports edges-out, and edges-in queries on which it is between the storage consumption of the tree based index, and the object cache. On the FB dataset (middle), we observe
a high compression rate of the working data compared to the original textual graph data. However, the tree-based approach consumes about 20% additional storage which is not a critical mass compared to the other approaches. The object cache requires 4000% compared to the original data, while the adjacency list is between the tree-based approach, and the object cache. However, non of the approaches exceed the original data size limit. On the GOG dataset (right), we observed a reduction of 17 MiB (22%) from the original textual dataset to the main-memory variant. As for the EEN dataset, the tree-based approach consumes most storage for indexing exact-match queries, followed by connected-by queries. The exact match requires about 160% additional storage, while the connected-by index requires about 125% additional storage. The other basic queries are indexed by the tree-based index with an overhead of at most 66% of the original storage. Again, the object cache requires around 140% of the original storage while the adjacency list between the tree-based index, and the object cache.

**Explanation and Interpretation**

Scanning can be always executed on the dataset without additional storage consumption for its maintenance. However, as we show in Section 5.3.4 on page 69 scanning should be avoided. The alternatives to scanning are the index structures adjacency list, object cache, and the tree based index. Among them, the tree-based index achieved the best storage consumption, i.e., it introduces the least storage overhead per basic pattern match. We mention here, that we originally proposed to index each possible basic pattern match in Section 4.1.2 on page 36 but consider the storage overhead here for each basic pattern match separately. Consequentially, if the tree-based index should replace the scanning approach for universal basic pattern matching, the required storage consumption is the sum of all basic pattern matches. We considered the per-pattern match query storage consumption for the tree-index due to a fair comparison to the other approaches. However, in contrast to the tree-based index, the native indexes adjacency list, and object cache require more storage in direct comparison. The reason is that these indexes benefit from direct access to a list of node identifier which is also their major drawback, i.e., the storage consumption linearly grow when the number of nodes in the graph linearly grows. This is in contrast to the tree-based index which depends on the number of edges rather than the number of distinct nodes. This aspect becomes dramatic for the FB dataset: this dataset has a huge number of nodes which are pointed at from other nodes but which itself do not point at other nodes. We already considered these „dead ends“ in previous discussions. However, a node which is pointed at but which does not point at another node is stored in both the object cache, and the adjacency list. In both index structures, these nodes consume storage but are not connected to a pre-computed resultset. For instance, in the FB dataset the majority of nodes are features which are pointed at other nodes, but the node itself point at nothing.
Table 5.3: Complexities of an *edges-out* query in a Graph $G = (V, E)$ on node $u$, additional storage consumption relative to working dataset size, and index build time. Best values are marked in bold font.

<table>
<thead>
<tr>
<th>Query Task</th>
<th>Scan</th>
<th>Native</th>
<th>Red-Black</th>
<th>Adjacency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find $u$</td>
<td>$\mathcal{O}(</td>
<td>E</td>
<td>)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>Determine $N(u)$</td>
<td>$\mathcal{O}(</td>
<td>N(u)</td>
<td>)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>Return $(N(u), \text{types})$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage overhead</th>
<th>Scan</th>
<th>Native</th>
<th>Red-Black</th>
<th>Adjacency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEN dataset</td>
<td>-</td>
<td>+ 108%</td>
<td>+ 67%</td>
<td>+ 83%</td>
</tr>
<tr>
<td>FB dataset</td>
<td>-</td>
<td>+ 3980%</td>
<td>+ 60%</td>
<td>+ 1740%</td>
</tr>
<tr>
<td>GOG dataset</td>
<td>-</td>
<td>+ 140%</td>
<td>+ 58%</td>
<td>+ 96%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index build time</th>
<th>Scan</th>
<th>Native</th>
<th>Red-Black</th>
<th>Adjacency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEN dataset</td>
<td>-</td>
<td>3 msec</td>
<td>60 msec</td>
<td>3.8 msec</td>
</tr>
<tr>
<td>FB dataset</td>
<td>-</td>
<td>0.07 msec</td>
<td>0.44 msec</td>
<td>0.00 msec</td>
</tr>
<tr>
<td>GOG dataset</td>
<td>-</td>
<td>100 msec</td>
<td>1100 msec</td>
<td>10 msec</td>
</tr>
</tbody>
</table>

5.4 Summary

In this chapter we provided a brief overview on implementation details of our evaluation prototype in Section 5.1, and evaluate the concepts covered in this thesis in Section 5.2, and in Section 5.3.

In our micro-benchmarks, we determined a suitable scan configuration for our case-study: a horizontal parallelization on a column-oriented storage layout with bulk-processing evaluation for large datasets (i.e., the scanning in memory-bound). We show that scanning is not a competitive option compared to the indexes introduced in this thesis when query performance is primary. If storage overhead is primary, scanning might be considered as an option. However, we argued efficient indexing can highly exploit pre-computed resultsets for single step traversals, such that the task of answering a query becomes the effort of finding the record, and copying a memory address.

**Query Execution**

In Table 5.3 we summarize the complexities for an *edges-out* query on a node $u$ in a graph-shaped dataset. The evaluation of an *edges-out* query consists of three sub-tasks: (1) find $u$, (2) determine its neighborhood $N(u)$, and (3) return $N(u)$ along with the relationship types. Once a resultset is available, the effort of sub-tasks (3) is constant for all approaches, since it is returning a memory address. The differences rely on the sub-tasks (1), and (2). While the native approach (object cache), and the adjacency list can access the pre-computed set for $u$ directly, both scanning and search-tree-based approaches must find $u$ first. Since the red-black tree is a balanced search
5.4. Summary

tree, $u$ can be found in logarithmic time w.r.t. the number of all edges. In contrast, scanning must access all edges to determine $\mathcal{N}(u)$ - independent of whether the scan is parallelized, or not. The sub-task (2) is constant for all approaches with exception of the scan-based approach. For the index approaches, the effort for sub-task (2) is constant since the indexes hold pre-computed result sets that are directly associated to search keys. In contrast, scanning must collect the information needed for the resultset while performing sub-task (1).

Consequentially, for node-centric traversal queries, indexes should be favored, and the primary difference is finding the search key, i.e., find node $u$ for a query on $u$.

**Introspection: Finding a Node**

In contrast to common belief, we do not observe a significant performance difference between the native and non-native indexing approaches for a single step traversal.

Both native approaches exploit direct access via node identifiers as positions in an array, while the tree-based index utilizes a binary search strategy. In fact, up to a dataset size of 1 GiB ($\approx 90$mio edges), a binary search finds the requested record in $\lceil \log_2(90 \cdot 10^6) \rceil = 27$ steps which does not introduce a notable overhead compared to directed access. To observe a notable difference, say 0.5 msec, the sum of all execution times of these 27 steps for a binary search must exceed this threshold of 0.5 msec. Such high execution time is unlikely on modern hardware given that dataset size, since a single step consists of a few CPU instruction, and little data that must be transferred to the CPU in order to be compared with the search key. The duration of a single binary search step dependents, of course, directly on the number of edges. For datasets beyond 1 GiB it is likely to observe a notable difference at a certain dataset size threshold.

**Index Storage Consumption**

When considering the required storage consumption, the native index approaches require unacceptable additional storage in our case-study. In Table 5.3, we show the storage consumption per approach and per dataset. In general, the tree-based approach should be favored, since its storage consumption dependents primary on the number of edges and, thus, the graph size. In contrast, the storage consumption of the native approaches dependents primary on the number of nodes. When a node is isolated, and therefore not contained in an edge, this node does not affect the storage consumption of the tree-based approach. In contrast, to achieve a constant time for sub-task (1), the native-indexes require an array where the $i$-th element correspond to the node with the id $i$. Hence, the more nodes in a dense graph are involved (or the more isolated nodes exists), the sparser this array becomes, and the more memory is wasted. This becomes dramatic for the facebook dataset.

**Relationship-Centric Queries**

Native approaches only support a subset of all possible basic pattern queries (c.f.,
Figure 5.6: Logical query plan over edge set $E$ of a graph $G = (V, E)$ to query *What are shared edge-relationships between $u \in V$ and $v \in V$?*. The left plan only considers *node-centric* queries, while the right plan takes advantages of a *relationship-centric* queries.

Table 4.1), i.e., native approaches consider node-centric navigation traversal queries (e.g., nodes-out or edges-out queries). Relationship-centric queries such as *What are the shared edge-relationships between a node $u$, and a node $v$, a node-centric approach must intersect edges-out of $u$ and edges-in of $v$ for both directions*. We show an example query plan in Figure 5.6. A relationship-centric query will utilize a connected-by query for both directions for the same tasks, and will be likely favored by a query optimizer. However, the more basic pattern queries are supported (e.g., by the tree-based index), the higher the storage consumption will be, since additional redundant data must be hold for additional indexing. In fact, we observed heavy storage consumption for the tree-based approach when supporting all basic pattern queries. Since supporting all basic pattern matches will improve the overall querying speed (and logical plan optimization potential), compression is an important topic here.

**Index Creation Costs**

In our case-study, the time it takes to instantly re-build an arbitrary indexes is less than 4 seconds. Of course, this time depends on the involved dataset, and relies on the fact that we optimize for read-only indexes in a main-memory based system. However, as the tree-based approach should be favored in terms of additional storage consumption, the tree-based approach is not the first choice in terms of index build times. We argue that for read-only (and append-only) datasets, the index must not be destroyed and completely re-built, typically. Hence, further optimization could cover this aspect which was out of the scope of this thesis.
6. Related Work

Specialized graph database provide a built-in support for navigation access patterns, such as traversals. This makes them a promising choice for exploration tasks, especially from a usability perspective due to their specialized query languages. From a performance perspective, there is an on-going discussion whenever graph databases outperform relational databases. In this section we highlight interesting research as our related work.

6.1 Graph Traversals in Relational Engines

Clearly, there is a strong business need for both data management scenarios, and also a strong need for a management solution capable to address both at once. Consequentially, there is on-going effort to integrate one data model into the other.

Recent research suggest that relational databases are competitive for certain graph specific tasks from a performance perspective, and graph databases do not perform well for bulk-intensive workloads in general. The database community addresses this issue by asking if the switch to graph-shaped data management really require to re-design databases, or if existing system can be either used for this novel use cases, or at least be adapted.

On the one hand, Gubichev et al. who provide empirical evidence that traditional relational databases and RDF stores provide competitive advantage for graph-specific pattern matching queries [GT14]. On the other hand, Angles et al. proposed a social network micro-benchmark for graph databases, RDF stores and relational databases. They conclude in contrast that relational databases are highly stressed to compute reachability queries in a reasonable time for a hop count larger than 4 [APPDSL13]. Also Vicknair et. al observed better performances on structural queries using graph
6. Related Work

Paradise et al. recently suggested an extensible graph traversal framework, Graphite, for columnar main-memory databases [PLB15]. They investigated how graph traversal can be integrated on top of a columnar storage. Thus, they study how the graph topology affect their two proposed traversal algorithm, namely a level-synchronous, and a fragmented-incremental traversal algorithm. Both algorithms in Graphite rely on high-efficient parallel scanning utilizing the capabilities of SAP Hana. While Paradise et al. study how improved query performances can be achieved due to a variability in the traversal algorithm, we fix a traversal algorithm and examine how a variability for each step in a traversal can be improved by alternatives to scan operations.

Whether RDBMS lack in general for graph processing is investigated by Fan et al. recently [FRP15]. They proposed GraIL that compiles vertex-centric queries formulated in a domain-specific language to SQL. Thus, they adapted the Bulk-Synchronous Parallel (BSP) programming model with message passing similar to Giraph for RDBMS by utilizing tables for the communication, and states. Similar, Jindal et al. suggested GraphQL as a graph query language for relational databases that compiles graph-specific queries into SQL [JM14]. Both Fan et al., and Jindal et al. compared their suggestions to the distributed graph processing framework Giraph. In contrast to them, we consider native and non-native indexing strategies for efficient graph traversals as the backbone of efficient generic graph algorithm. Although both hint to performance opportunities using a Graph-SQL compiler, we believe a compiling approach to SQL will leave optimization potential untouched since it is unlikely the RDBMS query optimizer considers graph-specific optimizations. In addition, we focus on explicit-non-distributed execution in this work. For non-distributed systems, Welc et al. found relational databases using an abstraction layer to compile graph specific queries (GreenMarl) provide competitive or superior performance for shortest-path algorithm (bidirectional Dijkstra) [WRW+13].

6.2 Comparison of Graph Databases, and - Models

All graph database have in common that the data is modeled in terms of nodes and edges. Some graph databases only rely on this, others add further semantic to records in terms of user-defined attributes (properties) attached to edges and nodes. Vicknair et al. compared a property graph database with a simple graph modeled in a RDBMS [VMZ+10]. Simple graphs are also used by Hong [HHL14]. Paradise et al., and Macko et al. focus on property graph databases [PLB15, MMS13]. Neumann et al. focus on RDF graph databases [NW08]. Gubichev et al. considered both property graph database and RDF graph databases [GT14].
6.3 Database Comparisons

Depending on the graph model, the graph-shaped data can be stored, and managed differently. Differences in the encoding must be considered since the data structure inherently affects the data access costs. Native graph databases such as Neo4j store graph-shaped data explicitly in a non-tabular fashion connecting related records [RWE13]. Other graph database such as RDF-3X use a logical table using $B^+$ trees [NW08]. Angles et al. encoded their graph-data as non-graph data during their evaluation. They use regular entity tables and added indexes to certain table attributes [APPDSLP13]. Vicknair et al. modeled the graph using two tables, one for nodes and one for edges where record properties are not considered [VMZ+10]. Similar, Paradise et al. used two tables in a columnar relational engine and added properties in the tables schema directly [PLB15]. Welc et al. used only on table with three columns [WRW+13].

6.3 Database Comparisons

As for the relational counterpart, there is no system that is the database. Depending on the systems focus, and client requirements, graph databases differ heavily in their implementations. Vicknair et al. compared MySQL and Neo4j [VMZ+10]. Gubichev et al. considered Virtuoso, Neo4j, and SAP HANA [GT14]. Neumann et al. considered RDF-3X, MonetDB, and PostgreSQL [NW08]. Welc et al. included Neo4j and Green-Marl into its evaluation [WRW+13].

6.4 Architecture, and Platform Comparison

Paradise et al. compared a main-memory based systems with a disk-based graph database system a RDBMS in their evaluation that achieved competitive performance for a hop count up to 8 [PLB15]. On the other hand, Vicknair et al. compared only disk-based architectures, but one native compiled and the other running in a virtual machine [VMZ+10]. Neumann et al. compared native compiled disk-based databases only [NW08]. Angles et al. considered primarily native compiled databases and one running in a virtual machine [APPDSLP13]. Welc et al. compared a query compiler to native code, and a graph database running in a virtual machine [WRW+13].

6.5 Comparison on Different Datasets

Gubichev et al. compared property graph databases on a semantic web dataset addressing an RDF encoding [GT14]. Vicknair et al. randomly generated graph data [VMZ+10]. In contrast, Paradise et al. used real world graph datasets ranging from infrastructure to social networks, and a patent network [PLB15]. Similar, Neumann et al. used three real world RDF datasets [NW08]. Macko et al. considered small-world graphs, and real-world graphs [MMS13]. Collaborative networks, citation networks, and web graphs are considered by Hong [HHL14].
7. Wrap-Up

In this chapter, we provide a summary to the work presented in this thesis, draw a conclusion, and give an overview of future work opportunities.

7.1 Summary

In this work, we addressed the question whether efficient navigation in graph-shaped data requires a native graph storage in context of main-memory databases.

We argued for a whitebox evaluation to avoid confounding variables, and to enable a fair comparison in an equal environment for native and non-native graph storage approaches.

In order to avoid bias due to some internal behavior of blackbox systems, we argue for a unified evaluation despite different storage approaches:

1. **One graph model.** The graph model have to be the intersection of the RDF model, and property graph model, i.e., the graph model has to be a directed multi graph.
2. **One graph encoding.** At its core, navigation in graph-shaped data operates on edges, i.e., the graph encoding must be limited to express the graph by its edge-set rather than by its properties or advanced meta data.
3. **One system under study.** To enable a fair comparison, each of the approaches under study must be equally handled by the system, i.e., the evaluation system must be able to manage both native and non-native graph storage.
4. **One architecture and platform.** To avoid bias due to architectural mismatch, or different platforms, the approaches under study must be executed not only in the same system, but also on the same architecture and platform.
5. **One evaluation environment.** A direct comparison of native and non-native graph storage approaches must run in the same evaluation environment, i.e., the evaluation must cover a representative query that stresses approaches equally.

To achieve the desired goal of a whitebox evaluation, we proceed as following:

1. **Unification.** We introduced a logical presentation to model graph edges, independent of the (graph) database in use. This model is a directed multi graph, where each edge is a triple consisting of dictionary encoded identifiers.

2. **Abstraction.** We established the concept of basic pattern matching which is a trinary equi-selection on our logical presentation. With basic pattern matching, we can express all possible navigation-related traversal queries (e.g., querying the neighborhood, or incident nodes given an edge).

3. **Mapping.** We showed how to use basic pattern matching queries to execute single step traversal queries which are the backbone of any graph traversal algorithm. We used single step traversals (i.e., the edges-out basic pattern) to implement a reachability query with breadth-first search.

4. **Implementation.** We introduced existing native and non-native graph storage approaches, and adapted them in a transparent and unified way in the context of main-memory databases. Namely, we considered scanning and tree-based indexing as non-native approaches, and adjacency lists and the object cache (from Neo4j) as native approaches for our evaluation.

Our evaluation includes in-depth experiments on artificial data, as well as reachability queries on real-world datasets. We evaluated native and non-native graph storage approaches considering the requirements mentioned above.

Our experiments show that each approach under study is affected by the graph datasets properties, such as the size, or the diameter. We observed similar performances for all indexes. In general, all indexes achieved query execution times up to 1/1000 of the scanning query execution times in direct comparison. Without the need for seeking in some external files on disk, accessing relevant records by the use of (non-native) tree-based indexes achieves comparable performances to native approaches.

### 7.2 Conclusion

High efficient traversal algorithm in main-memory systems require indexing adjacent records (node-centric queries) and incident relationships (relationship-centric queries) rather than the property of being a native graph storage or a non-native graph storage. This means, it is the fast access to relevant records that counts for efficient traversal, and
not the storage layout of the single record, or the set of records: in main-memory systems, the query engine favors late materialization, i.e., operating on record positions in certain record containers (e.g., tables) instead of concrete values as long as possible. Indexing adjacent records and incident relationships is a question how to organize these positions for fast access, while the concrete record storage layout is secondary. Hence, efficient single step traversals in main-memory graph-shaped data rely on indexing rather than on storage layout.

7.3 Future Work

The following outlines further research directions related to this thesis:

**Compression.** We showed how tree-based indexing of basic pattern matching queries can lead to superior query performances compared to scanning for navigation queries. Nevertheless, supporting all basic pattern matching queries using tree-based indexes, we have to create several separate indexes. This leads to high storage consumption. Future work may consider compression techniques for multi-dimensional index structures in order to minimize the additional storage consumption to a feasible level.

**Parallelized reachability queries.** We showed that each index under study is nearly optimal in terms of access costs, i.e., returning the resultset is cheap since it is copying a memory address, and accessing the relevant record is done either in constant time, or in logarithmic time depending on the number of edges in total. However, our experiments hint that on large-scale dataset (e.g., web graphs) even this query performance will lead to unacceptable query times. Thus, to address large-scale dataset performance issues, we have to investigate advanced parallelized reachability queries.

**Enable graph navigation in RDBMS.** The navigation in graph-shaped data as introduced in this work relies on unique identifiers for both, nodes, and relationships stored in an RDF-like graph. As we showed, this representation suites well for basic pattern matching queries, and thus enables generic navigation-centric single step traversal functions. In order to close the gap between RDBMS and graph databases in terms of navigation capabilities, future work should cover how tuples from different tables in a RDBMS can be efficiently mapped to global, unique identifiers. This enables efficient graph-navigation in RDBMS using basic pattern matching queries.
Bibliography


[RPBS07] Jaap C. Reijneveld, Sophie C. Ponten, Henk W. Berendse, and Cornelis J. Stam. The application of graph theoretical analysis to complex


Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.