An Evaluation of Graph Prose: Graph Query PROcessing with Search Engines

Author: Shreeraksha Achutha

June 25, 2018

Advisors:
M.Sc. Gabriel Campero Durand
Data and Knowledge Engineering Group

Prof. Dr. rer. nat. habil. Gunter Saake
Data and Knowledge Engineering Group
Achutha, Shreeraksha:
An Evaluation of Graph Prose: Graph Query PROcessing with Search Engines
Abstract

Network analysis is used to study real world relationships, after modeling them as graphs. Graph Databases, which are non-relational databases, play a vital role in helping companies to store and analyze this network-oriented data. Not only social networking services, but also various industries and domains require graph databases to store, query and analyzed connected data. Along with this, a variety of websites and other systems provide some sort of full-text search capability over the data stored, helping users to find what they are looking for in a very short span of time. Some systems that can enhance graph databases to provide full-text search capabilities are ElasticSearch and Apache Solr, which are Lucene-based search engines.

Considering the benefits from both types of systems, Graph Databases and Search engines, we undertake in this study an evaluation on potentials for a more perfect union between them. We study the potential of Search engines to support Graph queries, complementing Graph Database engines. For our evaluation we have considered Neo4j and JanusGraph as Graph Databases and ElasticSearch as a Search Engine. We have considered the LDBC Social Network Benchmark Data and some queries from the Business Intelligence Workload.

We take a close look into limitations and opportunities from the data mapping between engines. We report for global queries speedups in the range of 100x and 5000x (for Neo4j and JanusGraph, respectively) when using the search engine in contrast to the native graph alternative, but for queries that require traversals while filtering on edge data the graph database maintains a superior performance with latencies around 4-10x smaller. As a conclusion, we show that indeed the graph databases can take advantage of the search engine to carry out global queries, but local queries should be supported by the graph database. Future work building on our findings should manage automated re-writes between engines.
Acknowledgements

By submitting this thesis, my long term association with Otto von Guericke University will come to an end.

First and foremost, I am grateful to my advisor M.Sc. Gabriel Campero Durand for his guidance, patience and constant encouragement without which this may not have been possible.

I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to write my Master’s thesis at his chair.

It has been a privilege for me to work in collaboration with the Data and Knowledge Engineering Group.

I would like to thank my family and friends, who supported me in completing my studies and in writing my thesis.
Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, June 25th 2018

________________________
Shreeraksha Achutha
# Contents

List of Figures xiv

1 Introduction 1

2 Background 5

2.1 Networks and Graph Models 6

2.1.1 Network Analysis 6

2.1.1.1 Social Networks 6

2.1.2 Graph Model 11

2.1.2.1 Property Graph Model 13

2.1.2.2 RDF Graph Model 14

2.2 Graph Data Management 15

2.2.1 Graph Databases 16

2.2.1.1 Property Graph Databases 17

2.2.1.2 RDF Databases 20

2.2.1.3 Query Processing and Optimization 21

2.2.1.4 Query Languages 21

2.2.1.5 Relational Databases and Full-Text Search Engines 23

2.3 Search engine: ElasticSearch 24

2.3.0.1 Document Indexing: 25

2.4 Integration of Graph Databases and Search Engines 29

2.5 Benchmark Used: LDBC 30

2.6 Summary 33

3 An Evaluation of Graph Prose: Graph PROcessing with Search Engines 35

3.1 Research Questions 35

3.2 Case Study: LDBC SNB Benchmark 36

3.2.1 Data Schema 36

3.2.2 Entities 36

3.2.3 Relations 38

3.2.4 Scale Factors 38

3.2.5 LDBC SNB Business Intelligence Queries 39

3.3 Selection of Tools 41
3.3.1 Neo4j with GraphAware .................................................. 41
3.3.2 JanusGraph ................................................................. 42
3.3.3 ElasticSearch ............................................................... 42
3.4 Experimental Setup ........................................................... 42
  3.4.0.1 Backend Configuration ............................................. 43
  3.4.0.2 Configuration for Mapping Tools ................................. 44
3.4.1 Dataset Generation ....................................................... 46
3.4.2 Dataset Loading ........................................................... 47
3.5 Summary ........................................................................... 47

4 Mappings from Graph Database to Search Engine ......................... 49
  4.1 Research Questions .......................................................... 49
  4.2 Mappings ....................................................................... 50
    4.2.1 Types of Mappings ...................................................... 50
      4.2.1.1 Basic Mapping ..................................................... 51
      4.2.1.2 Mapping Relationships with Efficient Use of Edge Data ...................................................... 51
      4.2.1.3 Mapping Relationships with Join Data Types ................................. 52
      4.2.1.4 Use of Mappings in the Queries of our Study .................. 54
    4.2.2 Performance Evaluation of Neo4j ................................... 54
    4.2.3 Performance Evaluation of JanusGraph ......................... 54
    4.2.4 Comparison of Results ................................................ 54
  4.3 Summary ........................................................................... 54

5 Global Graph Query Rewrites ....................................................... 55
  5.1 Research Questions .......................................................... 55
  5.2 Global Graph query Evaluation .......................................... 55
    5.2.1 Rewrite for Global Query and Profile of Execution ............ 56
    5.2.2 Performance Evaluation of Neo4j ................................... 60
    5.2.3 Performance Evaluation of JanusGraph ............................ 63
    5.2.4 Comparison of Results ................................................ 63
  5.3 Summary ........................................................................... 65

6 Adjacency Graph Query Rewrites .................................................. 67
  6.1 Research Questions .......................................................... 67
  6.2 Adjacency Graph query Evaluation ...................................... 67
    6.2.1 Adjacency Query: Query 18 ........................................ 68
      6.2.1.1 Rewrite for Adjacency Query ................................ 69
      6.2.1.2 Performance Evaluation of Neo4j ............................. 71
      6.2.1.3 Comparison of Results .......................................... 71
    6.2.2 Adjacency Query: Query 3 ........................................... 72
      6.2.2.1 Rewrite for Adjacency Query ................................ 72
      6.2.2.2 Performance Evaluation of Neo4j ............................. 73
      6.2.2.3 Comparison of Results .......................................... 74
    6.3 Adjacency Query: Query 12 ............................................. 74
      6.3.1 Rewrite for Adjacency Query ..................................... 75
      6.3.2 Performance Evaluation of Neo4j ................................. 76
      6.3.3 Comparison of Results .............................................. 76
6.4 Adjacency Query: Query 14 .................................................. 76
  6.4.1 Rewrite for Adjacency Query ............................................ 78
  6.4.2 Performance Evaluation of Neo4j ....................................... 79
  6.4.3 Comparison of Results ................................................... 79
6.5 Summary ........................................................................... 80

7 Conclusion and Future Work ............................................... 81
  7.1 Summary ......................................................................... 81
  7.2 Threats to validity ............................................................ 82
  7.3 Future work and Concluding Remarks ................................. 83

Bibliography ........................................................................... 85
List of Figures

1.1 Response time of alternative implementations for calculating the average degree centrality over different relations on the Pokec dataset [Dur17] ... 2

1.2 Graph Prose: queries are rewritten from graph to search engines, supported by effective mappings ... ... 3

2.1 Network Structure ... ... 7

2.2 Social Roles and equivalences ... ... 10

2.3 Graph model with Node Properties ... ... 12

2.4 Property Graph and its concepts ... ... 13

2.5 There are numerous types of graphs. Many of the formalisms described here can be mixed and matched in order to provide the modeler with more expressiveness ... ... 15

2.6 A property graph is a directed, labeled, attributed, multi-graph. The edges are directed, vertexes/edges are labeled, vertexes/edges have associated key/value pairs meta data (i.e. properties), and there can be multiple edges between any two vertexes. ... ... 16

2.7 Comparison of Graph Databases ... ... 18

2.8 Neo4j Native Graph Database Architecture ... ... 19

2.9 Neo4j Node and Relationship Store File Record Structure ... ... 20

2.10 Simple Graph Database Model ... ... 22

2.11 SQL Query ... ... 23

2.12 CYPHER Query ... ... 23

2.13 Parallel concepts between ElasticSearch and SQL Database ... ... 24

2.14 High-level Architecture of Commercial Quality Full-Text Engines ... ... 25

2.15 Shards across two nodes ... ... 27
List of Figures

2.16 Lucene Segment Search .................................................. 28
2.17 ElasticSearch Architecture .......................................... 29
2.18 Example Graph Pattern .................................................. 32
3.1 LDBC SNB Data Schema .................................................. 37
3.2 Neo4j Characteristics ...................................................... 41
3.3 GraphAware Neo4j-To-ElasticSearch : Workflow ............. 45
3.4 Neo4j configuration for GraphAware ............................... 46
3.5 LDBC Datagen Params.ini configuration ............................ 46
3.6 LDBC SNB data loaded into Neo4j ..................................... 47
4.1 New fields TGTID, SRCID added in CSV ....................... 52
4.2 New fields SRCID and TGTID in ElasticSearch .............. 53
5.1 Query 1 : Description and Pattern .................................. 56
5.2 ElasticSearch Query : Posting Summary ......................... 58
5.3 Neo4j Cypher Query : Posting Summary ......................... 60
5.4 Neo4j Output : Posting Summary .................................... 61
5.5 Profile for Execution of Global Query in Neo4j .............. 62
5.6 ElasticSearch Output : Posting Summary ......................... 64
5.7 Performance evaluation for Query 1 ............................... 65
6.1 Query 18 : Description and Pattern ............................... 69
6.2 Query 18 : Neo4j Cypher Query ..................................... 71
6.3 Performance evaluation for Query 18 ............................... 72
6.4 Query 3 : Description and Pattern .................................. 73
6.5 Query 3 : Neo4j Cypher Query ..................................... 74
6.6 Performance evaluation for Query 3 ............................... 75
6.7 Query 12 : Description and Pattern ............................... 76
6.8 Query 12 : Neo4j Cypher Query ................................... 77
6.9 Performance evaluation for Query 12 ............................... 77
6.10 Query 14 : Description and Pattern ............................... 78
6.11 Query 14 : Neo4j Cypher Query ................................... 79
6.12 Performance evaluation for Query 14 ............................. 80
1. Introduction

In this chapter, we present the motivation behind this Thesis, describe its goals and outline.

Network analysis can be used to study real world networks of relationships, leading to business and real world opportunities. Social networks are one kind of these networks, connecting individuals through relationships that are social in nature, such as friendship, collaboration, etc. Graphs, in turn, are a simple modeling tool used to represent and work with those networks, by considering them as nodes and edges (also called as vertexes and relationships).

Graph Databases are a kind of data management system that has graph structures as an underlying logical model. Instead of tables or documents, users have the notion that their data is stored in entities that are either nodes or relationships, and in properties that are assigned to those entities. Graph databases are designed to allow very fast and deep traversals through the graph elements, using the concept of index-free adjacency [RNT12]. These systems are very fast when searching for relationships of type friends of friends (e.g, k-hop traversals).

In order to understand better the expected use cases for such databases, there has been an industry-based effort, lead by the LDBC Council, to form benchmarks for graph-like data management. Within these benchmarks practitioners have proposed a set of workloads that seek to represent real-world scenarios. To date this council has proposed 3 workloads [ATB+14]:

- Interactive workloads consist of queries that explore a graph around a given node (i.e., short read queries) while simultaneously managing update operations.

- Business intelligence workloads are formed by queries that explore vast portions of the graph with complex queries involving nested grouping operations and filtering criteria.
- *Graph analytics workloads* focus on single algorithms at a time, which in its majority, involve the complete graph and require large-scale processing.

Authors have also proposed that apart from considering workloads, the emerging graph queries themselves can be grouped [KWY12] into three classes: mining queries (e.g. finding frequent subgraphs from a large graph), matching queries (e.g. finding a specific, explicit pattern) and selection queries (i.e., including skyline queries, where the task is to find an implicit structure). Following these classes, we observe that the first two workloads of the LDBC benchmark are composed primarily of matching queries.

In order to support workloads similar to the business intelligence and interactive workloads, graph databases offer diverse query languages such as Cypher and Gremlin. Furthermore, to provide an enhanced search functionality for end users graph databases are integrated with search engines, helping users to find what they are looking for by performing full-text queries. Some graph databases that provide integration with search engines are Neo4j (e.g., through the Graphaware plugin), Orient-DB and JanusGraph (for which the integration does not require external plugins).

Though a large body of research has studied optimizations of these graph queries through the graph database itself (e.g., through better indexing, query processing, etc.) [FPP16], there is today almost no single study devoted to how the companion search engine can be employed to support them. This is not a difficult problem in itself, but due to the relative novelty of graph technologies and the slow standardization of technologies connecting search engines and graph databases, it has received insufficient attention.

Previous findings report potential gains in using the search engine for graph queries. By leveraging the aggregation framework, authors achieve speedups of 150x [Figure 1.1] when compared to a graph-only execution using Apache Titan (now JanusGraph) [Dur17]. However these results do not study in depth the mapping possibilities, nor do authors use benchmark queries.

![Figure 1.1: Response time of alternative implementations for calculating the average degree centrality over different relations on the Pokec dataset](Dur17)
Motivated by these previous findings and by possible improvements to such research, we propose in this study to evaluate the potential of search engines to support graph queries, complementing the graph database query engine. For this we focus clearly on an extension of traditional queries which only address a given type of engine, such that they start as graph queries and are fully or partly rewritten to be executed by the search engine, with support from effective mappings. We call this approach *Graph Prose* (see Figure 1.2).

![Figure 1.2: Graph Prose: queries are rewritten from graph to search engines, supported by effective mappings](image)

For studying the proposed approach we select Neo4j and JanusGraph as graph databases, and ElasticSearch as a search engine. We evaluate the impact of hand-tuned rewriting of queries across engines from the business intelligence workload of the LDBC Social Network benchmark. We selected queries from this workload since they are more complex than those of the interactive workload. In our re-writes we employ features from the query interface of ElasticSearch, which to the best of our knowledge capture the best capacities of this engine, such as scripted fields and functionalities for scrolling the results. We report our findings for global queries, yielding speedups of 100x when using aggregations. We also compare global queries in JanusGraph vs. Neo4j, finding the latter to produce better performance. We also report that for Neo4j the search engine performs 10x slower than the graph engine for queries that require traversals from a node.

In addition, we find that there is a crucial impact of graph to document mappings, in the utility of the search engine. For example, the common practice of some plugins to map nodes and relationships to separate indexes limits the utility of the search
engine, preventing certain kinds of queries. As a result from our observation we are able
to suggest a series of steps to improve the mappings, contributing to future work in
leveraging the search engine as part of a graph query engine.

Our work is organized in the following way:

- **Introduction:** We present the motivation behind the thesis, describe goals and
  outline its organization (Chapter 1).
- **Background:** We present an overview of the theoretical background and state of
  the art relevant to the current research work (Chapter 2).
- **An Evaluation of Graph Prose: Graph Query Processing with Search Engines:**
  We define a set of core research questions regarding the potential of
  the search engine to support graph queries, complementing the graph database
  query engine, using formal benchmarks. We present our experimental setup with
  selected tools, Architecture, Configuration, Benchmarks and Hardware used in our
  application for graph querying optimizations (Chapter 3).
- **Mappings from Graph Database to Search Engine:** We discuss, based on
  the chosen tools, our observations regarding possible mappings between graph
  entities and search engine documents. We discuss how different mappings impact
  the utility of the search engine for graph queries (Chapter 4).
- **Global Graph Query Re-writes:** We present our evaluation on the gains
  possible from rewriting global graph queries (i.e., no more than 1 hop involved)
  such that they could be executed directly in the search engine (Chapter 5).
- **Adjacency Graph Query Re-writes:** We report our results from evaluating
  the gains possible from rewriting adjacency queries (i.e., only a small number of
  hops is involved) such that they could be executed directly in the search engine
  (Chapter 6).
- **Conclusion and Future Work:** We conclude our studies in this chapter (Chapter 7).
2. Background

In this chapter, we present an overview of the theoretical background and state of the art relevant to the current research work. Since our work studies the role of the search engine to support graph database queries not by running them on the native graph storage but by running them on the search engine, here we aim to provide sufficient information for understanding the context of our research, and to present with care the main ideas necessary for understanding our research questions and focus.

We outline this chapter as follows:

- **Networks and Graph Models:** We begin by introducing Networks and some concepts of network analytics, which provide context for our research in Section 2.1. In our selection of examples we focus on OLAP/Analysis queries which belong to the domain of network analysis, and are not concerned with data loading/updates or point queries, which belong more to data management. We also focus on Social Networks as a representative domain. We also introduce in this section logical models for graphs (RDF, property graph, hyper-graph).

- **Graph Data Management:** Next we present the data management context in which graph technologies are being developed, in Section 2.2. We introduce graph databases like Neo4j or JanusGraph in Section 2.2.1. These choices, in turn are supported through languages like OpenCypher or Gremlin.

- **Search Engine: ElasticSearch:** Here we discuss the fundamentals on how search engines store data and the types of queries supported by them in Section 2.3, with a focus on ElasticSearch and Lucene as representatives.

- **Integration of Graph Databases and Search Engines:** Based on our research we discuss on search engines and technologies involved in integrating data from both engines in Section 2.4.
• **Benchmark used:** LDBC: In order to help understand our evaluation, we offer a brief discussion on the benchmark used in our evaluation, the LDBC benchmark [Section 2.5](#).

• **Summary:** We conclude this chapter by summarizing our studies in [Section 2.6](#).

## 2.1 Networks and Graph Models

### 2.1.1 Network Analysis

Graphs represent real world networks, consisting of entities that are connected to each other through relationships. Some examples of networks are *Social Networks* where we model the relationships between people. *Transportation Networks* where the connectivity between locations (e.g., roads, flight paths) and characteristics of these connections (e.g., cost of a transport fare) are modeled.

By modeling our data as a network we can gain insight about which entities or nodes are important as broadcasters or influencer’s in a social network. Additionally, we can optimize transportation between cities (i.e., through path finding), or we can leverage the network structure to find communities in the network enabling data analysis at community (rather than global) level.

Networks are described by two sets of items, *nodes* and *edges* together these form a network, otherwise known in mathematical terms as a graph. Nodes and Edges can have metadata associated with them. In our example (see [Figure 2.1](#)) there are two friends ‘Emli’ and ‘Saint’ who met on the ‘21st of January 2018’. In this case nodes are ‘Emli’ and ‘Saint’ with metadata stored in them, as key-value pairs of properties like ‘Id’ and ‘Age’. The friendship, represented as a line between the two nodes, may have metadata such as ‘date’ that represents the date on which ‘Emli’ and ‘Saint’ first met.

#### 2.1.1.1 Social Networks

Social networks are networks where the entities are individuals and the relationships that connect the individuals are social in nature. For example, friendships, or communication, etc. Given that in recent years social networking platforms have become omnipresent in our society, social network analysis is becoming a more common task, with potential gains for diverse use cases. In this section we present such area, with the intention of providing context for the kinds of queries and analysis that our study can consider.

Social networking platforms are websites or applications, or any media, that enable individuals to engage in relationships and express digitally some of the connections that they have in real life. Some examples are Facebook, Google+ and Twitter. While these examples are some of the few sites or services which come to our mind when we think of social networks, these do not express the entire scope of social networking platform. Such systems can also include:
2.1. Networks and Graph Models

- Multi-Media sharing sites using social networks, like Youtube.

- Professional networking sites, like LinkedIn and Xing, providing career-related opportunities.

- Information sites used to seek solutions for everyday problems, in line with the ideas of user-generated content, as epitomized in the term Web 2.0. For example, if we are thinking on how to improve food habits while living in a certain area, we can find various blogs and websites like, DIY community, Calorie counter and food diary, related to it. Wikipedia can also be considered an information site.

- Educational sites like EdX, Udacity and Coursera, where students can interact with other students online and do projects.

- Forums and Q&A sites, like Quora, Stackoverflow, etc.

- Reputation systems, like Uber, Airbnb, Blablacar, and others. Where users offer services and are given reputation points, to manage the interactions.

In spite of diversity of domains, all these platforms share several common characteristics, such as having users and content as entities, and of capturing interactions, social roles, relationships, etc, among them.

Social Network Analysis (SNA) is a field of study that focuses on the structure of relationships in social groups and organizations. Below we present a brief, illustrative description of terms commonly used in Social Networking Analysis [CF12]. We also summarize some of these terms in Table 2.1. We can organize these terms according to
Background

SNA Terms | Meaning
---|---
Social network, or Sociogram | Graph
Actor | Node
Link | Edges
Ego | Current node under discussion
Alters | Other nodes as viewed from Ego node
Eigenvector centrality | Measures the influence of a node in the network
Betweenness centrality | Stress

Table 2.1: Social Network Analysis Terminology

their underlying concepts, such as centrality, connectedness, social roles and scope of analysis.

Centrality

The notion of centrality aims to provide a score to a node based on how connected it is to the complete network, or alternatively on the quality/importance of the connections.

Below are some methods to measure the centrality, with different characteristics: [CF12] [BINF12] [HR05]

Degree centrality: The degree centrality of a node can be often simply calculated as the number of edges (i.e., the degree) of a given type or direction directly connected to a node. Hence, nodes with a higher degree are considered to be more central. This measure is only statistical and it does not say anything clear about the node w.r.t the complete graph.

Eigenvector centrality: Used to measure the influence of a node in a network. It assigns relative scores to each node in the network, by following the notion that nodes with high-scores propagate through their connections a high score to connected nodes, while low-scoring nodes propagate lesser scores [CF12]. Algorithms that calculate this measure are usually iterative in nature and must converge such that the scores assigned are faithful to the expected results.

Closeness centrality: Nodes which are in the shortest distance to all the neighboring nodes in the graph are considered to have high closeness centrality.

Betweenness centrality: Nodes with shortest-paths achieve high values in this centrality measure.

Flow centrality: In contrast to betweenness and closeness centrality, which consider shortest-paths, flow centrality attempts to measure the abundance of paths that flow through a node.
Bonacich centrality: This calculates the degree of the indirect neighbors these are the first eigenvector of the graph.

Centrality measures can be divided into two categories, those that consider the network flow only (e.g., degree centrality), and can be calculated through local measures, and those that are based on random walks (i.e., like eigenvector centrality), requiring iterative algorithms.

Connectedness

Connectedness measures attempt to model the importance of a node in keeping the graph as a single connected component. Stress is a measurement used to capture this notion.

Stress: Find all the shortest paths between all pairs of nodes in a graph. The stress of an edge is the number of these shortest paths that the edge belongs to [CF12].

Social Roles

This refers to the position of a actor in society. In contrast to centrality or connectedness measures, social roles can include more domain-specific semantics into it.

Roles are based on the relationships that actors have with other actors [CF12]. For example, the “husband” role is defined in part as being linked to another actor in a “wife” role. In other words, social roles could be thought of as representing regularities (i.e. common patterns) in the relationships between actors. Actors playing a particular social role have to be equivalent/similar to each other by some metric. In general, the following three kinds of similarities are considered, in decreasing order of constraints:

- **Structural equivalence**: Two actors $u$ and $v$ in a graph $G = (V, E)$ are structurally equivalent iff
  $$\forall x \in V, \quad (u, x) \in E \iff (v, x) \in E$$
  and
  $$\forall x \in V, \quad (x, u) \in E \iff (x, v) \in E$$
  They are connected to a similar set of nodes, with the arrows pointing in the same directions. Two structurally equivalent actors can exchange their positions without changing the network.

- **Automorphic equivalence**: Two actors $u$ and $v$ of a labeled graph $G$ are automorphically equivalent iff all the actors of $G$ can be re-labeled to form an isomorphic graph with the labels of $u$ and $v$ interchanged. Two automorphically equivalent vertexes share exactly the same label-independent properties as shown in Figure 2.2.

- **Regular equivalence**: If $G = (V, E)$ is a connected graph and $\equiv$ is an equivalence relation on $V$, then $\equiv$ is a regular equivalence iff [CF12]
  $$\forall a, b, c \in V, \quad a \equiv b \iff \begin{cases} (a, c) \in E \Rightarrow \exists d \in V \text{ such that } (b, d) \in E \text{ and } d = c \\ (c, a) \in E \Rightarrow \exists d \in V \text{ such that } (d, b) \in E \text{ and } d = c \end{cases}$$
Two actors $u$ and $v$ are regularly equivalent if they are equally related to equivalent others.

**Scope of Analysis**

Graphs can be analyzed at macro (i.e., global), meso (i.e., community) or micro levels (i.e., a very local neighborhood).

*Ego-centric networks (ego only)*: Ego-centric methods focus on the individual, but not the whole network. This network collects a single hop (or a few of them) from the ego node. The analysis of ego-networks gives a general texture of the network as a whole, by using the individual as a representative [CF12].

*Ego-centric networks (with alter connections)*: Since it’s not always possible to track all ego-centric networks from all nodes. Another approach is to begin with a selection of main nodes (egos), and identify to which nodes they are connected [CF12].

**Summary on Social Network Analysis Terminology**

In this subsection we have briefly presented, for illustration, some basic concepts from SNA. These concepts corresponded either to graph statistics (e.g., centrality and connectedness measures) that could be calculated with algorithms, to pattern searches (e.g. to determine social roles and equivalences), and to the scope of analysis. With these concepts we hope to have illustrated the kinds of queries that can be posed to a graph, once the data is loaded into a graph database. Namely, queries to determine global statistics (requiring algorithmic support), queries to match or find patterns, and queries taking place at different levels of the graph. In the next sections we talk about
2.1. Networks and Graph Models

models for graphs, graph data management alternatives, and we conclude the chapter by presenting a state-of-the-art benchmark for graph data management, consisting of queries and workloads that replicate some of the analysis concepts we have discussed in this section.

2.1.2 Graph Model

We live in the era of information, where information is everything and everywhere, and can be accessed from anywhere. For example if we type in Google for Big data we receive millions of results. For these results to be fetched they should be noted down and stored in a database or search engine. Often the results can be stored in a graph database, since graphs are useful to model and follow the relevance of items in a network [Why17].

Graph databases can be composed of billions of nodes and are mainly focused on relationships. Previously, data management systems relied on structured query languages and relational data, but for cases where the data is huge/irregular developers have found it difficult and slow to communicate with each row and column, or to scale out the processing [RWE13a].

As data size increased it was difficult for users of structured query languages to find the relationships between the datasets. Graph databases came as one of many solutions for these issues, as they represent unstructured data and focus on relationships. Today we can find several companies and research institutes performing big data analytics using graph databases and network-based analysis to boost their efficiency.

G(V,E) describes the standard mathematical definition of a graph [W+01]. Here V is a non empty set of vertexes (also called nodes) and E is an edge set, where each edge has either two vertexes associated with it (though they can be the same vertex) as endpoints,

\[ E \subseteq V \times V \times L \]

L is a set of labels that are attached to the different nodes and relationships. Labels are a named graph construct. They are used to group sets of nodes. Nodes can have multiple labels and they can also be queried using labels. Edges also carry labels. For example in Figure 2.3, label Tournament and Game are related by the relationship HAS_GAMES, so the labels assigned represent the kind of relationship linking these nodes, and the role that each plays.

For modeling real-world observations as a graph, it is important to note that in nodes we attach properties to represent entity attributes (e.g. in our example we can see that Tournament is attached to the property name:'League of legends'), while in relationships we tend to attach properties to represent the strength/weight or quality of that relationship. Graphs are a useful way of modeling variably structured data.

As mentioned previously, labels can be used for querying. One query on labels could ask the database to find all of the nodes labeled Player. Labels can also be used for
indexing. For example, we can tell the database to index all of the nodes in the dataset which contain the label Player. As a result, labels are essential for managing graph data, acting as an approach for typing the entities. Typed entities can also have their own schema.

Apart from labels, we can further extend graphs such that nodes/edges carry extra properties. This enhances the expressiveness of graph models.

![Graph model with Node Properties](image)

Figure 2.3: Graph model with Node Properties

Models are not just important for discussing graphs, they are specifically essential for graph databases, because a clear model provides a way to standardize the features that can be expected from using the graph database [Why17].

Models for databases can be characterized by three basic components, logical data components, query and transformation languages, and integrity constraints. Based on this, graph database models present a structure where data and the operations follow graph models and integrity constraints can be defined using such models. These characteristics make graph database models easy to apply for representing unstructured data.

An important aspect of these models is that in databases built over such models, the separation between schema and data is less strict or evident than in the classical relational model [Why17].

In the next section we will formally present the property graph model and the RDF model, which are two underlying data models which are employed as logical models for designing graph databases.
2.1. Networks and Graph Models

2.1.2.1 Property Graph Model

Property graphs are one of the standard logical models for working with graphs. Property graphs are named as such because they allow to store properties for nodes and relationships. Furthermore, relationships and nodes will have one special attribute that allows to group them in classes, for nodes this attribute are labels, for relationships these are types. Thus, property graphs are attributed, labeled and typed. In addition, relationships can be directed, which are represented with arrowheads, as shown in Figure 2.4. Vertexes and edges are equipped with collection of key-value pairs. Property graphs are highly expressive and hence used by most graph models [Tho18].

Property graphs contain nodes (vertexes) and relationships (edges); nodes and relationships may or may not have properties. Nodes represent entity types. Relationships represent the connectedness, and since they are named they bring clarity and context to the nodes. Hence, the labeled property graph model is a general purpose graph data model which provides structure (connectedness) and meaning (naming and labeling) for modeling networks as graphs, and for storing such data.

Apart from the property graph, there are different types of graphs which are common in theory. Here we list some of these types [RN10]:

- **Directed graph**: Graphs in which edges are directed (asymmetric relationship). The line connecting vertexes has an arrow at the end.
• **Undirected graph**: It’s a special kind of directed graph since every undirected (symmetric relationship) edge represents a relationship that goes both ways. It is equivalent to two edges going in opposite directions.

• **Multi graph**: There can be multiple edges between the same two vertexes.

• **Weighted graph** (directed/undirected): Sometimes edges carry weights, weights are numbers. They are used to represent the strength of ties or transition probabilities.

• **Vertex/Edge-labeled graph**: Vertexes can be labeled or the edges can be typed to represent the way in which two vertexes are related (e.g. friendships).

• **Vertex/Edge-attributed graph**: Attributes can be appended as meta data to vertexes or edges.

• **Semantic graph**: It models cognitive structures such as the relationship between concepts and the instances of those concepts. Unlike the other definitions given in this list, this type of graph can be generalized to constitute a graph database model.

• **Half-edge graph**: Graphs that contain unary edges (i.e. an edge that only “connects” one vertex).

• **Pseudo graph**: It’s used to denote a reflexive relationship.

• **Hyper graph**: An edge could connect an arbitrary number of vertexes. This type of graph can also be used as a basis for a graph database model.

• **RDF graph**: A Resource Description Framework (RDF) graph restrict the vertex/edge labels to Uniform Resource Identifiers (URIs). RDF graphs can be semantic graphs. Through its use of URIs for referring to real-world entities, this model is connected to the semantic Web.

For our study we focus on the **property graph model**, which is supported by most graph systems. The property graph model, also known as “property graph”, is a directed, labeled, attributed, multi-graph. Graphs of this form allow for the representation of labeled vertexes, labeled edges, and attribute meta data (i.e. properties) for both vertexes and edges. Figure 2.6 gives an example of property graph. The high expressiveness of the property graph, which can express also RDF, makes the property graph one of the most popular graph data type.

### 2.1.2.2 RDF Graph Model

The RDF model, also called a Triple store is a simple yet expressive model used for managing graph data. It defines data in the form of subject predicate object items, called
2.2. Graph Data Management

Figure 2.5: There are numerous types of graphs. Many of the formalisms described here can be mixed and matched in order to provide the modeler with more expressiveness \( [\text{RN10}] \).

RDF triples \( [\text{AG17}] \). RDF can be used to model ontologies and semantic information, including linked open data and the semantic web.

SPARQL is a standard query language for RDF databases. It supports advanced features like property paths, aggregate functions and sub queries \( [\text{AG17}] \).

With this we conclude our presentation on graph models, which are necessary to introduce the fundamentals for graph data management. Next will discuss about Graph data management.

2.2 Graph Data Management

Graph Data Management can be classified into two groups: Graph databases and Graph processing frameworks. Although these two groups are able to handle similar workloads, they provide different approaches for storing and querying the graph data.

Graph databases aim at persistent management of the graph data, allowing to store and access graph data a persistent state. Furthermore they support multiuser access and can have ACID guarantees. Graph databases provide database services like user interface, query language, data definition and manipulation, query optimizer, storage engine, database engine, tuning, backup and recovery.

Graph processing frameworks provide large-scale processing and analysis of large graphs in a distributed environment with many machines. They do not focus on multi-user support, but instead focus on optimizing a distributed bulk or streaming execution.
Figure 2.6: A property graph is a directed, labeled, attributed, multi-graph. The edges are directed, vertexes/edges are labeled, vertexes/edges have associated key/value pairs meta data (i.e. properties), and there can be multiple edges between any two vertexes.

In these systems graphs are processed in memory and are managed by distinct and distributed nodes. Some programming abstractions are offered to end-users for working with these systems, like vertex-centric or graph-centric processing [YBT+16].

2.2.1 Graph Databases

These are mainly used for storing and querying graph data. The design of graph databases is based on the graph model, which determines the data structures, query operations and integrity constraints. Most graph databases implement either the property graph model or the RDF graph model.

Graph databases have been classified by authors into two categories: Native graph storage and non-native graph storage [RWE13b]. Native graph storage provides specialized indexes and data layouts for storing and querying graphs. Non-native graph storage use other types of database systems (e.g. relational databases or document stores) to store graph data they also implement query interfaces to execute the graph queries.

Index free adjacency is the key differentiator of native graph processing. It makes sure that for all nodes their relationships are stored adjacent to them, and thus can be accessed directly from the node without requiring further global lookups. During query processing this helps in speeding up the retrieval of data without a need for indexes [RN12]. ACID guarantees can be provided to ensure that the stored graphs aren’t corrupted over time.
AllegroGraph, Bitsy, Cayley, GraphBase, Graphd, HyperGraphDB, IBM System G, imGraph, InfiniteGraph, InfoGrid, Neo4j, Sparksee/DEX, Trinity and TurboGraph come under the group of Native graph storage. Under the group Non-native graph storage are JanusGraph (which supports Apache Cassandra, Apache HBase and Oracle BerkeleyDB as storage backends), FlockDB (a distributed graph-oriented database which uses MySQL as the storage engine), OrientDB and ArangoDB (which are document-store databases adapted to graphs), OQGRAPH (a graph computation engine for MySQL), VelocityGraph (an object database supporting graphs), Horton (based on the cloud programming infrastructure Orleans), and Oracle extensions for managing spatial and graph data as explained in [AG17].

2.2.1.1 Property Graph Databases

In this section we present 2 specific property graph databases: Neo4j and JanusGraph.

Neo4j-Native storage

The architecture of Neo4j is presented in Figure 2.8. From this image we can understand the following:

The Neo4j Native graph database architecture consists first of access components, which include support for the Cypher language, for a Core API (for low-level operations) and for a Traversal API. Cypher is the declarative language used by Neo4j. Using the Traversal API Neo4j can offer a data traversal interface for developers using Java or supported high-level languages.

The page cache contains temporary storage for Neo4j, helping to manage data residing on disk. The Transaction Management component satisfies the ACID properties of Neo4j, which are useful in maintaining data integrity. The transaction log contains the details of operations performed by Neo4j, enabling recovery and roll-backs.

Neo4j Storage

Based on descriptions by authors from the Neo4j system [RWE13a], we can explain the storage of this database as follows:

The data in Neo4j is stored through the component called Record Files, which manages store files. Neo4j stores nodes, relationships, labels, and properties in separate store files. Nodes are stored in a specialized node store file called neostore.nodestore.db, each record in a node store is defined to be 15 bytes in length. Thus, records are fixed-size by default. The first byte in a node record is the in-use flag. This tells the database whether the record is currently being used to store a node, or whether it can be used to store a new node. The next four bytes represent the ID of the first relationship connected to that node, and the following four bytes represent the ID of the first property for that node. The subsequent five bytes for labels point to the label store entry for this node. The final bytes are reserved for extra flags. One of these is a flag used to identify if nodes are densely connected. The rest of the space is reserved for future use.
### Figure 2.7: Comparison of Graph Databases

|                     | Neo4j | Amazon Neptune | Google Spanner | Amazon Timestream | Gexf | AllegroGraph | AllegroGraph Studio Edition | AllegroGraph Desktop | AllegroGraph Engine | AllegroGraph Studio Edition | AllegroGraph Desktop | AllegroGraph Engine | AllegroGraph Engine | AllegroGraph Engine | AllegroGraph Engine | AllegroGraph Engine | AllegroGraph Engine | AllegroGraph Engine | AllegroGraph Engine |
|---------------------|-------|----------------|----------------|-------------------|------|--------------|----------------------------|---------------------|----------------------|----------------------------|--------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| **Single-mode**     | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Multi-mode**      | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Integration**     | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Storage**         | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Graph Algorithms**| X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Query Languages** | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **API**             | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Graph Models**    | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |
| **Data Model**      | X     | X              | X              | X                 | X    | X            | X                          | X                   | X                    | X                            | X                  | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    | X                    |

**Note:** The table above shows a comparison of various graph database systems. Each row represents a feature, and the columns represent different graph databases. The checkmark (X) indicates the presence of the feature, while the absence is denoted by a blank space.
Relationships are stored in a relationship store file called neostore.relationshipstore.db, this store consists also of fixed-sized records of 33 bytes in length. Each relationship record contains the 8 bytes for IDs of the nodes at the start and end of the relationship, 4 bytes contain a pointer to the relationship type (which is stored in the relationship type store), 16 bytes are pointers for the next and previous relationship records for each of the start and end nodes, and a flag indicating whether the current record is the first in a so-called relationship chain.

Additional properties of vertexes and edges are physically stored in the neostore.propertystore.db file, like node and relationship stores, property records are of a fixed size.

The use of fixed-size records for nodes helps to search for them in the store file with a relative ease. Based on the storage format, we can say that graph database could be good for local access patterns.

JanusGraph

JanusGraph is a graph database based on non-native storage, which is used for storing and querying graphs, usually across multi-machine clusters. Unlike Neo4j, which has a specific storage model for storing graph data, JanusGraph’s data storage layer is pluggable, which means a middleman software component tells JanusGraph how to communicate with the data store. Implementations of the pluggable storage layer are called storage backends. JanusGraph supports for various storage backends: Apache Cassandra, Apache HBase, Google Cloud Bigtable and Oracle BerkeleyDB. JanusGraph supports the Gremlin
query language, which is supported by the general TinkerPop graph query framework, which acts as a query engine for JanusGraph.

After presenting Neo4j and JanusGraph as two examples of property graph databases, we talk about RDF databases in the next small section.

2.2.1.2 RDF Databases

Databases supporting the RDF model can also be classified into native and non-native RDF databases [AG17]. Jena, RDF-3X, 4store, and TripleBit are some of the native RDF databases. Among the non-native RDF databases are OpenLink Virtuoso, Sesame, and DB2RDF, which are implemented on top of relational database systems. Furthermore, RDF can store data in four categories: triples table, property table, column store, and RDF graph base store [AG17].

RDF uses the triple store structure where RDF data is stored in a 3-column table where each column consists of Subject, Predicate, and Object. To evaluate the SPARQL queries self-joins are used. To improve the performance of queries in this storage model, exhaustive indexing methods are adopted that create a full set of SPO permutations of indexes. RDF-3X, Sesame, 3store, BrightstarDB are some of the systems implementing triple tables. The second approach is to store RDF data in a property table with Subject as the first column and the list of distinct predicates as the remaining columns. A single property table can be extremely sparse and contains many null values. Thus multiple-property tables with different clusters of properties are proposed as an optimization technique. Jena, Oracle, and BitMat are examples of systems implementing property tables.

RDF data can also be stored by using multiple two-column tables, one for each unique predicate. The first column is for Subject whereas the other column is for Object. This
method is called column store with vertical partitioning, can be implemented over row-oriented or column-oriented database systems. Finally, a graph based approach focuses on storing RDF data as a graph. In this case, the RDF triples must be modeled as classical graph nodes and edges, and the SPARQL queries must be transformed into graph queries. Ontotext GraphDB, gStore, Stardog, Blazegraph, TrinityRDF and GE are some of RDF graph based stores [De18].

2.2.1.3 Query Processing and Optimization

Using Query Processing frameworks we can transform high level graph queries into simple and efficient low level language programs. Query optimization helps to find out the efficient execution plan that will help to minimize the runtime. Query execution plans are a series of steps that a query undergoes when it is received by the database system for update or retrieval of information.

2.2.1.4 Query Languages

Most of the relational databases use structured query language. But as graph databases emerged the requirement for a powerful query language has become essential. Graph Models like Figure 2.10 are perfect for describing the graph database, but when it comes to storing the data in database to retrieve, create and manipulate data we need a query language. Query languages are similar to human readable languages used to communicate with the database. Just as graph database models are easy to understand even the query languages in Graph Databases are made easier to understand. Some of them are Cypher, Gremlin, SPARQL, etc. More recently there has also been an effort from the LDBC Council to introduce more expressive languages with a built-in support for path-query evaluation [AAB+18].

We will talk about Gremlin and Cypher next. These are examples of imperative and high-level pattern-matching languages [Bry15]:

SQL queries are more about joining tables and graph queries are about connecting nodes using relationships. Neo4j’s Cypher helps to represent graphs as diagrams, making it a suitable language for describing graphs. In Figure 2.10 we show an example of a query pattern for three mutual friends; we can express this pattern in Cypher using an example from other authors [RWE13a]: \((emil)\leftarrow[:KNOWS]-[jim]][:KNOWS]-\rightarrow(ian)\) where \emil, textit jim, ian\ are the nodes which are connected by the relationship \textit{KNOWS}. We can see that Cypher naturally follows the way we draw the graph, with the additional observation that it supports pattern-matching queries.

Gremlin is a query language from JanusGraph used for storage and retrieval of data and also modify data in graph. It easily expresses complex graph traversals and mutation operators. It is a path oriented language that shares the approach of function chaining that is used in functional programming. Gremlin queries are evaluated from left to right and are written as a chain of operations/functions. For example, let us consider a gremlin query: \(g.V().has('name','hercules').out('father').out('father').values('name')===>saturn\ [Jan17], where
2. Background

Figure 2.10: Simple Graph Database Model

- g: is current graph traversal.
- V: represents all the vertexes in the graph.
- has('name', 'hercules'): filters the vertexes based on the property name 'hercules'.
- out('father'): traverse through all the outgoing edges of father from Hercules.
- out('father'): traverse through all the outgoing edges of father from Hercules’ father’s vertex (i.e. Jupiter).
- name: from ‘hercules’ vertex’s grandfather get the property name.

All together these steps form a path-like traversal query. Results can be demonstrated by decomposing each step. While constructing larger complex query chains this way of representing traversal/query can be helpful.

Gremlin is dedicated for traversal querying and best for high level traversing, but Cypher, being more imperative can, in principle be further optimized to find the best traversing solution.

For illustration purposes we show example query snippets for comparing SQL Figure 2.11 with CYPHER queries Figure 2.12. In SQL we see, just a SELECT DISTINCT from the OrganizationName table. In Cypher, the same query is mapped to MATCH, a simple pattern that contains all nodes with the label OrganizationName, and RETURN DISTINCT among them. In our example Figure 2.11 joins are performed on three tables.
university, student name and university address to get the University 'Ovgu' and we can see that in Figure 2.12. JOINS are much simpler in the CYPHER query when compared with the SQL query.

```sql
SELECT DISTINCT c.OrganisationName
FROM university AS c
JOIN Student_name AS s ON (c.StudentID = s.StudentID)
JOIN university_address AS ua ON (s.UniversityID = ua.UniversityID)
WHERE c.UniversityName = 'Ovgu';
```

Figure 2.11: SQL Query

```cypher
MATCH (:University {UniversityName:'Ovgu'})
  <-[:STUDIES]-(:Student)
RETURN DISTINCT c.OrganisationName AS Organisation;
```

Figure 2.12: CYPHER Query

### 2.2.1.5 Relational Databases and Full-Text Search Engines

Traditional Relational Databases are power houses of many software applications since the 1980s. Full text search engines evolved much later than them, with the advent of the Web. The emergence of unstructured text data and the need for querying such data with relevance ranking motivated organizations to use non-traditional databases and, as a result, text search engines emerged after traditional databases as a solution to store and query unstructured text data. The development also forced many companies to move the entire system into text search engine [Mil09].

Indexing (i.e., the storing of data in a search engine) begins when data is inserted into the main document index, this index inserts data in each row of the document. Additional fields like document title can be inserted at the same time. If the data is taken from relational databases, the primary key is represented as a Key in the document index. After inserting data the external document is opened by the indexing engine, where it creates an ordered word list to load to the main index. This process usually involves parsing and stemming (aspects related to text processing), and the creation of postings lists, a form of index which enable the system to track the mention of key terms in documents, next to the position in which such mention occurred. These lists also track the frequency of term usage per documents, enabling in turn a different ranking per match using scoring techniques like TF-IDF [BYRN+09].

This process for indexing is repeated for each document (which can also correspond to a row in a relational database). When users input a query or when the query arrives programmatically, the full text searches in the sorted word index and identifies the document that contains the requested term. Finally the engine creates the list of documents that qualify the search request, ranking the results according to internal
scoring functions. Some of the advantages of Search Engines over traditional databases are:

- Full text search engines are particularly used to handle textual data.
- Index structure of full text search engine can facilitate certain types of global queries.
- These Search Engines also offer query operators like word density, statistically based similarity, language stemming.
- They allow for hybrid searches that can search in both textual data and traditional fielded data in the same query.
- Using relevancy weights it offers fine tuned search results that can serve different user expectations.

2.3 Search engine: ElasticSearch

Search Engines (SEs) are apparently the biggest database management systems in the world. For example, commercial search engines handle very large amounts of data and, in the case of Google, more than 3.5 billion queries per day.\(^1\)

\(^1\)For example, see: [http://www.internettlivestats.com/google-search-statistics/](http://www.internettlivestats.com/google-search-statistics/)
ElasticSearch is a free open source search engine, or a distributed inverted index. It is built on top of Apache Lucene. Apache Lucene is a high performance Java library for search engine core components, including scoring function and writing files to disk. ElasticSearch uses Lucene for distributed full text search and analytics capabilities.

**How do Search Engines store the data?**

Traditionally data has been stored in the form of rows and columns in a relational database which was equivalent of using spreadsheet. In a search engine an index is like a database which helps the user to search for documents. Below Figure 2.14 is a comparison of ElasticSearch along with SQL databases.

<table>
<thead>
<tr>
<th>MySQL</th>
<th>ElasticSearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Index</td>
</tr>
<tr>
<td>Table</td>
<td>Type</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
</tr>
<tr>
<td>Schema</td>
<td>Mapping</td>
</tr>
<tr>
<td>SQL</td>
<td>Query DSL</td>
</tr>
<tr>
<td>SELECT * FROM</td>
<td>GET <a href="http://localhost:9200">http://localhost:9200</a>...</td>
</tr>
<tr>
<td>table...</td>
<td>PUT <a href="http://localhost:9200">http://localhost:9200</a>...</td>
</tr>
<tr>
<td>UPDATE table</td>
<td></td>
</tr>
<tr>
<td>SET...</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.14: Parallel concepts between ElasticSearch and SQL Database

ElasticSearch is a distributed document store. It stores documents in the form of JSON format. We can host multiple indexes on one ElasticSearch installation. Each index will consist of multiple 'types'. In ElasticSearch we can query multiple types and multiple indexes in one single query. As it stores individual fields in a document which is indexed; it can store and retrieve complex data structures, such as geographic spatial data. As soon as a document is stored in ElasticSearch, it can be retrieved from any node in the cluster. Here the data in all fields are indexed by default, which means every field has a committed inverted index which helps in fast retrieval of data. Optimistic version control is used, when needed, to ensure that data is not lost while being accessed by multiple processes \[ec18\].

**2.3.0.1 Document Indexing:**

ElasticSearch is comprised of *clusters* which consist of one or more nodes with the same cluster name; and each cluster name consists of one master node, and *nodes* that communicate with each other to read and write to an index. Clusters should also have a unique name to prevent unnecessary joining of nodes \[ec18\].
A document not only consists of data but also of the _index itself, which is where the document is stored, the _type which is the class of the object that the document contains and the _id, a unique identifier that represents the document.

Now let's provide more detail of these three elements:

- **_index**: It is a collection of documents that are grouped together. For example, all the Person details are stored in a Person index and all the Message details will be stored in a Message index.

- **_type**: It is a concept akin to a ‘table’ in a relational database. Each type is defined as a list of fields that can be specified for documents of that type. The mapping process from input data to document types can be user-defined, determining how each field in the document is analyzed.

- **_field**: Documents consist of a list of fields or key-value pairs. The value can be simple value like strings, integers or dates; or nested structures like an array belonging to an object. Fields are similar to a column in a table of a relational database. Each field has a field type which shows the type of data that can be stored in that field.

ElasticSearch consists of shards, which are single Lucene indexes, these indexes point to primary and replica shards. Input data is distributed across shards.

Fast data retrieval in ElasticSearch is achieved by using Lucene. Each index in ElasticSearch is comprised of shards across one or many nodes. In Figure 2.15 we see an example of an ElasticSearch cluster with two nodes, two indexes (Index-node-Person and Index-node-Country) and five shards in each node, it can also have zero to many replica shards that duplicates the data. In Figure 2.15 there are three primary shards and three replica shards. The first write happens in the primary shard. Primary shards are not limited to a single node. If any node fails, a functioning node from the replica shards is promoted to a primary shard status. Before the data is moved to a replica shard it is written by a primary shard, such that it can hold the latest values. Data can be read from both primary and replica shards.

We can interact with ElasticSearch using the Json Rest API. It consists of CAT commands which tells the status of cluster.

**ElasticSearch Storage**

Lucene spreads its index across several disk files, each format is built for a specific purpose. These files are organized into logical “segments” that represents subsets of documents across the corpus i.e, the entire set of documents in an index. For instance if there are two documents doc1 and doc2 which contain the field “tag” and type “string”, with text “small data” and a doc3 with the same structure but the tag contains the string “big data”. Lucene stores the data in the form of inverted index as follows:
2.3. Search engine: ElasticSearch

Figure 2.15: Shards across two nodes

1. big=[doc1, doc2]
2. data=[doc1, doc2, doc3]
3. small=[doc3]

So we can say that it is an index of terms to the document but not index of documents to terms. When we search for “big” we get the document doc1 and doc2. If we search for “small” we get document doc3. And finally when we search for “data” we get all the three documents. This is the core data structure in Lucene and is called Inverted index.

Each segment index Figure 2.16\[2\] maintains the following:

- Field Names: This contains the set of field names used in the index.
- Stored Field Values: This contains, for each document, a list of attribute-value pairs, where the attributes are field names.

[https://blog.parse.ly/post/1691/lucene/]
• Term Dictionary: A dictionary containing all of the terms used in all of the indexed fields of all of the documents.

• Term Frequency Data: For each term in the dictionary, the numbers of all the documents that contain that term, and the frequency of the term in that document.

• Term Proximity Data: For each term in the dictionary, the positions that the term occurs in each document.

Index store large amount of data that can exceed hardware limit of a single node. For instance if there is a single index with billions of documents in it and take up 1TB of disk space, it may not fit in single node or the performance might get lowered. So to solve this issue ElasticSearch introduces concept called Shards where the index is sub divided into pieces called shards [Figure 2.17](https://www.slideshare.net/inovex/realtime-data-analytics-mit-elasticsearch). Each shard is a full functional independent “index”. A shard is a single lucene index, it allows to scale/split data horizontally and allows to distribute and parallelize operations across shards which helps to increase the performance/throughput. In case if primary shards goes offline or fails for whatever reason, ElasticSearch allows to make one or more copies of these primary shards called Replica Shards. Which can be used when primary shard fails, it provides high availability and allows to scale out the search volume as searches can be executed in replicas in parallel. By default ElasticSearch has 5 Primary Shards and 5 Replica Shards, makes a total of 10 shards per index.
2.4. Integration of Graph Databases and Search Engines

Connected data is all around us and enables new types of applications. As we already discussed, Graph Databases are developed to store, manage and query data efficiently that are highly variable and richly connected. Graph queries allow to access data exploiting connections between elements, but they don’t enable efficient textual searches. On the other hand, search engines like ElasticSearch provide fast, reliable, and easy-to-tune textual searches.

In order to leverage such functionalities and provide efficient search to end users, data can be projected from the graph in a document format and stored in indexes in ElasticSearch. There, such documents can be analyzed according to the mapping defined for the index, and then become available for access using textual searches.

The updates in the graph database should be reflected in the ElasticSearch indexes. Ideally, the users should be able to define the graph entities to be replicated as well as the semantics of the documents that will be sent to ElasticSearch. We can call this configuration “replication mapping”.

Graph Databases usually provide an integration with Search Engines to provide the full-text search functionality. Some of the Graph Databases that offer integration with Search Engines are Neo4j JanusGraph, ArangoDB, and OrientDB.

Based on the storage format only, we can plausibly expect that the Search Engine could be good for global access patterns.

Figure 2.17: ElasticSearch Architecture
Orient DB is a multi-model database management system which supports Document stores, Key-value stores, and Graph DBMS. OrientDB has one index per OrientDB node. It supports replication to ES, with its own replication and consistency mechanisms. The goal of indexing in ElasticSearch is to be able to query the data with full-text search. When the replication process starts scanning OrientDB, first it creates/replaces a unique vertex, for example, let's call it “checkpoint vertex,” which contains the start date of the scan. Every time an application modifies OrientDB, it reads the checkpoint vertex and sets the modification date of each indexed vertex/edge to its date. If a scan is started during the modification, the checkpoint vertex has been changed and the transaction will fail. For deletes, a vertex describing the delete has to be created. There are some drawbacks with this:

- The application has to know either what is indexed in ES, or it has to set a date on every vertex/edge.
- Transactions have to be used when we want to modify one vertex/edge.

With this we conclude our presentation on graph databases, search engines, and their integration. In the next section we discuss the benchmark that we used to evaluate potentials exposed by the graph database/search engine integration: the LDBC SNB Benchmark.

### 2.5 Benchmark Used: LDBC

The new era of data economy, based on large, distributed, and complexly structured data sets, has brought on new and complex challenges in the field of data management and analytics. These data sets, usually modeled as large graphs, have attracted both industry and academia, due to the new opportunities in research and innovation that they offer. This situation has also opened the door for new companies to emerge, offering new non-relational and graph-like technologies that are called to play a significant role in the upcoming years.

The change in the data paradigm calls for new standardized benchmarks to test these new emerging technologies, setting up a framework where different systems can compete and be compared in a fair way. Such benchmarks aim to help technology providers to identify the bottlenecks and gaps of their systems and, in general, drive the research and development of new information technology solutions. Without the existence of standard benchmarks, the uptake of these technologies is at risk by not providing the industry with clear, user-driven targets for performance and functionality.

The LDBC Social Network Benchmark (LDBC SNB) aims at being a complete standard benchmark by setting rules for the evaluation of graph-like data management technologies. LDBC SNB is designed to be a plausible look-alike of all the aspects of operating a social network site, as one of the most representative and relevant use cases of modern graph-like applications. LDBC SNB includes the Interactive Workload which consists...
2.5. Benchmark Used: LDBC

of a user-centric transactional-like workload interleaved with simple analytical queries; and the Business Intelligence Workload, including more complex analytic queries to respond to business-critical questions. Initially, a graph analytics workload, for large-scale processing, was also included in the roadmap of LDBC SNB, but this was finally delegated to the Graphalytics benchmark project, which was adopted as an official LDBC graph analytics benchmark. LDBC SNB and Graphalytics combined target a broad range of systems with different nature and characteristics, aiming at capturing the essential features of these scenarios while abstracting away details of specific business deployments [Erl15], [ATB+14], [Bon13], [SPPA+18].

LDBC is a EU project responsible for specifying benchmarks, benchmarking procedures and with the goal to develop industry-strength benchmarks for graph and RDF data management systems. The LDBC benchmark council was co-founded by graph database companies like Neo Technologies and Sparsity Technologies, and the RDF database companies Ontotext and OpenLink Systems [Erl15].

LDBC SNB aims at being a complete benchmark, designed with the following goals in mind [Erl15]:

- **Rich coverage**: LDBC SNB is intended to cover most demands encountered in the management of complexly structured data.

- **Modularity**: LDBC SNB is broken into parts that can be individually addressed.

- **Reasonable implementation cost**: For a product offering relevant functionality, the effort for obtaining initial results with SNB should be small, in the order of days.

- **Relevant selection of challenges**: Benchmarks are known to direct product development in certain directions. LDBC SNB is informed by the state-of-the-art in database research so as to offer optimization challenges for years to come while not having a prohibitively high threshold for introduction of new challenges.

- **Reproducibility and documentation of results**: LDBC SNB will specify the rules for full disclosure of benchmark execution and for auditing of benchmark runs. The workloads may be run on any equipment but the exact configuration and price of the hardware and software must be disclosed.

Three different formats are supported by DATAGEN, the tool that generates data (i.e., graphs) according to specifications, of LDBC benchmarks [Erl15]:

- **CSV**: Data output in CSV format, one file per different entity and on file per different relation. Also, there is a file per those attributes whose cardinality is larger than one (i.e. Person.email, Person.speaks, etc).

- **CSVMergeForeign**: Similar to CSV format, but in this case, those relations of the form 1 to 1 and 1 to N, are stored in the tail entity file as a foreign keys.
Turtle: Dataset in Turtle format for RDF systems.

Graph Patterns: To illustrate queries, we use graph patterns as shown in Figure 2.18, with the following notation:

- Nodes are marked as entityName: EntityType (camel case notation for both, starting with a lowercase character for the first and an uppercase character for the second). If the entityName is not used in the query results, aggregations or calculations, and not referenced in the query specification, the entityName can be omitted.
- Positive conditions for edges are denoted with solid lines.
- Negative conditions for edges, i.e. edges that are not allowed in the graph, are denoted with dashed red lines.
- Edges without direction imply that there must be an edge in at least one of the directions.
- Filtering conditions are typeset in italic, e.g. id = $tag.
- Attributes that should be returned are denoted in sans-serif font, e.g. name.
- Variable length paths, i.e. edges that can be traversed multiple times are denotes with min...max, e.g. replyOf or knows 1 . . . 2. By default, the value of min is 1, and the value of max is unlimited.
- Aggregations are shown in dashed boxes with the type of aggregation (count, sum, avg, etc.) in the upper right corner.

Social Network Benchmark: It is the first LDBC benchmark which has a social network similar to Facebook. The dataset consists of Person, which has majority of data as Messages in the form of Post, comments or as discussions. Main aim of SNB is to make the generated data more realistic as possible for benchmark queries to exhibit the desired effects.
Apart from this schema and benchmark, LDBC has developed the Semantic Publishing Benchmark (SPB): Used to manage RDF system using data publishing case. Inputs provided to this benchmark is mainly by media organizations which makes heavy use of RDF. It mainly deals with large volume of streaming content, news articles from media or publishing organizations. SPB consists of a Data Generator and a Query Driver which has two workloads: basic and advanced. Data generator is used for producing synthetic data, it consists a set of real knowledge data and ontologies provided by GeoNames, The BBC, and DBpedia. Various scales of data can be generated by SNP data generator that is from 1M triples to billion. Data generated is saved in proper RDF format. The data generated can be loaded into the RDF benchmark using test driver. Once the data is loaded query substitution parameters are generated by analyzing the various statistics about the loaded data.

2.6 Summary

In this section we presented the relevant related work which is the basis of our work for evaluation of graph databases, and search engines over graph queries. We introduced networks and graph data models and graph data management; graph databases and search engines and also the integration of graph databases with search engines. We also discussed about the benchmark used.

In the next chapter we introduce our research questions, before moving on to the evaluations. We present the case study from the benchmark that we used for data generation and data loading, along with the GraphAware neo4j-to-elasticsearch replication module and the experimental settings.
3. An Evaluation of Graph Prose: 
Graph PROcessing with Search Engines

In this chapter, we introduce the precise evaluation methods that we seek to use in our research. The outline for this chapter is as follows:

- **Research Questions**: First we provide several research questions that we aim to address in our study (Section 3.1).
- **Experimental Setup**: Here we discuss about the Architecture, Configuration, Benchmarks and Hardware used for our experimental evaluation (Section 3.4).
- **Summary**: We recapitulate the main points of the chapter (Section 3.5).

### 3.1 Research Questions

In order to study the potential uses of the search engine to support graph queries we propose the following research questions:

1. What are possible mappings between graph entities and search engine documents? How do these different mappings impact the utility of the search engine for graph queries?

2. What are the gains possible from rewriting global graph queries (i.e., no more than 1 hop involved) such that they could be executed directly in the search engine, assuming a naive mapping?

3. What are the gains possible from rewriting adjacency queries (i.e., more than 1 hop involved) such that they could be executed directly in the search engine, assuming a naive mapping?
To answer our research questions we have selected the following:

- **As a case study**: LDBC SNB Benchmark, focusing on the BI queries. We have selected this as a generally accepted standard benchmark with an expressive set of queries.

- **As systems for evaluation**: Neo4j with GraphAware and JanusGraph, and ElasticSearch. We have selected these systems both as examples of property graph databases, and as examples of native and plugin-based support for graph-to-search-engine integration.

### 3.2 Case Study: LDBC SNB Benchmark

In this section we discuss the LDBC SNB Benchmark, we explain the schema, consisting of entities and relationships from a social network model as it evolves through years. In addition we explain the different scale factor configurations for the data generator and the business intelligence queries.

#### 3.2.1 Data Schema

The schema defines the structure of the data used in the benchmark, which specifies different entities, their attributes and their relations. Data includes entities such as Person, Organizations, Places, etc; corresponding to the goal of the benchmark to represent a social network [ABLP+14].

#### 3.2.2 Entities

- **City**: A sub-class of a Place. City entities are used to specify where person is located in, as well as where universities are located in.

- **Comment**: A sub-class of a Message, and represents a reply(comment) made to an existing message by person.

- **Company**: A sub-class of an Organization, that represents where persons work and which country the company is located in.

- **Country**: A sub-class of a Place, and represents a continent of the world, also represents where the person who created message is located in.

- **Forum**: Point where people post messages. Forums consists of tags(topics) which people in the forum are talking about. It has a member and a moderator. They are distinguished by their titles. The attributes of Forum entity are id, title, creationDate.

- **Message**: Entity that represents a message created by a person, likes to the message and replies to the message. The attributes of Message abstract entity are id, browserUsed, Location, creationDate, Content, Length.
3.2. Case Study: LDBC SNB Benchmark

Figure 3.1: LDBC SNB Data Schema

• **Organization**: Entity which is an institution of the real world. Name, id, url are the attributes of Organization entity.

• **Person**: Real world entity created by person when he/she joins the network, and contains various information about the person. Person interact by friendship relations and sharing messages, replies to messages and likes to messages. Id, firstName, lastName, gender, birthDay, email, speaks, browserUsed, locationIP, creationDate are the attributes of Person entity.

• **Place**: Place in the world which is an entity with City, Country and Continent as sub-classes. Id, name, url are the attributes of Place entity.

• **Post**: A sub-class of Message, that is posted in a forum. Posts are created by persons into the forums where they belong. Posts contain either content or imageFile, always one of them but never both. The attributes of Post entity are language and imageFile/content.
**Tag:** Tags (topics/post) are used to specify the forum topics, as well as the topics a person is interested in. Tag can have type TagClass. Id, name, url are the attributes of Tag entity.

**TagClass:** A class used to build a hierarchy of tags. TagClass can be subclass of itself. Id, name, url are the attributes of TagClass entity.

**University:** A sub-class of Organization, and it represents an institution where persons study.

### 3.2.3 Relations

In the above section we have discussed about the entities in the LDBC data, now we will discuss about the relationships in the LDBC data. Entities and Relations are connected by the “id” attribute.

In the Table 3.1 we list the relationships and their relationships with entities:

<table>
<thead>
<tr>
<th>Relationships</th>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>containerOf</td>
<td>Forum[1]</td>
<td>Post[1..*]</td>
</tr>
<tr>
<td>hasCreator</td>
<td>Message[0..*]</td>
<td>Person[1]</td>
</tr>
<tr>
<td>hasInterest</td>
<td>Person[0..*]</td>
<td>Tag[0..*]</td>
</tr>
<tr>
<td>hasMember</td>
<td>Forum[0..*]</td>
<td>Person[1..*]</td>
</tr>
<tr>
<td>hasModerator</td>
<td>Forum[0..*]</td>
<td>Person[1]</td>
</tr>
<tr>
<td>hasTag</td>
<td>Message[0..*]</td>
<td>Tag[0..*]</td>
</tr>
<tr>
<td>hasTag</td>
<td>Forum[0..*]</td>
<td>Tag[0..*]</td>
</tr>
<tr>
<td>hasType</td>
<td>Tag[0..*]</td>
<td>TagClass[0..*]</td>
</tr>
<tr>
<td>isLocatedIn</td>
<td>Company[0..*]</td>
<td>Country[1]</td>
</tr>
<tr>
<td>isLocatedIn</td>
<td>Message[0..*]</td>
<td>Country[1]</td>
</tr>
<tr>
<td>isLocatedIn</td>
<td>Person[0..*]</td>
<td>City[1]</td>
</tr>
<tr>
<td>isLocatedIn</td>
<td>University[0..*]</td>
<td>City[1]</td>
</tr>
<tr>
<td>isPartOf</td>
<td>City[1..*]</td>
<td>Country[1]</td>
</tr>
<tr>
<td>isPartOf</td>
<td>Country[1..*]</td>
<td>Continent[1]</td>
</tr>
<tr>
<td>isSubclassOf</td>
<td>TagClass[0..*]</td>
<td>TagClass[0..*]</td>
</tr>
<tr>
<td>knows</td>
<td>Person[0..*]</td>
<td>Person[0..*]</td>
</tr>
<tr>
<td>likes</td>
<td>Person[0..*]</td>
<td>Message[0..*]</td>
</tr>
<tr>
<td>replyOf</td>
<td>Comment[0..*]</td>
<td>Message[1]</td>
</tr>
<tr>
<td>studyAt</td>
<td>Person[0..*]</td>
<td>University[0..*]</td>
</tr>
<tr>
<td>workAt</td>
<td>Person[0..*]</td>
<td>Company[0..*]</td>
</tr>
</tbody>
</table>

Table 3.1: Relationships connected through entities of different types

### 3.2.4 Scale Factors

LDBC SNB defines a set of scale factors (SFs), targeting systems of different sizes and budgets. SFs are computed based on the ASCII size in Gigabytes of the generated
output files using the CSV serializer. For example, SF 1 corresponds roughly to 1 GB in CSV format, SF 3 roughly to 3 GB and so on and so forth. The proposed standard SFs are the following: 1, 3, 10, 30, 100, 300, 1000. The Test Sponsor may select the SF that better fits their needs, by properly configuring the DATAGEN.

The size of the resulting dataset, is mainly affected by the following configuration parameters: the number of persons and the number of years simulated. Different SFs are computed by scaling the number of Persons in the network, while fixing the number of years simulated. Table 3.2 shows the parameters used in each of the SFs [EALP+15].

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>1</th>
<th>3</th>
<th>10</th>
<th>30</th>
<th>100</th>
<th>300</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Persons</td>
<td>11K</td>
<td>27K</td>
<td>73K</td>
<td>182K</td>
<td>499K</td>
<td>1.25M</td>
<td>3.6M</td>
</tr>
<tr>
<td>Number of years</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.2: Parameter for each scale factor [EALP+15]

3.2.5 LDBC SNB Business Intelligence Queries

<table>
<thead>
<tr>
<th>1</th>
<th>Posting summary</th>
<th>13</th>
<th>Popular tags per month in a country</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Top tags for country, age, gender, time</td>
<td>14</td>
<td>Top thread initiators</td>
</tr>
<tr>
<td>3</td>
<td>Tag evolution</td>
<td>15</td>
<td>Social normals</td>
</tr>
<tr>
<td>4</td>
<td>Popular topics in a country</td>
<td>16</td>
<td>Experts in social circle</td>
</tr>
<tr>
<td>5</td>
<td>Top posters in a country</td>
<td>17</td>
<td>Friend triangles</td>
</tr>
<tr>
<td>6</td>
<td>Most active posters of a given topic</td>
<td>18</td>
<td>Persons with a given number of posts</td>
</tr>
<tr>
<td>7</td>
<td>Most authoritative users on a given topic</td>
<td>19</td>
<td>Strangers interaction</td>
</tr>
<tr>
<td>8</td>
<td>Related topics</td>
<td>20</td>
<td>High level topics</td>
</tr>
<tr>
<td>9</td>
<td>Forums with related tags</td>
<td>21</td>
<td>Zombies in a country</td>
</tr>
<tr>
<td>10</td>
<td>Central Person for a tag</td>
<td>22</td>
<td>International dialog</td>
</tr>
<tr>
<td>11</td>
<td>Unrelated replies</td>
<td>23</td>
<td>Holiday destinations</td>
</tr>
<tr>
<td>12</td>
<td>Trending posts</td>
<td>24</td>
<td>Messages by topic and continent</td>
</tr>
</tbody>
</table>

Table 3.3: 24 BI Queries

The complete set of LDBC queries is shown in Table 3.3. To provide a better context for the features that they test in a database, we also include a summary of the special CYPHER operators used in these queries Table 3.4. The CYPHER version of all the queries is already provided by the LDBC benchmark, as part of code pertaining to already supported graph databases.

From these queries we have considered a small subset for our experiments, as we aim to evaluate rewrites on few of these queries. We have sorted the queries based on the number of nodes(vertexes) and relationships(edges) in each query. In each corresponding chapter we discuss why we have selected such a query for our study. In the following we provide a brief list of the queries selected as examples of global and local queries:

- **Query 1. Posting summary:** Find all the messages created before a given date.
<table>
<thead>
<tr>
<th>CYPHER feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATES</td>
<td>Takes set of values and calculates aggregated values over them. Some of the aggregation functions are sum(), avg(), min(), max(), count()</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Retrieves only unique rows depending on the columns that have been selected to the output.</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Constraints the number of rows in the output.</td>
</tr>
<tr>
<td>OPTIONAL MATCH</td>
<td>It matches the patterns against your graph database, but if no matches are found, OPTIONAL MATCH will use for missing parts of the pattern.</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>It is a sub-class following RETURN or WITH, it specifies output should be sorted and how.</td>
</tr>
<tr>
<td>REGEX</td>
<td>Used for filtering regular expression. We can match regular expressions using &quot;~regexp'</td>
</tr>
<tr>
<td>UNION</td>
<td>It combines the results of two or more queries into a single result set that includes all the rows that belong to all the queries in the union. The number and names of the columns must be identical.</td>
</tr>
<tr>
<td>UNWIND</td>
<td>Expands list into sequence of rows.</td>
</tr>
<tr>
<td>WITH</td>
<td>Allows query parts to be chained together piping the results from one to be used as starting point or criteria for next.</td>
</tr>
</tbody>
</table>

Table 3.4: CYPHER Query Features

- **Query 3. Tag Evolution:** Given a ‘month’ of the given ‘year’ find the Tags that were used in Messages and the Tags that were used during the next month.

- **Query 18. How many persons have a given number of posts:** For each Person, count the number of Messages they made (messageCount). Count only Messages that contain the given attributes.

- **Query 12. Trending Posts:** Find all Messages that received more than a given number of likes which are created after a given ‘date’.

- **Query 14. Top thread initiators:** Given ‘startDate’ and ‘endDate’ find the number of Posts created within that time interval for each Person and also the number of Messages in each of their reply trees, including the root Post of each tree.

After introducing the characteristics of the benchmark data, and the specific queries that we consider we introduce the systems that we test with: Neo4j with GraphAware, JanusGraph and ElasticSearch.
3.3 Selection of Tools

In this section we establish the tools that we selected to execute our evaluation.

3.3.1 Neo4j with GraphAware

The Property Graph model is a standard logical model for working with graphs. Neo4j is one of the popular Graph Databases, based on the Property Graph Model. It uses the Cypher Query Language. Neo4j provides expected database characteristics like ACID transaction compliance to ensure data integrity, cluster support, drivers for popular programming languages like Java, Python, .Net and JavaScript. In addition it claims to provide constant time traversal in big graphs, enabling it to scale to billions of entities due to efficient representation of nodes and relationships.

GraphAware is a family of plugins for Neo4j that manages the replication mapping of graph data into a search engine like ElasticSearch. The GraphAware Neo4j to ElasticSearch plugin makes this configuration easy and flexible. It uses a JSON format and the Spring Expression Language.

The GraphAware Neo4j to ElasticSearch plugin is an extension that takes care of replicating updates to the Graph as JSON documents to ElasticSearch. It serves as a common identifier for both ElasticSearch documents and Neo4j nodes and relationships. Based on replication mapping, it does the following:

- Inspect transactional changes asynchronously.
• Determine if an update is relevant to be replicated.
• Transform changes into JSON and send to ElasticSearch.

GraphAware uses the neo4j-uuid plugin to provide effortless integration with all of the GraphAware modules. The module uses the Spring Expression Language library offering the possibility to access node/relationship properties or define some conditionals in an expressive manner via a Java interface, making it possible to handle cases like [Bac13]:

• Time-based indexes
• Filtering nodes based on property or labels
• Indexing nodes or relationships in more than one index

3.3.2 JanusGraph

JanusGraph is an open source non-native graph database. It acts both as a stand-alone server, called as Gremlin-server, and a language-embedded client. JanusGraph can run in memory, without any backend, and it can also use as backends key-column value or key-value stores like BerkeleyDB, H-Base and Cassandra. In addition it can use search engines as indexes, which act through in-build data replication through user-defined mappings.

3.3.3 ElasticSearch

ElasticSearch is an open source, distributed, RESTful and analytics search engine built on top of Apache Lucene. As explained in Section 2.3 ElasticSearch uses inverted indexing of documents which allows very fast full-text searches. It can scale to petabytes of data easily.

With this we conclude our brief presentation on the systems used for our evaluation, in the next section we discuss the hardware employed for the test, next do data on versions of library.

3.4 Experimental Setup

Our evaluation was executed on a multi-core machine composed of 2 processors (8 cores in total) with 500 GB of memory. The application was running on Ubuntu 16.04 and java-1.8.0-openjdk-amd64. Detailed information of these processors is listed below.

• Product name: Intel(R) Core(TM) CPU 660 v2 @ 3.33GHz

[http://janusgraph.org/]
3.4. Experimental Setup

- Number of Cores: 2
- Number of Threads: 2
- Processor Base Frequency: 3.33 GHz
- Cache: 4096 KB
- Processor: 2

And the operation system information:

- LSB Version: core-9.20160110ubuntu0.2-amd64:core-9.20160110ubuntu0.2-noarch: security- 9.20160110ubuntu0.2-amd64:security-9.20160110ubuntu0.2-noarch
- Distributor ID: Ubuntu
- Description: Ubuntu 16.04.3 LTS
- Release: 16.04
- Codename: xenial

Most of the companies that use Neo4j, ElasticSearch and JanusGraph, work on hardware or cloud-based offerings. Our evaluation was set up using the above mentioned hardware.

3.4.0.1 Backend Configuration

For Neo4j and ElasticSearch there are many different versions available. We decided to use the following version which are both the latest, and suitable for our evaluations:

- **Neo4j**
  
  We have used Neo4j Community Edition 3.3.1 On Debian in our experiment. Prerequisites to install Neo4j on Debian is:
  
  - Neo4j 3.3.1 requires the Java 8 runtime to be installed.

  Using ’sudo apt-get install neo4j=3.3.1’ command we can install the Neo4j Community Edition.

- **ElasticSearch**
  
  In our evaluation we have used ElasticSearch 5.1.1 debian package.

- **LDBC Dataset : Software Requirements**

  1. LDBC DATAGEN 0.2.5 – https://github.com/ldbc/ldbc_snb_datagen : The data generator used to generate the LDBC datasets.

• **GraphAware**
  1. graphaware-neo4j-to-elasticsearch-3.3.1.51.7.jar
  2. graphaware-server-community-all-3.3.1.51.jar
  3. graphaware-uuid-3.3.1.51.14.jar

The requirements for executing LDBC SNB are limited to pure software requirements. LDBC SNB does not impose the usage of any specific type of system, as it targets systems of different nature and characteristics.

For loading LDBC data into databases we required the following software:

• **DATAGEN–ldbc-snb-datagen [Arn14]**: The data generator used to generate the datasets of the benchmark.

• **DATA IMPORT-ldbc-snb-implementations**: Used for loading generated datasets into the database.

For the latency measurements we used the Neo4j Browser Interface, the Java System timers (i.e., System.nanoTime()) and the ElasticSearch CURL-response times. For each evaluation we carried out 20 repetitions, and we report the average execution times.

### 3.4.0.2 Configuration for Mapping Tools

In [Chapter 4](#), we discuss about the types of mapping available between graph and search engines, and about our specific choices for this. In this section we disclose the configuration we used for the GraphAware tool, which is the tool needed for mapping between Neo4j and ElasticSearch; in addition we present cursory information to understand the tool.

**GraphAware : Neo4j-to-ElasticSearch**

The GraphAware Neo4j to ElasticSearch plugin is an extension that takes care of replicating updates to the Graph as JSON documents to ElasticSearch.

GraphAware replicates data from Graph database to Search engine as shown in workflow Figure 3.3 where the Neo4j transaction data is translated to a set of NodeCreated, RelationshipUpdated, NodeDeleted objects. Every object passes through replication mapping definitions and if the condition is satisfied the ElasticSearch actions along with JSON document content is returned back as a set of actions which will be executed in bulk against ElasticSearch cluster. Every mapping definition is responsible for evaluating the condition along with the index and type of the document. Using graph object properties, nodes, labels, relationship types finally JSON representation of the document is built. 
For setting up GraphAware we needed to install and configure the plugin by placing them into the Neo4j directories. We also added the below configuration to Neo4j.conf file.

3.4.1 Dataset Generation

The LDBC data set needs to be generated and preprocessed before loading it to the database. The LDBC data generator uses Apache Hadoop version 2.6.0. to ease the execution of Hadoop in a provided ‘run.sh’ script. The following variables were configured in the run.sh script [Arn14]:

- **PARAM-GENERATION**: We have used the datagen with scalefactor 1 in for our implementation.

The main configuration was set through the file params.ini in the ldbc-snb-datagen directory as shown in Figure 3.5. We disclose thus the configuration for the scale factor and the corresponding serializers that we used.

After configuring the run.sh script and executing it a “Social Network” folder is created which contains the ldbc data in the form csv files.

Ldbc data contains the entities like City, Comment, Country, Organization, Company, Forum, Message, Tag, Person, TagClass, University, Post and relationships like containerOf, hasCreator, hasInterest, hasMember, hasTag, hasType, isLocatedIn, isPartOf, isSubclassOf, knows, likes, replyOf, studyAt, workAt.

Ldbc data schema Figure 3.1 defines the structure of the data used. We used the generated data, without modifications, save for changes needed for more efficient mapping. These changes are explained in Chapter 5.
3.5. Summary

3.4.2 Dataset Loading

For full disclosure and to make our work artifact reproducible, we disclose the small changes we performed to make the generated data compatible with Neo4j. For Janus-Graph there were no necessary changes.

The CSV files generated by ldbc datagen required a bit of preprocessing before loading into graph database like replacing headers with Neo4j compatible ones, replacing labels (for example: city to City), conversion of date and datetime formats using the convert-csvs.sh script in ldbc_snb_Implementations.

Just for illustration, in Figure 3.6 we show a snapshot describing the loaded data in Neo4j. Here entities(nodes) are stored in separate store file and relationship types are stored in separate relationship store file respectively.

![Database Information](image)

Figure 3.6: LDBC SNB data loaded into Neo4j

With this we conclude our disclosure of the datasets, systems and benchmarks used for our evaluation. We expect that the information provided in this chapter can contribute towards making our experiments a reproducible artifact. In the next section we summarize and conclude this chapter.

3.5 Summary

In this chapter we proposed a list of research questions. These questions cover the different mappings used to evaluate the performance of Search Engines, and also benefits of using global and local queries over the Search Engine.
In this chapter we also discussed about the benchmark used, data generation and data schema. We concluded this chapter with the experimental setup, the backend versions and some details about our data loading process.

In the next chapter we discuss about the different mappings possible from graph database to search engine and also how these mappings impact the performance potential from using the search engine.
4. Mappings from Graph Database to Search Engine

In this brief chapter we will discuss about different types of mappings from graph database to search engines and the impact of these mappings on the utility of search engines. This information is necessary to understand the tests that we carry out to answer our research questions, in the subsequent chapters. The outline for this chapter is as follows:

- **Research Questions**: We start by defining the research questions that are discussed in this chapter (Section 4.1).
- **Mappings**: Here we discuss different methods to map data from graph to search engine (Section 4.2).
- **Summary**: We summarize our chapter in this section (Section 4.3).

### 4.1 Research Questions

In this chapter we address the following research question:

1. What are possible mappings between graph entities and search engine documents? How do these different mappings impact the utility of the search engine for graph queries?

These are introductory questions that we answer by studying the possible implementation choices, rather than through implementation. The second part of the research question could not be addressed by implementation (i.e., comparing an efficient mapping against one based on join data types), since in our choice of benchmark we did not find a specific query that could enable us to compare the utility of alternative mappings.
4.2 Mappings

Here we discuss on the alternatives to map the data from a graph to a search engine representation, following our observations from studying the possibilities offered by the search engine and the graph databases. We establish the alternatives that we select for our further evaluation. We also mention the fields that we created (which were not originally there), in order to support better mappings.

4.2.1 Types of Mappings

- **Basic Mapping**: By default in all tools we have found nodes and edges are mapped and indexed separately. This can be done either using GraphAware Neo4j-to-ElasticSearch replication, or through the JanusGraph default configuration. Unlike the Neo4j case, where all fields are mapped by default once the plugin is set and the entities are configured to be mapped (using a configuration file), in JanusGraph the user must decide which fields from the entities are mapped.

- **Mapping relationships**:
  - Efficient use of edge data: In order to make efficient use of edge data, it necessary that the edge documents contain information about the vertexes that are connected to them. This is the prerequisite to enable navigation from edges to vertexes when using the search engine only. For this to be realized it is necessary first that the ids of vertexes are present, without any encoding, in the search engine representation of the vertex. This is not the case, by default, of the JanusGraph mapping, therefore it is necessary that we explicitly set the ids of the vertexes as a new property to be mapped. Once the ids of the vertexes are available, it is also necessary to explicitly copy the ids of the vertexes to the edges. Thanks to this, traversals in the search engine can be supported by the terms-lookup mechanism, which requires querying one index type, to get values, and searching over the other index using a term query, which is like a join, and is inefficient.
  - Inheritance relationships: GraphAware treats inheritance relationships in a specialized way by replicating each child as the parent type, and duplicating the data. This actually provides an opportunity for traversing because hopping through inheritance relationships can be avoided, with queries only directed to type of the parent document. However it is necessary to mark in the parent document the type of the child, such that the types can be distinguished. We achieve this through an extra property that acts as a flag indicating the type for each parent document (i.e., either parent, or what specific kind of child document).
  - Join Data Types for parent-child relations and Nested relationships: A Join Data Type is a special field in a search engine like ElasticSearch that creates parent/child relationships within documents of the same index. Nested relationships, on the other hand, are a modeling alternative where we simply
create nested mappings, which means that a document will contain a list of sub-documents within. Both Join Data Types and Nested relationships could be adopted for efficient mapping of relationships. However these approaches are not currently supported by our chosen tools. GraphAware indexes node and relationship data into separate indexes, and ElasticSearch does not support using the Join Data Type over separate indexes. Though JanusGraph can map the data to a same index, it does not support the Join Data Type as a data type and does not accept the mapping. Furthermore, nested documents are not supported by any of the tools.

4.2.1.1 Basic Mapping

In our study we have used GraphAware Neo4j-ElasticSearch module to replicate the data in Neo4j graph database to ElasticSearch. Using this module for replication all the nodes where indexed separately to ElasticSearch from Neo4j and all the relationships where also indexed separately.

4.2.1.2 Mapping Relationships with Efficient Use of Edge Data

In our study we adopt a basic solution for mapping relationships. We enhance the document representation with the necessary information to make efficient use of edge data, and hence we can employ the terms-lookup mechanism from ElasticSearch for queries which require some traversing, where we query one index type get the ids from that type and look for that id in other index type. This basic solution also enables us to use aggregations on ids which can be an efficient tool for solving some kinds of queries.

Data Manipulation (Handling)

To map the data from graph to search engine we have updated the csv files from LDBC Data with some fields which where not present initially. The following fields where added into csv files:

- SRCID (source id) and TGTID (target id) fields where added to the relationship files as seen in Figure 4.1.

For example, in csv file comment_hasCreator_person.csv we have added two new columns TGTID and SRCID of type Long, where TGTID is taken from the START_ID of Comment and SRCID is from the END_ID of Person. So each relationship was mapped to particular nodes using the TGTID and SRCID, once the data is updated and loaded into the search engine. We can see in Figure 4.2 how updated data with SRCID and TGTID is reflected in the search Engine.

Using the SRCID and TGTID we where able use the terms lookup mechanism from ElasticSearch to get the ids of one index type and look for those ids in the other index type.
4.2.1.3 Mapping Relationships with Join Data Types

In traversal queries using the search engine hops can be the “choke-points”. When using a basic approach (described in the preceding subsection), the processing for a transitive closure\(^1\) would work as follows:

- Given some source ids, just traverse one at a time through the relationships, until there are no more results. In each hop there is the requirement to store the newly found ids as source ids for the following hop.

In ElasticSearch the Join Data Types could provide an alternative way for this processing. To set these data types we need to carry out the following steps:

- We need to create the parents first.
- Then we need to create the children. For this we have to supply the child document with the information of the document, and also the information of the parentID. This means that we will have to enrich the vertex data with some edge data. We also observe the limitation that we can only have one parent per child. This means that data is like a tree (where each node does not have multiple parents), and it is not like a more general graph.

We find that in the LDBC schema there are few cases that match the hierarchical expectation of the join data type. One case is the scenario of Messages and Replies, where a reply cannot come from different messages.

---

\(^1\)We can define, informally, a transitive closure, starting at a given vertex and using a given evaluation criteria, as the collection of entities that are found by hopping in one given direction from the origin vertex and evaluating the criteria (e.g. that edges are of a certain relationship), until no more hops are possible.
Given a parentId, the search engine is capable of answering a query about all documents having that as a parent. This means that for the traversal use case in our evaluation, we would have to first determine the parentIds for the source (e.g. from the message), and then do separate queries (e.g. on the replies) for each hop. This, does not change (apart from the fact that we use a single index and document type), the procedure without the Join Data Types.

Therefore we consider that there is no specific use case where join data types can be exploited in the LDBC schema.
A use case for the Join Data Type would work as follows: Given a vertex we want to find the connected vertexes in one hop, which also match certain criteria on the vertexes themselves. For example, we could search for all vertexes that have a given vertex as a parent, but that also fulfill some other criteria on vertex properties. This could benefit from join data types since the vertex filtering criteria can be applied in the same query than the traversal, which is not the case when the traversal happens on the edges alone.

4.2.1.4 Use of Mappings in the Queries of our Study

As stated in Chapter 3, we have selected a small subset of queries for our evaluation. In this subsection we briefly discuss the mappings used for the queries.

Query 1 is a query over a single node type, and it does not have any joins or relationships in it. For our evaluation we use aggregations and the scripting language from ElasticSearch, along-side the Neo4j Cypher query. The basic mapping by GraphAware was sufficient to query directly into that particular node and get the results. More details on this query processing is discussed in Chapter 5.

The remaining queries, that is Query 3, Query 18, Query 12 and Query 14, are queries with more than one node type and one or more relationship types. We mapped the relationships with the alternative that we called “efficient mapping of edge data”, and thus we were able to use the terms-lookup mechanism as the choice for supporting traversals.

4.3 Summary

In this chapter we introduce the different possible mappings to map data from graph database to search engines. We also discussed about the data handling to map relationships into search engines. And also about how join data type works in ElasticSearch and why we could not use join data type for our evaluation. As a result we do not provide a performance evaluation to answer the research questions for this chapter, instead our discussion is based on the study of implementation choices when adopting the specific graph databases and search engine that we studied.
5. Global Graph Query Rewrites

In this chapter we discuss about the gains possible from rewriting global graph queries, where no more than one hop is required, such that they could be executed directly in the search engine. The outline for this chapter is as follows:

- **Research Questions**: We start by defining the research questions that we address through experimental evaluation in this chapter (Section 5.1).

- **Global Graph query Evaluation**: Next we study, through an implementation and query re-writes, the gains possible by evaluating the global graph queries in the search engine (Section 5.2).

- **Summary**: We conclude by summarizing the findings in our chapter (Section 5.3).

### 5.1 Research Questions

In this chapter we answer the following research question:

2. What are the gains possible from rewriting global graph queries (i.e., no more than 1 hop involved) such that they could be executed directly in the search engine, assuming a naive mapping?

In the next section we discuss how we addressed this question.

### 5.2 Global Graph query Evaluation

In order to answer our research question we select a global query from the LDBC SNB-BI workload, provide a rewrite such that it can be answered through the search engine alone, and compare the performance of this method against the execution on Neo4j and JanusGraph. We present first the results for Neo4j and then for JanusGraph.
5.2.1 Rewrite for Global Query and Profile of Execution

Global graph queries are OLAP queries which search through a large amount of nodes, with complex analysis, but require no hop or not more than one hop.

In our study we have found one global query from the 24 LDBC BI queries which requires not more than one hop. We can see the pattern see the description of the query along with parameters to be given as input and the expected output in sorted order, in Figure 5.1

<table>
<thead>
<tr>
<th>query</th>
<th>BI</th>
<th>read</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Posting summary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pattern</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>desc.</td>
<td>Given a date, find all Messages created before that date. Group them by a 3-level grouping:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. by year of creation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. for each year, group into Message types: is Comment or not</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. for each year-type group, split into four groups based on length of their content</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0: 0 &lt;= length &lt; 40 (short)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 1: 40 &lt;= length &lt; 80 (one liner)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 2: 80 &lt;= length &lt; 160 (tweet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 3: 160 &lt;= length (long)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>params</td>
<td>1 date Date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 messageYear</td>
<td>32-bit Integer</td>
<td>R</td>
<td>Year of the Message</td>
</tr>
<tr>
<td>2 isComment</td>
<td>Boolean</td>
<td>M</td>
<td>true for Comments, false for Posts</td>
</tr>
<tr>
<td>3 lengthCategory</td>
<td>String</td>
<td>C</td>
<td>0 for short, 1 for one-liner, 2 for tweet, 3 for long</td>
</tr>
<tr>
<td>4 messageCount</td>
<td>32-bit Integer</td>
<td>A</td>
<td>Total number of Messages in that group</td>
</tr>
<tr>
<td>5 averageMessageLength</td>
<td>32-bit Integer</td>
<td>A</td>
<td>Average length of the Message content in that group</td>
</tr>
<tr>
<td>6 sumMessageLength</td>
<td>32-bit Integer</td>
<td>A</td>
<td>Sum of all Message content lengths</td>
</tr>
<tr>
<td>7 percentageOfMessages</td>
<td>32-bit Float</td>
<td>A</td>
<td>Number of Messages in group as a percentage of all messages created before the given date</td>
</tr>
<tr>
<td>sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 year</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 isComment</td>
<td>false (true, i.e. the ordering puts Posts first, and Comments second)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 lengthCategory</td>
<td>order based on the length of the category, 0 (short), 1 (one liner), etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.1: Query 1: Description and Pattern

We can describe the query as follows: Given a creationDate, find all the messages that were created before such date. Once we get a list of results, then those messages should

be grouped by several criteria: \textit{year of creation}, whether the \textit{message is a comment or not}, and \textit{length category} for the message, for each year.

Length categories are defined as follows:

1. If length $< 40$ return 0
2. If length $> 40$ but $< 80$ return 1
3. if length $> 80$ but $< 160$ return 2
4. if length $\leq 160$ return 3

Apart from the grouping criteria, the output should include the \textit{total number of messages per group}, \textit{average message length per group}, \textit{sum of message length per group} and \textit{percentage of messages per group (with respect to the total retrieved items)}.

After establishing the query itself, we can now describe how it was written for the search engine.

\textbf{Rewrite for Global Query}

The ElasticSearch query is as shown in Figure 5.2. We will explain more in detail about the query below:

- Initially we connect to the ElasticSearch browser using 'localhost:9200', since we are querying on one of the nodes will search in index \textit{neo4j-index-test-node}.
- \textit{size} is mentioned to define how many terms bucket should be returned out of overall terms list, by default the bucket for top ten terms are returned.
- Then will filter all the messages with creationDate $\leq 20110817111021570$.
- We have used terms aggregation from ElasticSearch bucket aggregation which is a multi-bucket value source based aggregation where buckets are dynamically built - one per unique value.
- Using terms aggregation we look for messages which are comment and return true or false, size=2 is mentioned as the isComment value can be true or false.
- Then we search for messages based on the length category using terms aggregation.
- Next we get the Year from creationDate using the scripting language provided by ElasticSearch. \textit{inline} specifies the source of the script. The scripting language is used to retrieve the custom parameters and apply to each document queried.
- We get the average message length using the \textit{avg} aggregation and the sum of message length using \textit{sum} from the ElasticSearch pipeline aggregation.
• We get the message count using the value_count aggregation.
• Percentage is calculated using the bucket script aggregation, a parent pipeline aggregation. It executes a script which can perform per-bucket computations on specified metrics in the parent multi-bucket aggregation. The specified metric must be numeric and the script must return a numeric value. We have calculated the percentage using the scripting language.

• Finally we get message count, average message length, sum of message length and percentage of message length for each year, based on the length category considering if the message is comment or not.

Profile for Execution of Global Query

In this brief subsection we describe the profiling results for executing the query over the search engine. These results are akin to the EXPLAIN query in SQL systems, and can be obtained by using the EXPLAIN API (_explain) of ElasticSearch.

1. As described in previous sections ElasticSearch data is stored in the form of shards, in our index neo4j-index-test-node is split into 5 shards as:
   - neo4j-index-test-node[0]
   - neo4j-index-test-node[1]
   - neo4j-index-test-node[2]
   - neo4j-index-test-node[3]
   - neo4j-index-test-node[4].

2. Each shard searches for the message with the given criteria i.e, creationDate lesser than the provided parameter (i.e., 20110817111021570).

3. The search is further decomposed into BooleanQuery, where the description is +ConstantScore(_type:Message), it is further decomposed into children of type ConstantScoreQuery where the description is ConstantScore(_type:Message) and the children is further grouped into children of type TermQuery where the description is: _type:Message and creationDate:[-9223372036854775808 TO 20110817111021570].

4. Its further decomposed into collector of name CancellableCollector and children of name MultiCollector.

5. Multicollector consists of children of name TotalHitCountCollector and ProfilingAggregator.

6. ProfilingAggregator is broken down into aggregations of type RangeAggregator, StringTermsAggregator, AvgAggregator, SumAggregator, ValueCountAggregator.

7. The same steps (from step 2 to step 6) are carried out for the remaining shards like neo4j-index-test-node[1], etc.
After presenting in this section our re-write of a graph query to be executed in a search engine, and some details about the profiling information, we present in the next section how the query is addressed by the connected graph databases, and we disclose the results from our performance evaluation of these alternatives.

5.2.2 Performance Evaluation of Neo4j

Global Query in Cypher

```cypher
MATCH (message:Message)
WHERE message.creationDate <= 20110817111021570
WITH toFloat(count(message)) AS totalMessageCount |
MATCH (message:Message)
WHERE message.creationDate <= 20110817111021570 
  AND message.content IS NOT NULL
WITH 
  totalMessageCount, 
  message, 
  message.creationDate/10000000000000 AS year
WITH 
  totalMessageCount, 
  year, 
  message:Comment AS isComment, 
  CASE 
    WHEN message.length < 40 THEN 0 
    WHEN message.length < 80 THEN 1 
    WHEN message.length < 160 THEN 2 
    ELSE 3 
  END AS lengthCategory, 
  count(message) AS messageCount, 
  floor(avg(message.length)) AS averageMessageLength, 
  sum(message.length) AS sumMessageLength
RETURN 
  year, 
  isComment, 
  lengthCategory, 
  messageCount, 
  averageMessageLength, 
  sumMessageLength, 
  messageCount / totalMessageCount AS percentageOfMessages
ORDER BY 
  year DESC, 
  isComment ASC, 
  lengthCategory ASC
```

Figure 5.3: Neo4j Cypher Query: Posting Summary
5.2. Global Graph query Evaluation

The Neo4j Cypher Query 1 is shown in Figure 5.3. The queries are provided as part of the LDBC repository, which includes some code for databases already studied with the benchmark.

The Cypher implementation for Query 1 can be explained, in detail, as follows:

1. First we search for all the messages with creationDate <= '20110817111021570', count all the messages that match the criteria and return it as totalMessageCount.
2. Second we check if the message content is not null.
3. Third we get year from creationDate by calculation creationDate/10000000000000000.
4. Fourth we check if the message is a comment or not and name it as isComment, if the message is comment it returns true.
5. Then messages are grouped based on the length, and this returned as a lengthCategory.
6. Count the messages based on the lengthCategory.
7. Calculate the sum of the length and return as sumMessageLength, count the average as averageMessageLength percentage for each group of length category as percentageOfMessages.
8. Finally sort the results with year in descending order, with the message iscomment in ascending and length category in ascending order.

![Figure 5.4: Neo4j Output : Posting Summary](image)

Profile for Execution of Global Query in Neo4j with Cypher

We were able to generate a profile of the execution of the Neo4j Cypher query, by using an EXPLAIN command, where a direct acyclic graph representing the physical query plan, is generated. These results are enclosed in Figure 5.5.
Figure 5.5: Profile for Execution of Global Query in Neo4j
After executing the query in the Neo4j browser we can see in that Neo4j has taken 25284 milliseconds to execute the query.

### 5.2.3 Performance Evaluation of JanusGraph

#### Global Query in JanusGraph

Below we will explain in detail about how we wrote the query in JanusGraph, using the Gremlin query language. We did not have any reference on how LDBC queries were to be written for Gremlin, thus we tried out different implementations and settled on the best of our possible expressions for the query.

We should note that we employ indexes in JanusGraph, to guarantee that our performance reports are not affected by omitting such configuration.

```java
g.V()
    .has("creationDate", P.lte(20110817111021570L)).
    has("vType", P.within("post","comment")).
    has("content").
    group().
    by(__.values("creationDate")).
    map(it->(Long (it.get())/10000000000000L)).
    by(__.group()).
    by(__.values("isComment")).
    by(__.values("length")).
    map(it->
        (Integer)(it.get())<40?0:
        (Integer)(it.get())<80?1:
        (Integer)(it.get())<160?2:
        3).
    groupCount()).

next();
```

To summarize the query, first, we filtered vertexes based on their creation date, their type, and their availability of content. Second, we grouped the results based on the year, then on the flag isComment, and finally on the length categories. We returned as a result the group count. We did not return the average, sum or percentages; however as we will see in Section 5.2.4, the performance for JanusGraph for this simpler query was lower than the others, therefore we believe that comparatively, the difference between the performance of JanusGraph will be even larger for the more complete query.

The output from JanusGraph for query 1 is as shown in Table 5.1. The time taken by the query to execute in JanusGraph is 1251149 milliseconds.

### 5.2.4 Comparison of Results

After executing the query we can see in that ElasticSearch has taken 256 milliseconds to execute the same query executed by Neo4j.
Figure 5.6: ElasticSearch Output : Posting Summary

Figure 5.7 presents the comparison of the execution time for the LDBC query 1: (Posting summary) in Neo4j, JanusGraph and ElasticSearch. Comparatively, the query rewritten for ElasticSearch-only runs 99x and 4887x faster than the counterpart queries over Neo4j and JanusGraph, respectively. Hence, we establish that ElasticSearch, and search engine processing can indeed be more efficient than graph engines for supporting global queries without joins/traversals. This result is accomplished by adopting terms aggregation and scripting query features from ElasticSearch, which contributed to faster data access and retrieval. Alternatively, for ElasticSearch we could have decomposed the query into sub-queries, leading to a different performance result than the one we report. We believe, nonetheless that our rewrite represents a efficient solution to answer the query, as shown in the performance results.

<table>
<thead>
<tr>
<th>year</th>
<th>isComment</th>
<th>lengthCategory</th>
<th>messageCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>false</td>
<td>0</td>
<td>113845</td>
</tr>
<tr>
<td>2010</td>
<td>false</td>
<td>2</td>
<td>33932</td>
</tr>
<tr>
<td>2010</td>
<td>false</td>
<td>3</td>
<td>4188</td>
</tr>
<tr>
<td>2010</td>
<td>true</td>
<td>0</td>
<td>244715</td>
</tr>
<tr>
<td>2010</td>
<td>true</td>
<td>1</td>
<td>36409</td>
</tr>
<tr>
<td>2010</td>
<td>true</td>
<td>2</td>
<td>81394</td>
</tr>
<tr>
<td>2010</td>
<td>true</td>
<td>3</td>
<td>5440</td>
</tr>
<tr>
<td>2011</td>
<td>false</td>
<td>0</td>
<td>181005</td>
</tr>
<tr>
<td>2011</td>
<td>false</td>
<td>2</td>
<td>36531</td>
</tr>
<tr>
<td>2011</td>
<td>false</td>
<td>3</td>
<td>4433</td>
</tr>
<tr>
<td>2011</td>
<td>true</td>
<td>0</td>
<td>272651</td>
</tr>
<tr>
<td>2011</td>
<td>true</td>
<td>1</td>
<td>39910</td>
</tr>
<tr>
<td>2011</td>
<td>true</td>
<td>2</td>
<td>90509</td>
</tr>
<tr>
<td>2011</td>
<td>true</td>
<td>3</td>
<td>6016</td>
</tr>
</tbody>
</table>

Table 5.1: JanusGraph Output : Posting Summary
5.3 Summary

In summary, in this chapter we studied the use of the search engine for supporting global graph queries that do not involve traversals. To this end we selected a query with this pattern from the LDBC-SNB BI Workload. We provided a novel rewrite of the query for the search engine, we used an implementation of the query for Neo4j using CYPHER and we offered a novel implementation of the query for JanusGraph using Gremlin. We provided, in addition, profiles describing the physical behavior of the query. From our evaluation we found that by using the search engine the query can be answered 99x faster than by using the Neo4j graph engine alone, and 4887x faster than when using the JanusGraph graph engine alone. In the next chapter we will present our results for a similar evaluation, but extended to local graph queries, where traversals are required.
6. Adjacency Graph Query Rewrites

In this chapter we will discuss about the gains possible from rewriting adjacency(local) queries where more than 1 hop is involved, such that they could be executed directly in the search engine. The outline for this chapter is as follows:

- **Research Questions**: We start by defining the research questions that we address through experimental evaluation in this chapter (Section 6.1).

- **Adjacency Graph query Evaluation**: Here we evaluate the gains possible by rewriting the adjacency graph queries, such that they could be executed by the search engine (Section 6.2).

- **Summary**: We summarize our chapter in this section. (Section 6.5).

### 6.1 Research Questions

In this chapter we answer the following research question:

3. What are the gains possible from rewriting adjacency queries (i.e., more than 1 hop involved) such that they could be executed directly in the search engine, assuming a naive mapping?

In the next section we present how we addressed this question.
6.2 Adjacency Graph query Evaluation

Here we discuss our evaluation on the gains possible from rewriting adjacency graph queries using Neo4j graph database and ElasticSearch as search engine and present our results.

Adjacency graph queries are OLTP queries which are like friends recent likes, that require more than one hop. In our study we have found that most of the LDBC SNB BI queries are adjacency queries with more than one hop.

As mentioned in Section 3.2.5, we selected a subset of queries that correspond to this pattern from the LDBC SNB BI workload, based on the number of nodes and relationships that need to be evaluated to answer the query. Specifically, to evaluate the adjacency queries we focused on queries 18, 3, 12 and 14. To our understanding these are queries with a moderate level of complexity within the workload.

In the next sections (Section 6.2.1, Section 6.2.2, Section 6.3 and Section 6.4) we discuss each query, how we rewrote it to be used for the search engine, the performance on Neo4j and then a comparison with the performance on ElasticSearch.

For our evaluation we do not include JanusGraph, since we were not able to achieve efficient executions for the queries. Perhaps with more work in evaluating optimizations we could have reached better performance with JanusGraph, but it was outside of the scope of this project.

6.2.1 Adjacency Query: Query 18

In Figure 6.1 we can see the description of the query along with parameters to be given as input and the expected output, in sorted order. This query answers to the question: How many persons have a given number of posts?

In this query we have to count the number of messages created by the particular person and for each message count value we have to count the number of persons with exactly that message count. Some initial filtering criteria must be fulfilled, like: the message content cannot be empty, the message length should be below a threshold, the creation date should be after a given date, and the messages should be written in a given language. To evaluate such criteria requires already to consider several entities. The language is retrieved from the Post entities, using the language attribute, while the content, message length and creation date are asserted on Message entities that are connected through a replyOf transitive closure relationship (i.e., any number of hops) with one of the Posts in the given language. Furthermore, messages are filtered on the above mentioned criteria. Finally, all the Person entities immediately connected through the hasCreator relationship with the messages selected become then the pivotal point for an aggregation on the count of messages from the previous criteria, being returned in sorted order.

Messages can be either a Post or a Comment.

6.2. Adjacency Graph query Evaluation

### Rewrite for Adjacency Query

Queries of this kind require joining two indexes i.e., in the Neo4j-GraphAware mapping, `neo4j-index-test-node` and `neo4j-index-test-relationship` (this is our current case). As stated previously, to perform this join with a Join Data Type in a single query is currently not supported by ElasticSearch. So, for now, we have used a terms query from ElasticSearch to get the ids from one index type and query the other index type using the ids from the previous step. This requires a certain effort and can’t be queried directly in ElasticSearch, but, for efficiency requires to use a Programming Language Client. If joins were better supported in search engines, then plausibly this query could have been answered with a single search engine query.

In the following we describe our rewrite for the query 18, using the terms lookup mechanism from ElasticSearch query builders to build a terms query from values contained in another document:
1. First we search in Messages for all the Messages matching the criteria and we get the ids.

```java
BoolQueryBuilder matchingPosts = new BoolQueryBuilder();
    matchingPosts.must(QueryBuilders.termQuery("_type", "Message"));
    matchingPosts.must(QueryBuilders.rangeQuery("creationDate")
        .gt(20110722000000000L));
    matchingPosts.must(QueryBuilders.rangeQuery("length").lte(20L));
SearchResponse response = client.prepareSearch()
    .setIndices("neo4j-index-test-node")
    .setQuery(matchingPosts)
    .setFetchSource("id", null)
    .setScroll(new TimeValue(60000))
    .setSize(10000).get();
```

2. Second we search in Messages for all the Posts matching the language criteria, in addition to the other criteria, and we get the ids.

```java
matchingPosts = new BoolQueryBuilder();
    matchingPosts.must(QueryBuilders.termQuery("_type", "Message"));
    matchingPosts.must(QueryBuilders.termQuery("isComment", "false"));
    matchingPosts.must(QueryBuilders.termQuery("language", "ar"));
    matchingPosts.must(QueryBuilders.rangeQuery("creationDate")
        .gt(20110722000000000L));
    response = client.prepareSearch()
        .setIndices("neo4j-index-test-node")
        .setQuery(matchingPosts)
        .setFetchSource("id", null)
        .setScroll(new TimeValue(60000))
        .setSize(10000).get();
```

3. Third, we hop as much times as needed (7 times), starting from those ids, through the Reply_Of relationship, so that we get the ids of all the messages in that language, which in addition fulfill the criteria preselected (this is asserted by looking up their ids in from those in the first step). This operation is akin to joins, because we query node ids over the edge documents (Reply_Of relationship).

```java
BoolQueryBuilder matchingMsgs = new BoolQueryBuilder();
    matchingMsgs.must(QueryBuilders.termQuery("_type", "REPLY_OF"));
    matchingMsgs.must(QueryBuilders.termsQuery("TGTID", idsNextHop));
SearchResponse response2 = client.prepareSearch()
    .setIndices("neo4j-index-test-relationship")
    .setQuery(matchingMsgs)
    .setFetchSource("SRCID", null)
```
4. Fourth, we filter out from the ids of the previous step, those that do not match the criteria on Messages, where we do a self-join.

5. Finally we use the ids from the previous step to search for people that created the messages (in Has_Creator) group them manually, which is again similar to a join.

```java
BoolQueryBuilder matchingPeople = new BoolQueryBuilder();
    matchingPeople.must(QueryBuilders.termQuery("_type", "HAS_CREATOR"));
    matchingPeople.must(QueryBuilders.termsQuery("SRCID", idListOfMatchingMessages));
    response2 = client.prepareSearch()
        .setIndices("neo4j-index-test-relationship")
        .setQuery(matchingPeople)
        .addAggregation(AggregationBuilders.terms("agg").field("TGTID")
                        .size(10000000).get();
```

6. The collected results are sorted using the Java arrays library, before returning to the user.

### 6.2.1.2 Performance Evaluation of Neo4j

The Neo4j Cypher version of the query is in Figure 6.2.

![Neo4j Cypher Query](image)

Figure 6.2: Query 18: Neo4j Cypher Query

### 6.2.1.3 Comparison of Results

We have executed the LDBC query 18, as described above, in both Neo4j and ElasticSearch. We find that the execution time taken by Neo4j CYPHER is 10x faster than the ElasticSearch, queries as shown in Figure 6.3.
This performance variation between Neo4j and ElasticSearch could be alleviated with
denormalization (where filtering criteria is copied to the edges, to avoid joining). In
addition, this might be improved by exploiting the Join Data Type framework, but, as
mentioned above. It is not currently supported.

6.2.2 Adjacency Query: Query 3

Here in Figure 6.4 we can see the description of the query along with parameters to
be given as input and the expected output in sorted order. This query portrays tag
evolution.

In this query we find the Tags that were used in Messages during the given month of
the given year and the Tags that were used during the next month. For the Tags, and
for both months, we compute the count of Messages. We should return the tag name
and the countMonth1 i.e, the occurrence of a tag during the given year and month and
countMonth2 i.e, the occurrence of a tag during the next month after the given year
and month. Finally we also return the diff, that is the difference between countMonth1
and countMonth2 in sorted order with a limit in the output size of 100.

6.2.2.1 Rewrite for Adjacency Query

We describe our rewrite of query 3 as follows:

1. First we search for messages with creationDate as year and month matching the
criteria. With year as 2010 and month as 10.

2. Second we search for the hasTag that have as source id (SRCID) the message ids that we have, and from that we get all the target Ids (TGTID).

3. Then we store in a local hash table each target Id with the count of how many times it appeared in the first month.

4. We repeat this for the second month. With the year as 2010 and month as 11.

5. Then, we calculate the difference (\(\text{diff}\)) between values, and we sort them descending, and get the top ids.

6. For each month we return the \textit{tag name}, \textit{countMonth1} (which we got from step 3) and \textit{countMonth2} (which we got from step 4) and the difference of \textit{countMonth1} and \textit{countMonth2} (retrieved from step 5).

### 6.2.2.2 Performance Evaluation of Neo4j

The Neo4j Cypher Query for Query 3 is as shown in Figure 6.5.
6. Adjacency Graph Query Rewrites

6.2.2.3 Comparison of Results

We have executed the LDBC query 3 in both Neo4j and ElasticSearch and can report that the execution time taken by Neo4j is 4x faster than the ElasticSearch counterpart, as seen in Figure 6.6.

6.3 Adjacency Query: Query 12

In Figure 6.7 we present the description of the query along with parameters to be given as input and the expected output in sorted order. This query corresponds to the study of trending posts.

![Cypher Query]

Figure 6.5: Query 3 : Neo4j Cypher Query

The query can be described as follows: Find all Messages created after a given date (exclusive), that received more than a given number of likes (likeThreshold). For each message with message id and creationdate, return the firstName and the lastName of the person who created the post and the likeCount i.e, the number of likes Post received in sorted order, with a cap of 100 on the output size.

6.3.1 Rewrite for Adjacency Query

We rewrite query 12 for the search engine as follows:

1. First, we search for messages and get the ids of those matching the criteria creationDate > 20110721220000000.

2. Second, we perform a terms query and aggregation from the ElasticSearch query builder on the likes relationship that have source ids within the message ids retrieved from the previous step, and we get all the target ids and the like counts, in sorted order. In this stage we can filter the messages by their like counts.

3. We perform a terms query on the hasCreator relationship for the messages retrieved, and we store the source id as message id and target id as person id.

4. We use the target id of hasCreator to retrieve the matching firstName and lastName from the person entities.

5. Finally, we assemble the results and sort them.
6. Adjacency Graph Query Rewrites

6.3.2 Performance Evaluation of Neo4j

The Neo4j Cypher version of query 12 is as shown in Figure 6.8.

6.3.3 Comparison of Results

We have executed the LDBC query 12 in both Neo4j and ElasticSearch and we have found that the execution time taken by Neo4j is 4.8x faster than the ElasticSearch rewrite, as seen in Figure 6.9.

6.4 Adjacency Query: Query 14

In Figure 6.10, we can see the description of the query 14, along with parameters to be given as input and the expected output in sorted order. This query computes the top thread initiators.

The query can be described as follows: For each Person, we count the number of Posts that they created in the time interval [startDate, endDate] (equivalent to the number of threads they initiated), and the number of Messages in each of their (transitive)
6.4. Adjacency Query: Query 14

Figure 6.8: Query 12: Neo4j Cypher Query

```
MATCH
  (message:Message)-[:HAS_CREATOR]-(creator:Person),
  (message)<-[like:LIKES]-(:Person)
WHERE message.creationDate > 2011072122000000
WITH message, creator, count(like) AS likeCount
WHERE likeCount > 400
RETURN
  message.id,
  message.creationDate,
  creator.firstName,
  creator.lastName,
  likeCount
ORDER BY
  likeCount DESC,
  message.id ASC
LIMIT 100
```

Figure 6.9: Performance evaluation for Query 12

reply trees, including the root Post of each tree. When calculating Message counts we only need to consider messages created within the given time interval. Return for each Person, the number of Posts that they created, and the count of all Messages that
appeared in the reply trees (including the Post at the root of the tree) that they created, in sorted order and with the output being limited to a size of 100.

<table>
<thead>
<tr>
<th>query</th>
<th>B1/ read / 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Top thread initiators</td>
</tr>
</tbody>
</table>

**Pattern**

For each Person, count the number of Posts they created in the time interval \([\text{startDate}, \text{endDate}]\) (equivalent to the number of threads they initiated) and the number of Messages in each of their (transitive) reply trees. When calculating Message counts only consider messages created within the given time interval.

Return each Person, number of Posts they created, and the count of all Messages that appeared in the reply trees (including the Post at the root of tree) they created.

**Parameters**

1. \(\text{startDate} \) \(\text{Date}\)
2. \(\text{endDate} \) \(\text{Date}\)

**Result**

1. \(\text{person.id} \) \(\text{64-bit Integer} \) \(\text{R}\)
2. \(\text{person.firstName} \) \(\text{String} \) \(\text{R}\)
3. \(\text{person.lastName} \) \(\text{String} \) \(\text{R}\)
4. \(\text{threadCount} \) \(\text{32-bit Integer} \) \(\text{A}\) The number of Posts created by that Person (the number of threads initiated)
5. \(\text{messageCount} \) \(\text{32-bit Integer} \) \(\text{A}\) The number of Messages created in all the threads this Person initiated

**Sort**

1. \(\text{messageCount} \) \(\downarrow\)
2. \(\text{person.id} \) \(\uparrow\)

**Limit**

100

Figure 6.10: Query 14 : Description and Pattern

### 6.4.1 Rewrite for Adjacency Query

Our rewrite of query 14 to be executed on the search engine can be explained as follows:

1. First, we select \(\text{posts} \) with \(\text{creationDate} \geq 20120531220000000 \) and \(\text{creationDate} \leq 20120630220000000\); we retrieve the list of post ids.

2. Second, we apply a terms query on has_Creator and search for creators that have as source id one item in the post ids. From this query we retrieve all target ids, which correspond to all creators of the matching posts.
3. Then we store in a hash table each target id, and the count of how many times they appeared, this is the `threadCount`, in addition this initializes the `messageCount` of `posts` per creator.

4. Third, we get the list of message ids that match the criteria `creationDate>20120531220000000` and `creationDate<20120630220000000`.

5. Then we perform aggregations in reply__Of over the target ids (using a terms query on message ids). With this step we are counting the first hop of all messages that match the criteria and are an immediate reply of the posts determined in the first step.

6. We add this to the hash table with the results from previous step, and this is the updated `MessageCount` to the first hop.

7. We loop on the previous two step, until no more messages can be retrieved. In this way we count the size of the transitive tree of replies.

8. We perform a terms query on the `person id` obtained from step 2., and get the person `firstName` and `lastName`.

9. Finally, for each person display their `person id`, `firstName`, `lastName`, `postCount`(threadCount) and `messageCount` in sorted order, limited to 100.

### 6.4.2 Performance Evaluation of Neo4j

The Neo4j Cypher version of query 14 is shown in Figure 6.11.

![Figure 6.11: Query 14 : Neo4j Cypher Query](image)

### 6.4.3 Comparison of Results

We have executed the LDBC query 14 both on Neo4j and ElasticSearch, and we have found that the execution time taken by Neo4j is 8x faster than the corresponding time for the ElasticSearch queries, as seen in Figure 6.12.
In summary, in this chapter we carried out a performance comparison of the execution of adjacency graph queries, both on a graph engine and a search engine. Our study was based on careful rewrites of the selected queries, striving to provide an efficient rewrite, using terms aggregations and advanced features, when possible. These rewrites required us to decompose each single graph query into several queries for the search engine, hence we employed a programming language client of ElasticSearch and managed intermediate results within this programming environment. We find that Neo4j can perform 10x, 4x, 4.8x and 8x faster than the search engine counterpart. We consider that the limited performance of the search engine for these tasks might alleviated through some data denormalization, which might aid in performing filtering queries while avoiding joins. Other performance improvements could possibly be achieved on later offerings of the search engine.

With this we conclude our evaluation on the applicability of the search engine for graph queries. In the next chapter we wrap-up our study by summarizing our findings from Chapter 5 and Chapter 6. We also disclose threats to the validity of our findings and we propose future work to advance the research we conducted in this project.
7. Conclusion and Future Work

7.1 Summary

The goal of our study was to evaluate the potential of the search engine to support graph queries, complementing the graph database query engine.

To this end we selected for our implementation an open-source distributed graph database: JanusGraph, and a commercial graph database Neo4j. As a representative of search engines, we selected the open source search and analytics engine ElasticSearch. In addition we choose the LDBC benchmark, employing the data generated by the benchmarking tool and a selection of queries from the Social Network Benchmark, and more specifically, from the Business Intelligence Workload. This workload was selected for our study, since it was developed to evaluate complex analytics involving vast areas of the graph and adjacency queries.

In order to study these alternatives, we proposed a series of research questions about gains possible by rewriting global and adjacency graph queries over search engines when compared to their execution over a graph database. While this may be a very limited focus and few people have studied it, executing graph queries over the search engine mappings can indeed lead to performance gains on occasions. In our work we are not comparing the performance of search engines and graph databases with the aim of replacing the latter with the former; instead we look for a more perfect union of both. To support our study we have replicated the graph database into our search engine, ElasticSearch, both using a module called “Graphware-neo4j-to-elasticsearch” and user-defined configurations for JanusGraph. We listed the possible alternatives to map graph data into the search engine, for efficient use of the latter, with the databases selected:

- Basic mapping, where each entity (i.e., nodes and relationships) is stored as a separate document.
• Improved mapping for relationships

  – Explicitly indexing the vertex ids, and the replicating the ids of the connected vertexes to the edges. Both of these changes are the prerequisites to enable traversals in a search engine.

  – Inheritance relationships are mapped in a special way in different tools. For example, GraphAware offers a native support wherewith child members of an isClassOf relationship are stored twice, following the schema of either the parent or the child.

  – Join Data Types and Nested Relationships are not supported in current mapping tools.

Based on a practical assessment we established that the domain of application for Join Data Types and Nested Relationships might still be limited, and that data denormalization might be necessary to benefit from such approaches, when they are supported. We evaluated the gains achieved by rewriting global and adjacency queries in both graph databases (Neo4j and JanusGraph) and search engine (ElasticSearch).

The findings of our experiments using these BI queries from LDBC SNB are included in Chapter 5 and Chapter 6.

We have found that search engines perform really well for global graph queries, achieving 99x and 4887x speedups for query 1 over the execution on graph databases.

For adjacency queries the search engine faces several challenges, and in our rewrites we are not able to surpass the performance of the graph query engine. We report that the graph engine performs, comparably, 10x, 4x, 4.8x and 8x faster. The results of our work may vary with different mappings and use of scripting features, nested queries and parent-child relationship from ElasticSearch query builder.

We hope that our study can contribute to bring attention to the role of the search engine as a complement to graph databases, and that improvements in mappings enable further gains. We also expect that our experience in rewriting queries could contribute towards a theory for effective automated rewrites.

In the next sections we disclose some threats to the validity of our findings and we propose future work:

7.2 Threats to validity

• Internal Threats:

  – Rewriting of global and adjacency queries using more features from search engine like use of scripting language might have contributed to better performance search engine. Future studies should also explore more query builder features and also join two indexes (nodes and relationships) into one index.
which will allow using parent-child relationship, scripting and nested query features from search engines which will impact in performance of search engine over graph queries in a better way. Similarly, our evaluation of JanusGraph is accomplished through our own implementation, and hence its efficiency is susceptible for improvements in a further implementation.

- In our observations of the results we are comparing a terminal curl request (for the case of ES-only), in-browser DB-request (for the case of Neo4j), requests from Java ES-clients and Java-based Gremlin clients (for the case of JanusGraph). As a consequence there might be some intermediate components that influence our observations, other than the search engine alone, e.g. such as the time for serializing the results back into the requesting clients. This could constitute an internal threat to the validity of our conclusions.

- **External Threats:**
  - We have used a single machine for our evaluation, with default configurations for Neo4j and JanusGraph. Perhaps the use of more parallel processing and optimizations, both in the graph and search engine could lead to different results.
  - Datasets, technologies, hardware, configuration and implementation choices might have affected in some ways the performance of our results to be replicated in other cases.
  - We have used only system generated dataset and queries, namely from LDBC in our evaluation. But researchers usually provide experiments both using specific system-generated datasets and real-world datasets. System-generated datasets are needed to verify the effectiveness of certain features but real world data is more authentic. Similarly, different queries could provide more immediately applicable results.
  - In our study of mappings, we decide not to evaluate the use of denormalization. Perhaps employing some amount of denormalization is common practice in real-world systems, and hence it might be acceptable to provide tests using such techniques.

### 7.3 Future work and Concluding Remarks

- **Can graph databases answer some search engine queries?:** In our experiment we have evaluated the potential of search engine over graph queries. So now future work might be if search engines queries can be executed in graph database and evaluate the potential of graph database over search engine queries (e.g. for aggregations, and for batch relevance scoring).

- **Research on Improving the Mapping:** If we had better mappings it would be easier to capitalize on the possibilities of the search engine for graph adjacency
queries. Perhaps scripting features from the search engine could be adopted for more fine-tuned mappings. It would be a useful feature, to reduce query time, to create nested documents containing vertexes and their relationships.

- **Research on Automated Query Rewrites**: In our work we establish that there is indeed potential for the search engine to support global graph queries. This, however, is still a knowledge available to developers but not to the query engine of a graph database. It would be a potentially rewarding job to work in including operations on the replicated search engine data as part of the graph engine query planning. Through this approach the graph engines and the search engines might be able to form a more perfect union.
Bibliography


[HR05] Robert A Hanneman and Mark Riddle. Introduction to social network methods, 2005. (cited on Page 8)


